МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
имени Н.Э. БАУМАНА

Учебное пособие

Учебно-методический комплект

по курсу "Основы конструкторско-технологической информатики" Конспект лекций

МГТУ имени Н.Э. Баумана

ОГЛАВЛЕНИЕ

Введение в среду разработки Microsoft Vidual Studio. Написание простейших программ.

Обзор среды разработки Microsoft Visual Studio

Visual C++ является частью Microsoft Visual Studio 2005 - комплекта средств разработки приложений. Visual C++ - это интегрированная среда разработки, и все создаваемые с помощью нее приложения представляют собой проекты.

Проект - это набор взаимосвязанных исходных файлов компиляция и компоновка, которых позволяет создать исполняемую Windows программу или DLL.

Исходные файлы проекта хранятся в отдельном каталоге, кроме того, проект часто зависит от внешних файлов, таких как подключаемых (include). В проекте Visual C++ завимозависимости между отдельными компонентами описаны в текстовом файле проекта с расширением VCPROJ. А специальный текстовый файл решения с расширением SLN содержит список всех проектов данного решения.

Решение (Solution) - набор проектов, объединенных вместе, которые решают одну задачу.

Для того чтобы начать работу с существующим проектом, необходимо открыть в Visual C++ соответствующий SLN файл. Типы файлов создаваемых в проекте Visual C++ указаны ниже:

Таблица 1.1
« Типы файлов проекта VS C++»

Расширение файла	**Описание**
APS	Поддержка просмотра ресурсов
BSC	Информация браузера
IDL	Файл на языке описания интерфейсов IDL
NCB	Поддержка просмотра классов
SLN	Файл решения
SUO	Поддержка параметров и конфигурации решения
VCPROJ	Файл проекта

С технической точки зрения Visual C++ представляет собой один из инструментов Visual Studio. С помощью этой интегрированной среды, вы можете использовать любые другие языки программирования, в том числе разработанные не Microsoft. Так выглядит открытый проект в среде MS Visual Studio:

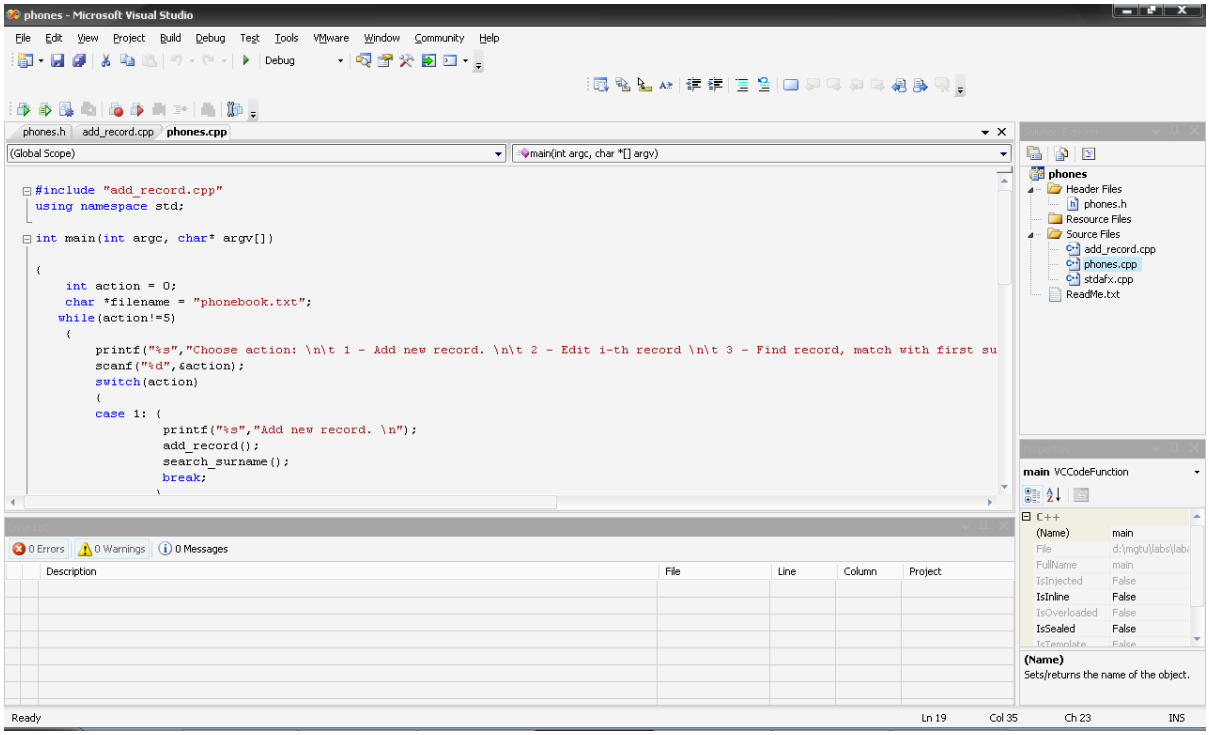


Рис. 1.1 Открытый проект MS Visual Studio

В рабочей области среды разработки содержится окно редактора (см. рис. 1.2) для ввода программного кода, окно Обозревателя решений и проектов (Solution Explorer, см. рис. 1.3) и окно Обозревателя свойств (Properties, см. рис. 1.4) текущего (выбранного) объекта.

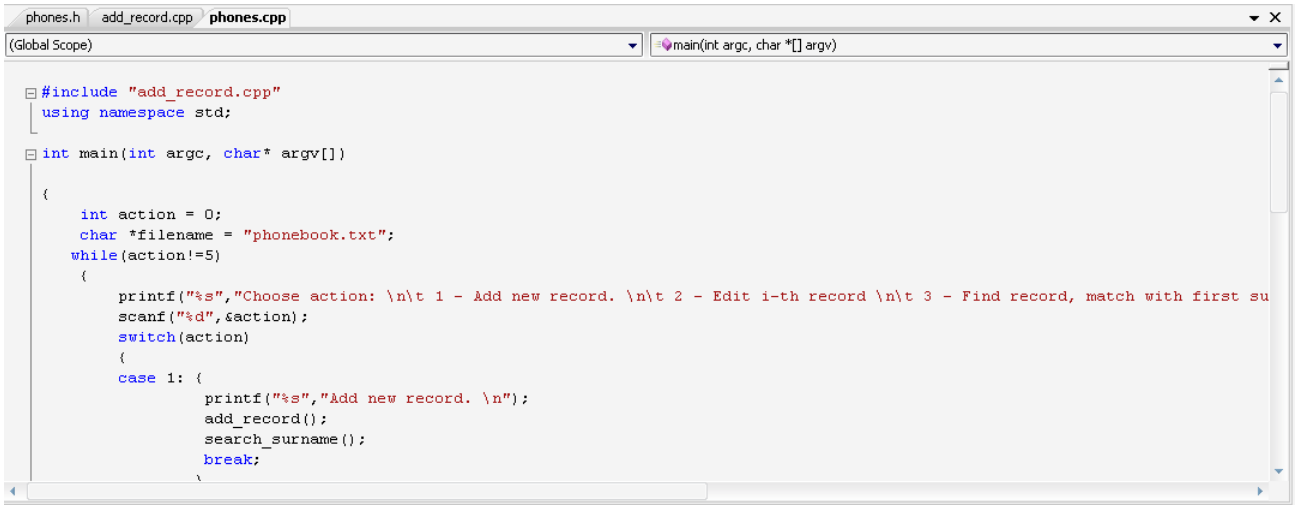


Рис. 1.2 Окно редактора кода.

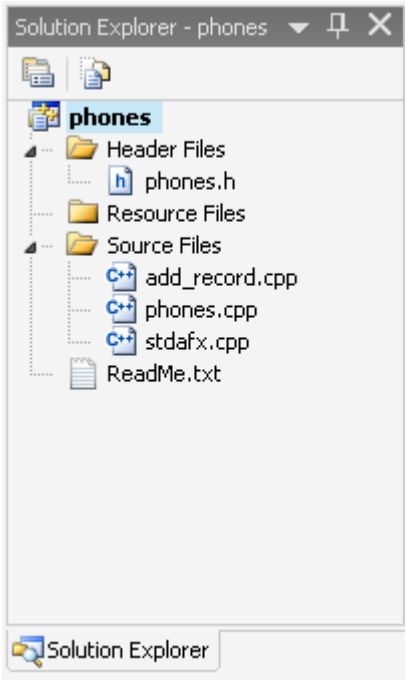


Рис. 1.3. Окно обозревателя решений и проектов.

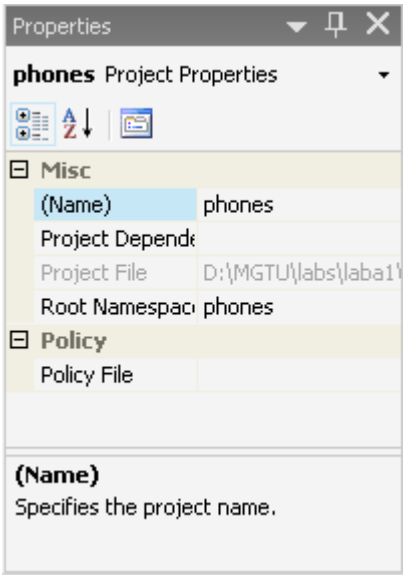


Рис. 1.4. Окно обозревателя свойств.

Основные мастера и утилиты MS Visual Studio 2005

Мастера для создания проектов:

1. **MFC Application Wizard (exe)** - мастер для создания проектов Windows-приложений на основе классов библиотеки MFC. Мастер предоставляет программисту богатый выбор настроек проекта. С его помощью можно создавать приложения с однодокументным, многодокументным или диалоговым интерфейсом. Однодокументное приложение позволяет пользователю работать только с одним файлом. Многодокументное приложение может одновременно предоставить работу с несколькими документами, каждым в собственном окне. Пользовательский интерфейс диалогового приложения представляет собой единственное диалоговое окно.
2. **MFC DLL Wizard** - этот мастер приложений позволяет создать структуру DLL, основанную на MFC. При помощи него можно определить характеристики будущей DLL.
3. **ATL Project Wizard** - это средство позволяет создать элемент управления ActiveX или сервер автоматизации, используя новую библиотеку шаблонов ActiveX (ActiveX Template Library - ATL). Опции этого мастера дают возможность выбрать активный сервер (DLL) или исполняемый внешний сервер (exe-файл).
4. **Custom Wizard** - при помощи этого средства можно создать пользовательские мастера AppWizard. Пользовательский мастер может базироваться на стандартных мастерах для приложений MFC или DLL, а также на существующих проектах или содержать только определяемые разработчиком шаги.
5. **Visual Studio Add-in Wizard** - мастер дополнений позволяет создавать дополнения к Visual Studio. Библиотека DLL расширений может поддерживать панели инструментов и реагировать на события Visual Studio.
6. **MFC ActiveX Control Wizard** - мастер элементов управления реализует процесс создания проекта, содержащего один или несколько элементов управления ActiveX, основанных на элементах управления MFC.
7. **Win32 Project Wizard** - этот мастер позволяет создать проект обычного Windows-приложения или динамически подключаемой библиотеки. Тип проекта определяется выбором соответствующих опций в диалоговых окнах мастера. Проект создается незаполненным, файлы с исходным кодом в него следует добавлять вручную.
8. **Win32 Console Application Wizard** - мастер создания проекта консольного приложения. Проект консольного приложения создается пустым, предполагая добавление файлов исходного текста в него вручную.

Разработка программ в среде Microsoft Visual Studio.

Создание проекта.

Программы (более часто именуемые *приложениями*), создаваемые в среде разработки MS VS.NET, представляются в виде *проекта*, понимаемого как объединение всех необходимых для построения программы файлов. Близкие по назначению проекты могут объединяться в наборы проектов – *решения* (*solutions*). Как результат, при начале разработки программы необходимо создать проект, размещаемый в создаваемое по умолчанию решение.

Для создания проекта необходимо выполнить:

1. Запустить MS VS.
2. Для создания нового проекта в диалоговом окне Начальная страница (Start Page) необходимо нажать кнопку Create New Project. В появившемся диалоговом окне New Project (см. рис. 1.5) нужно выполнить следующие действия:
   - В поле Name задать имя создаваемого проекта
   - В поле Location установить папку для размещения файлов проекта
   - В области Project Types выбрать вариант Visual C++ Projects,
   - В области Templates выбрать вариант Console Application.
   - По завершении всех перечисленных действий необходимо нажать кнопку OK.

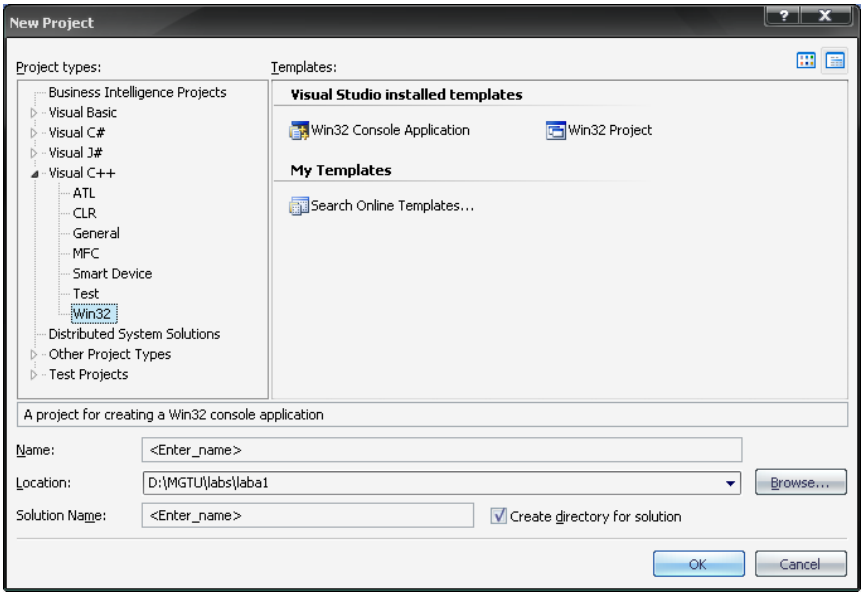


Рис. 1.5 Окно создания нового проекта.

Основным для начальной работы в среде разработки MS VS .NET является редактор программного кода. Редактор MS VS .NET обеспечивает все стандартные действия, которые доступны для любого другого редактора (набор программного кода, редактирование, копирование, вставка, поиск и т.д.) и, кроме того, обладает большим набором дополнительных возможностей, значительно помогающих разработчикам создавать большие и сложные программные системы. Ориентируясь на начальное знакомство со средой разработки, рассмотрим несколько полезных свойств редактора кода, которые могут оказать заметное практическое содействие программисту при подготовке даже самых простых программ.

Автоматическая проверка правильности текста.

Редактор программного кода поддерживает оперативную (в процессе ввода текста) проверку правильности ввода программы – ключевые слова алгоритмического языка опознаются и выделяются (обычно синим) цветом. При этом, если использование ключевых слов происходит неправильно (не соответствует синтаксическим правилам языка программирования) данное ключевое слово будет подчеркиваться красной волнистой линией. Как результат, при наборе программного кода следует внимательно следить за цветовой окраской ключевых слов и выделением синтаксически неправильных конструкций программного кода.

Получение справочной информации.

Для получения справочной информации нужно установить текстовый курсор на элемент программы, для которого необходимо наличие справки, и нажать клавишу F1 (следует отметить, что справка будет выдана на английском языке; кроме того, получение справки возможно только в случае, если на компьютере установлена справочная служба MSDN Library – данная служба поставляется при приобретении MS VS). Как правило, получаемая информация содержит всю необходимую для программиста информацию, обеспечивая, тем самым, действенную помощь при разработке программ. В большинстве случаев, справочная информация дополнена примерами практически использования рассматриваемых элементов.

Автоматизированная поддержка набора текста.

Для оказания максимального содействия программисту для быстрого и безошибочного набора программного кода в редакторе среды MS VS имеется специальная служба IntelliSense, которая обеспечивает: отображение списка методов и полей для классов, структур, пространства имен и других элементов кода (вывод списка осуществляется автоматически после ввода имени и последующего за ним одного из разделителей "." (точка), "->" или "::"; выбор нужного варианта может быть выполнен, например, при помощи двойного щелчка мыши на требуемой строке списка или при помощи последовательного нажатия клавиш <Tab> и <Enter>); отображение информации о параметрах для методов и функций – вывод данной информации также осуществляется автоматически после ввода имени метода или функции (см. Рис. 1.6). Отображение краткого описания элементов кода программы (вывод описания происходит при наведении указателя мыши на нужный элемент кода). Завершение слов при наборе наименований команд и имен функций (для использования этой возможности следует набрать несколько первых символов вводимого имени и нажать одновременно клавиши <Ctrl> и <Пробел>, выбор нужного варианта, как и ранее, производится при помощи двойного щелчка мыши или клавиш <Tab> и <Enter>). Автоматическое сопоставление правильности расстановки скобок (набираемые скобки },],), #endif выделяются более темным цветом вместе с соответствующей открывающейся скобкой).

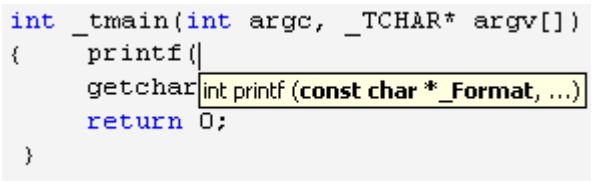


Рис. 1.6. Всплывающая подсказка

Процедура построения исполняемой программы.

Для выполнения программы, подготовленной на алгоритмическом языке, необходимо осуществить достаточно длинную цепочку технологических действий – программу нужно откомпилировать и убедиться, что в ней отсутствуют синтаксические ошибки, далее программу надо собрать ("слинковать") вместе со всеми используемыми служебными модулями - в результате в рамках платформы MS VS получается готовая к исполнению сборка (assembly) на промежуточном языке (Microsoft Intermediate Language, MSIL или просто IL). При запуске на выполнение сборка должна быть переведена с промежуточного языка в исполняемую программу в командах компьютера, на котором будет работать сборка – реализацию данного шага выполняют JIT-компиляторы общей среды выполнения (Common Language Runtime, CLR) платформы MS .NET (англ. JIT – Just In Time – в нужный момент).

Запуск сборки на выполнение.

Построение сборки (команда Build пункта меню Build) и запуск ее на выполнение (команда Start Debugging пункта меню Debug) могут быть выполнены раздельно, однако достаточным является и применение одной команды Start Debugging, т.к. при выполнении этой команды проверяется соответствие имеющейся сборки и программного кода в редакторе и, если после времени построения последнего варианта сборки в программном коде были поведены какие-либо изменения, то автоматически будет вызван JIT-компилятор и сформирован новый вариант сборки. Выполнение команды Start Debugging, как можно увидеть в пункте меню, можно обеспечить и простым нажатием клавиши F5. При запуске на выполнение подготовленной на предшествующих шагах программы могут возникнуть две различные ситуации:

− программа подготовлена правильно, в этом случае запуск сборки произойдет без обнаружения ошибок, на экране дисплея мелькнет окно вывода результатов и практически моментально исчезнет. Для наблюдения итогов выполнения программы окно вывода надо задержать – это можно обеспечить, например, при помощи вызова процедуры ввода перед завершением метода main. В этом случае при переходе на вызов метода **getchar()** выполнение программы будет приостановлено и мы получим возможность рассмотрения результатов вывода программы (см. рис. 1.7).

Рис. 1.7 Окно выполняемой программы.

Для продолжения работы программы достаточно нажать клавишу Enter (отметим еще раз, что метод ввода getchar() используется в данном примере только для организации приостановки окна вывода, а не для реального ввода данных);

− другая ситуация возникает при обнаружении ошибок при построении сборки – в этом случае, естественно, выполнение сборки невозможно и для ее подготовки необходимо найти и исправить все имеющие ошибки в программном коде программы.

Обработка синтаксических ошибок.

При обнаружении синтаксических ошибок, компилятор в диалоговом окне Microsoft Development Environment выводит сообщение There were build errors. Continue? для ответа на которое следует нажать кнопку «Нет». В результате компиляция программы завершается, в окне Output выводится сообщение Build: 0 succeeded, 1 failed, 0 skipped и для каждой обнаруженной ошибки в окне Task List приводится ее краткое описание. Так, например, если в правильной программе нашего учебного примера убрать символ ";" после функции getchar(), сообщение об ошибке имеет вид (см. рис. 1.8).

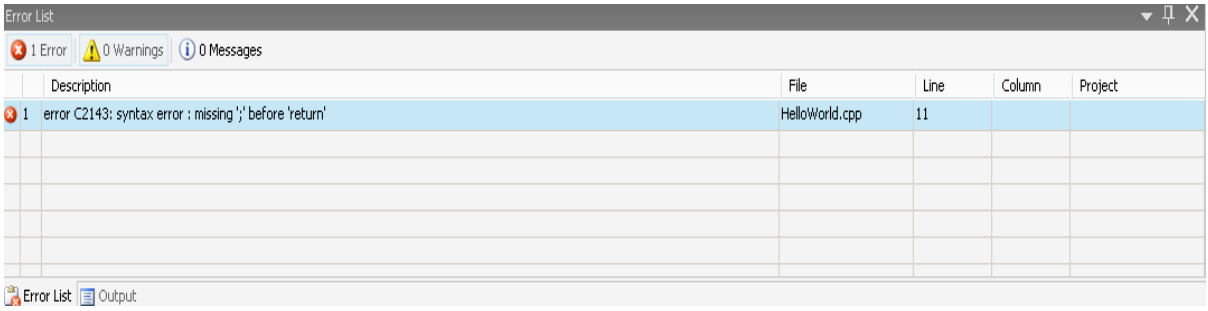


Рис. 1.8 Сообщение о синтаксической ошибке.

syntax error : missing ';' before 'return'. Сообщение об ошибке можно выделить и, нажав клавишу F1, получить справочную информацию по допущенной ошибке. Нажав клавишу Enter (или выполнив двойной щелчок мыши) можно перейти в окно редактора на строчку с оператором, в котором была обнаружена ошибка. Следует отметить, что "правильное понимание" выдаваемых компилятором сообщений требует определенной практики (если бы компилятор мог абсолютно точно выделять ошибочные ситуации в программе, то тогда исправление ошибок могло бы происходить автоматически).

Методы поиска ошибок (отладки)

Процесс выявления причин обнаруженной ошибки, определение места (локализация) ошибки в программе и исправление ошибочно реализованного программного кода обычно называется отладкой. Как правило, при отладке существует некоторая предварительная стадия, во время которой программист выдвигает те или иные предложения о причинах ошибочной работы и проводит визуальный анализ (инспекцию) программного кода – к сожалению, данной формой отладки многие программисты пренебрегают, хотя эффективность такого способа отладки является достаточно высокой. Если при инспекции кода выявить ошибки не удается, далее наступает основной способ отладки – отладочное выполнение программы (или трассировка), в ходе которого работа программы может быть приостановлена для просмотра значений тех или иных переменных программы с целью обнаружения ситуаций, когда эти значения не соответствуют предполагаемых. Тем самым, задача трассировки – обнаружения информационных признаков проявления ошибки.

Пошаговое выполнение программы. Для выполнения программы в пошаговом режиме (в режиме трассировки) используются четыре команды, которые доступны из меню Debug, панели инструментов Debug и клавиш быстрого вызова:

− команда Step Info (клавиша F11) обеспечивает последовательное, строка за строкой, выполнение программного кода программы (включая содержимое вызываемых методов),
Команда Step Over (клавиша F10) обеспечивает, как и предшествующая команда Step Info, последовательное выполнение программы, но при этом вызов методов рассматривается как один неделимый шаг (т.е. без перехода внутрь вызываемых методов), − Команда Step Out (клавиша Shift+F11) обеспечивает выполнение всех оставшихся строк программного кода текущего выполняемого метода без останова, позволяя выполнить быстрый переход в последнюю точку вызова,
Команда Run to Cursor (клавиша Ctrl+F10) обеспечивает выполнение без останова программного кода между текущей строки останова и позицией курсора (в зависимости от настроек параметров среды MS VS данная команда может отсутствовать в пункте меню Debug).

Удобным средством указания точек останова процесса выполнения программы является использование контрольных точек (breakpoints). Для определения контрольной точки необходимо щелкнуть мышкой на вертикальной полосе слева от нужной строки программного кода; повторный щелчок отменяет установки контрольной точки. В ходе выполнения программы при попадании на контрольную точку происходит останов; для продолжения работы необходимо выполнить команду Continue пункта меню Debug.

Наблюдение значений переменных. Для наблюдения значений переменных в момент останова выполнения программы достаточно расположить указатель мыши на имени переменной – в результате значение переменной появится в виде всплывающей подсказки. Дополнительная возможность для наблюдения значений переменных состоит в использовании специальных окон наблюдения:

Окно Autos отображает значения всех переменных, используемых в текущей и предшествующей строках точки останова программы; в окне отображаются названия переменных, их тип и значения; окно Autos обычно располагается в нижней левой части экрана и для его высветки необходимо щелкнуть мышью на ярлычке с названием окна; − Окно Locals отличается от предшествующего окна Autos тем, что отображает значения всех переменных текущей области видимости (т.е. переменных текущего выполняемого метода или его локального блока);

Удобный способ добавления переменных в окна наблюдения состоит в использовании техники "Взять и перенести" (выделить имя переменной, нажать левую кнопку мыши и, не отпуская ее, переместить указатель мыши в окно наблюдения, после чего отпустить конку мыши). Для удаления переменных из окна наблюдения достаточно выделить соответствующую строку и нажать клавишу <Delete>. Близким по назначению к окнам Watch является окно Quick Watch, которое дополнительно позволяет изменять значения наблюдаемых переменных; для высветки окна необходимо выделить нужную переменную и выполнить команду Quick Watch пункта меню Debug.

ЗАДАНИЯ

1. Написать по аналогии с примером «Hello, world» программу выводящую приветствие типа «Good afternoon, your name!», (создать новый проект).
2. Написать программу спрашивающую у пользователя имя, считывающую введенное имя и выводящую приветствие. Для ввода имени использовать функцию scanf(“%s”, &name) и переменную char name[16].

Типы переменных языка C++

Основные типы данных

Чтобы реализовать алгоритм, программам необходимо работать с данными - числами, символами, т.е. объектами, которые несут в себе информацию, предназначенную для использования. Некоторые данные устанавливаются равными определенным значениям еще до того, как программа начинает выполняться, а после ее запуска такие значения сохраняются неизменными на всем протяжении работы программ. Эти *данные* называются константами. Данные, которые могут изменяться, или же им могут быть присвоены значения во время выполнения программы, называются переменными. Различие между переменной и константой очевидно: во время выполнения программы значение переменной может быть изменено (например, с помощью присваивания), а значение константы изменить нельзя.

Кроме различия между переменными и константами существует еще различие между типами данных. Некоторые данные в программе являются числами, некоторые - символами. Компилятор должен уметь идентифицировать и обрабатывать данные любого типа. В языке Си предусмотрено использование нескольких основных типов данных. Если величина есть константа, то компилятор может распознать ее тип только по тому виду, в котором она присутствует в программе. В случае переменной необходимо, чтобы ее тип был объявлен в операторе описания. В стандарте языка Си используется семь ключевых слов, указывающих на различные типы данных:

int
long
short
unsigned
char
float
double

Первые четыре ключевых слова используются для представления *целых*, т.е. целых чисел без десятичной, дробной части. Если мы хотим подчеркнуть, что целое не может быть отрицательным, то нужно к целому подписывать ключевое слово unsigned, например, unsigned short, char предназначено для указания на буквы и другие символы. float, double используются для представления чисел с десятичной точкой. Типы, обозначаемые этими ключевыми словами, можно разделить на два класса по принципу размещения в памяти машины. Первые пять ключевых слов определяют целые типы данных, последние два - типы данных с плавающей точкой.

Описание различных типов, переменные и константы

Целые числа. У целого числа никогда не бывает дробной части. Представив целое число в двоичном виде, его нетрудно разместить в машине. Например, число 3 в двоичном виде выглядит как 11. Если его поместить в слово 32-разрядной машины, необходимо первые 30 бит установить в 0, а последние 2 бита - в 1.

Числа с плавающей точкой. Числа с плавающей точкой соответствуют тому, что математики называют вещественными числами. Способ кодирования, используемый для помещения в память числа с плавающей точкой, полностью отличается от размещения целого числа. Числа с плавающей точкой представляют в виде дробной части и порядка числа, а затем обе части размещают в памяти.

Все данные типов *int, short, long* являются числами со знаками, т.е. значениями этих типов могут быть только целые числа - положительные, отрицательные и нуль. Один бит используется для указания знака числа, поэтому максимальное число со знаком, которое можно представить в слове, меньше, чем максимальное число без знака.

Описание данных *целого типа*. При описании данных необходимо написать только тип, за которым должен следовать список имен переменных. Например, int dog, rad, nina. В качестве разделителя между именами переменных необходимо использовать запятую. Целые константы. Согласно правилам языка Си, число без десятичной точки и без показателя степени рассматривается как целое. Поэтому компилятор по записи константы определяет, целая она или вещественная. Если нужно ввести константу типа long, то нужно указать признак L или l в конце числа. Если при записи константы целое начинается с цифры 0, то эта константа интерпретируется как восьмеричное число, если же целое начинается с символа 0x или 0X - как шестнадцатеричное число.

Инициализация переменных *целого типа*. Константы применяются при инициализации переменных. Это означает присваивание переменной некоторого значения перед началом обработки. Можно инициализировать переменную в операторе описания.

Например,

```
int dog=5;
int rad=077;
int nina =0X99;
```

Описание данных типа *unsigned*. Этот тип является модификатором типов: int, short, long. Мы можем использовать комбинацию ключевых слов unsigned int, unsigned short, unsigned long, т.е. переменная не может принимать отрицательного значения. Для указания типа unsigned int достаточно написать unsigned. Целые беззнаковые константы записываются так же, как и обычные константы, запрещено только использование знака минус. Например:

```
short s = 65535;
unsigned short u = 65535;
printf("short = %d \n unsigned short = %d ", s, u);
```

Выдас для первой переменной значение -1, а для второй 65535.

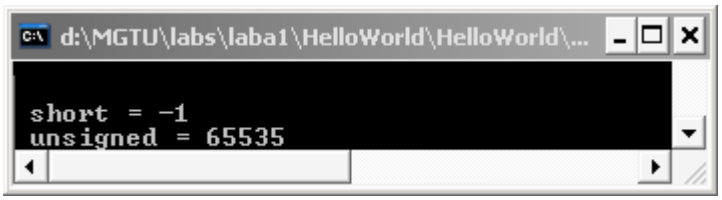


Рис. 1.9 Вывод значений short и unsigned short

Описание данных типа *char*. Этот тип определяет целые числа без знака в диапазоне от 0 до 255. Такое целое обычно размещается в одном байте памяти. Для описания символьной переменной применяется ключевое слово char. Правила описания более чем одной переменной и инициализации переменных остаются теми же, что и для других основных типов. Например:

```
char dog, cat;
```
Символьные константы. Символы в языке Си заключаются в апострофы. Например:
```
char dog;
dog='b';
```
Если апострофы опущены, то компилятор считает, что используется неописанная переменная b. В стандартном языке Си значением переменной или константы типа char могут быть только одиночные символы. Примеры символьных констант:
```
'A', 'a', '7', '$'.
```
Специальные (управляющие) символьные константы.

Новая строка (перевод строки)	'\n'
Горизонтальная табуляция	'\t'
Вертикальная табуляция	'\v'
Возврат на шаг	'\b'
Возврат каретки	'\r'
Перевод формата	'\f'
Обратная косая	'\\'
Апостроф	'\''
Кавычки	'\"'
Нулевой символ (пусто)	'\0'

Строковые константы.

Строковая константа представляется последовательностью символов кода ASCII, заключенной в кавычки. Например:
```
"Это строковая константа"
```
В конце каждой строки компилятор помещает нулевой символ '\0', отмечающий конец данной строки. Каждая строковая константа, даже если она идентична другой строковой константе, сохраняется в отдельном месте памяти. Если необходимо ввести в строку символ кавычек ("), то перед ними надо поставить символ обратной косой (\). В строку могут быть введены любые специальные символьные константы, перед которыми стоит символ \. Символ \ и следующий за ним символ новой строки игнорируется.

Данных типа float и double. Числа с плавающей точкой в языке С++ описываются типом float. Числа с плавающей точкой аналогичны числам в обычной алгебраической записи, используемой при работе с очень большими или малыми числами.

Обычно для размещения в памяти числа с плавающей точкой отводится 32 бита - 8 бит для представления порядка и знака и 24 бита - для мантиссы, т.е. коэффициента при степени десяти. Для представления данных типа *double* (вычисление с двойной точностью) используется удвоенное число битов. Другой способ определения данных типа *double* заключается в использовании ключевых *слов long float*. Переменные с плавающей точкой описываются и инициализируются таким же образом, что и переменные целого типа. Например:
```
float dog, cat, bigword=5.77e+34;
```
Константы с плавающей точкой. В языке Си имеется несколько возможностей записи констант с плавающей точкой. Наиболее общая форма записи константы - это последовательность десятичных цифр со знаком, включающим в себя десятичную точку, затем символ e или E и показатель степени по основанию 10 со знаком. Знак (+) можно не писать. Ниже приведено несколько правильно записанных *констант* с плавающей точкой:
```
   1.1e+12
   3.14159
```
Символьные строки. Символьная строка - это последовательность символов, возможно пустая (""). Рассмотрим пример:
```
"Большой спрос на образование в области
 информационных дисциплин объясняется не только
 популярностью компьютеров в современном обществе,
 но и реальной пользой от их применения."
```
Кавычки не являются частью строки. Они вводятся только для того, чтобы отметить ее начало и конец. В языке Си нет специального типа, который можно было бы использовать для описания строк. Вместо этого строки представляются в виде "набора" элементов типа *char*. Это означает, что символы в строке можно представить расположенными в соседних ячейках памяти - по одному символу в ячейке. Символ \0 в языке Си используется для того, чтобы отмечать конец строки. Нуль-символ не выводится на печать и в таблице кода ASCII (American Standard Code for Information Interchange) имеет номер 0. Наличие нуль-символа означает, что количество ячеек массива символов должно быть на одну больше, чем число символов строки.

Массив можно представить как совокупность нескольких ячеек памяти, объединенных в одну строку. *Массив* - это упорядоченная последовательность элементов *данных* одного типа. *Массив* из 177 ячеек памяти, в каждую из которых, к примеру, помещается один символ типа *char*, можно задать с помощью оператора описания:

```
char String[177];
```

Квадратные скобки указывают, что переменная String - *массив* из 177 элементов, а *char* задает тип каждого элемента. Длину *строки* в символах (без завершающего символа) определяет функция *strlen( )*. Обращение к ней в нашем примере выглядит так:

```
strlen(String);
```

Результат - целое число.

ЗАДАНИЯ

1. Напишите программу, которая переводит секунды в дни, часы, минуты и секунды. Количество переводимых секунд запрашивать у пользователя.

```
Результат выводить в виде:
N секунд это:
дней:   d
часов:  h
минут:  m
секунд: s
```

Для ввода чисел используйте оператор scanf(“%format”,a), где
format: %d – для переменных типа int, %3.2f – для переменных типа double, float (первое число – количество символов целой части, второе число – количество десятичных разрядов), %c – для переменных типа char, %s – для строковых переменных.

2. Напишите программу, переводящую скорость из километров в час в метры в секундах.
Количество переводимых км/ч запрашивать у пользователя.
Результат выводить в виде: `N км/ч = M м/с`

Арифметические и логические операции

Оператор присваивания

Оператор присваивания является основой любого алгоритмического языка (см. лекцию 3). В Си он записывается с помощью символа равенства, например, строка

```
x = 100;
```

означает присвоение переменной x значения 100. Для сравнения двух значений используется двойное равенство ==, например, строка

```
bool f = (2 + 2 == 5);
```

присваивает логической переменной f значение false (поскольку 2+2 не равно пяти, логическое выражение в скобках ложно).

Непривычным для начинающих может быть то, что оператор присваивания "=" в Си - бинарная операция, такая же, как, например, сложение или умножение. Значением операции присваивания = является значение, которое присваивается переменной, стоящей в левой части. Это позволяет использовать знак присваивания внутри выражения, например, x = (y = sin(z)) + 1.0;

Здесь в скобках стоит выражение *y = sin(z),* в результате вычисления которого переменной y присваивается значение *sin z*. Значением этого выражения является значение, присвоенное переменной y, т.е. *sin z*. К этому значению затем прибавляется единица, т.е. в результате переменной x присваивается значение *sin z+1*.

Выражения, подобные приведенному в этом примере, иногда используются, когда необходимо запомнить значение подвыражения (в данном случае *sin (z)*) в некоторой переменной (в данном случае y), чтобы затем не вычислять его повторно. Еще один пример:

```
n = (k = 3) + 2;
```

В результате переменной k присваивается значение 3, а переменной n - значение 5. Конечно, в нормальных программах такие выражения не встречаются.

Арифметические операции

К четырем обычным арифметическим операциям сложения +, вычитания -, умножения * и деления / в Си добавлена операция нахождения остатка от деления первого целого числа на второе, которая обозначается символом процента %. Приоритет у операции вычисления остатка

% такой же, как и у деления или умножения. Отметим, что операция % перестановочна с операцией изменения знака (унарным минусом), например, в результате выполнения двух строк
```
x = -(5 % 3);
y = (-5) % 3;
```
обеим переменным x и y присваивается отрицательное значение -2.

Операции увеличения и уменьшения

В Си добавлены операции увеличения и уменьшения на единицу, которые, к примеру, очень удобно применять к счетчикам. Операция увеличения записывается с помощью двух знаков сложения ++, операция уменьшения - с помощью двух минусов --. Например, операция ++, примененная к целочисленной переменной i, увеличивает ее значение на единицу:
```
++i;   эквивалентно i = i+1
```
Операции увеличения и уменьшения на единицу можно применять только к дискретным типам - целочисленным переменным различного вида и указателям. Операцию нельзя применять к вещественным переменным! Например, следующий фрагмент программы является ошибочным:
```
double x;
. . .
++x;  // Ошибка! Операция ++ неприменима
      //         к вещ. переменной
```
Операции увеличения ++ и уменьшения -- на единицу имеют префиксную и суффиксную формы. В префиксной форме операция записывается перед переменной, как в приведенных выше примерах. В суффиксной форме операция записывается после переменной:
```
++x;    // Префиксная форма
x--;    // Суффиксная форма
```
Разница между префиксной и суффиксной формами проявляется только при вычислении сложных выражений. Если используется префиксная форма операции ++, то сначала переменная увеличивается, и только после этого ее новое значение используется в выражении. При использовании суффиксной формы значение переменной сначала используется в выражении и только затем увеличивается. Примеры:
```
int x = 5, y = 5, a, b;
a = (++x) + 2; // переменной a присваивается значение 8
b = (y++) + 2; // переменной b присваивается значение 7
```
С логической точки зрения, префиксная операция более естественна (при использовании суффиксной формы надо сперва вычислить сложное выражение и только затем вернуться к увеличению переменной, т.е. операция ++ выполняется не в момент ее использования, а как бы откладывается на потом).

Операции "увеличить на", "домножить на" и т.п.

В большинстве алгоритмов при выполнении операции сложения чаще всего переменная-результат операции совпадает с первым аргументом:
```
x = x + y;
```
Здесь складываются значения двух переменных x и y, результат помещается в первую переменную x. Таким образом, значение переменной x увеличивается на значение y. Подобные фрагменты встречаются в программах гораздо чаще, чем фрагменты вида
```
x = y + z;
где аргументы и результат различны.
```
Оператор += читается как "увеличить на". Строка
```
x += y;    // Увеличить значение x на y
```
эквивалентна в Си строке
```
x = x + y; // x присвоить значение x + y,
```
но короче и нагляднее.

Аналогично, для арифметических операций +, -, *, /, % можно использовать операции
```
+= увеличить на
-= уменьшить на
*= домножить на
/= поделить на
%= поделить с остатком на
```
к примеру, строка

```
x *= 2.0;
```
удваивает значение вещественной переменной x.
Операторы вида ?= можно использовать даже для операций ?, которые записываются двумя символами. Например, операции логического умножения и сложения, они записываются в Си как && (двойной амперсенд) и || (двойная вертикальная черта). Соответственно, логические операторы "домножить на" и "увеличить на" записываются в виде &&= и ||=, например,
```
bool x, y;
x &&= y;    // эквивалентно x = x && y;
x ||= y;    // эквивалентно x = x || y;
```

Логические операции

В Си используются следующие обозначения для логических операций:
```
|| логическое "или" (логическое сложение)
&& логическое "и" (логическое умножение)
! логическое "не" (логическое отрицание)
```
Примеры логических выражений:
```
bool a, b, c, d;
int x, y;

a = b || c;             // логическое "или"
d = b && c;             // логическое "и"
a = !b;                 // логическое "не"
a = (x == y);           // сравнение в правой части
c = (x > 0 && y != 1); // c истинно, когда
                          // оба сравнения истинны
```
Самый высокий приоритет у операции логического отрицания, затем следует логическое умножение, самый низкий приоритет у логического сложения.

Чрезвычайно важной особенностью операций логического сложения и умножения является так называемое "сокращенное вычисление" результата. А именно, при вычислении результата операции логического сложения или умножения всегда сначала вычисляется значение первого аргумента. Если оно истинно в случае логического сложения или ложно в случае логического умножения, то второй аргумент операции не вычисляется вовсе. Результат операции полагается истинным в случае логического сложения или ложным в случае логического умножения.

Операции сравнения

Операция сравнения сравнивает два выражения. В результате вырабатывается логическое значение - true или false (истина или ложь) в зависимости от значений выражений. Примеры:
```
int x, y;
if (x == y) {do smth}; // true, если x равно y, иначе false
if (x == x) {do smth}; // всегда true
if (2 < 1) {do smth};;  // всегда false
```
Операции сравнения в Си обозначаются следующим образом:
```
== равно,  != не равно,
> больше,  >= больше или равно,
< меньше,  <= меньше или равно.
```

Управляющие конструкции

Управляющие конструкции позволяют организовывать циклы и ветвления в программах. В Си всего несколько конструкций.

Фигурные скобки

Фигурные скобки позволяют объединить несколько элементарных операторов в один составной оператор, или блок. Во всех синтаксических конструкциях составной оператор можно использовать вместо простого. В Си в начало блока можно помещать описания локальных переменных. Локальные переменные, описанные внутри блока, создаются при входе в блок и уничтожаются при выходе из него.

Приведем фрагмент программы, обменивающий значения двух вещественных переменных:
```
double x, y;
```

```
. . .
{
    double tmp = x;
    x = y;
    y = tmp;
}
```
Здесь, чтобы обменять значения двух переменных x и y, мы сначала запоминаем значение x во вспомогательной переменной tmp. Затем в x записывается значение y, а в y - сохраненное в tmp предыдущее значение x. Поскольку переменная tmp нужна только внутри этого фрагмента, мы заключили его в блок и описали переменную tmp внутри этого блока. По выходу из блока память, занятая переменной tmp, будет освобождена.

Оператор if

Оператор *if* ("если") позволяет организовать ветвление в программе. Он имеет две формы: оператор "если" и оператор "если...иначе". Оператор "если" имеет вид
```
if (условие)
    действие;
оператор "если...иначе" имеет вид
if (условие)
    действие1;
else
    действие2;
```
В качестве условия можно использовать любое выражение логического или целого типа. Напомним, что при использовании целочисленного выражения значению "истина" соответствует любое ненулевое значение. При выполнении оператора "если" сначала вычисляется условное выражение после if. Если оно истинно, то выполняется действие, если ложно, то ничего не происходит. Например, в следующем фрагменте в переменную m записывается максимальное из значений переменных x и y:
```
double x, y, m;
. . .
m = x;
if (y > x)
    m = y;
```
При выполнении оператора "если...иначе" в случае, когда условие истинно, выполняется действие, записанное после if; в противном случае выполняется действие после else. Например, предыдущий фрагмент переписывается следующим образом:
```
double x, y, m;
. . .
if (x > y)
    m = x;
else
    m = y;
```
Когда надо выполнить несколько действий в зависимости от истинности условия, следует использовать фигурные скобки, объединяя несколько операторов в блок, например,
```
double x, y, d;
. . .
if (d > 1.0) {
    x /= d;
    y /= d;
}
```
Здесь переменные x и y делятся на d только в том случае, когда значение d больше единицы.
Фигурные скобки можно использовать даже, когда после if или else стоит только один оператор. Они улучшают структуру текста программы и облегчают ее возможную модификацию. Пример:
```
double x, y;
. . .
if (x != 0.0) {
    y = 1.0;
}
```

Если нужно будет добавить еще одно действие, выполняемое при условии "x отлично от нуля", то мы просто добавим строку внутри фигурных скобок.

Выбор из нескольких возможностей: if...else if...

Несколько условных операторов типа "если...иначе" можно записывать последовательно (т.е. действие после else может снова представлять собой условный оператор). В результате реализуется выбор из нескольких возможностей. Конструкция выбора используется в программировании очень часто. Пример: дана вещественная переменная x, требуется записать в вещественную переменную y значение функции sign(x):

```
sign(x) = -1, при x < 0
sign(x) = 1,  при x > 0
sign(x) = 0,  при x = 0
```

Это делается с использованием конструкции выбора:

```
double x, s;
. . .
if (x < 0.0) {
    s = (-1.0);
}
else if (x > 0.0) {
    s = 1.0;
}
else {
    s = 0.0;
}
```

При выполнении этого фрагмента сперва проверяется условие x < 0.0. Если оно истинно, то выполняется оператор s = (-1.0); иначе проверяется второе условие x > 0.0. В случае его истинности выполняется оператор s = 1.0, иначе выполняется оператор s = 0.0. Фигурные скобки здесь добавлены для улучшения структурности текста программы.

В любом случае, в результате выполнения конструкции выбора исполняется лишь один из операторов (возможно, составных). Условия проверяются последовательно сверху вниз. Как только находится истинное условие, то производится соответствующее действие и выбор заканчивается.

Цикл while

Конструкция цикла "пока" соответствует циклу while в Си:

```
while (условие)
    действие;
```

Цикл while называют *циклом с предусловием*, поскольку условие проверяется перед выполнением тела цикла.

Цикл while выполняется следующим образом: сначала проверяется условие. Если оно истинно, то выполняется действие. Затем снова проверяется условие; если оно истинно, то снова повторяется действие, и так до бесконечности. Цикл завершается, когда условие становится ложным. Пример:

```
int n, p;
. . .
p = 1;
while (2*p <= n)
    p *= 2;
```

В результате выполнения этого фрагмента в переменной p будет вычислена максимальная степень двойки, не превосходящая целого положительного числа n.

Если условие ложно с самого начала, то действие не выполняется ни разу. Это очень облегчает программирование и делает программу более надежной, поскольку исключительные ситуации автоматически правильно обрабатываются. Так, приведенный выше фрагмент работает корректно при n = 1 (цикл не выполняется ни разу).

При ошибке программирования цикл может никогда не кончиться. Чтобы избежать этого, следует составлять программу таким образом, чтобы некоторая ограниченная величина, от которой прямо или косвенно зависит условие в заголовке цикла, монотонно убывала или возрастала после каждого выполнения тела цикла. Это обеспечивает завершение цикла. В приведенном выше фрагменте такой величиной является значение p, которое возрастает вдвое после каждого выполнения тела цикла.

Рассмотрим построение цикла "пока" на примере программы вычисления квадратного корня методом деления отрезка пополам.

Пример: вычисление квадратного корня методом деления отрезка пополам

Пусть надо найти квадратный корень из неотрицательного вещественного числа a с заданной точностью ε. Задача сводится к нахождению корня функции $y = x^2-a$ на отрезке [0,b], где b = max(1,a). На этом отрезке функция имеет ровно один корень, поскольку она монотонно возрастает и на концах отрезка принимает значения разных знаков (или нулевое значение при a = 0 или a = 1).

Идея алгоритма состоит в том, что отрезок делится пополам и выбирается та половина, на которой функция принимает значения разных знаков. Эта операция повторяется до тех пор, пока длина отрезка не станет меньше, чем ε. Концы текущего отрезка содержатся в переменных x0, x1. В данном случае функция монотонно возрастает при $x \geq 0$. Инвариантом цикла является утверждение о том, что функция принимает отрицательное или нулевое значение в точке x0 и положительное или нулевое значение в точке x1. Цикл рано или поздно завершается, поскольку после каждого выполнения тела цикла длина отрезка [x0,x1] уменьшается в два раза.

Приведем полный текст программы:

```
#include <stdio.h> // Описания стандартного ввода-вывода

int main() {
    double a;   // Число, из которого извлекается корень
    double x, x0, x1;      // [x0, x1] - текущий отрезок
    double y;              // Значение ф-ции в точке x
    double eps = 0.000001; // Точность вычисления корня

    printf("Введите число a:\n");
    scanf("%lf", &a);

    if (a < 0.0) {
        printf("Число должно быть неотрицательным.\n");
        return 1; // Возвращаем код
    }             //     некорректного завершения

    // Задаем концы отрезка
    x0 = 0.0;
    x1 = a;
    if (a < 1.0) {
        x1 = 1.0;
    }

    // Утверждение: x0 * x0 - a <= 0,
    //              x1 * x1 - a >= 0

    while (x1 - x0 > eps) {
        // Инвариант: x0 * x0 - a <= 0,
        //            x1 * x1 - a >= 0
        x = (x0 + x1) / 2.0; // середина отрезка [x0,x1]
        y = x * x - a;       // значение ф-ции в точке x

        if (y >= 0.0) {
            x1 = x; // выбираем левую половину отрезка
        }
        else {
            x0 = x; // выбираем правую половину отрезка
        }
    }
```

```
    // Утверждение: x0 * x0 - a <= 0,
    //              x1 * x1 - a >= 0,
    //              x1 - x0 <= eps
    x = (x0 + x1) / 2.0; // Корень := середина отрезка

    // Печатаем ответ
    printf("Квадратный корень = %lf\n", x);

    return 0; // Возвращаем код успешного завершения
}
```

Выход из цикла break, переход на конец цикла continue

Если необходимо прервать выполнение цикла, следует использовать оператор

```
break;
```

Оператор `break` применяется внутри тела цикла, заключенного в фигурные скобки. Пример: требуется найти корень целочисленной функции `f(x)`, определенной для целочисленных аргументов.

```
int f(int x);   // Описание прототипа функции
. . .
int x;
. . .
// Ищем корень функции f(x)
x = 0;
while (1) {
    if (f(x) == 0) {
        break;  // Нашли корень
    }
    // Переходим к следующему целому значению x
    //    в порядке 0, -1, 1, -2, 2, -3, 3, ...
    if (x >= 0) {
        x = (-x - 1);
    }
    else {
        x = (-x);
    }
}
// Утверждение: f(x) == 0
```

Здесь используется бесконечный цикл "while (1)". Выход из цикла осуществляется с помощью оператора "break".

Иногда требуется пропустить выполнение тела цикла при каких-либо значениях изменяющихся в цикле переменных, переходя к следующему набору значений и очередной итерации. Для этого используется оператор

```
continue;
```

Оператор continue, так же, как и break, используется лишь в том случае, когда тело цикла состоит более чем из одного оператора и заключено в фигурные скобки. Его следует понимать как переход на фигурную скобку, закрывающую тело цикла.

Цикл for

Популярный в других языках программирования параметрический цикл в языке Си реализуется с помощью цикла for. Он выглядит следующим образом:

```
for (инициализация; условие продолжения; итератор)
    тело цикла;
```

Инициализация выполняется один раз перед первой проверкой условия продолжения и первым выполнением тела цикла. Условие продолжения проверяется перед каждым выполнением тела цикла. Если условие истинно, то выполняется тело цикла, иначе цикл завершается. Итератор выполняется после каждого выполнения тела цикла (перед следующей проверкой условия продолжения).

Поскольку условие продолжения проверяется перед выполнением тела цикла, цикл for является, подобно циклу while, циклом с предусловием. Если условие продолжения не выполняется изначально, то тело цикла не выполняется ни разу.

Рассмотрим пример суммирования массива с использованием цикла for:

```
double a[100]; // Массив a содержит не более 100 эл-тов
int n;         // Реальная длина массива a (n <= 100)
double sum;    // Переменная для суммы эл-тов массива
int i;         // Переменная цикла
. . .
sum = 0.0;
for (i = 0; i < n; ++i) {
    sum += a[i]; // Увеличиваем сумму на a[i]
}
```

Здесь целочисленная переменная i используется в качестве переменной цикла. В операторе инициализации переменной i присваивается значение 0. Условием продолжения цикла является условие i<n. Итератор ++i увеличивает переменную i на единицу. Таким образом, переменная i последовательно принимает значения 0, 1, 2,..., n-1. Для каждого значения i выполняется тело цикла.

ЗАДАНИЯ

1. Напишите функцию, реверсирующую строку (переставляющую
символы в строке в обратном порядке).

2. Вычислить y=x1+x1x2+x1x2x3+:+ x1x2 *** xm,
где m - либо номер первого отрицательного элемента массива x,
либо общее количество элементов n в массиве x.

3. Напишите программу, печатающую таблицу умножения для чисел от 0 до 9 в десятичной системе счисления.

Представление программы в виде функций

Прототипы функций

Функция - это самостоятельная единица программы, созданная для решения конкретной задачи. Функция в языке С играет ту же роль, что и подпрограммы или процедуры в других языках. Функциями удобно пользоваться, например, если необходимо обработать один и тот же код программы. Как и переменные, функции надо объявлять (declare). Функцию необходимо объявить до её использования.

Каждая функция языка С имеет имя и список аргументов (формальных параметров). Функции могут возвращать значение. Это значение может быть использовано далее в программе. Так как функция может вернуть какое-нибудь значение, то обязательно нужно указать тип данных возвращаемого значения. Если тип не указан, то по умолчанию предполагается, что функция возвращает целое значение (типа int). После имени функции принято ставить круглые скобки (это касается вызова функции её объявления и описания). В этих скобках перечисляются параметры функции, если они есть. Если у функции нет параметров, то при объявлении и при описании функции вместо <список параметров> надо поставить void - пусто.

Объявление (прототип) функции имеет вид:

```
тип <имя функции>(список параметров);
```

Обратите внимание на то, что при описании функции после заголовка функции тип <имя функции>(список параметров) точка с запятой не ставится, а при объявлении функции точка с запятой ставится. Прототип функции сообщает информацию об имени функции, типе возвращаемого значения, количестве и типах ее аргументов. Пример:

```
int gcd(int x, int y);
```

Описан прототип функции gcd, возвращающей целое значение, с двумя целыми аргументами. Имена аргументов x и y здесь являются лишь комментариями, не несущими никакой информации для компилятора. Их можно опускать, например, описание

```
int gcd(int, int);
```

является вполне допустимым.

Вызов функции делается следующим образом:

```
<имя функции>(параметры);
```

или

```
<переменная>=<имя функции>(параметры);
```

```
int n=0;
n = gcd(10,20);
```

При вызове функции так же ставиться точка с запятой.

Формальный аргумент - переменная в вызываемой программе, а *фактический аргумент* - конкретное значение, присвоенное этой переменной вызывающей программой. *Фактический аргумент* может быть константой, переменной или более сложным выражением. Независимо от типа *фактического аргумента* он вначале вычисляется, а затем его величина передается *функции*. *Фактический аргумент* - это конкретное значение, которое присваивается переменной, называемой *формальным аргументом*. Если для связи с некоторой *функцией* требуется более одного аргумента, то наряду с именем *функции* можно задать список аргументов, разделенных запятыми. Например:

```
print_num(i,j){
int i,j;
   printf("значение i=%d. Значение j=%d.", i,j);
}
```

Обращение в программе к данной *функции* будет таковым:

```
print_num(6,19);
```

В языке Си функциям передаются *значения* фактических параметров. При вызове функции значения параметров копируются в аппаратный стек. Следует четко понимать, что изменение формальных параметров в теле функции не приводит к изменению переменных вызывающей программы, передаваемых функции при ее вызове, - ведь функция работает не с самими этими переменными, а с копиями их значений. Если необходимо, чтобы функция могла изменить значения переменных вызывающей программы, надо передавать ей указатели на эти переменные. Тогда функция может записать любую информацию по переданным адресам. В Си таким образом реализуются выходные и входно-выходные параметры функций.

Описания прототипов функций обычно выносятся в заголовочные файлы, см. раздел 3.1. Для коротких программ, которые помещаются в одном файле, описания прототипов располагают в начале программы. Рассмотрим пример такой короткой программы.

Пример: вычисление наибольшего общего делителя

Программа вводит с клавиатуры терминала два целых числа, затем вычисляет и печатает их наибольший общий делитель. Непосредственно вычисление наибольшего общего делителя реализовано в виде отдельной функции

```
int gcd(int x, int y);
```

(gcd - от слов greatest common divizor). Основная функция main лишь вводит исходные данные, вызывает функцию gcd и печатает ответ. Описание прототипа функции gcd располагается в начале текста программы, затем следует функция main и в конце - реализация функции gcd. Приведем полный текст программы:

```
#include <stdio.h> // Описания стандартного ввода-вывода

int gcd(int x, int y); // Описание прототипа функции

int main() {
    int x, y, d;
    printf("Введите два числа:\n");
    scanf("%d%d", &x, &y);
    d = gcd(x, y);
    printf("НОД = %d\n", d);
    return 0;
}

int gcd(int x, int y) { // Реализация функции gcd
    while (y != 0) {
        // Инвариант: НОД(x, y) не меняется
        int r = x % y;  // Заменяем пару (x, y) на
        x = y;          // пару (y, r), где r --
        y = r;          // остаток от деления x на y
    }
    // Утверждение: y == 0
    return x;   // НОД(x, 0) = x
}
```

ЗАДАНИЯ

1. Дана строка символов S. Напечатайте все входящие в эту строку заглавные латинские буквы 'A', ... , 'Z' по одному разу в алфавитном порядке с указанием числа вхождений каждой буквы в строку. Проверку на признак заглавной буквы и подсчет количества вхождений в строку оформить в виде функции.
2. Напишите программу, содержащую функцию удаления из вектора элемента перед заданным. Заданный элемент определяется по его порядковому номеру.
3. Пусть в векторе вещественных чисел содержатся результаты измерения некоторой величины x в нескольких опытах. Вычислите среднее значение величины x по формуле $M[X]=\sum_{i=0}^{n-1} x_i$ /n, где n – количество измерений, и дисперсию по формуле: $D[X]=\sum_{i=0}^{n-1}(M[X]-x_i)^2$ /n. Вычисление среднего арифметического M[X] и дисперсии D[X] организовать в виде отдельных функций.

Работа с файлами

Стандартная библиотека Си содержит набор функций для работы с файлами. Эти функции описаны в стандарте ANSI. Отметим, что файловый ввод-вывод не является частью языка Си, и ANSI-функции - не единственное средство ввода-вывода. Так, в операционной системе Unix более популярен другой набор функций ввода-вывода, который можно использовать не только для работы с файлами, но и для обмена по сети.

Открытие файла: функция fopen

Для доступа к файлу применяется тип данных FILE. Это структурный тип, имя которого задано с помощью оператора typedef в стандартном заголовочном файле "stdio.h". Программисту не нужно знать, как устроена структура типа файл: ее устройство может быть системно зависимым, поэтому в целях переносимости программ обращаться явно к полям струтуры FILE запрещено. Тип данных "указатель на структуру FILE используется в программах как черный ящик: функция открытия файла возвращает этот указатель в случае успеха, и в дальнейшем все файловые функции применяют его для доступа к файлу.

Прототип функции открытия файла выглядит следующим образом:

```
FILE *fopen(const char *path, const char *mode);
```

Здесь path - путь к файлу (например, имя файла или абсолютный путь к файлу), mode - режим открытия файла. Строка mode может содержать несколько букв. Буква "r" (от слова read) означает, что файл открывается для чтения (файл должен существовать). Буква "w" (от слова write) означает запись в файл, при этом старое содержимое файла теряется, а в случае отсутствия файла он создается. Буква "a" (от слова append) означает запись в конец существующего файла или создание нового файла, если файл не существует.

В некоторых операционных системах имеются различия в работе с текстовыми и бинарными файлами (к таким системам относятся MS DOS и MS Windows; в системе Unix различий между текстовыми и бинарными файлами нет). В таких системах при открытии бинарного файла к строке mode следует добавлять букву "b" (от слова binary), а при открытии текстового файла -- букву "t" (от слова text). Кроме того, при открытии можно разрешить выполнять как операции чтения, так и записи; для этого используется символ + (плюс). Порядок букв в строке mode следующий: сначала идет одна из букв "r", "w", "a", затем в произвольном порядке могут идти символы "b", "t", "+". Буквы "b" и "t" можно использовать, даже если в операционной системе нет различий между бинарными и текстовыми файлами, в этом случае они просто игнорируются.

Значения символов в строке mode сведены в следующую таблицу:

r	Открыть существующий файл на чтение
w	Открыть файл на запись. Старое содержимое файла теряется, в случае отсутствия файла он создаётся.
a	Открыть файл на запись. Если файл существует, то запись производится в его конец.
t	Открыть текстовый файл.
b	Открыть бинарный файл.
+	Разрешить и чтение, и запись.

Несколько примеров открытия файлов:

```
FILE *f, *g, *h;
. . .
// 1. Открыть текстовый файл "abcd.txt" для чтения
f = fopen("abcd.txt", "rt");

// 2. Открыть бинарный файл "c:\Windows\Temp\tmp.dat"
// для чтения и записи
g = fopen("c:/Windows/Temp/tmp.dat", "wb+");
```

```
// 3. Открыть текстовый файл "c:\Windows\Temp\abcd.log"
// для дописывания в конец файла
h = fopen("c:\\Windows\\Temp\\abcd.log", "at");
```

Обратите внимание, что во втором случае мы используем обычную косую черту / для разделения директорий, хотя в системах MS DOS и MS Windows для этого принято использовать обратную косую черту \. Дело в том, что в операционной системе Unix и в языке Си, который является для нее родным, символ \ используется в качестве экранирующего символа, т.е. для защиты следующего за ним символа от интерпретации как специального. Поэтому во всех строковых константах Си обратную косую черту надо повторять дважды, как это и сделано в третьем примере. Впрочем, стандартная библиотека Си позволяет в именах файлов использовать нормальную косую черту вместо обратной; эта возможность была использована во втором примере.

В случае удачи функция fopen открытия файла возвращает ненулевой указатель на структуру типа FILE, описывающую параметры открытого файла. Этот указатель надо затем использовать во всех файловых операциях. В случае неудачи (например, при попытке открыть на чтение несуществующий файл) возвращается нулевой указатель. При этом глобальная системная переменная errno, описанная в стандартном заголовочном файле "errno.h, содержит численный код ошибки. В случае неудачи при открытии файла этот код можно распечатать, чтобы получить дополнительную информацию:

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

FILE *f = fopen("filnam.txt", "rt");
if (f == NULL) {
    printf(
        "Ошибка открытия файла с кодом %d\n",
        errno
    );
    . . .
}
```

В приведенном выше примере при открытии файла функция fopen в случае ошибки возвращает нулевой указатель на структуру FILE. Чтобы проверить, произошла ли ошибка, следует сравнить возвращенное значение с нулевым указателем. Для наглядности стандартный заголовочный файл "stdio.h" определяет символическую константу NULL как нулевой указатель на тип void:

```
#define NULL ((void *) 0)
```

Сделано это вроде бы с благой целью: чтобы отличить число ноль от нулевого указателя. При этом язык Си, в котором контроль ошибок осуществляется недостаточно строго, позволяет сравнивать указатель общего типа void * с любым другим указателем. Между тем,в Си вместо константы NULL всегда можно использовать просто 0, и вряд ли от этого программа становится менее понятной.

Диагностика ошибок: функция perror

Использовать переменную errno для печати кода ошибки не очень удобно, поскольку необходимо иметь под рукой таблицу возможных кодов ошибок и их значений. В стандартной библиотеке Си существует более удобная функция perror, которая печатает системное сообщение о последней ошибке вместо ее кода. Печать производится на английском языке, но есть возможность добавить к системному сообщению любой текст, который указывается в качестве единственного аргумента функции perror. Например, предыдущий фрагмент переписывается следующим образом:

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

FILE *f = fopen("filnam.txt", "rt");
if (f == 0) {
    perror("Не могу открыть файл на чтение");
    . . .
}
```

Функция perror печатает сначала пользовательское сообщение об ошибке, затем после двоеточия системное сообщение. Например, при выполнении приведенного фрагмента в случае ошибки из-за отсутствия файла будет напечатано

```
Не могу открыть файл на чтение: No such file or directory
```

Функции бинарного чтения и записи fread и fwrite

После того как файл открыт, можно читать информацию из файла или записывать информацию в файл. Рассмотрим сначала функции бинарного чтения и записи fread и fwrite. Они называются бинарными потому, что не выполняют никакого преобразования информации при вводе или выводе (с одним небольшим исключением при работе с текстовыми файлами, которое будет рассмотрено ниже): информация хранится в файле как последовательность байтов ровно в том виде, в котором она хранится в памяти компьютера.

Функция чтения fread имеет следующий прототип:

```
size_t fread(
    char *buffer,    // Массив для чтения данных
    size_t elemSize, // Размер одного элемента
    size_t numElems, // Число элементов для чтения
    FILE *f          // Указатель на структуру FILE
);
```

Здесь size_t определен как беззнаковый целый тип в системных заголовочных файлах. Функция пытается прочесть numElems элементов из файла, который задается указателем f на структуру FILE, размер каждого элемента равен elemSize. Функция возвращает реальное число прочитанных элементов, которое может быть меньше, чем numElems, в случае конца файла или ошибки чтения. Указатель f должен быть возвращен функцией fopen в результате успешного открытия файла. Пример использования функции fread:

```
FILE *f;
double buff[100];
size_t res;

f = fopen("tmp.dat", "rb"); // Открываем файл
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для чтения");
    exit(1);  // завершить работу с кодом 1
}

// Пытаемся прочесть 100 вещественных чисел из файла
res = fread(buff, sizeof(double), 100, f);
// res равно реальному количеству прочитанных чисел
```

В этом примере файл "tmp.dat" открывается на чтение как бинарный, из него читается 100 вещественных чисел размером 8 байт каждое. Функция fread возвращает реальное количество прочитанных чисел, которое меньше или равно, чем 100. Функция fread читает информацию в виде потока байтов и в неизменном виде помещает ее в память. Следует различать текстовое представление чисел и их бинарное представление. В приведенном выше примере числа в файле должны быть записаны в бинарном виде, а не в виде текста. Для текстового ввода чисел надо использовать функции ввода по формату, которые будут рассмотрены ниже.

Открытие файла как текстового с помощью функции fopen, например:

```
FILE *f = fopen("tmp.dat", "rt");
```

вовсе не означает, что числа при вводе с помощью функции fopen будут преобразовываться из текстовой формы в бинарную. Из этого следует только то, что в

операционных системах, в которых строки текстовых файлов разделяются парами символами "\r\n" (они имеют названия CR и LF - возврат каретки и продергивание бумаги, Carriage Return и Line Feed), при вводе такие пары символов заменяются на один символ "\n" (продергивание бумаги). Обратно, при выводе символ "\n" заменяется на пару "\r\n". Такими операционными системами являются MS DOS и MS Windows. В системе Unix строки разделяются одним символом "\n" (отсюда проистекает обозначение "\n", которое расшифровывается как new line). Таким образом, внутреннее представление текста всегда соответствует системе Unix, а внешнее - реально используемой операционной системе. Отметим также, что создатели операционной системы компьютеров Apple Macintosh выбрали, чтобы жизнь не казалась скучной, третий, отличный от двух предыдущих, вариант: текстовые строки разделяются одним символом "\r".

Такое представление текстовых файлов восходит к тем уже далеким временам, когда еще не было компьютерных мониторов и для просмотра текста использовались электрифицированные пишущие машинки или посимвольные принтеры. Текстовый файл фактически представлял собой программу печати на пишущей машинке и, таким образом, содержал команды возврата каретки и продергивания бумаги в конце каждой строки.

Функция бинарной записи в файл fwrite аналогична функции чтения fread. Она имеет следующий прототип:

```
size_t fwrite(
    char *buffer,    // Массив записываемых данных
    size_t elemSize, // Размер одного элемента
    size_t numElems, // Число записываемых элементов
    FILE *f          // Указатель на структуру FILE
);
```

Функция возвращает число реально записанных элементов, которое может быть меньше, чем numElems, если при записи произошла ошибка - например, не хватило свободного пространства на диске. Пример использования функции fwrite:

```
FILE *f;
double buff[100];
size_t num;
. . .

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для записи");
    exit(1);  // завершить работу программы с кодом 1
}

// Записываем 100 вещественных чисел в файл
res = fwrite(buff, sizeof(double), 100, f);
// В случае успеха res == 100
```

Закрытие файла: функция fclose

По окончании работы с файлом его надо обязательно закрыть. Система обычно запрещает полный доступ к файлу до тех пор, пока он не закрыт. (Например, в нормальном режиме система запрещает одновременную запись в файл для двух разных программ.) Кроме того, информация реально записывается полностью в файл лишь в момент его закрытия. До этого она может содержаться в оперативной памяти (в так называемой файловой кеш-памяти), что при выполнении многочисленных операций записи и чтения значительно ускоряет работу программы.

Для закрытия файла используется функция fclose с прототипом

```
int fclose(FILE *f);
```

В случае успеха функция fclose возвращает ноль, при ошибке -- отрицательное значение (точнее, константу конец файла EOF, определенную в системных заголовочных файлах как минус единица). При ошибке можно воспользоваться функцией perror, чтобы

напечатать причину ошибки. Отметим, что ошибка при закрытии файла - явление очень редкое (чего не скажешь в отношении открытия файла), так что анализировать значение, возвращаемое функцией fclose, в общем-то, не обязательно. Пример использования функции fclose:

```
FILE *f;

f = fopen("tmp.res", "wb"); // Открываем файл "tmp.res"
if (f == 0) { // При ошибке открытия файла
    // Напечатать сообщение об ошибке
    perror("Не могу открыть файл для записи");
    exit(1);  // завершить работу программы с кодом 1
}

. . .

// Закрываем файл
if (fclose(f) < 0) {
    // Напечатать сообщение об ошибке
    perror("Ошибка при закрытии файла");
}
```

Пример: подсчет числа символов и строк в текстовом файле. В качестве содержательного примера использования рассмотренных выше функций файлового ввода приведем программу, которая подсчитывает число символов и строк в текстовом файле. Программа сначала вводит имя файла с клавиатуры. Для этого используется функция scanf ввода по формату из входного потока, для ввода строки применяется формат "%s. Затем файл открывается на чтение как бинарный (это означает, что при чтении не будет происходить никакого преобразования разделителей строк). Используя в цикле функцию чтения fread, мы считываем содержимое файла порциями по 512 байтов, каждый раз увеличивая суммарное число прочитанных символов. После чтения очередной порции сканируется массив прочитанных символов и подсчитывается число символов "\n" продергивания бумаги, которые записаны в концах строк текстовых файлов как в системе Unix, так и в MS DOS или MS Windows. В конце закрывается файл и печатается результат.

```
//
// Файл "wc.cpp"
// Подсчет числа символов и строк в текстовом файле
//
#include <stdio.h>  // Описания функций ввода-вывода
#include <stdlib.h> // Описание функции exit

int main() {
    char fileName[256]; // Путь к файлу
    FILE *f;            // Структура, описывающая файл
    char buff[512];     // Массив для ввода символов
    size_t num;         // Число прочитанных символов
    int numChars = 0;   // Суммарное число символов := 0
    int numLines = 0;   // Суммарное число строк := 0
    int i;              // Переменная цикла

    printf("Введите имя файла: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); // Открываем файл на чтение
    if (f == 0) { // При ошибке открытия файла
        // Напечатать сообщение об ошибке
        perror("Не могу открыть файл для чтения");
        exit(1);  // закончить работу программы с кодом 1
                  // ошибочного завершения
    }
```

```
    while ((num = fread(buff, 1, 512, f)) > 0) { // Читаем
        // блок из 512 символов. num -- число реально
        // прочитанных символов. Цикл продолжается, пока
        // num > 0

        numChars += num; // Увеличиваем число символов

        // Подсчитываем число символов перевода строки
        for (i = 0; i < num; ++i) {
            if (buff[i] == '\n') {
                ++numLines; // Увеличиваем число строк
            }
        }
    }

    fclose(f);

    // Печатаем результат
    printf("Число символов в файле = %d\n", numChars);
    printf("Число строк в файле = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы: она применяется к собственному тексту, записанному в файле "wc.cpp. Введите имя файла: wc.cppЧисло символов в файле = 1635

```
Число строк в файле = 50
```

Форматный ввод-вывод: функции fscanf и fprintf

В отличие от функции бинарного ввода fread, которая вводит байты из файла без всякого преобразования непосредственно в память компьютера, функция форматного ввода fscanf предназначена для ввода информации с преобразованием ее из текстового представления в бинарное. Пусть информация записана в текстовом файле в привычном для человека виде (т.е. так, что ее можно прочитать или ввести в файл, используя текстовый редактор). Функция fscanf читает информацию из текстового файла и преобразует ее во внутреннее представление данных в памяти компьютера. Информация о количестве читаемых элементов, их типах и особенностях представления задается с помощью формата. В случае функции ввода формат - это строка, содержащая описания одного или нескольких вводимых элементов. Форматы, используемые функцией fscanf, аналогичны применяемым функцией scanf, они уже неоднократно рассматривались. Каждый элемент формата начинается с символа процента "%". Наиболее часто используемые при вводе форматы приведены в таблице:

%d	целое десятичное число типа int (d - от decimal)
%lf	вещ. число типа double (lf - от long float)
%c	один символ типа char
%s	ввод строки. Из входного потока выделяется слово, ограниченое пробелами или символами перевода строки '\n'. Слово помещается в массив символов. Конец слова отмечается нулевым байтом.

Прототип функции fscanf выглядит следующим образом:

int fscanf(FILE *f, const char *format, ...);

Многоточие здесь означает, что функция имеет переменное число аргументов, большее двух, и что количество и типы аргументов, начиная с третьего, произвольны. На самом деле, фактические агрументы, начиная с третьего, должны быть указателями на вводимые переменные. Несколько примеров использования функции fscanf:

```
int n, m; double a; char c; char str[256];
```

```
    FILE *f;
    . . .
    fscanf(f, "%d", &n);  // Ввод целого числа
    fscanf(f, "%lf", &a); // Ввод вещественного числа
    fscanf(f, "%c", &c);  // Ввод одного символа
    fscanf(f, "%s", str); // Ввод строки (выделяется очередное
                          // слово из входного потока)
    fscanf(f, "%d%d", &n, &m); // Ввод двух целых чисел
```

Функция fscanf возвращает число успешно введенных элементов. Таким образом, возвращаемое значение всегда меньше или равно количеству процентов внутри форматной строки.

Функция fprintf используется для форматного вывода в файл. Данные при выводе преобразуются в их текстовое представление в соответствии с форматной строкой. Ее отличие от форматной строки, используемой в функции ввода fscanf, заключается в том, что она может содержать не только форматы для преобразования данных, но и обычные символы, которые записываются без преобразования в файл. Форматы, как и в случае функции fscanf, начинаются с символа процента "%". Они аналогичны форматам, используемым функцией fscanf. Небольшое отличие заключается в том, что форматы функции fprintf позволяют также управлять представлением данных, например, указывать количество позиций, отводимых под запись числа, или количество цифр после десятичной точки при выводе вещественного числа.

Прототип функции fprintf выглядит следующим образом:

```
int fprinf(FILE *f, const char *format, ...);
```

Многоточие, как и в случае функции fscanf, означает, что функция имеет переменное число аргументов. Количество и типы аргументов, начиная с третьего, должны соответствовать форматной строке. В отличие от функции fscanf, фактические аргументы, начиная с третьего, представляют собой выводимые значения, а не указатели на переменные. Для примера рассмотрим небольшую программу, выводящую данные в файл "tmp.dat":

```
#include <stdio.h>  // Описания функций ввода вывода
#include <math.h>   // Описания математических функций
#include <string.h> // Описания функций работы со строками

int main() {
    int n = 4, m = 6; double x = 2.;
    char str[256] = "Print test";
    FILE *f = fopen("tmp.dat", "wt"); // Открыть файл
    if (f == 0) {                     // для записи
        perror("Не могу открыть файл для записи");
        return 1; // Завершить программу с кодом ошибки
    }
    fprintf(f, "n=%d, m=%d\n", m, n);
    fprintf(f, "x=%.4lf, sqrt(x)=%.4lf\n", x, sqrt(x));
    fprintf(
        f, "Строка \"%s\" содержит %d символов.\n",
        str, strlen(str)
    );
    fclose(f);  // Закрыть файл
    return 0;   // Успешное завершение программы
}
```

В результате выполнения этой программы в файл "tmp.dat" будет записан следующий текст:

```
n=6, m=4
x=2.0000, sqrt(x)=1.4142
```

```
Строка "Print test" содержит 10 символов.
```
В последнем примере форматная строка содержит внутри себя двойные апострофы. Это специальные символы, выполняющие роль ограничителей строки, поэтому внутри строки их надо экранировать (т.е. защищать от интерпретации как специальных символов) с помощью обратной косой черты \, которая, напомним, в системе Unix и в языке Си выполняет роль защитного символа. Отметим также, что мы воспользовались стандартной функцией sqrt, вычисляющей квадратный корень числа, и стандартной функцией strlen, вычисляющей длину строки.

Понятие потока ввода или вывода

В операционной системе Unix и в других системах, использующих идеи системы Unix (например, MS DOS и MS Windows), применяется понятие потока ввода или вывода. Поток представляет собой последовательность байтов. Различают потоки ввода и вывода. Программа может читать данные из потока ввода и выводить данные в поток вывода. Программы можно запускать в конвейере, когда поток вывода первой программы является потоком ввода второй программы и т.д. Для запуска двух программ в конвейере используется символ вертикальной черты | между именами программ в командной строке. Например, командная строка:

```
ab | cd | ef
```

означает, что поток вывода программы ab направляется на вход программе cd, а поток вывода программы cd - на вход программе ef. По умолчанию, потоком ввода для программы является клавиатура, поток вывода назначен на терминал (или, как говорят программисты, на консоль). Потоки можно переправлять в файл или из файла, используя символы больше > и меньше <, которые можно представлять как воронки. Например, командная строка

```
abcd > tmp.res
```

перенаправляет выходной поток программы abcd в файл "tmp.res", т.е. данные будут выводиться в файл вместо печати на экране терминала. Соответственно, командная строка

```
abcd < tmp.dat
```

заставляет программу abcd читать исходные данные из файла "tmp.dat" вместо ввода с клавиатуры.

В Си работа с потоком не отличается от работы с файлом. Доступ к потоку осуществляется с помощью переменной типа FILE *. В момент начала работы Си-программы открыты три потока:

- stdin -- стандартный входной поток. По умолчанию он назначен на клавиатуру;
- stdout -- стандартный выходной поток. По умолчанию он назначен на экран терминала;
- stderr -- выходной поток для печати информации об ошибках. Он также назначен по умолчанию на экран терминала.

Переменные stdin, stdout, stderr являются глобальными, они описаны в стандартном заголовочном файле "stdio.h. Операции файлового ввода-вывода могут использовать эти потоки, например, строка

```
fscanf(stdin, "%d", &n);
```

вводит значение целочисленной переменной n из входного потока. Строка

```
fprintf(stdout, "n = %d\n", n);
```

выводит значение переменой n в выходной поток. Строка

```
fprintf(stderr, "Ошибка при открытии файла\n");
```

выводит указанный текст в поток stderr, используемый обычно для печати сообщений об ошибках. Функция perror также выводит сообщения об ошибках в поток stderr.

По умолчанию, стандартный выходной поток и выходной поток для печати ошибок назначены на экран терминала. Однако операция перенаправления вывода в файл > действует только на стандартный выходной поток. Например, в результате выполнения командной строки

```
abcd > tmp.res
```

обычный вывод программы abcd будет записываться в файл "tmp.res", а сообщения об ошибках по-прежнему будут печататься на экране терминала. Для того чтобы

перенаправить в файл "tmp.log" стандартный поток печати ошибок, следует использовать командную строку

```
abcd 2> tmp.log
```

(между двойкой и символом > не должно быть пробелов). Двойка здесь означает номер перенаправляемого потока. Стандартный входной поток имеет номер 0, стандартный выходной поток - номер 1, стандартный поток печати ошибок - номер 2. Данная команда перенаправляет только поток stderr, поток stdout по-прежнему будет выводиться на терминал. Можно перенаправить потоки в разные файлы:

```
abcd 2> tmp.log > tmp.res
```

Таким образом, существование двух разных потоков вывода позволяет при необходимости направить нормальный вывод и вывод информации об ошибках в разные файлы.

Функции scanf и printf ввода и вывода в стандартные потоки

Поскольку ввод из стандартного входного потока, по умолчанию назначенного на клавиатуру, и вывод в стандартный выходной поток, по умолчанию назначенный на экран терминала, используются особенно часто, библиотека функций ввода-вывода Си предоставляет для работы с этими потоками функции scanf и printf. Они отличаются от функций fscanf и fprintf только тем, что у них отсутствует первый аргумент, означающий поток ввода или вывода. Строка

```
scanf(format, ...); // Ввод из станд. входного потока
эквивалентна строке
fscanf(stdin, format, ...); // Ввод из потока stdin
Аналогично, строка
printf(format, ...); // Вывод в станд. выходной поток
эквивалентна строке
fprintf(stdout, format, ...); // Вывод в поток stdout
```

Функции текстового преобразования sscanf и sprintf

Стандартная библиотека ввода-вывода Си предоставляет также две замечательные функции sscanf и sprintf ввода и вывода не в файл или поток, а в строку символов (т.е. массив байтов), расположенную в памяти компьютера. Мнемоника названий функций следующая: в названии функции fscanf первая буква f означает файл (file), т.е. ввод производится из файла; соответственно, в названии функции sscanf первая буква s означает строку (string), т.е. ввод производится из текстовой строки. (Последняя буква f в названиях этих функций означает форматный). Первым аргументом функций sscanf и sprintf является строка (т.е. массив символов, ограниченный нулевым байтом), из которой производится ввод или в которую производится вывод. Эта строка как бы стоит на месте файла в функциях fscanf и fprintf.

Фунции sscanf и sprintf удобны для преобразования данных из текстового представления во внутреннее и обратно. Например, в результате выполнения фрагмента

```
char txt[256] = "-135.76"; double x;
sscanf(txt, "%lf", &x);
```

текстовая запись вещественного числа, содержащаяся в строке txt, преобразуется во внутреннее представление вещественного числа, результат записывается в переменную x. Обратно, при выполнения фрагмента

```
char txt[256]; int x = 12345;
sprintf(txt, "%d", x);
```

значение целочисленной переменной x будет преобразовано в текстовую форму и записано в строку txt, в результате строка будет содержать текст "12345", ограниченный нулевым байтом.

Для преобразования данных из текстового представления во внутреннее в стандартной библиотеке Си имеются также функции atoi и atof с прототипами

```
int atoi(const char *txt);    // текст => int
double atof(const char *txt); // текст => double
```

Функция atoi преобразует текстовое представление целого числа типа int во внутреннее. Соответственно, функция atof преобразует текстовое представление вещественного числа типа double. Мнемоника имен следующая:

- atoi означает "character to integer";
- atof означает "character to float".

Прототипы функций atoi и atof описаны в стандартном заголовочном файле "stdlib.h", а не "stdio.h", поэтому при их использовании надо подключать этот файл:

```
#include <stdlib.h>
```

(вообще-то, это можно делать всегда, поскольку "stdlib.h" содержит описания многих полезных функций, например, функции завершения программы exit, генератора случайных чисел rand и др.).

Другие полезные функции ввода-вывода

Стандартная библиотека ввода-вывода Си содержит ряд других полезных функций ввода-вывода. Отметим некоторые из них.

Посимвольный ввод-вывод

int fgetc(FILE *f);	ввести символ из потока f
int fputc(int c, FILE *f);	вывести символ в поток f

Построковый ввод-вывод

char *fgets(char *line,int size, FILE *f);	ввести строку из потока f
char *fputs(char *line, FILE *f);	вывести строку в поток f

Позиционирование в файле

int fseek(FILE *f, long offset, int whence);	установить текущую позицию в файле f
long ftell(FILE *f);	получить текущую позицию в файле f
int feof(FILE *f);	проверить,достигнут ли конец файла f

Функция fgetc возвращает код введенного символа или константу EOF (определенную как минус единицу) в случае конца файла или ошибки чтения.

Функция fputc записывает один символ в файл. При ошибке fputc возвращает константу EOF (т.е. отрицательное значение), в случае удачи - код выведенного символа c (неотрицательное значение).

В качестве примера использования функции fgetc перепишем рассмотренную ранее программу wc, подсчитывающую число символов и строк в текстовом файле:

```
//
// Файл "wc1.cpp"
// Подсчет числа символов и строк в текстовом файле
// с использованием функции чтения символа fgetc
//
#include <stdio.h>  // Описания функций ввода-вывода

int main() {
    char fileName[256]; // Путь к файлу
    FILE *f;            // Структура, описывающая файл
    int c;              // Код введенного символа
    int numChars = 0;   // Суммарное число символов := 0
    int numLines = 0;   // Суммарное число строк := 0

    printf("Введите имя файла: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); // Открываем файл
    if (f == 0) { // При ошибке открытия файла
        // Напечатать сообщение об ошибке
        perror("Не могу открыть файл для чтения");
        return 1; // закончить работу программы с кодом 1
```

```
    }

    while ((c = fgetc(f)) != EOF) { // Читаем символ
        // Цикл продолжается, пока c != -1 (конец файла)

        ++numChars; // Увеличиваем число символов

        // Подсчитываем число символов перевода строки
        if (c == '\n') {
            ++numLines; // Увеличиваем число строк
        }
    }

    fclose(f);

    // Печатаем результат
    printf("Число символов в файле = %d\n", numChars);
    printf("Число строк в файле = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы wc1 в применении к собственному тексту, записанному в файле "wc1.cpp:

```
Введите имя файла: wc1.cpp
Число символов в файле = 1334
Число строк в файле = 44
```

Функция `fgets` с прототипом

```
char *fgets(char *line, int size, FILE *f);
```

выделяет из файла или входного потока f очередную строку и записывает ее в массив символов line. Второй аргумент size указывает размер массива для записи строки. Максимальная длина строки на единицу меньше, чем size, поскольку всегда в конец считанной строки добавляется нулевой байт. Функция сканирует входной поток до тех пор, пока не встретит симол перевода строки "\n" или пока число введенных символов не станет равным size - 1. Символ перевода строки "\n" также записывается в массив непосредственно перед терминирующим нулевым байтом. Функция возвращает указатель line в случае успеха или нулевой указатель при ошибке или конце файла.

Раньше в стандартную библиотеку Си входила также функция gets с прототипом

```
char *gets(char *line);
```

которая считывала очередную строку из стандартного входного потока и помещала ее в массив, адрес которого являлся ее единственным аргументом. Отличие от функции fgets в том, что не указывается размер массива line. В результате, подав на вход функции gets очень длинную строку, можно добиться переполнения массива и стереть или подменить участок памяти, используемый программой. Если программа имеет привилегии суперпользователя, то применение в ней функции gets открывает путь к взлому системы, который использовался хакерами. Поэтому в настоящее время функция gets считается опасной и применение ее не рекомендовано. Вместо gets следует использовать fgets с третьим аргументом stdin.

При выполнении файловых операций исполняющая система поддерживает указатель текущей позиции в файле. При чтении или записи n байтов указатель текущей позиции увеличивается на n; таким образом, чтение или запись происходят последовательно. Библиотека ввода-вывода Си предоставляет, однако, возможность нарушать эту последовательность путем позиционирования в произвольную точку файла. Для этого используется стандартная функция fseek с прототипом

```
int fseek(FILE *f, long offset, int whence);
```

Первый аргумент f функции определяет файл, для которого производится операция позиционирования. Второй аргумент offset задает смещение в байтах, оно может быть как положительным, так и отрицательным. Третий аргумент whence указывает, откуда

отсчитывать смещение. Он может принимать одно из трех значений, заданных как целые константы в стандартном заголовочном файле "stdio.h":

SEEK_CUR	смещение отсчитывается от текущей позиции
SEEK_SET	смещение отсчитывается от начала файла
SEEK_END	смещение отсчитывается от конца файла

Например:
```
fseek(f, 0, SEEK_SET);
```
устанавливает текущую позицию в начало файла. Фрагмент
```
fseek(f, -4, SEEK_END);
```
устанавливает текущую позицию в четырех байтах перед концом файла. Наконец, фрагмент
```
fseek(f, 12, SEEK_CUR);
```
продвигает текущую позицию на 12 байтов вперед.
Отметим, что смещение может быть положительным даже при использовании константы SEEK_END (т.е. при позиционировании относительно конца файла): в этом случае при следующей записи размер файла соответственно увеличивается.

Функция возвращает нулевое значение в случае успеха и отрицательное значение EOF (равное -1) при неудаче - например, если указанное смещение некорректно при заданной операции или если файл или поток не позволяет выполнять прямое позиционирование.

Узнать текущую позицию относительно начала файла можно с помощью функции ftell с прототипом
```
long ftell(FILE *f);
```
Функция ftell возвращает текущую позицию (неотрицательное значение) в случае успеха или отрицательное значение -1 при неудаче (например, если файл не разрешает прямое позиционирование).
Наконец, узнать, находится ли текущая позиция в конце файла, можно с помощью функции feof с прототипом
```
int feof(FILE *f);
```
Она возвращает ненулевое значение (т.е. истину), если конец файла достигнут, и нулевое значение (т.е. ложь) в противном случае. Например, в следующем фрагменте в цикле проверяется, достигнут ли конец файла, и, если нет, считывается очередной байт:
```
FILE *f;
. . .
while (!feof(f)) {     // цикл пока не конец файла
    int c = fgetc(f); // | прочесть очередной байт
    . . .             // | . . .
}                     // конец цикла
```

Работа с текстами

Стандартная библиотека Си предоставляет набор функций для работы с текстами. К сожалению, большая часть из них ориентирована на представление символов в виде одного байта (во время разработки языка Си кодировка Unicode, в которой на символ отводится два байта, еще не существовала). Функции можно разделить на две группы:

1. функции, определяющие тип символа, - является ли он буквой, цифрой, пробелом, знаком препинания и т.п. Это функции описаны в стандартном заголовочном файле "ctype.h". Увы, функции, касающиеся букв, работают только для латинского алфавита;
2. функции для работы с текстовыми строками. Строкой в Си считается последовательность байтов, ограниченная в конце нулевым байтом. Функции работы со строками описаны в стандартном заголовочном файле "string.h".

Определение типов символов

Библиотека Си предоставляет следующие функции для определения типа символов, описанные в стандартном заголовочном файле "ctype.h":

int isdigit(int c);	символ с - цифра;
int isalpha(int c);	с - латинская буква;
int isspace(int c);	с - пробел, перевод строки и т.п.;
int ispunkt(int c);	с - знак препинания;
int isupper(int c);	с - прописная латинская буква;
int islower(int c);	с - строчная латинская буква;
int toupper(int c);	если с -- лат. буква, то преобразовать с к прописной букве;
int tolower(int c);	если с -- лат. буква, то преобразовать с к строчной букве.

Функции, начинающиеся с префикса is, возвращают ненулевое значение (т.е. истину), если символ с кодом с принадлежит указанному классу, и нулевое значение (ложь) в противном случае. Функции toupper и tolower преобразуют латинские буквы к верхнему или нижнему регистру, на остальных символах они действуют тождественно.
В качестве примера использования функции isspace модифицируем программу "wc.cpp", подсчитывающую число строк и символов в текстовом файле. Добавим в нее подсчет слов. Будем считать словами любые связные группы символов, разделенные пробелами, табуляциями или разделителями строк.

```
//
// Файл "wc2.cpp"
// Подсчет числа символов, слов и строк в текстовом файле
//
#include <stdio.h>  // Описания функций ввода-вывода
#include <ctype.h>  // Описания типов символов

int main() {
    char fileName[256]; // Путь к файлу
    FILE *f;            // Структура, описывающая файл
    int c;              // Код введенного символа
    int numChars = 0;   // Суммарное число символов := 0
    int numLines = 0;   // Суммарное число строк := 0
    int numWords = 0;   // Суммарное число слов := 0
    bool readingWord = false; // Читаем слово := false

    printf("Введите имя файла: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); // Открываем файл
    if (f == 0) { // При ошибке открытия файла
        // Напечатать сообщение об ошибке
        perror("Не могу открыть файл для чтения");
        return 1; // закончить работу программы с кодом 1
    }

    while ((c = fgetc(f)) != EOF) { // Читаем символ
        // Цикл продолжается, пока c != -1 (конец файла)

        ++numChars; // Увеличиваем число символов

        // Подсчитываем число символов перевода строки
        if (c == '\n') {
            ++numLines; // Увеличиваем число строк
        }

        // Подсчитываем число слов
        if (!isspace(c)) {      // если с не пробел
            if (!readingWord) { //  если не читаем слово
                ++numWords;     //   увеличить число слов
                readingWord = true; // читаем слово:=true
```

```
            }                       //  конец если
        } else {                    // иначе
            readingWord = false;//  читаем слово:=false
        }                           // конец если
    }

    fclose(f);

    // Печатаем результат
    printf("Число символов в файле = %d\n", numChars);
    printf("Число слов = %d\n", numWords);
    printf("Число строк = %d\n", numLines);

    return 0; // Возвращаем код успешного завершения
}
```

Пример выполнения программы wc2 в применении к собственному тексту, записанному в файле "wc2.cpp":

```
Введите имя файла: wc2.cpp
Число символов в файле = 1998
Число слов = 260
Число строк = 57
```

Работа с текстовыми строками

Стандартная библиотека Си предоставляет средства вычисления длины строки, копирования, сравнения, соединения (конкатенации) строк, поиска вхождений одной строки в другую. Функции описаны в стандартном заголовочном файле "string.h". Прототипы наиболее часто используемых функций приведены ниже.

Определение длины строки:

```
size_t strlen(const char *s); //длина строки;
```

Копирование строк:

```
char *strcpy(char *dst, const char *src); /*копировать строку src в строку dst;*/
char *strncpy(char *dst, const char *src, size_t maxlen); /*копировать строку src в dst, не более maxlen символов;*/
char *strcat(char *dst, const char *src); /*копировать строку src в конец dst (конкатенация строк).*/
```

Работа с произвольными массивами байтов:

```
void *memmove(void *dst, const void *src, size_t len); /*копировать область памяти с адресом src размером len байтов в область памяти с адресом dst;*/
void *memset(void *dst, int value, size_t len); /*записать значение value в каждый из len байтов, начиная с адреса dst.*/
```

Сравнение строк:

```
int strcmp(const char *s1, const char *s2); /*лексикографическое сравнение строк s1 и s2. Результат нулевой, если строки равны, отрицательный, если первая строка меньше второй, и положительный, если первая строка больше второй;*/
int strncmp(const char *s1, const char *s2, size_t maxlen); /*сравнение строк s1 и s2, сравнивается не более maxlen символов; */
int memcmp(const void *m1, const void *m2, size_t len);/* сравнение областей памяти с адресами m1 и m2 размером len каждая. */
```

Поиск:

```
char *strchr(const char *s, int c); /*найти первое вхождение символа c в строку s. Функция возвращает указатель на найденный символ или ноль в случае неудачи;*/
char *strstr(const char *s1, const char *s2); /*найти первое вхождение строки s2 в строку s1. Функция возвращает указатель на найденную подстроку в s1, равную строке s2, или ноль в случае неудачи.*/
```

Функции библиотеки ввода-вывода

Функции для работы с файлами

fopen – открывает поток, связанный с файлом filename и типом доступа type.
FILE *fopen(char *filename, char *type);
fclose – закрывает поток stream.
int fclose(FILE *stream);
fcloseall – закрывает все открытые потоки.
int fcloseall(void);
remove – удаляет файл с именем filename.
int remove(char *filename);
rename – переименовывает файл oldname в файл newname.
int rename(char *oldname, char *newname);
ftell – возвращает положение указателя текущей позиции файла, связанного с потоком stream. Значение возвращается в виде смещения в байтах относительно начала файла. Значение, возвращаемое функцией ftell, в дальнейшем можно использовать при вызове функции fseek. Ftell возвращает положение указателя текущей позиции при успешном завершении.При ошибке возвращается значение.
long int ftell(FILE *stream);
fseek – устанавливает адресный указатель файла, соответствующий потоку stream, в новую позицию, которая расположена по смещению offset относительно места в файле, определенного параметром fromtwhere. Параметр fromtwhere может иметь одно из трех значений 0, 1 или 2, которые представлены тремя символическими константами, определенными в файле stdio.h, следующим образом:
SEEK_SET(0) – начало файла, SEEK_CUR(1) – позиция текущего указателя файла, SEEK_END(2) – конец файла(EOF); Функция fseek возвращает значение 0, если указатель файла успешно перенесен, и ненулевое значение в случае неудачного завершения.
int fseek(FILE *stream, long int offset, int fromwhere);
fgetpos – сохраняет позицию указателя файла, связанного с потоком stream, в месте, указываемом параметром pos. При успешном завершении fgetpos возвращает 0.
int fgetpos(FILE *stream, fpos_t *pos);
Здесь и далее fpos_t - предварительно объявленный тип typedef long fpos_t.
fsetpos - устанавливает указатель текущей позиции файла, связанного с потоком stream в новую позицию, которая определяется значением, получаемым предшествующим вызовом функции fgetpos. При успешном завершении fsetpos возвращает 0.
Int fsetpos(FILE *stream, const fpos_t *pos);
fgetc – получает символ из потока stream.
int fgetc(FILE *stream);
fgetchar – получает символ из потока stdin.
int fgetchar(void);
fgets – получает строку s длиной не более n символов из потока stream.
char *fgets(char *s, int n, FILE *stream);
fputc – выводит символ c в поток stream.
int fput(int c, FILE *stream);
fputchar- выводит символ c в поток stdout.
int fputchar(int c);
fputs – выводит строку символов string в поток stream.
int fputs(char *string, FILE *stream);
gets – получает строку символов s из потока stdin.
char *gets(char *s);
getc – выводит из потока stream символ этого потока.
int getc(FILE *stream);
getchar – выводит символ из потока stdin.
int getchar(void);
putc – выводит символ c в поток stream.
int putc(int c, FILE *stream);
putchar – выводит символ c в поток stdout.
int putchar(int c);
puts – выводит строку s в поток stdout.
int puts(const char *s);

putw – помещает в поток stream целое значение w.
int putw(int w, FILE *stream);
getw – вводит из потока stream целое число.
int getw(FILE *stream);
fread – считывает n элементов данных длиной size из потока stream по адресу ptr.
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
Здесь и далее size_t – предварительно объявленный в библиотеке тип typedef usigned size_t.
fwrite – записывает n элементов данных длиной size из ptr и поток stream.
Size_t fwrite(void *ptr, size_t size, size_t n, FILE *stream);
Функции форматированного ввода-вывода
printf – производит форматированный вывод в stdout.
int printf(const char *format [,argument,…]);
scanf – выполняет форматированный вывод из потока stdin.
int scanf(const char *format [,adress,…]);
fprintf – посылает форматированный вывод в поток stream.
int fprintf(FILE *stream, const char *format [,argument,…]);
fscanf – выполняет форматированный ввод из потока stream.
int fscanf(FILE *stream, const char *format [,adress,…]);
sprintf – производит форматированный вывод в сстроку buffer.
int sprintf(char *buffer, const char *format [,argument,…]);
sscanf – выполняет форматированный ввод из строки buffer.
Int sscanf(const char *buffer, const char *format [,adress,…]);

Функции ввода-вывода на экран(conio.h)

cprintf – осуществляет форматированный вывод на экран.
int cprintf(const char *format [,argument,…]);

Файлы и их назначение

Заголовочный файл	Назначение
assert.h	Диагностика программ
ctype.h	Преобразование и проверка символов
errno.h	Проверка ошибок
float.h	Работа с числами с плавающей точкой
limits.h	Определение размеров целочисленных типов
locale.h	Поддержка интернациональной среды
match.h	Математическая библиотека
setjmp.h	Возможность нелокальных переходов
signal.h	Обработка сигналов
stdarg.h	Поддержка функций с неопределенным числом параметров
stddef.h	Разное
stdio.h	Библиотека стандартного ввода-вывода
stdlib.h	Функции общего назначения
string.h	Функции работы со строками символов
time.h	Функции работы с датами и временем

getch – читает один символ с консоли без вывода его на экран.
int getch(void);
getche – считывает один символ с консоли и отображает его в текущем текстовом окне экрана.
int getche(void);
putch – выводит символ на экран.
int putch(int c);

Каждая библиотечная функция , определенная стандартом языка С, имеет прототип в соответствующем заголовочном файле. В соответствие со стандартом ANSI языка С должно быть 15 следующих заголовочных файлов.На самом деле каждый из компиляторов содержит , как правило больше заголовочных файлов, например –

библиотеку функций работы с графическим экраном graphics.h и библиотеку функций для работы с текстовым экраном conio.h.

ЗАДАНИЯ

1. Составьте программу, которая распечатывает последние n строк файла ввода.

2. Напишите программу, которая читает 2 файла и печатает их вперемежку: одна строка из первого файла, другая - из второго. Придумайте, как поступить, если файлы содержат разное число строк.

3. Дан текстовый файл f. Найти и распечатать самую длинную строку этого файла.

4.Пусть имеется файл, содержащий программу на Си. Напишите программу, удаляющую из текста программы на С комментарии.

ЗАДАНИЯ НА ДОМ

1. Пусть файл содержит записи о сотрудниках некоторого учреждения. Каждая запись включает порядковый номер записи, фамилию, имя сотрудника, его номер телефона и возраст:

```
1 Иванов Иван 1234567 25
2 Петров Петр 2345678 27
…
```

Напишите программу, которая выводит меню с доступными операциями:

```
1 - добавить новую запись (при добавлении новой записи проверять нет ли уже такой записи в файле)
2 - редактировать запись (вводится номер записи)
3 - искать запись по фамилии (если имеются однофамильцы, выводить все записи с одинаковыми фамилиями)
4 - удалить запись по порядковому номеру
5 - завершить работу программы
```

Каждую из операций оформить в виде отдельной функции.

2. Пусть файл содержит информацию о наличии книг в магазине. Каждая запись включает номер ISBN книги, автора, название, количество книг:

```
9666965348 Стругацкие Град обреченный 258
5845908914 Керниган, Ритчи Язык программирования С 129
…
```

Напишите программу, которая выводит меню с доступными операциями:

```
1 - добавить новую книгу (при добавлении проверять нет ли уже такой книги)
2 - редактировать запись (вводится номер ISBN)
3 - поиск книг одного автора (вывести все книги по введенной фамилии автора)
4 - изменить количество книг (по введенному номеру ISBN)
5 - завершить работу программы
```

Каждую из операций оформить в виде отдельной функции

Программа должна иметь следующую структуру:
Lab1.c – файл, содержащий функцию main
Lab1_func.c – файл, содержащий реализацию функций программы.
Lab1.h – заголовочный файл, содержащий глобальные переменные и заголовки функций.

Варианты:

1. задание 1 реализовать функции 1,3,5
2. задание 1 реализовать функции 2,4,5
3. задание 2 реализовать функции 1,3,5
4. задание 2 реализовать функции 2,4,5

Массивы

Массив в Си — это набор элементов одного типа, обратиться к которым можно по индексу. Элементы массивов в С расположены друг за другом в памяти компьютера. Двумерные массивы Си — это прямоугольная таблица чисел. Двумерный массив С состоит из рядов и столбцов. Первый индекс двумерного массива С — это число рядов, а второй индекс — это число столбцов. Описание массива в Си состоит из имени базового типа, названия массива и его размера, который указывается в квадратных скобках. Размер массива обязательно должен быть целочисленной константой или константным выражением. Примеры:

```
int a[10];
char c[256];
double d[1000];
```

В первой строке описан массив целых чисел из 10 элементов. Подчеркнем, что нумерация в Си всегда начинается с нуля, так что индексы элементов массива изменяются в пределах от 0 до 9. Во второй строке описан массив символов из 256 элементов (индексы в пределах 0...255), в третьей - массив вещественных чисел из 1000 элементов (индексы в пределах 0...999). Для доступа к элементу массива указывается имя массива и индекс элемента в квадратных скобках, например:

```
a[10], c[255], d[123]
```

Объявление массива сводится к указанию типа его элементов и количества элементов по каждому измерению:

```
#define Nmax 50
/*директива define указывает процессору на то, что встречающееся в коде обозначение Nmax следует заменить на значение 50 */
char a1[20],a2[2][80];
int b1[25],b2[Nmax];
```

По такому объявлению компилятор будет знать, сколько места в оперативной памяти понадобится для хранения такого массива. Для глобальных массивов место в памяти будет выделено в момент запуска программы, а для локальных – в момент вызова соответствующей функции.

Объявление массива можно совместить с его инициализацией, т.е. с присвоением начальных значений всем элементами массива или только нескольким первым элементам:

```
char a[7]="Привет";
char b[7]={'П','р','и','в','е','т',0x0};
char c[]="Привет";
float d[10]={1.,2.,3.,4.};
int q[2][3]={{1,2,3},
             {4,5,6}};
```

Обратите внимание на инициализацию символьных массивов a, b и c. В первом случае значения элементов массива совпадают с символами указанной строковой константы. Хотя значащих символов там 6, не следует забывать и о невидимом признаке конца строки – байте с нулевым значением. В случае инициализации массива b каждый его элемент задан символьной константой, не забыть и признак конца строки. Самый удобный способ использован при инициализации массива c – вместо того, чтобы указывать количество элементов, здесь заданы пустые скобки. Компилятор по заданному значению текстовой константы сам определит нужное количество байтов. Это позволяет избежать ненужных ошибок при подсчете количества символов в достаточно длинных строках. При инициализации массива d вместо десяти значений заданы только четыре. Это означает, что указанные величины будут присвоены только первым четырем элементам. Остальные элементы не инициализированы. В случае если d является глобальным массивом, значения этих элементов будут равны 0 (память, выделяемая глобальным данным, предварительно чистится). Если массив d локализован в какой-то функции, то значения его элементов, начиная с d[4], предсказать невозможно – там будет находиться "мусор", которая при повторных вызовах функции может оказаться разной.

Инициализация двумерных массивов выглядит более естественно, если значения элементов строк располагать друг под другом (так как это сделано в инициализации массива q).

Оператор sizeof возвращает размер всего массива в байтах, а не в элементах массива. В данном примере:

```
sizeof(a) = 10*sizeof(int) = 40,
sizeof(c) = 256*sizeof(char) = 256,
sizeof(d) = 1000*sizeof(double) = 8000
```

ЗАДАНИЯ

1. Написать программу поиска минимального элемента массива из n элементов.

2.Дана последовательность из 20 целых чисел. Определить количество инверсий в этой последовательности (то есть пар таких элементов, в которых большее число находится слева от меньшего: x[i] > x[j] при i<j).

3. Вычислить скалярное произведения двух векторов.

Сложность алгоритмов.

Алгоритм сортировки — это алгоритм для упорядочения элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма. Нам уже известно, что правильность - далеко не единственное качество, которым должна обладать хорошая программа. Одним из важнейших является эффективность, характеризующая прежде всего время выполнения программы T(n) для различных входных данных (параметра n).

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- Время — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны *худшее*, *среднее* и *лучшее* поведение алгоритма в терминах мощности входного множества *A*. Если на вход алгоритму подаётся множество *A*, то обозначим $n = |A|$. Для типичного алгоритма хорошее поведение — это O(n*log(n)) и плохое поведение — это $O(n^2)$. Идеальное поведение для упорядочения — O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в Ω(n*log(n)) сравнениях. Тем не менее, существует алгоритм сортировки Хана (Yijie Han) с вычислительной сложностью O(n*log(log(n))*log(log(log(n)))), использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за O-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике). Также существует понятие сортирующих сетей. Предполагая, что можно одновременно (например, при параллельном вычислении) проводить несколько сравнений, можно отсортировать *n* чисел за $O(\log^2(n))$ операций. При этом число *n* должно быть заранее известно;
- Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. Как правило, эти алгоритмы требуют O(log(n)) памяти. При оценке не учитывается место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.

Методы сортировки массивов

Сортировка числовых и нечисловых данных – одна из важнейших процедур обработки информации, т.к. она существенно ускоряет последующий поиск тех или иных объектов. О том, какое внимание уделяется различным алгоритмам сортировки, свидетельствует специальный том Д.Кнута "Искусство программирования для ЭВМ: Сортировка и поиск" объемом порядка 840 стр. Надо отметить, что оценка трудоемкости различных методов сортировки представляет собой довольно сложную математическую задачу. Те оценки, которые приведены ниже, заимствованы из литературных источников. Рассмотрим несколько разных алгоритмов сортировки – от самых простых и самых медленных до одного из наиболее эффективных.

Сортировка методом пузырька

Идея метода состоит в сравнении двух соседних элементов, в результате чего меньшее число (более легкий "пузырек") перемещается на одну позицию влево. Обычно просмотр организуют с конца, и после первого прохода самое маленькое число перемещается на первое место. Затем все повторяется от конца массива до второго элемента и т.д. Известен и другой вариант пузырьковой сортировки, в котором также сравнивают два соседних элемента, и если хотя бы одна из смежных пар была переставлена, то просмотр начинают с самого начала.

Функция bubble, реализующая первый алгоритм пузырьковой сортировки приведена ниже:

```
void bubble(int *x, int n)
{ register int i,j;
  int tmp;
  for(i=1;i<n;i++)
    for(j=n-1;j>=i; j--)
      if(x[j-1]>x[j])
        { tmp=x[j-1]; x[j-1]=x[j]; x[j]=tmp; }
```

```
}
```

Более известный алгоритм пузырьковой сортировки реализован в функции bubble1. В ней использована флажковая переменная q, которая принимает ненулевое значение в случае перестановки какой-либо смежной пары:

```
void bubble1(int *x, int n)
{ register int i,j;
  int tmp,q;
m: q=0;
  for(i=1;i<n-1;i++)
    if(x[i]>x[i+1])
      { tmp=x[i]; x[i]=x[i+1]; x[i+1]=tmp; q=1;}
  if(q) goto m;
}
```

Пузырьковая сортировка неплохо работает, когда в исходных данных многие элементы уже упорядочены. Если исходный массив уже отсортирован, то работа функции ограничивается первым проходом. В худшем случае (массив упорядочен по убыванию) количество сравнений составляет n*(n-1)/2, а количество перестановок достигает 3*n*(n-1)/2. Среднее количество перестановок равно 3*n*(n-1)/4.

Сортировка методом отбора

Идея метода: находится элемент с наименьшим значением и меняется местами с первым элементом. Среди оставшихся элементов ищется наименьший, который меняется со вторым и т.д. Функция select, реализующая такую процедуру, приведена ниже:

```
void select(int *x, int n)
{ register int i,j,k;
  int q,tmp;
  for(i=0; i<n-1;i++)
    { q=0; k=i; tmp=x[i];
      for(j=i+1; j<n; j++)
        { if(x[j]<tmp)
            { k=j; tmp=x[j]; q=1; }
        }
      if(q) { x[k]=x[i]; x[i]=tmp; }
    }
}
```

Сортировка методом отбора является n-квадратичным алгоритмом. Внешний цикл выполняется n-1 раз, а внутренний n/2. В результате производится n*(n-1)/2 сравнений, что замедляет работы алгоритма при большом количестве элементов в массиве. В наилучшем случае, если список уже упорядочен требуется сравнить только n-1 элемент и каждое сравнение требует трех промежуточных шагов. Наихудший случай аппроксимирует количество перестановок. Количество перестановок для среднего случае вычисляется по формуле n(log n + y), y – константа Эйлера (0.577216).

Оценка трудоемкости метода отбора:

- количество сравнений – n*(n-1)/2;
- количество перестановок:
  - в лучшем случае – 3*(n-1)
  - в худшем случае –n2/4+3*(n-1)
  - в среднем – n*(log n +0.577216)

Хотя количество сравнений для пузырьковой сортировки и сортировки методом отбора одинаковы, однако, для сортировки отбором показатель количества перестановок в среднем случае намного лучше. Тем не менее, существуют еще более совершенные методы сортировки.

Сортировка методом вставки

Идея метода: последовательное пополнение ранее упорядоченных элементов. На первом шаге сортируются два первые элемента. Затем на свое место среди них вставляется третий элемент. К трем упорядоченным добавляется четвертый, который занимает свое место в четверке и т.д. Примерно так игроки упорядочивают свои карты при сдаче их по одной. Функция insert, реализующая описанную процедуру, приведена ниже:

```
void insert(int *x, int n)
```

```
{ register int i,j;
  int tmp;
  for(i=1;i<n;i++)
    { tmp=x[i];
      for(j=i-1;j>=0 && tmp<x[j]; j--)
        x[j+1]=x[j];
      x[j+1]=tmp;
    }
}
```

В отличии от метода пузырьковой сортировки и сортировки методом отбора количество сравнений, имеющих место в процессе сортировки методом вставки, зависит от исходной упорядоченности списка. Если массив уже отсортирован, то все равно потребуется 2*(n-1) сравнение. Если массив упорядочен по убыванию, то число сравнений возрастает до n*(n+1)/2. По этой причине в наихудших случаях алгоритм вставки так же плох, как и пузырьковый метод или метод отбора. В среднем случае он лишь немногим лучше предыдущих методов. Однако, поведение алгоритма сортировки методом вставки естественно. Это означает, что если массив уже отсортирован в нужном порядке, алгоритм проводит наименьшее количество вычислений, а если массив отсортирован в обратном порядке – его работа наиболее интенсивна.

Сортировка методом Шелла

В 1959 году сотрудник фирмы IBM D.L. Shell предложил оригинальный алгоритм сортировки. По его предложению сначала сортируются элементы, отстоящие друг от друга на 3 позиции, затем – на две позиции и, наконец, сортируются смежные элементы. В дальнейшем экспериментальным путем были найдены более удачные расстояния между сортируемыми элементами: 9→5→3→2→1. Среднее время работы усовершенствованного алгоритма Шелла порядка $n^{1.2}$. Это существенно лучше, чем характерная для трех предыдущих методов величина порядка n^2.

```
void shell(int *x, int n)
{ register int i,j,gap,k;
  int xx;
  char a[5]={9,5,3,2,1};
  for(k=0;k<5;k++)
    { gap=a[k];
      for(i=gap;i<n;i++)
        { xx=x[i];
          for(j=i-gap; xx<x[j] && j>=0; j=j-gap)
            x[j+gap]=x[j];
          x[j+gap]=xx;
        }
     }
}
```

Быстрая сортировка

Известный математик C.A.R. Hoare в 1962 году опубликовал алгоритм быстрой сортировки, за которым закрепилось название quicksort. Основная идея быстрой сортировки напоминает метод поиска делением пополам. Сначала выбирается граничный элемент в сортируемом массиве, его называют компарандом. Все, что больше этого элемента переносится в правую часть массива, а все, что меньше – в левую. После первого шага средний элемент оказывается на своем месте. Затем аналогичная процедура повторяется для каждой половины массива. На каждом последующем шаге размер обрабатываемого фрагмента массива уменьшается вдвое. Количество операций, которое требуется для реализации этой процедуры, оценивается константой $n*\log_2 n$. Это еще быстрее, чем сортировка Шелла. В отличие от предыдущих функций быстрая сортировка

оформлена из двух функций – quick, которая допускает принятое в других функциях обращение, и рекурсивной процедуры qs:

```
void quick(int *x, int n)
{ qs(x,0,n-1); }
//---------------------------------
void qs(int *x,int left,int right)
{ register int i,j;
  int xx,tmp;
  i=left; j=right;
  xx=x[(left+right)/2];
  do { while(x[i]<xx && i<right)i++;
       while(xx<x[j] && j>left) j--;
       if(i<=j)
         { tmp=x[i]; x[i]=x[j];
           x[j]=tmp; i++; j--;
         }
      }
   while(i<=j);
   if(left<j) qs(x,left,j);
   if(i<right)qs(x,i,right);
}
```

Здесь функция quick(int *x, int n) задает вызов основной сортиру.щей функции qs(int *x,int left,int right). Среднее число выполняемых сравнений равно: n*log n. Среднее количество перестановок приблизительно равно: (n*log n)/6. Эти показатели существенно меньше, чем у рассмотренных выше методов сортировки. Следует иметь в виду, если значение-компаранд для каждого из разделов является максимальным, то на практике метод быстрой сортировки вырождается в метод «медленной» сортировки с временем выполнения, пропорциональным квадрату количества элементов.

Указатели

Указатели - это переменные, которые хранят адреса объектов. В неявном виде указатели присутствовали и в других языках программирования, но в Си они используются гораздо чаще, а работа с указателями организована максимально просто.

При описании указателя надо задать тип объектов, адреса которых будут содержаться в нем. Перед именем указателя при описании ставится звездочка, чтобы отличить его от обычной переменной. Примеры описаний указателей:

```
int *a, *b, c, d;
char *e;
void *f;
```

В первой строке описаны указатели a и b на тип int и простые переменные c и d типа int (c и d - не указатели!). С указателями возможны следующие два действия:

1. присвоить указателю адрес некоторой переменной. Для этого используется операция взятия адреса, которая обозначается амперсандом &. Например, строка

```
a = &c;
```

   указателю a присваивает значение адреса переменной c;

2. получить объект, адрес которого содержится в указателе; для этого используется операция звездочка '*', которая записывается перед указателем. (Заметим, что звездочкой обозначается также операция умножения.) Например, строка

```
d = *a;
```

   присваивает переменной d значение целочисленной переменной, адрес которой содержится в a. Так как ранее указателю a был присвоен адрес переменной c, то в результате переменной d присваивается значение c, т.е. данная строка эквивалентна следующей:

```
d = c;
```

Описание указателей

Конструкции массива и указателя при описании типа можно применять многократно в произвольном порядке. Кроме того, можно описывать прототип функции. Таким образом можно строить сложные описания вроде "массив указателей", "указатель на указатель", "указатель на массив", "функция, возвращающая значение типа указатель", "указатель на функцию" и т.д. Правила здесь таковы:

приоритеты конструкций описания распределены следующим образом:

операция * определения указателя имеет самый низкий приоритет. Например, описание

```
int *x[10];
```

означает "массив из 10 элементов типа указатель на int". Здесь к имени переменной x сначала применяется операция определения массива [] (квадратные скобки), поскольку она имеет более высокий приоритет, чем звездочка. Затем к полученному массиву применяется операция определения указателя. В результате получается "массив указателей", а не указатель на массив! Если нам нужно определить указатель на массив, то следует использовать круглые скобки при описании:

```
int (*x)[10];
```

Здесь к имени x сначала применяется операция * определения указателя;

операции определения массива [] (квадратные скобки после имени) и определения функции (круглые скобки после имени) имеют одинаковый приоритет, более высокий, чем звездочка.

Примеры:

```
int f();
```

Описан прототип функции f без аргументов, возвращающей значение типа int.

```
int (*f())[10];
```

Описан прототип функции f без аргументов, возвращающей значение типа указатель на массив из 10 элементов типа int;

последний пример уже не является очевидным.

Общий алгоритм разбора сложного описания можно охарактеризовать как чтение изнутри. Сначала находим описываемое имя. Затем определяем, какая операция применяется к имени первой. Если нет круглых скобок для группировки, то это либо определение указателя (звездочка слева от имени), либо определение массива (квадратные скобки справа от имени), либо определение функции (круглые скобки справа от имени). Таким образом получается первый шаг сложного описания. Затем находим следующую

операцию описания, которая применяется к уже выделенной части сложного описания, и повторяем это до тех пор, пока не исчерпаем все описание.

При описании указателя модификатор const, записанный до звездочки, означает, что описан указатель на константный объект, т.е. на объект, менять который нельзя или запрещено. Например, в строке:

```
const char *p;
```

описан указатель на константную строку (массив символов, менять который запрещено).

Указатели на константные объекты используются в Си чрезвычайно часто. Причина состоит в том, что константный указатель позволяет прочесть объект и при этом гарантирует, что объект не будет испорчен в результате ошибки программирования, т.к. константный указатель не дает возможности изменить объект.

Константный указатель ссылается на константный объект, однако, содержимое самого указателя может изменяться. Например, следующий фрагмент вполне корректен:

```
const char *str = "e2e4";
. . .
str = "c7c5";
```

Здесь константный указатель str сначала содержит адрес константной строки "e2e4". Затем в него записывается адрес другой константной строки "c7c5".

В Си можно также описать указатель, значение которого не может быть изменено; для этого модификатор const указывается после звездочки. Например, фрагмент кода:

```
int i;
int * const p = &i;
```

навечно записывает в указатель p адрес переменной i, перенаправить указатель p на другую переменную уже нельзя. Строка:

```
p = &n;
```

является ошибкой, т.к. указатель p - константа, а константе нельзя присвоить новое значение. Указатели, значения которых изменять нельзя, используются в Си значительно реже, в основном при заполнении константных таблиц.

Представление через указатели матриц и многомерных массивов

Специального типа данных матрица или многомерный массив в Си нет, однако, можно использовать массив элементов типа массив. Например, переменная a представляет матрицу размера 3×3 с вещественными элементами:

```
double a[3][3];
```

Для обращения к элементу матрицы надо записать его индексы в квадратных скобках, например, выражение `a[i][j]` представляет собой элемент матрицы a в строке с индексом i и столбце с индексом j. Элемент матрицы можно использовать в любом выражении как обычную переменную (например, можно читать его значение или присваивать новое).

Такая реализация матрицы удобна и максимально эффективна с точки зрения времени доступа к элементам. У нее только один существенный недостаток: так можно реализовать только матрицу, размер которой известен заранее. Язык Си не позволяет описывать массивы переменного размера, размер массива должен быть известен до начала работы программы еще на стадии компиляции.

Пусть нужна матрица, размер которой определяется во время работы программы. Тогда пространство под нее надо захватывать в динамической памяти с помощью функции malloc языка Си. При этом в динамической памяти захватывается линейный массив и возвращается указатель на него. Рассмотрим вещественную матрицу размером m строк на n столбцов. Выделение памяти выполняется с помощью функции malloc языка Си:

```
double *a;
. . .
a = (double *) malloc(m * n * sizeof(double));
```

При этом считается, что элементы матрицы будут располагаться в массиве следующим образом: сначала идут элементы строки с индексом 0, затем элементы строки

с индексом 1 и т.д., последними идут элементы строки с индексом m − 1. Каждая строка состоит из n элементов, следовательно, индекс элемента строки i и столбца j в линейном массиве равен i * n + j (действительно, поскольку индексы начинаются с нуля, то i равно количеству строк, которые нужно пропустить, i * n - суммарное количество элементов в пропускаемых строках; число j равно смещению внутри последней строки). Таким образом, элементу матрицы в строке i и столбце j соответствует выражение a[i * n + j]. Этот способ представления матрицы удобен и эффективен. Его основное преимущество состоит в том, что элементы матрицы хранятся в непрерывном отрезке памяти. Во-первых, это позволяет оптимизирующему компилятору преобразовывать текст программы, добиваясь максимального быстродействия; во-вторых, при выполнении программы максимально используется механизм кеш-памяти, сводящий к минимуму обращения к памяти и значительно ускоряющий работу программы.

В некоторых книгах по Си рекомендуется реализовывать матрицу как массив указателей на ее строки, при этом память под каждую строку выделяется отдельно в динамической памяти:

```
double **a; // Адрес массива указателей
int m, n;   // Размеры матрицы: m строк, n столбцов
int i;
. . .
// Захватывается память под массив указателей
a = (double **) malloc(m * sizeof(double *));

for (i = 0; i < m; ++i) {
    // Захватывается память под строку с индесом i
    a[i] = (double *) malloc(n * sizeof(double));
}
```

После этого к элементу a_{ij} можно обращаться с помощью выражения a[i][j].Несмотря на всю сложность этого решения, никакого выигрыша нет, наоборот, программа проигрывает в скорости. Причина состоит в том, что матрица не хранится в непрерывном участке памяти, это мешает как оптимизации программы, так и эффективному использованию кеш-памяти.

Многомерные массивы реализуются аналогично матрицам. Например, вещественный трехмерный массив размера 4 × 4 × 2 описывается как: `double a[4][4][2];`
обращение к его элементу с индексами x, y, z осуществляется с помощью выражения
`a[x][y][z]`
Многомерные массивы переменного размера с числом индексов большим двух встречаются в программах довольно редко, но никаких проблем с их реализацией нет: они реализуются аналогично матрицам. Например, пусть надо реализовать трехмерный вещественный массив размера m × n × k. Захватывается линейный массив вещественных чисел размером m * n * k:

```
double *a;
    . . .
    a = (double *) malloc(m * n * k * sizeof(double));
```

Доступ к элементу с индексами x, y, z осуществляется с помощью выражения `a[(x * n + y) * k + z]`.

Арифметика указателей

С указателями можно выполнять следующие операции:

- сложение указателя и целого числа, результат - указатель;
- увеличение или уменьшение переменной типа указатель, что эквивалентно прибавлению или вычитанию единицы;
- вычитание двух указателей, результат - целое число.

Прибавление к указателю p целого числа n означает увеличение адреса, который содержится в переменной p, на суммарный размер n элементов того типа, на который ссылается указатель. Указатель как бы сдвигается на n элементов вправо, если считать,

что индексы элементов массива возрастают слева направо. Аналогично вычитание целого числа n из указателя означает сдвиг указателя влево на n элементов. Пример:

```
int *p, *q;
int a[100];
p = &(a[5]); // записываем в p адрес 5-го
             //     элемента массива a
p += 7;      // p будет содержать адрес 12-го эл-та
q = &(a[10]);
--q;         // q содержит адрес элемента a[9]
```

Значение указателя при прибавлении к нему целого числа n увеличивается на произведение n на количество байтов, занимаемое одним элементом того типа, на который ссылается указатель. В программировании это называют масштабированием. Разность двух указателей - это количество элементов данного типа, которое умещается между двумя адресами. Результатом вычитания указателей является целое число. Физически оно вычисляется как разность значений двух адресов, деленная на размер одного элемента заданного типа. Операции сложения указателя с целым числом и разности двух указателей взаимно обратны:

```
int *p, *q;
int a[100];
int n;
p = &(a[5]);
q = &(a[12]);
n = q - p;      // n == 7
q = p + n;      // q == &(a[12])
```

Подчеркнем, что указатели нельзя складывать! В отличие от разности указателей, операция сложения указателей (т.е. сложения адресов памяти) абсолютно бессмысленна.

```
int *p, *q, *r;
int a[100];
p = &(a[5]);
q = &(a[12]);
r = p + q;  // Ошибка! Указатели нельзя складывать.
```

ЗАДАНИЯ

Работу с массивами организовать через указатели.

1.Переворот одномерного целочисленного массива – перестановка его элементов в обратном порядке. Выделить в отдельную функцию процедуру инвертирования.

2.Перестановка головы и хвоста массива без использования промежуточного массива. Алгоритм этой процедуры заключается он в том, что надо последовательно выполнить 3 инвертирования – головы массива, хвоста массива и всего массива целиком. Инвертирование выделить в отдельную функцию.

1 2 3 4 5 6 7 8 9 10
Голова: 1 2 3 4 5 6
Хвост: 7 8 9 10
Результат: 7 8 9 10 1 2 3 4 5 6

3.Определить количество разных элементов в целочисленном массиве. Алгоритм, выполняющую основную задачу выделить в отдельную функцию.

4. Написать программу перемножения квадратных матриц. Операцию перемножение вынести в отдельную функцию.

Представление программы в виде функций

Функции разбивают большие вычислительные задачи на более мелкие и позволяют воспользоваться тем, что уже сделано другими разработчиками, а не начинать создание программы каждый раз "с нуля". В выбранных должным образом функциях "упрятаны" несущественные для других частей программы детали их функционирования, что делает программу в целом более ясной и облегчает внесение в нее изменений.

Язык проектировался так, чтобы функции были эффективными и простыми в использовании. Обычно программы на Си состоят из большого числа небольших функций, а не из немногих больших. Программу можно располагать в одном или нескольких исходных файлах. Эти файлы можно компилировать отдельно, а загружать вместе, в том числе и с ранее откомпилированными библиотечными функциям

Проиллюстрируем механизм определения функции на примере функции `power(m, n)`, которая возводит целое m в целую положительную степень n. Так, power(2, 5) имеет значение 32. На самом деле для практического применения эта функция малопригодна, так как оперирует лишь малыми целыми степенями, однако она вполне может послужить иллюстрацией. Итак, мы имеем функцию power и главную функцию main, пользующуюся ее услугами, так что вся программа выглядит следующим образом:

```
#include <stdio.h>
int power(int m, int n);
/* тест функции power */
main()
{
    int i;
    for (i = 0: i < 10, ++i)
        printf(“%d %d %d\n”, i, power(2,i), power(-3,i));
    return 0;
}

/* возводит base в n-ю степень, n >= 0 */
int power(int base, int n)
{   int i, p;
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Определение любой функции имеет следующий вид:

```
тип-результата имя-функции (список параметров, если он есть)
{
    объявления
    инструкции
}
```

Определения функций могут располагаться в любом порядке в одном или в нескольких исходных файлах, но любая функция должна быть целиком расположена в каком-то одном. Если исходный текст программы распределен по нескольким файлам, то, чтобы ее скомпилировать и загрузить, вам придется сказать несколько больше, чем при использовании одного файла. В следующей строке из функции main к power обращаются дважды.

```
printf(“%d %d %d\n”, i, power(2,i), power(-3,i));
```

При каждом вызове функции power передаются два аргумента, и каждый раз главная программа main в ответ получает целое число, которое затем приводится к должному формату и печатается. Внутри выражения power(2, i) представляет собой целое значение точно так же, как 2 или i.

В первой строке определения power:

```
int power(int base, int n);
```

указываются типы параметров, имя функции и тип результата. Имена параметров локальны внутри power, это значит, что они скрыты для любой другой функции, так что

остальные подпрограммы могут свободно пользоваться теми же именами для своих целей. Последнее утверждение справедливо также для переменных `i` и `p`: `i` в `power` и `i` в `main` не имеют между собой ничего общего.

Далее *параметром* мы будем называть переменную из списка параметров, заключенного в круглые скобки и заданного в определении функции, а *аргументом* - значение, используемое при обращении к функции. Иногда в том же смысле мы будем употреблять термины *формальный аргумент* и *фактический аргумент*.

Значение, вычисляемое функцией `power`, возвращается в `main` с помощью инструкции `return`. За словом *return* может следовать любое выражение:

```
return выражение;
```

Функция не обязательно возвращает какое-нибудь значение. Инструкция return без выражения только передает управление в ту программу, которая ее вызвала, не передавая ей никакого результирующего значения. То же самое происходит, если в процессе вычислений мы выходим на конец функции, обозначенный в тексте последней закрывающей фигурной скобкой. Возможна ситуация, когда вызывающая функция игнорирует возвращаемый ей результат.

Поскольку main есть функция, как и любая другая она может вернуть результирующее значение тому, кто ее вызвал, - фактически в ту среду, из которой была запущена программа. Обычно возвращается нулевое значение, что говорит о нормальном завершении выполнения. Ненулевое значение сигнализирует о необычном или ошибочном завершении.

Объявление

```
int power(int m, int n);
```

стоящее непосредственно перед `main`, сообщает, что функция power ожидает двух аргументов типа *int* и возвращает результат типа *int*. Это объявление, называемое *прототипом функции*, должно быть согласовано с определением и всеми вызовами *power*. Если определение функции или вызов не соответствует своему прототипу, это ошибка.

Аргументы. Вызов по значению

В Си все аргументы функции передаются “по значению”. Это следует понимать так, что вызываемой функции посылаются значения ее аргументов во временных переменных, а не сами аргументы. Такой способ передачи аргументов несколько отличается от “вызова по ссылке” в Фортране и спецификации var при параметре в Паскале, которые позволяют подпрограмме иметь доступ к самим аргументам, а не к их локальным копиям.

Главное отличие заключается в том, что в Си вызываемая функция не может непосредственно изменить переменную вызывающей функции: она может изменить только ее частную, временную копию. В качестве примера приведем еще одну версию функции power, в которой как раз использовано это свойство.

```
/* power: возводит base в n-ю степень; n >= 0*/
int power(int base, int n)
{
    int p;
    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр n выступает здесь в роли временной переменной, в которой циклом for в убывающем порядке ведется счет числа шагов до тех пор, пока ее значение не станет нулем. При этом отпадает надобность в дополнительной переменной i для счетчика цикла. Что бы мы ни делали с n внутри power, это не окажет никакого влияния на сам аргумент, копия которого была передана функции power при ее вызове.

При желании можно сделать так, чтобы функция смогла изменить переменную в вызывающей программе. Для этого последняя должна передать адрес подлежащей изменению переменной (*указатель* на переменную), а в вызываемой функции следует

объявить соответствующий параметр как указатель и организовать через него косвенный доступ к этой переменной.

Механизм передачи массива в качестве аргумента несколько иной. Когда аргументом является имя массива, то функции передается значение, которое является адресом начала этого массива; никакие элементы массива не копируются. С помощью индексирования относительно полученного значения функция имеет доступ к любому элементу массива.

Внешние переменные

Программа на Си обычно оперирует с множеством внешних объектов: переменных и функций. Прилагательное "внешний" (`external`) противоположно прилагательному "внутренний", которое относится к аргументам и переменным, определяемым внутри функций. Внешние переменные определяются вне функций и потенциально доступны для многих функций. Сами функции всегда являются внешними объектами, поскольку в Си запрещено определять функции внутри других функций. По умолчанию одинаковые внешние имена, используемые в разных файлах, относятся к одному и тому же внешнему объекту (функции). (В стандарте это называется редактированием внешних связей (линкованием) (external linkage).)

Поскольку внешние переменные доступны всюду, их можно использовать в качестве связующих данных между функциями как альтернативу связей через аргументы и возвращаемые значения. Для любой функции внешняя переменная доступна по ее имени, если это имя было должным образом объявлено.

Если число переменных, совместно используемых функциями, велико, связи между последними через внешние переменные могут оказаться более удобными и эффективными, чем длинные списки аргументов. Внешние переменные полезны, так как они имеют большую область действия и время жизни. Автоматические переменные существуют только внутри функции, они возникают в момент входа в функцию и исчезают при выходе из нее. Внешние переменные, напротив, существуют постоянно, так что их значения сохраняются и между обращениями к функциям. Таким образом, если двум функциям приходится пользоваться одними и теми же данными и ни одна из них не вызывает другую, то часто бывает удобно оформить эти общие данные в виде внешних переменных, а не передавать их в функцию и обратно через аргументы.

Области видимости

Функции и внешние переменные, из которых состоит Си-программа, каждый раз компилировать все вместе нет никакой необходимости. Исходный текст можно хранить в нескольких файлах. Ранее скомпилированные программы можно загружать из библиотек. В связи с этим возникают следующие вопросы:

• Как писать объявления, чтобы на протяжении компиляции используемые переменные были должным образом объявлены?

• В каком порядке располагать объявления, чтобы во время загрузки все части программы оказались связаны нужным образом?

• Как организовать объявления, чтобы они имели лишь одну копию?

• Как инициализировать внешние переменные?

Начнем с того, что разобьем программу-калькулятор на несколько файлов. Конечно, эта программа слишком мала, чтобы ее стоило разбивать на файлы, однако разбиение программы позволит продемонстрировать проблемы, возникающие в больших программах.

Областью видимости имени считается часть программы, в которой это имя можно использовать. Для автоматических переменных, объявленных в начале функции, областью видимости является функция, в которой они объявлены. Локальные переменные разных функций, имеющие, однако, одинаковые имена, никак не связаны друг с другом. То же

утверждение справедливо и в отношении параметров функции, которые фактически являются локальными переменными.

Область действия внешней переменной или функции простирается от точки программы, где она объявлена, до конца файла, подлежащего компиляции. Например, если *main*, *sp*, *val*, *push* и *pop* определены в одном файле в указанном порядке, т. е.

```
main() {...}
int sp = 0;
double val[MAXVAL];
void push(double f) {...}
double pop(void) {...}
```

то к переменным *sp* и *val* можно адресоваться из *push* и *pop* просто по их именам; никаких дополнительных объявлений для этого не требуется. Заметим, что в *main* эти имена не видимы так же, как и сами *push* и *pop*. Однако, если на внешнюю переменную нужно сослаться до того, как она определена, или если она определена в другом файле, то ее объявление должно быть помечено словом extern.

Важно отличать *объявление* внешней переменной от ее *определения*. Объявление объявляет свойства переменной (прежде всего ее тип), а определение, кроме того, приводит к выделению для нее памяти. Если строки

int sp;
double val[MAXVAL];

расположены вне всех функций, то они *определяют* внешние переменные *sp* и *val*, т. е. отводят для них память, и, кроме того, служат объявлениями для остальной части исходного файла. А вот строки

extern int sp;
extern double val[];

объявляют для оставшейся части файла, что *sp* - переменная типа *int*, а *val* - массив типа *double* (размер которого определен где-то в другом месте); при этом ни переменная, ни массив не создаются, и память им не отводится.

На всю совокупность файлов, из которых состоит исходная программа, для каждой внешней переменной должно быть одно-единственное *определение*; другие файлы, чтобы получить доступ к внешней переменной, должны иметь в себе объявление *extern*. (Впрочем, объявление *extern* можно поместить и в файл, в котором содержится определение.) В определениях массивов необходимо указывать их размеры, что в объявлениях *extern* не обязательно. Инициализировать внешнюю переменную можно только в определении. Хотя вряд ли стоит организовывать нашу программу таким образом, но мы определим push и pop в одном файле, а val и sp - в другом, где их и инициализируем. При этом для установления связей понадобятся такие определения и объявления:

В файле 1:

```
extern int sp;
extern double val[];
void push(double f) {...}
double pop(void) {...}
```

В файле2:

```
int sp = 0;
double val[MAXVAL];
```

Поскольку объявления *extern* находятся в начале *файла1* и вне определений функций, их действие распространяется на все функции, причем одного набора объявлений достаточно для всего *файла1*. Та же организация *extern*-объявлений необходима и в случае, когда программа состоит из одного файла, но определения sp и val расположены после их использования.

Представление кода программы в нескольких файлах

Заголовочные файлы

Представим себе, что компоненты программы имеют существенно большие размеры, и зададимся вопросом, как в этом случае распределить их по нескольким файлам. Программу *main* поместим в файл, который мы назовем *main.c*; функции *push, pop* и их переменные расположим во втором файле, *stack.c*; а функцию *getop* - в третьем, *getop.c*. Наконец, функции *getch* и *ungetch* разместим в четвертом файле *getch.c*; мы отделили их от остальных функций.

Существует еще один момент, о котором следует предупредить читателя, - определения и объявления совместно используются несколькими файлами. Мы бы хотели, насколько это возможно, централизовать эти объявления и определения так, чтобы для них существовала только одна копия. Тогда программу в процессе ее развития будет легче и исправлять, и поддерживать в нужном состоянии. Для этого общую информацию расположим в заголовочном файле *calc.h*, который будем по мере необходимости включать в другие файлы. В результате получим программу, файловая структура которой показана ниже:

main.c:
```
#include <stdio.h>
#include <stdlib.h>
#include "calc.h"
#define MAXOP 100
main() {
    ...
}
```

calc.h:
```
#define NUMBER '0'
void push(double);
double pop(void);
int getop(char[]);
int getch(void);
void ungetch(int);
```

getop.c:
```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
getop (){
    ...
}
```

getch.c:
```
#include <stdio.h>
#define BUFSIZE 100
char buf[BUFSIZE];
intbufp = 0;
int getch(void) {
    ...
}
void ungetch(int) {
    ...
}
```

stack.c:
```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
```

```
double val[MAXVAL];
void push(double) {
    ...
}
double pop(void) {
    ...
}
```
Неизбежен компромисс между стремлением, чтобы каждый файл владел только той информацией, которая ему необходима для работы, и тем, что на практике иметь дело с большим количеством заголовочных файлов довольно трудно. Для программ, не превышающих некоторого среднего размера, вероятно, лучше всего иметь один заголовочный файл, в котором собраны вместе все объекты, каждый из которых используется в двух различных файлах; так мы здесь и поступили. Для программ больших размеров потребуется более сложная организация с большим числом заголовочных файлов.

Статические переменные

Переменные *sp* и *val* в файле *stack.c*, а также *buf* и *bufp* в *getch.c* находятся в личном пользовании функций этих файлов, и нет смысла открывать к ним доступ кому-либо еще. Указание `static`, примененное к внешней переменной или функции, ограничивает область видимости соответствующего объекта концом файла. Это способ скрыть имена. Так, переменные *buf* и *bufp* должны быть внешними, поскольку их совместно используют функции *getch* и *ungetch*, но их следует сделать невидимыми для "пользователей" функций *getch* и *ungetch*.

Статическая память специфицируется словом `static`, которое помещается перед обычным объявлением. Если рассматриваемые нами две функции и две переменные компилируются в одном файле, как в показанном ниже примере:

```
static char buf[BUFSIZE]; /* буфер для ungetch */
static int bufp = 0;      /* след. свободная позиция в buf */

int getch(void) {...}

void ungetch(int c) {...}
```

то никакая другая программа не будет иметь доступ ни к *buf*, ни к *bufp*, и этими именами можно свободно пользоваться в других файлах для совсем иных целей. Точно так же, помещая указание static перед объявлениями переменных *sp* и *val*, с которыми работают только *push* и *pop*, мы можем скрыть их от остальных функций.

Указание static чаще всего используется для переменных, но с равным успехом его можно применять и к функциям. Обычно имена функций глобальны и видимы из любого места программы. Если же функция помечена словом static, то ее имя становится невидимым вне файла, в котором она определена.

Объявление static можно использовать и для внутренних переменных. Как и автоматические переменные, внутренние статические переменные локальны в функциях, но в отличие от автоматических, они не возникают только на период работы функции, а существуют постоянно. Это значит, что внутренние статические переменные обеспечивают постоянное сохранение данных внутри функции.

ЗАДАНИЯ НА ДОМ

Методы сортировки должны быть реализованы в виде функций. Работу с массивами организовать через указатели. Код программы должен располагаться в нескольких файла: в заголовочном файле определить необходимые переменные и заголовки функций, во втором файле должен располагаться код функций сортировки, в главном файле – вызов функции main.

1. Написать программу сортировки массивов методом отбора и методом Шелла. Сравнить скорость работы указанных методов сортировки.

2. Написать программу сортировки массивов методом вставки и с помощью быстрой сортировки. Сравнить скорость работы указанных методов сортировки.

3. Написать программу сортировки массивов методом отбора и методом быстрой сортировки. Сравнить скорость работы указанных методов сортировки.

4. Написать программу сортировки массивов методом вставки и методом Шелла. Сравнить скорость работы указанных методов сортировки.

Для вычисление времени работы алгоритма использовать функцию gettimeofday(&tv,NULL) из sys/time.h

```
#include <sys/time.h>
unsigned long GetTickCount()
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return (tv.tv_sec*1000+tv.tv_usec/1000);
}
```

Тип char

В языке С среди базовых типов данных строк как таковых не оказалось. Вместо этого язык С предлагает использовать одномерные символьные массивы, в которых хранятся те же строки в виде последовательности однобайтовых символов, завершающихся байтом с нулевым кодом. Для этого признака конца строки в состав Escape- последовательностей включен специальный код '\0', хотя можно было бы воспользоваться и другой комбинацией – '\0x0'.
В языке С++ появился гораздо более удобный класс строковых данных string. Но знакомство с этим классом мы отложим на более поздний срок.
Значения строковых констант или начальные значения строковых "переменных" в отличие от символьных данных заключаются в двойные кавычки:

```
  const char c1[]="ABCDEFGH";
  char str1[]="1234";
  char letter[]="a";
  char symbol='a';
```

Байт с нулевым кодом система автоматически добавляет вслед за последним символом строки. Поэтому для символьного массива c1 будет выделено 9 байтов, а для символьного массива str1 – 5 байтов. Обратите внимание на то, что массив letter занимает в памяти 2 байта, тогда как символьная переменная symbol – 1 байт. Так как строка представлена одномерным массивом символов, то доступ к каждому ее символу осуществляется самым обычным способом:

c1[3] – четвертый символ в массиве c1 (т.е. буква 'D')
str1[0] – первый символ в массиве str1 (т.е. цифра '1')

Значение любого символа в строковой "переменной" можно изменить во время работы программы:

```
 str1[3]='L';
```

Запись за пределы строки может непредсказуемым образом повлиять на последующую работу программы:

```
  str1[4]=5; //Байт с признаком конца строки испорчен
  str1[5]=6; //Испорчен байт с какими-то данными
```

Список форматных указателей функции scanf предусматривает возможность ввода значений односимвольных (%c) и многосимвольных (%s) переменных:

```
  #include <stdio.h>
void main()
{ char ch1,ch2;
  char str1[10];
  scanf("%c %c",&ch1,&ch2);
  scanf("%s",str1);
.....................
```

Обратите внимание на то, что для ввода данных в скалярные переменные ch1 и ch2 в списке ввода необходимо указывать их адреса (&ch1, &ch2), а при вводе в массив str1 – достаточно написать его имя. Дело в том, что имя массива одновременно выполняет роль адреса своего первого элемента. Поэтому str1 и &str1[0] указывают на один и тот же адрес. Ввод значений символьных переменных ch1 и ch2 можно организовать одним из двух способов. Во-первых, в строке ввода можно набрать два требуемых символа либо слитно, либо разделяя их хотя бы одним пробелом и нажать клавишу Enter. Во-вторых, можно набрать первый символ, предназначенный для переменной ch1, и нажать клавишу Enter. Затем повторить аналогичным образом ввод следующего символа.

Некоторые функции работы со строками: для обработки строк, представленных одномерными символьными массивами, в библиотеке системных функций предусмотрено довольно много различных операций. Прототипы этих функций сгруппированы в

заголовочном файле string.h и большинство их названий начинается с префикса str (от string). Условимся о некоторых обозначениях аргументов и их типах, чтобы не повторять их в приведенной таблице:

- S, S1,S2 – указатель на символьный массив (как правило, имя массива);
- CS – указатель типа const char * (т.е. неизменяемый массив или строковая константа – источник данных);
- ch – код символа, обычно числовое значение типа int;
- k – количество символов.

strlen(CS)	Возвращает количество символов в строке S
strcpy(S1,CS2)	Копирует содержимое CS2 в S1, возвращает указатель на S1
strncpy(S1,CS2,k)	Копирует первые k символов из CS2 в S1, возвращает указатель на S1
strcat(S1,CS2)	Приписывает содержимое CS2 в конец S1, возвращает указатель на S1 (длина массива S1 должна предусматривать такое расширение)
strncat(S1,CS2,k)	Присоединяет первые k символов CS2 к содержимому S1, возвращает указатель на S1
strcmp(CS1,CS2)	Возвращаемое значение равно 0, если CS1=CS2, больше 0, если CS1>CS2, и меньше 0, если CS1<CS2
strncmp(CS1,CS2,k)	Сравниваются только первые k символов строк CS1 и CS2
stricmp(CS1,CS2)	При сравнении игнорируется разница между кодами больших и малых букв
strchr(CS,ch)	Строка CS сканируется слева направо до обнаружения символа ch. Если он найден, возвращаемый указатель "смотрит" на этот символ в строке CS, если такого символа нет, то возвращаемый указатель равен null (т.е. 0)
strrchr(CS,ch)	Аналогичный поиск с конца строки CS.
strstr(CS1,CS2)	Поиск первого вхождения строки CS2 в строку CS1. Если поиск завершен успешно, возвращается указатель на первый символ найденной подстроки. В противном случае возвращается null

Работа с памятью

В традиционных языках программирования, таких как Си, Фортран, Паскаль, существуют три вида памяти: статическая, стековая и динамическая. Конечно, с физической точки зрения никаких различных видов памяти нет: оперативная память - это массив байтов, каждый байт имеет адрес, начиная с нуля. Когда говорится о видах памяти, имеются в виду способы организации работы с ней, включая выделение и освобождение памяти, а также методы доступа.

Статическая память

Статическая память выделяется еще до начала работы программы, на стадии компиляции и сборки. Статические переменные имеют фиксированный адрес, известный до запуска программы и не изменяющийся в процессе ее работы. Статические переменные создаются и инициализируются до входа в функцию main, с которой начинается выполнение программы.

Существует два типа статических переменных:

глобальные переменные - это переменные, определенные вне функций, в описании которых отсутствует слово static. Обычно описания глобальных переменных, включающие слово extern, выносятся в заголовочные файлы (h-файлы). Слово extern означает, что переменная описывается, но не создается в данной точке программы. Определения глобальных переменных, т.е. описания без слова extern, помещаются в файлы реализации (c-файлы или cpp-файлы).

статические переменные - это переменные, в описании которых присутствует слово static. Как правило, статические переменные описываются вне функций. Такие статические переменные во всем подобны глобальным, с одним исключением: область видимости статической переменной ограничена одним файлом, внутри которого она определена, - и, более того, ее можно использовать только после ее описания, т.е. ниже по тексту. По этой причине описания статических переменных обычно выносятся в начало файла. В отличие

от глобальных переменных, статические переменные никогда не описываются в h-файлах (модификаторы extern и static конфликтуют между собой).

Стековая, или локальная, память

Локальные, или стековые, переменные - это переменные, описанные внутри функции. Память для таких переменных выделяется в аппаратном стеке. Память выделяется в момент входа в функцию или блок и освобождается в момент выхода из функции или блока. При этом захват и освобождение памяти происходят практически мгновенно, т.к. компьютер только изменяет регистр, содержащий адрес вершины стека.
Локальные переменные можно использовать при рекурсии, поскольку при повторном входе в функцию в стеке создается новый набор локальных переменных, а предыдущий набор не разрушается. Всегда следует избегать использования глобальных и статических переменных, если можно обойтись локальными.

Недостатки локальных переменных являются продожением их достоинств. Локальные переменные создаются при входе в функцию и исчезают после выхода из нее, поэтому их нельзя использовать в качестве данных, разделяемых между несколькими функциями. К тому же, размер аппаратного стека не бесконечен, стек может в один прекрасный момент переполниться (например, при глубокой рекурсии), что приведет к катастрофическому завершению программы. Поэтому локальные переменные не должны иметь большого размера. В частности, нельзя использовать большие массивы в качестве локальных переменных.

Динамическая память, или куча.

Помимо статической и стековой памяти, существует еще практически неограниченный ресурс памяти, которая называется динамическая, или куча (heap). Программа может захватывать участки динамической памяти нужного размера. После использования ранее захваченный участок динамической памяти следует освободить.
Под динамическую память отводится пространство виртуальной памяти процесса между статической памятью и стеком. Обычно стек располагается в старших адресах виртуальной памяти и растет в сторону уменьшения адресов. Программа и константные данные размещаются в младших адресах, выше располагаются статические переменные. Структура динамической памяти автоматически поддерживается исполняющей системой языка Си или C++. Динамическая память состоит из захваченных и свободных сегментов, каждому из которых предшествует описатель сегмента. При выполнении запроса на захват памяти исполняющая система производит поиск свободного сегмента достаточного размера и захватывает в нем отрезок требуемой длины. При освобождении сегмента памяти он помечается как свободный, при необходимости несколько подряд идущих свободных сегментов объединяются.

В языке Си для захвата и освобождения динамической памяти применяются стандартные функции malloc и free, описания их прототипов содержатся в стандартном заголовочном файле "stdlib.h". (Имя malloc является сокращением от memory allocate - "захват памяти".) Прототипы этих функций выглядят следующим образом:

```
void *malloc(size_t n); // Захватить участок памяти
                        // размером в n байт
void free(void *p); // Освободить участок
                    // памяти с адресом p
```

Здесь n - это размер захватываемого участка в байтах, size_t - имя одного из целочисленных типов, определяющих максимальный размер захватываемого участка. Функция malloc возвращает адрес захваченного участка памяти или ноль в случае неудачи (когда нет свободного участка достаточно большого размера). Функция free освобождает участок памяти с заданным адресом. Для задания адреса используется указатель общего типа void*. После вызова функции malloc его необходимо привести к указателю на

конкретный тип, используя операцию приведения типа. Например, в следующем примере захватывается участок динамической памяти размером в 4000 байтов, его адрес присваивается указателю на массив из 1000 целых чисел:

```
int *a;     // Указатель на массив целых чисел
. . .
a = (int *) malloc(1000 * sizeof(int));
```

Выражение в аргументе функции malloc равно 4000, поскольку размер целого числа sizeof(int) равен четырем байтам. Для преобразования указателя используется операция приведения типа (int *) от указателя обобщенного типа к указателю на целое число.

Пример: печать n первых простых чисел.
Рассмотрим пример, использующий захват динамической памяти. Требуется ввести целое цисло n и напечатать n первых простых чисел. (Простое число - это число, у которого нет нетривиальных делителей.) Используем следующий алгоритм: последовательно проверяем все нечетные числа, начиная с тройки (двойку рассматриваем отдельно). Делим очередное число на все простые числа, найденные на предыдущих шагах алгоритма и не превосходящие квадратного корня из проверяемого числа. Если оно не делится ни на одно из этих простых чисел, то само является простым; оно печатается и добавляется в массив найденных простых.

Поскольку требуемое количество простых чисел n до начала работы программы неизвестно, невозможно создать массив для их хранения в статической памяти. Выход состоит в том, чтобы захватывать пространство под массив в динамической памяти уже после ввода числа n. Вот полный текст программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main() {
    int n;  // Требуемое количество простых чисел
    int k;  // Текущее количество найденных простых чисел
    int *a; // Указатель на массив найденных простых
    int p;  // Очередное проверяемое число
    int r;  // Целая часть квадратного корня из p
    int i;  // Индекс простого делителя
    bool prime; // Признак простоты

    printf("Введите число простых: ");
    scanf("%d", &n);
    if (n <= 0)   // Некорректное значение =>
        return 1; // завершаем работу с кодом ошибки

    // Захватываем память под массив простых чисел
    a = (int *) malloc(n * sizeof(int));

    a[0] = 2; k = 1;     // Добавляем двойку в массив
    printf("%d ", a[0]); // и печатаем ее

    p = 3;
    while (k < n) {
        // Проверяем число p на простоту
        r = (int)(               // Целая часть корня
            sqrt((double) p) + 0.001
        );
        i = 0;
        prime = true;
        while (i < k && a[i] <= r) {
            if (p % a[i] == 0) { // p делится на a[i]
```

```
                prime = false;   // => p не простое,
                break;           // выходим из цикла
            }
            ++i; // К следующему простому делителю
        }
        if (prime) {  // Если нашли простое число,
            a[k] = p; // то добавляем его в массив
            ++k;      // Увеличиваем число простых
            printf("%d ", p); // Печатаем простое число
            if (k % 5 == 0) { // Переход на новую строку
                printf("\n"); // после каждых пяти чисел
            }
        }

        p += 2; // К следующему нечетному числу
    }
    if (k % 5 != 0) {
        printf("\n"); // Перевести строку
    }
    // Освобождаем динамическую память
    free(a);
    return 0;
}
```

Пример работы данной программы:

```
Введите число простых: 50
2 3 5 7 11
13 17 19 23 29
31 37 41 43 47
53 59 61 67 71
73 79 83 89 97
101 103 107 109 113
127 131 137 139 149
151 157 163 167 173
179 181 191 193 197
199 211 223 227 229
```

Операторы new и delete языка C++.

В языке C++ для захвата и освобождения динамической памяти используются операторы new и delete. Они являются частью языка C++, в отличие от функций malloc и free, входящих в библиотеку стандартных функций Си.

Пусть T - некоторый тип языка Си или C++, p - указатель на объект типа T. Тогда для захвата памяти размером в один элемент типа T используется оператор new:

```
T *p;
p = new T;
```

Например, для захвата восьми байтов под вещественное число типа double используется фрагмент

```
double *p;
p = new double;
```

При использовании new, в отличие от malloc, не нужно приводить указатель от типа void* к нужному типу: оператор new возвращает указатель на тип, записанный после слова new.

Оператор new удобен еще и тем, что можно присвоить начальное значение объекту, созданному в динамической памяти (т.е. выполнить инициализацию объекта). Для этого начальное значение записывается в круглых скобках после имени типа, следующего за словом new. Например, в приведенной ниже строке захватывается память под вещественное число, которому присваивается начальное значение 1.5:

```
double *p = new double(1.5);
```

Этот фрагмент эквивалентен фрагменту

```
double *p = new double;
```

```
*p = 1.5;
```
С помощью оператора new можно захватывать память под массив элементов заданного типа. Для этого в квадратных скобках указывается длина захватываемого массива, которая может представляться любым целочисленным выражением. Например, в следующем фрагменте в динамической памяти захватывается область для хранения вещественной матрицы размера m*n:
```
double *a;
int m = 100, n = 101;
a = new double[m * n];
```
Такую форму оператора new иногда называют векторной.
Оператор delete освобождает память, захваченную ранее с помощью оператора new, например,
```
double *p = new double(1.5); // Захват и инициализация
. . .
delete p; // Освобождение памяти
```
Если память под массив была захвачена с помощью векторной формы оператора new, то для ее освобождения следует использовать векторную форму оператора delete, в которой после слова delete записываются пустые квадратные скобки:
```
double *a = new double[100]; // Захватываем массив
. . .
delete[] a; // Освобождаем массив
```

ЗАДАНИЕ
Модифицировать программу нахождения простых чисел, приведенную выше, с использованием операторов new и delete.

Структуры данных

Структуры

Структура — это конструкция, которая позволяет объединить несколько переменных с разными типами и именами в один составной объект. Она позволяет строить новые типы данных языка
Описание структуры выглядит следующим образом:
```
struct имя_структуры {
    описание полей структуры
};
```
имя_структуры — это любое имя, соответствующее синтаксису языка;
описания полей структуры — любая последовательность описаний переменных, имена и типы этих переменных могут быть произвольными. Эти переменные называются полями структуры.
Заканчивается описание структуры закрывающей фигурной скобкой. За закрывающей фигурной скобкой в описании структуры обязательно следует точка с запятой, в отличие от конструкции составного оператора, не следует забывать об этом.
Рассмотрим пример: опишем вектор в трехмерном пространстве, который задается тремя вещественными координатами x, y, z:
```
struct R3Vector {
    double x;
    double y;
    double z;
};
```
Таким образом, вводится новый тип "struct R3Vector"; объект этого типа содержит внутри себя три вещественных поля с именами x, y, z. После того как структура определена, можно описывать переменные такого типа, при этом в качестве имени типа следует использовать выражение struct R3Vector. Например, в следующей строке описываются два вещественных вектора в трехмерном пространстве с именами u, v:
```
struct R3Vector u, v;
```

С объектами типа структура можно работать как с единым целым, например, копировать эти объекты целиком:

```
struct R3Vector u, v;
. . .
u = v;  // Копируем вектор как единое целое
```

В этом примере вектор v копируется в вектор u; копирование структур сводится к переписыванию области памяти. Сравнивать структуры нельзя:

```
struct R3Vector u, v;
. . .
if (u == v) { // Ошибка! Сравнивать структуры нельзя
    . . .
}
```

Имеется также возможность работать с полями структуры. Для этого используется операция точка ".": пусть s — объект типа структура, f — имя поля структуры. Тогда выражение

```
s.f
```

является полем f структуры s, с ним можно работать как с обычной переменной. Например, в следующем фрагменте в вектор w записывается векторное произведение векторов u и v трехмерного пространства: w = u × v.

```
struct R3Vector u, v, w;
. . .
// Вычисляем векторное произведение w = u * v
w.x = u.y * v.z - u.z * v.y;
w.y = (-u.x) * v.z + u.z * v.x;
w.z = u.x * v.y - u.y * v.x;
```

В приведенных примерах все поля структуры R3Vector имеют один и тот же тип double, однако это совершенно не обязательно. Полями структуры могут быть другие структуры, никаких ограничений нет. Пример: плоскость в трехмерном пространстве задается точкой и вектором нормали, ей соответствует структура R3Plane. Точке трехмерного пространства соответствует структура R3Point, которая определяется аналогично вектору. Полное описание всех трех структур:

```
struct R3Vector { // Вектор трехмерного пространства
    double x;
    double y;
    double z;
};
struct R3Point { // Точка трехмерного пространства
    double x;
    double y;
    double z;
};
struct R3Plane { // Плоскость в трехмерном пр-ве
    struct R3Point origin;  // точка в плоскости
    struct R3Vector normal; // нормаль к плоскости
};
```

Пусть plane — это объект типа плоскость. Для того, чтобы получить координату x точки плоскости, надо два раза применить операцию "точка" доступа к полю структуры:

```
plane.origin.x
```

Структуры и указатели

Указатели на структуры используются довольно часто. Указатель на структуру S описывается обычным образом, в качестве имени типа фигурирует struct S*. Например, в следующем фрагменте переменная p описана как указатель на структуру **S**:

```
struct S { . . . }; // Определение структуры S
struct S *p; // Описание указателя на структуру S
```

Описание структуры может содержать указатель на структуру того же типа в качестве одного из полей. Язык Си допускает использование указателей на структуры, определение которых еще не завершено. Например, рассмотрим структуру TreeNode (вершина дерева), которая используется при определении бинарного дерева. Она содержит указатели на родительский узел и на левого и правого сыновей, которые также имеют тип struct TreeNode:

```
struct TreeNode {   // Вершина дерева
    struct TreeNode *parent; // Указатель на отца,
    struct TreeNode *left;   //   на левого сына,
    struct TreeNode *right;  //   на правого сына
    int value;              // Значение в вершине
};
```

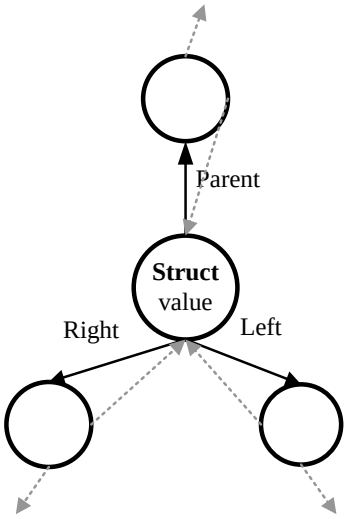


Рис. 3.1 Схема структуры TreeNode

Здесь при описании полей parent, left, right используется тип указатель на структуру TreeNode, определение которой еще не завершено, что допустимо в языке C++.

Для доступа к полям структуры через указатель на структуру служит операция стрелочка, которая обозначается двумя символами −> (минус и знак больше), их нужно рассматривать как одну неразрывную лексему (т.е. единый знак, единое слово). Пусть S — имя структуры, f — некоторое поле структуры S, p — указатель на структуру S. Тогда выражение p−>f обозначает поле f структуры S (само поле, а не указатель не него!). Это выражение можно записать, используя операцию звездочка (доступ к объекту через указатель), p−>f ~ (*p).f но, конечно, первый способ гораздо нагляднее. (Во втором случае круглые скобки вокруг выражения *p обязательны, поскольку приоритет операции точка выше, чем операции звездочка.)

Пример: рекурсивный обход дерева:

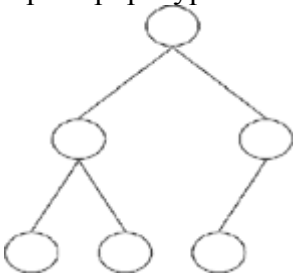


Рис. 3.2 Схема дерева

В качестве примера использования указателей на структуры приведем фрагмент программы, вычисляющий количество вершин бинарного дерева. Бинарным деревом

называется связный граф без циклов, у которого одна вершина отмечена как корневая, а все вершины упорядочены иерархически по длине пути от корня к вершине. У каждой вершины должно быть не больше двух сыновей, причем задан их порядок (левый и правый сыновья).

Вершина дерева описывается структурой TreeNode, которая рассматривалась выше. Если у вершины один из сыновей отсутствует, то соответствующий указатель содержит нулевой адрес.

Для подсчета числа вершин дерева используем функцию numNodes с прототипом

```
int numNodes(const struct TreeNode *root);
```

Ей передается константный указатель на корневую вершину дерева или поддерева. Функция возвращает суммарное число вершин дерева или поддерева. Эта функция легко реализуется с помощью рекурсии: достаточно подсчитать число вершин для каждого из двух поддеревьев, соответствующих левому и правому сыновьям корневой вершины, сложить их и прибавить к сумме единицу. Если левый или правый сын отсутствует, то соответствующее слагаемое равно нулю. Вот фрагмент программы, реализующий функцию numNodes.

```
// Описание структуры, представляющей вершину дерева
struct TreeNode {
    struct TreeNode *parent; // Указатель на отца,
    struct TreeNode *left;   //   на левого сына,
    struct TreeNode *right;  //   на правого сына
    void *value;             // Значение в вершине
};
// Рекурсивная реализация функции,
// вычисляющей число вершин дерева.
// Вход: указатель на корень поддерева
// Возвращаемое значение: число вершин поддерева
int numNodes(const struct TreeNode *root) {
    int num = 0;
    if (root == 0) { // Для нулевого указателя на корень
        return 0;    // возвращаем ноль
    }

    if (root->left != 0) {           // Есть левый сын =>
        num += numNodes(root->left); // вызываем функцию
    }                                // для левого сына

    if (root->right != 0) {          // Есть правый сын =>
        num += numNodes(root->right); // вызываем ф-цию
    }                                  // для правого сына

    return num + 1; // Возвращаем суммарное число вершин
}
```

Здесь неоднократно применялась операция стрелочка –> для доступа к полю структуры через указатель на нее.

При создании переменной типа структуры память под все элементы структуры выделяется последовательно для каждого элемента. Рассмотрим пример выделения памяти под структуру:

```
struct structA {
    char cA;
    char sA[2];
    float fA;};
```

При создании переменной структурного типа structA будет выделено 7 байтов. Элементы структуры будут размещены в памяти в следующем порядке:

char cA	char sA[2]	float fA

1	2	3	4	5	6	7

Простейшие структуры данных. Стек. Очередь.

Наиболее важными из простейших структур данных являются стек и очередь. Эти структуры встречаются в программировании буквально на каждом шагу, в самых разнообразных ситуациях.

Очередь

Очередь как структура данных понятна даже людям, не знакомым с программированием. Очередь содержит элементы, как бы выстроенные друг за другом в цепочку. У очереди есть начало и конец. Добавлять новые элементы можно только в конец очереди, забирать элементы можно только из начала. В отличие от обычной очереди, которую всегда можно при желании покинуть, из середины программистской очереди удалять элементы нельзя. Очередь можно представить в виде трубки. В один конец трубки можно добавлять шарики — элементы очереди, из другого конца они извлекаются. Элементы в середине очереди, т.е. шарики внутри трубки, недоступны. Конец трубки, в который добавляются шарики, соответствует концу очереди, конец, из которого они извлекаются — началу очереди. Таким образом, концы трубки не симметричны, шарики внутри трубки движутся только в одном направлении.

Рис. 3.3 Очередь

В принципе, можно было бы разрешить добавлять элементы в оба конца очереди и забирать их также из обоих концов. Такая структура данных в программировании тоже существует, ее название — "дек", от англ. Double Ended Queue, т.е. очередь с двумя концами. Дек применяется значительно реже, чем очередь.

Использование очереди в программировании почти соответствует ее роли в обычной жизни. Очередь практически всегда связана с обслуживанием запросов, в тех случаях, когда они не могут быть выполнены мгновенно. Очередь поддерживает также порядок обслуживания запросов. Рассмотрим, к примеру, что происходит, когда человек нажимает клавишу на клавиатуре компьютера. Тем самым человек просит компьютер выполнить некоторое действие. Например, если он просто печатает текст, то действие должно состоять в добавлении к тесту одного символа и может сопровождаться перерисовкой области экрана, прокруткой окна, переформатированием абзаца и т.п. Любая, даже самая простая, операционная система всегда в той или иной степени многозадачна. Это значит, что в момент нажатия клавиши операционная система может быть занята какой-либо другой работой. Тем не менее, операционная система ни в какой ситуации не имеет права проигнорировать нажатие на клавишу. Поэтому происходит прерывание работы компьютера, он запоминает свое состояние и переключается на обработку нажатия на клавишу. Такая обработка должна быть очень короткой, чтобы не нарушить выполнение других задач. Команда, отдаваемая нажатием на клавишу, просто добавляется в конец очереди запросов, ждущих своего выполнения. После этого прерывание заканчивается, компьютер восстанавливает свое состояние и продолжает работу, которая была прервана нажатием на клавишу. Запрос, поставленный в очередь, будет выполнен не сразу, а только когда наступит его черед.

Реализация очереди на базе массива. ***Как уже было сказано, программисту массив дан свыше, все остальные структуры данных нужно реализовывать на его основе. Конечно, такая реализация может быть многоэтапной, и не всегда массив выступает в качестве непосредственной базы реализации. В случае очереди наиболее популярны две реализации: непрерывная на базе массива и ссылочная реализация, или реализация на базе списка. При непрерывной реализации очереди в качестве базы выступает массив фиксированной длины N, таким образом, очередь ограничена и не может содержать более N элементов. Индексы элементов массива изменяются в***

пределах от 0 до N - 1. Кроме массива, реализация очереди хранит три простые переменные: индекс начала очереди, индекс конца очереди, число элементов очереди. Элементы очереди содержатся в отрезке массива от индекса начала до индекса конца.

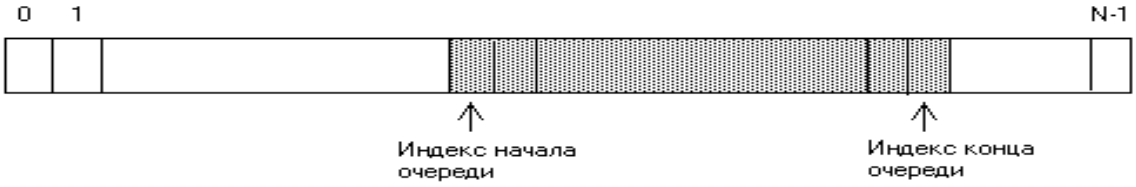


Рис. 3.4 Очередь на базе массива.

При добавлении нового элемента в конец очереди индекс конца сперва увеличивается на единицу, затем новый элемент записывается в ячейку массива с этим индексом. Аналогично, при извлечении элемента из начала очереди содержимое ячейки массива с индексом начала очереди запоминается в качестве результата операции, затем индекс начала очереди увеличивается на единицу. Как индекс начала очереди, так и индекс конца при работе двигаются слева направо.

Стек

Стек — самая популярная и, пожалуй, самая важная структура данных в программировании. Стек представляет собой запоминающее устройство, из которого элементы извлекаются в порядке, обратном их добавлению. Это как бы неправильная очередь, в которой первым обслуживают того, кто встал в нее последним. В программистской литературе общепринятыми являются аббревиатуры, обозначающие дисциплину работы очереди и стека. Дисциплина работы очереди обозначается FIFO, что означает первым пришел — первым уйдешь (First In First Out). Дисциплина работы стека обозначается LIFO, последним пришел — первым уйдешь (Last In First Out).

Стек можно представить в виде трубки с подпружиненным дном, расположенной вертикально. Верхний конец трубки открыт, в него можно добавлять, или, как говорят, заталкивать элементы. Общепринятые английские термины в этом плане очень красочны, операция добавления элемента в стек обозначается push, в переводе "затолкнуть, запихнуть". Новый добавляемый элемент проталкивает элементы, помещенные в стек ранее, на одну позицию вниз. При извлечении элементов из стека они как бы выталкиваются вверх, по-английски pop ("выстреливают").

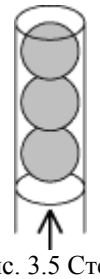


Рис. 3.5 Стек.

Примером стека может служить стог сена, стопка бумаг на столе, стопка тарелок и т.п. Отсюда произошло название стека, что по-английски означает стопка. Тарелки снимаются со стопки в порядке, обратном их добавлению. Доступна только верхняя тарелка, т.е. тарелка на вершине стека.

Стек применяется довольно часто, причем в самых разных ситуациях. Объединяет их следующая цель: нужно сохранить некоторую работу, которая еще не выполнена до конца, при необходимости переключения на другую задачу. Стек используется для временного сохранения состояния не выполненного до конца задания. После сохранения состояния компьютер переключается на другую задачу. По окончании

ее выполнения состояние отложенного задания восстанавливается из стека, и компьютер продолжает прерванную работу.

Реализация стека на базе массива. Реализация стека на базе массива является классикой программирования. Иногда даже само понятие стека не вполне корректно отождествляется с этой реализацией.

Базой реализации является массив размера N, таким образом, реализуется стек ограниченного размера, максимальная глубина которого не может превышать N. Индексы ячеек массива изменяются от 0 до N - 1. Элементы стека хранятся в массиве следующим образом: элемент на дне стека располагается в начале массива, т.е. в ячейке с индексом 0. Элемент, расположенный над самым нижним элементом стека, хранится в ячейке с индексом 1, и так далее. Вершина стека хранится где-то в середине массива. Индекс элемента на вершине стека хранится в специальной переменной, которую обычно называют указателем стека (по-английски Stack Pointer или просто SP).

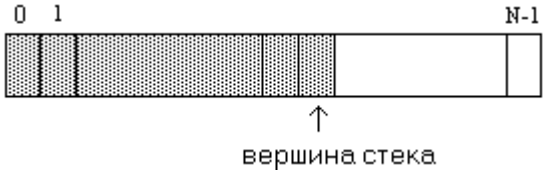


Рис. 3.6 Стек на баземассива.

Когда стек пуст, указатель стека содержит значение минус единица. При добавлении элемента указатель стека сначала увеличивается на единицу, затем в ячейку массива с индексом, содержащимся в указателе стека, записывается добавляемый элемент. При извлечении элемента из стека сперва содержимое ячейки массива с индексом, содержащимся в указателе стека, запоминается во временной переменной в качестве результата операции, затем указатель стека уменьшается на единицу.

В приведенной реализации стек растет в сторону увеличения индексов ячеек массива. Часто используется другой вариант реализации стека на базе вектора, когда дно стека помещается в последнюю ячейку массива, т.е. в ячейку с индексом N - 1. Элементы стека занимают непрерывный отрезок массива, начиная с ячейки, индекс которой хранится в указателе стека, и заканчивая последней ячейкой массива. В этом варианте стек растет в сторону уменьшения индексов. Если стек пуст, то указатель стека содержит значение N (которое на единицу больше, чем индекс последней ячейки массива).

Пример реализации стека: реализация включает два файла: "streal.h", в котором описывается интерфейс исполнителя "Стек", и "streal.cpp", реализующий функции работы со стеком. Слово real обозначает вещественное число. Используется вариант реализации стека на базе массива, описанный ранее: стек растет в сторону увеличения индексов массива. Пространство под массив элементов стека захватывается в динамической памяти в момент инициализации стека. Функции инициализации st_init передается размер массива, т.е. максимально возможное число элементов в стеке. Для завершения работы стека нужно вызвать функцию st_terminate, которая освобождает захваченную в st_init память. Ниже приведено содержимое файла "streal.h", описывающего интерфейс стека.

```
// Файл "streal.h"
// Стек вещественных чисел, интерфейс
//
#ifndef ST_REAL_H
#define ST_REAL_H

// Прототипы функций, реализующих предписания стека:

void st_init(int maxSize); // Начать работу (вх: цел
                           //     макс. размер стека)
void st_terminate();    // Закончить работу
void st_push(double x); // Добавить эл-т (вх: вещ x)
double st_pop();        // Взять элемент: вещ
double st_top();        // Вершина стека: вещ
int st_size();          // Текущий размер стека: цел
```

```
bool st_empty();         // Стек пуст? : лог
int st_maxSize();        // Макс. размер стека: цел
bool st_freeSpace();     // Есть свободное место? : лог
void st_clear();         // Удалить все элементы
double st_elementAt(int i); // Элемент стека на
                            //   глубине (вх: i): вещ
#endif
// Конец файла "streal.h"
```

Отметим, что директивы условной трансляции

```
#ifndef ST_REAL_H
#define ST_REAL_H
. . .
#endif
```

используются для предотвращения повторного включения h-файла: при первом включении файла определяется переменная препроцессора ST_REAL_H, а директива "#ifndef ST_REAL_H" подключает текст, только если эта переменная не определена. Такой трюк используется практически во всех h-файлах. Нужен он потому, что одни h-файлы могут подключать другие, и без этого механизма избежать повторного включения одного и того же файла трудно. Файл "streal.cpp" описывает общие статические переменные, над которыми работают функции, соответствующие предписаниям стека, и реализует эти функции.

```
// Файл "streal.cpp"
// Стек вещественных чисел, реализация
//
#include <stdlib.h>
#include <assert.h>
#include "streal.h" // Подключить описания функций стека

// Общие переменные для функций, реализующих
// предписания стека:
static double *elements = 0; // Указатель на массив эл-тов
                             //     стека в дин. памяти
static int max_size = 0;     // Размер массива
static int sp = (-1);        // Индекс вершины стека

// Предписания стека:

void st_init(int maxSize) { // Начать работу (вх:
                            //     макс. размер стека)
    assert(elements == 0);
    max_size = maxSize;
    elements = (double *) malloc(
        max_size * sizeof(double)
    );
    sp = (-1);
}

void st_terminate() { // Закончить работу
    if (elements != 0) {
        free(elements);
    }
}

void st_push(double x) { // Добавить эл-т (вх: вещ x)
    assert(              // утв:
        elements != 0 && //   стек начал работу и
        sp < max_size-1  //   есть своб. место
```

```
    );
    ++sp;
    elements[sp] = x;
}

double st_pop() { // Взять элемент: вещ
    assert(sp >= 0); // утв: стек не пуст
    --sp;            // элемент удаляется из стека
    return elements[sp + 1];
}

double st_top() { // Вершина стека: вещ
    assert(sp >= 0); // утв: стек не пуст
    return elements[sp];
}

int st_size() { // Текущий размер стека: цел
    return (sp + 1);
}

bool st_empty() { // Стек пуст? : лог
    return (sp < 0);
}

int st_maxSize() { // Макс. размер стека: цел
    return max_size;
}

bool st_freeSpace() { // Есть своб. место? : лог
    return (sp < max_size - 1);
}

void st_clear() { // Удалить все элементы
    sp = (-1);
}

double st_elementAt(int i) { // Элемент стека на
                             //   глубине (вх: i): вещ
    assert(                     // утв:
        elements != 0 &&        // стек начал работу и
        0 <= i && i < st_size() // 0 <= i < размер стека
    );
    return elements[sp - i];
}
// Конец файла "streal.cpp"
```

В реализации стека неоднократно использовалась функция assert. Фактическим аргументом функции является логическое выражение. Если оно истинно, то ничего не происходит; если ложно, то программа завершается аварийно, выдавая диагностику ошибки.

Стековый калькулятор и обратная польская запись формулы.

В 1920 г. польский математик Ян Лукашевич предложил способ записи арифметических формул, не использующий скобок. В привычной нам записи знак операции записывается между аргументами, например, сумма чисел 2 и 3 записывается как 2 + 3. Ян Лукашевич предложил две другие формы записи: префиксная форма, в которой знак операции записывается перед аргументами, и постфиксная форма, в которой знак операции записывается после аргументов. В префиксной форме сумма чисел 2 и 3 записывается как + 2 3, в постфиксной — как 2 3 +. В честь Яна Лукашевича эти формы записи называют прямой и обратной польской записью.

В польской записи скобки не нужны. Например, выражение

```
(2+3)*(15-7)
```

записывается в прямой польской записи как

```
* + 2 3 - 15 7,
```
в обратной польской записи — как
```
2 3 + 15 7 - *.
```
Если прямая польская запись не получила большого распространения, то обратная оказалась чрезвычайно полезной. Неформально преимущество обратной записи перед прямой польской записью или обычной записью можно объяснить тем, что гораздо удобнее выполнять некоторое действие, когда объекты, над которыми оно должно быть совершено, уже даны.

Обратная польская запись формулы позволяет вычислять выражение любой сложности, используя стек как запоминающее устройство для хранения промежуточных результатов.

Для вычисления выражения надо сначала преобразовать его в обратную польскую запись (при некотором навыке это легко сделать в уме). В приведенном выше примере выражение (2+3)*(15-7) преобразуется к
```
2 3 + 15 7 - *
```
Затем обратная польская запись просматривается последовательно слева направо. Если мы видим число, то просто вводим его в калькулятор, т.е. добавляем его в стек. Если мы видим знак операции, то нажимаем соответствующую клавишу калькулятора, выполняя таким образом операцию с числами на вершине стека.

Изобразим последовательные состояния стека калькулятора при вычислении по приведенной формуле. Сканируем слева направо ее обратную польскую запись:
```
2 3 + 15 7 - *
```
Стек вначале пуст. Последовательно добавляем числа 2 и 3 в стек.

```
|   | вводим число 2 ⟶ | 2 | вводим число 3 ⟶ | 3 |
                                               | 2 |
```
Далее читаем символ + и нажимаем на клавишу + калькулятора. Числа 2 и 3 извлекаются из стека, складываются, и результат помещается обратно в стек.

```
| 3 | выполняем сложение ⟶ | 5 |
| 2 |
```
Далее, в стек добавляются числа 15 и 7.

```
| 5 | вводим число 15 ⟶ | 15 | вводим число 7 | 7  |
                        | 5  |                | 15 |
                                                | 5  |
```
Читаем символ - и нажимаем на клавишу - калькулятора. Со стека при этом снимаются два верхних числа 7 и 15 и выполняется операция вычитания. Причем уменьшаемым является то число, которое было введено раньше, а вычитаемым — число, введенное позже. Иначе говоря, при выполнении некоммутативных операций, таких как вычитание или деление, правым аргументом является число на вершине стека, левым — число, находящееся под вершиной стека.

```
| 7  | выполняем вычитание ⟶ | 8 |
| 15 |                       | 5 |
| 5  |
```
Наконец, читаем символ * и нажимаем на клавишу * калькулятора. Калькулятор выполняет умножение, со стека снимаются два числа, перемножаются, результат помещается обратно в стек.

```
| 8 | выполняем умножение ⟶ | 40 |
| 5 |
```
Число 40 является результатом вычисления выражения. Оно находится на вершине стека и высвечивается на дисплее стекового калькулятора.

ЗАДАНИЕ НА ДОМ

Написать программу, реализующую калькулятор. В проект должны входить три файла: "streal.h", "streal.cpp" и "stcalc.cpp". Первые два файла реализуют стек вещественных чисел, эта реализация уже рассматривалась выше. Файл "stcalc.cpp" реализует стековый

калькулятор на базе стека, его и нужно добавить к проекту. Пользователь вводит строку из чисел и арифметических операций – программа должна распарсить строку, записать числа в стек, а при наличии арифметической операции выполнить ее и поместить результат обратно в стек. Пользователь может ввести число с клавиатуры, это число просто добавляется в стек. При вводе одного из четырех знаков арифметических операций +, -, *, / программа извлекает из стека два числа, выполняет указанное арифметическое действие над ними и помещает результат обратно в стек. Значение результата отображается также на дисплее. Кроме арифметических операций, пользователь может ввести название одной из стандартных функций: sin, cos, exp, log (натуральный логарифм). При этом программа извлекает из стека аргумент функции, вычисляет значение функции и помещает его обратно в стек. При желании список стандартных функций и возможных операций можно расширить. Каждую команду стекового калькулятора нужно реализовать в виде отдельной функции. Например, вычитание реализуется с помощью функции onSub():

```
static void onSub() {
    double y, x;
    if (st_size() < 2) {
        printf("Stack depth < 2.\n");
        return;
    }
    y = st_pop();
    x = st_pop();
    st_push(x - y);
    display();
}
```

Двоичные файлы. Формат BMP.

Так сложилось, что файлы делятся на текстовые и нетекстовые (последние иногда называют двоичными, или бинарными), файл, содержащий программу на Си, — текстовый; файл, который вы можете создать, используя, например, любой текстовый редактор (FAR или кому что нравится), — тоже текстовый. А вот файл, содержащий, например, рисунок в формате bmp, JPEG, да и в любом другом графическом формате — двоичный. Текстовые файлы отличаются от двоичных двумя особенностями: во-первых, они делятся на строки, каждая из которых заканчивается "переводом строки", состоящим из двух символов с кодами 0x0D 0x0A; во-вторых, текстовые файлы заканчиваются "признаком конца файла" — символом с кодом 0x1A (точнее, должны заканчиваться, это условие соблюдается не всегда).

При чтении текстового файла (потока) функции Си преобразуют "признак конца строки", т.е. последовательность символов 0x0D 0x0A, в один символ 0x0A (' \n'), а "признак конца файла (потока)" — в значение EOF *(End Of File).* Константа EOF определена в заголовочном файле stdio.h и обычно равна -1.

При чтении двоичных потоков никаких преобразований не производится. То, с каким потоком мы собираемся работать — текстовым или двоичным, указывается при его открытии. Один из способов открытия потока — использование функции fopen. Тип потока указывается в строке параметров, которая является вторым аргументом этой функции. Пример: f=fopen ("test.ext", " rt"} ; — открытие текстового файла test.ext для

чтения и связывание его с файловой переменной f. На то, что файл открывается как текстовый, указывает буква "t" в строке "rt". Чтобы открыть этот файл как бинарный, надо использовать букву "b": f=fopen ("test.ext", "rb").

В BMP-файле имеются две части — так называемый заголовок, в котором хранится информация о картинке (соответствующая структура приведена ниже, правда, приходится отметить, что лишь небольшое количество полей заголовка нами будет использовано), и собственно изображение. Изображение хранится по точкам, в построчной развертке, начиная с *нижней* строки картинки.

По решению разработчиков формат Bmp-файла не привязан к конкретной аппаратной платформе. Этот файл состоит из четырех частей: заголовка, информационного заголовка, таблицы цветов (палитры) и данных изображения. Если в файле хранится изображение с глубиной цвета 24 бита (16 млн. цветов), то таблица цветов может отсутствовать, однако в 256-цветном случае она есть. Структура каждой из частей файла, хранящего 256-цветное изображение, дана в таблице ниже.

Заголовок файла начинается с сигнатуры «BM», а затем идет длина файла, выраженная в байтах. Следующие 4 байта зарезервированы для дальнейших расширений формата, а заканчивается этот заголовок смещением от начала файла до записанных в нем данных изображения. При 256 цветах это смещение составляет 1078 — именно столько и пришлось пропустить в нашей прошлой программе, чтобы добраться до данных.

Информационный заголовок начинается с собственной длины (она может изменяться, но для 256-цветного файла составляет 40 байт) и содержит размеры изображения, разрешение, характеристики представления цвета и другие параметры.

Ширина и высота изображения задаются в точках растра и пояснений, пожалуй, не требуют.

Количество плоскостей могло применяться в файлах, имеющих небольшую глубину цвета. При числе цветов 256 и больше оно всегда равно 1, поэтому сейчас это поле уже можно считать устаревшим, но для совместимости оно сохраняется.

Глубина цвета считается важнейшей характеристикой способа представления цвета в файле и измеряется в битах на точку. В данном случае она равна 8.

Компрессия. В Bmp-файлах обычно не используется, но поле в заголовке для нее предусмотрено. Обычно она равна 0, и это означает, что изображение не сжато. В дальнейшем будем использовать только такие файлы.

Размер изображения — количество байт памяти, требующихся для хранения этого изображения, не считая данных палитры.

Горизонтальное и вертикальное разрешения измеряются в точках растра на метр. Они особенно важны для сохранения масштаба отсканированных картинок. Изображения, созданные с помощью графических редакторов, как правило, имеют в этих полях нули.

Число цветов позволяет сократить размер таблицы палитры, если в изображении реально присутствует меньше цветов, чем это допускает выбранная глубина цвета. Однако на практике такие файлы почти не встречаются. Если число цветов принимает значение, максимально допустимое глубиной цвета, например 256 цветов при 8 битах, поле обнуляют.

Число основных цветов — идет с начала палитры, и его желательно выводить без искажений. Данное поле бывает важно тогда, когда максимальное число цветов дисплея было меньше, чем в палитре Bmp-файла. При разработке формата, очевидно, принималось, что наиболее часто встречающиеся цвета будут располагаться в начале таблицы. Сейчас этого требования практически не придерживаются, т. е. цвета не упорядочиваются по частоте, с которой они встречаются в файле. Это очень важно, поскольку палитры двух разных файлов, даже составленных из одних и тех же цветов, содержали бы их (цвета) в разном порядке, что могло существенно осложнить одновременный вывод таких изображений на экран.

За информационным заголовком следует таблица цветов, представляющая собой массив из 256 (по числу цветов) 4-байтовых полей. Каждое поле соответствует своему цвету в палитре, а три байта из четырех — компонентам синей, зеленой и красной составляющих для этого цвета. Последний, самый старший байт каждого поля зарезервирован и равен 0.

После таблицы цветов находятся данные изображения, которое по строкам растра записано снизу вверх, а внутри строки — слева направо. Так как на некоторых платформах невозможно считать единицу данных, которая меньше 4 байт, длина каждой строки выровнена на границу в 4 байта, т. е. при длине строки, некратной четырем, она дополняется нулями. Это обстоятельство обязательно надо учитывать при считывании файла, хотя, возможно, лучше заранее позаботиться, чтобы горизонтальные размеры всех изображений были кратны 4.

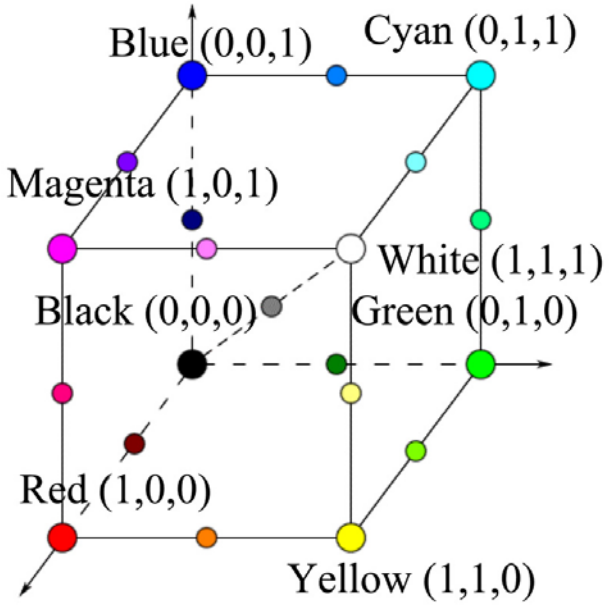
Структура Bmp-файла

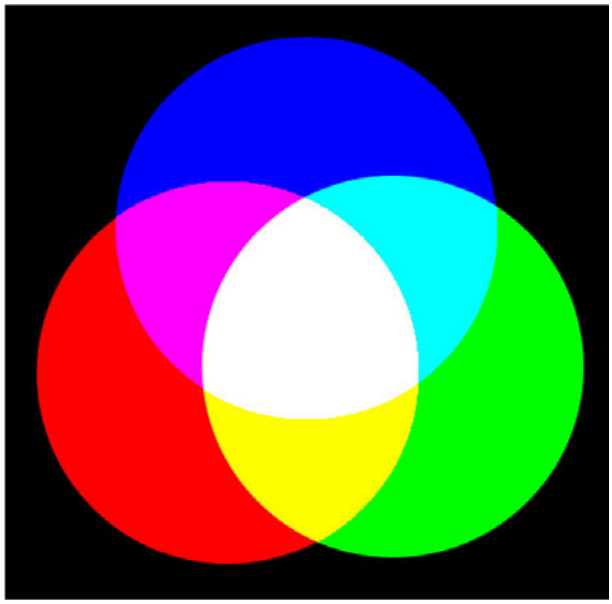
Имя	Длина	Смещение	Описание
Заголовок файла (BitMapFileHeader)			
Type	2	0	Сигнатура "BM"
Size	4	2	Размер файла
Reserved 1	2	6	Зарезервировано
Reserved 2	2	8	Зарезервировано
OffsetBits	4	10	Смещение изображения от начала файла
Информационный заголовок (BitMapInfoHeader)			
Size	4	14	Длина заголовка
Width	4	18	Ширина изображения, точки
Height	4	22	Высота изображения, точки
Planes	2	26	Число плоскостей
BitCount	2	28	Глубина цвета, бит на точку
Compression	4	30	Тип компрессии (0 - несжатое изображение)
SizeImage	4	34	Размер изображения, байт
XpelsPerMeter	4	38	Горизонтальное разрешение, точки на метр
YpelsPerMeter	4	42	Вертикальное разрешение, точки на метр
ColorsUsed	4	46	Число используемых цветов (0 - максимально возможное для данной глубины цвета)
ColorsImportant	4	50	Число основных цветов
Таблица цветов (палитра) (ColorTable)			
ColorTable	1024	54	256 элементов по 4 байта
Данные изображения (BitMap Array)			
Image	Size	1078	Изображение, записанное по строкам слева направо и снизу вверх

Представление изображений. RGB-модель.

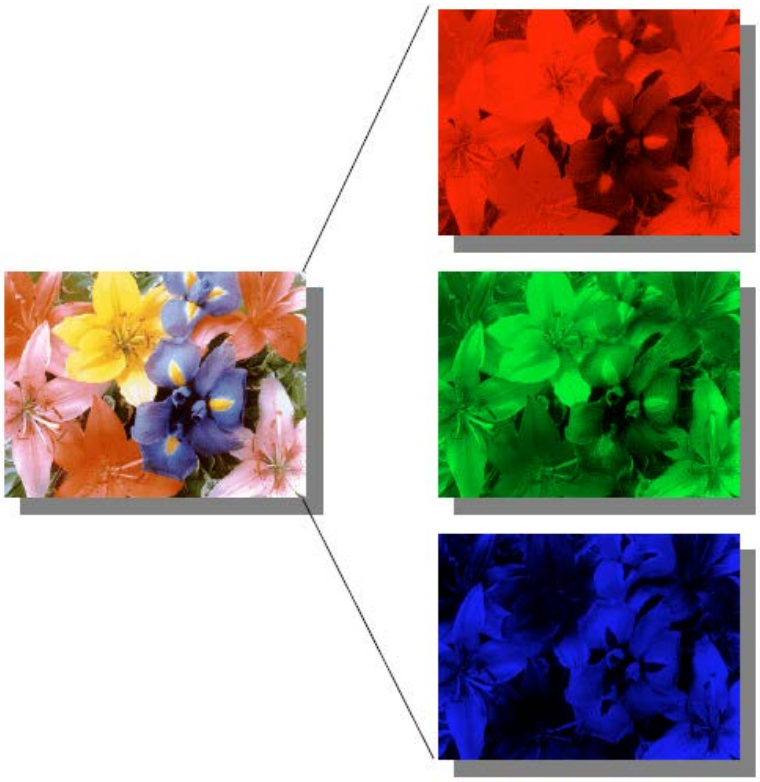
Компьютер может работать только с цифровыми данными. Для того чтобы обработать с помощью компьютера изображение, его нужно выразить в цифровом виде. Существуют два основных способа цифрового представления изображений: растровый и векторный. Для векторной графики характерно разбиение изображения на ряд графических примитивов – точки, прямые, ломаные, дуги, полигоны. Таким образом, появляется возможность хранить не все точки изображения, а координаты узлов примитивов и их свойства (цвет, связь с другими узлами и т. д.). Итак, под растровым (bitmap, raster) понимают способ представления изображения в виде совокупности отдельных точек (пикселей) различных цветов или оттенков. Это наиболее простой способ представления изображения, ибо таким образом видит наш глаз. Все изображения можно подразделить на две группы с палитрой и без нее. У изображений с палитрой в пикселе (pixel - picture element) хранится число - индекс в некотором одномерном векторе цветов называемом палитрой. Чаще всего встречаются палитры 16 и 256 цветов.

Цвета образуются в природе различным образом. С одной стороны, источники света (солнце, лампочки, экраны компьютеров и телевизоров) излучают свет различных длин волн, воспринимаемый глазом как цветной свет. Попадая на поверхности несветящихся предметов, свет частично поглощается, а частично отражается. Отраженное излучение воспринимается глазом как окраска предметов. Таким образом, цвет объекта возникает в результате излучения или отражения. Описание цвета объекта в первом случае отличается от второго, т. е. применяются разные модели цвета. Модель RGB описывает излучаемые цвета и основана на трех базовых цветах — Red (Красный), Green (Зеленый), Blue (Синий). Остальные цвета образуются при смешивании этих трех основных. При сложении (смешении) двух лучей основных цветов результат светлее составляющих. Цвета этого типа называются аддитивными.





Из смешения красного и зеленого получается желтый, зеленого и синего — голубой, синий и красный дают пурпурный. Если смешиваются все три цвета, в результате образуется белый. Смешав три базовых цвета в разных пропорциях (с разными яркостями), можно получить все многообразие оттенков. Если говорить о растровом изображении в модели RGB**,** то каждый его пиксел представляется яркостями трех базовых цветов: красного, зеленого и синего. Как уже говорилось выше, яркости пикселов хранятся в каналах. Таким образом, для RGB-изображения требуется три канала.



Цветовой канал — это полутоновое изображение, отражающее распределение яркостей соответствующего базового цвета. Если документ имеет модель Grayscale, то содержимое единственного канала и образует изображение. Редактирование канала и редактирование изображения в этих случаях одно и то же.

Если модель документа — RGB, то изображения в красном, зеленом и синем канале, накладываясь друг на друга, образуют цветную картинку. При этом, заметьте, цвета смешиваются аддитивно, как лучи света. Это значит, что при наложении результат осветляется. Чем светлее канал, тем больше базового цвета содержится в изображении.

ЗАДАНИЯ:

1. Разделить произвольное (нарисованное или считанное из BMP файла) изображение по каналам RGB. Продемонстрировать исходное и полученные изображения.
2. Преобразовать произвольное (нарисованное или считанное из BMP файла) цветное изображение в оттенки серого. Продемонстрировать исходное и полученные изображение.

Алгоритм сжатия изображений без потерь RLE.

Типичное изображение, полученное цифровой фотокамерой, имеет разрешение порядка 3000×2000, т.е. около 6 мегапикселей; для передачи цвета обычно используется 24 бита на пиксель. Таким образом, объем исходных данных составляет порядка 17 мегабайт. Для профессиональных устройств ввода изображений размер получаемого растра может быть значительно больше, а глубина цвета - достигать 48 бит на пиксель. Соответственно, размер одного изображения может быть больше 200 мегабайт. Поэтому весьма актуальными являются алгоритмы сжатия изображений, или, иными словами, алгоритмы, которые позволяют уменьшить объем данных, представляющих изображение.

Существуют два основных класса алгоритмов:

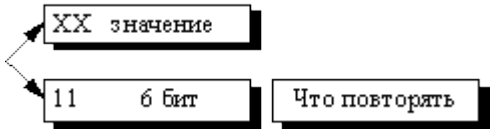
1. A называется алгоритмом сжатия без потерь (англ. lossless compression), если существует алгоритм A^{-1} (обратный к A) такой, что для любого изображения I $A(I) = I_1$ и $A^{-1}(I_1) = I$. Изображение I задано как множество значений атрибутов пикселей; после применения к I алгоритма A получаем набор данных I_1. Сжатие без потерь применяется в таких графических форматах представления изображений, как: GIF, PCX, PNG, TGA, TIFF, множество собственных форматов от производителей цифровых фотокамер, и т.д.);
2. A называется алгоритмом сжатия с потерями (англ. lossy compression), если он не обеспечивает возможность точного восстановления исходного изображения. Парный к A алгоритм, обеспечивающий примерное восстановление, будем обозначать как A^*: для изображения I $A(I) = I_1$, $A^*(I_1) = I_2$ и при этом полученное восстановленное изображение I_2 не обязательно точно совпадает с I. Пара A, A^* подбирается так, чтобы обеспечить большие коэффициенты сжатия и тем не менее сохранить визуальное качество, т.е. добиться минимальной разницы в восприятии между I и I_2. Сжатие с потерями применяется в следующих графических форматах: JPEG, JPEG2000 и т.д.

Первый вариант алгоритма RLE

Данный алгоритм необычайно прост в реализации. Групповое кодирование — от английского Run Length Encoding (RLE) — один из самых старых и самых простых алгоритмов архивации графики. Изображение в вытягивается в цепочку байт по строкам растра. Само сжатие в RLE происходит за счет того, что в исходном изображении встречаются цепочки одинаковых байт. Замена их на пары <счетчик повторений, значение> уменьшает избыточность данных. Рассмотрим, например, следующую строку: `abbbbzzzzddaaaaaffffuu` после применения группового сжатия эта последовательность будет преобразована в: `1a4b4z2d5a4f2u.` В данном алгоритме признаком счетчика (counter) служат единицы в двух верхних битах считанного байта. Соответственно оставшиеся 6 бит расходуются на счетчик, который может принимать значения от 1 до 64. Строку из 64 повторяющихся байтов мы превращаем в два байта, т.е. сожмем в 32 раза. Если же кодируемая последовательность содержит единичный байт, значение которого больше 191, то он так же кодируется парой <счетчик, значение>.

Алгоритм декомпрессии при этом выглядит так: Считываем счетчик

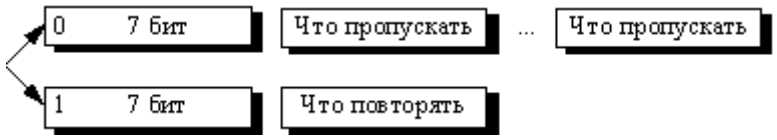
```
Initialization(...);
do {
    byte = ImageFile.ReadNextByte();
    if(является счетчиком(byte)) {
        counter = Low6bits(byte)+1;
        value = ImageFile.ReadNextByte();
        for(i=1 to counter)
            DecompressedFile.WriteByte(value)
        }
    else {
        DecompressedFile.WriteByte(byte)
} while(ImageFile.EOF());
```



Алгоритм рассчитан на деловую графику — изображения с большими областями повторяющегося цвета. Ситуация, когда файл увеличивается, для этого простого алгоритма не так уж редка. Ее можно легко получить, применяя групповое кодирование к обработанным цветным фотографиям. Для того, чтобы увеличить изображение в два раза, его надо применить к изображению, в котором значения всех пикселов больше двоичного 11000000 и подряд попарно не повторяются. Данный алгоритм реализован в формате PCX.

Второй вариант алгоритма RLE

Второй вариант этого алгоритма имеет больший максимальный коэффициент архивации и меньше увеличивает в размерах исходный файл. Признаком повтора в данном алгоритме является единица в старшем разряде соответствующего байта:



Алгоритм декомпрессии для него выглядит так: считываем счетчик повторяемых или оригинальных байт. Если первый бит счетчика установлен в 1, значит следующий байт следует повторить n раз, n={0;127}. Как извлечь младшие 7 разрядов из байта см. в описании лабораторной работы 1 (побитовые логические операции). Если первый бит счетчика установлен в 0, значит следующие n байт (n={0;127}) следует записать по одному разу.

```
Initialization(...);
do {
    byte = ImageFile.ReadNextByte();
    counter = Low7bits(byte)+1;
    if(если признак повтора(byte)) {
        value = ImageFile.ReadNextByte();
        for (i=1 to counter)
        CompressedFile.WriteByte(value)
    }
    else {
    for(i=1 to counter){
        value = ImageFile.ReadNextByte();
        CompressedFile.WriteByte(value)
    }
    CompressedFile.WriteByte(byte)
} while(ImageFile.EOF());
```

Как можно легко подсчитать, в лучшем случае этот алгоритм сжимает файл в 64 раза (а не в 32 раза, как в предыдущем варианте), в худшем увеличивает на 1/128. Средние показатели степени компрессии данного алгоритма находятся на уровне показателей первого варианта.

Похожие схемы компрессии использованы в качестве одного из алгоритмов, поддерживаемых форматом TIFF, а также в формате TGA.

ЗАДАНИЯ:

1. Изобразите блок-схему енкодера первого варианта алгоритма RLE.
2. Изобразите блок-схему декодера первого варианта алгоритма RLE.
3. Изобразите блок-схему енкодера второго варианта алгоритма RLE.
4. Изобразите блок-схему декодера второго варианта алгоритма RLE.

Характеристики алгоритма RLE:

Коэффициенты компрессии:
первый вариант: 32, 2, 0,5; (лучший, средний, худший коэффициенты);
второй вариант: 64, 3, 128/129; (лучший, средний, худший коэффициенты).

Класс изображений: ориентирован алгоритм на изображения с небольшим количеством цветов: деловую и научную графику.

Симметричность: примерно единица.

Характерные особенности: К положительным сторонам алгоритма, пожалуй, можно отнести только то, что он не требует дополнительной памяти при архивации и разархивации, а также быстро работает. Интересная особенность группового кодирования состоит в том, что степень архивации для некоторых изображений может быть существенно повышена всего лишь за счет изменения порядка цветов в палитре изображения.

ЗАДАНИЯ НА ДОМ:

1. Реализовать RLE encoder/decoder по первому алгоритму.
2. Реализовать RLE encoder/decoder по второму алгоритму.

Программа должна предоставлять пользователю возможность: загрузить произвольный bmp-файл, сжать его согласно варианту задания, загрузить сжатый файл, декодировать сжатый файл.

При сдаче лабораторной предоставить два файла, один из которых демонстрирует эффективность работы алгоритма, а второй – наоборот, показывает, увеличение объема сжатого файла по сравнению с исходным.

Понятие сокета

Сокет (socket) - это конечная точка сетевых коммуникаций. Он является чем-то вроде "портала", через которое можно отправлять байты во внешний мир. Приложение просто пишет данные в сокет; их дальнейшая буферизация, отправка и транспортировка осуществляется используемым стеком протоколов и сетевой аппаратурой. Чтение данных из сокета происходит аналогичным образом.В программе сокет идентифицируется дескриптором - это просто переменная типа int. Программа получает дескриптор от операционной системы при создании сокета, а затем передаёт его сервисам socket API для указания сокета, над которым необходимо выполнить то или иное действие.

Атрибуты сокета

С каждым сокет связываются три атрибута: *домен, тип и протокол*. Эти атрибуты задаются при создании сокета и остаются неизменными на протяжении всего времени его существования. Для создания сокета используется функция socket, имеющая следующий прототип.

```
int socket(int domain, int type, int protocol);
```

Домен определяет пространство адресов, в котором располагается сокет, и множество протоколов, которые используются для передачи данных. Чаще других используются домены Unix и Internet, задаваемые константами AF_UNIX и AF_INET соответственно (префикс AF означает "address family" - "семейство адресов"). При задании AF_UNIX для передачи данных используется файловая система ввода/вывода Unix. В этом случае сокеты используются для межпроцессного взаимодействия на одном компьютере и не годятся для работы по сети. Константа AF_INET соответствует Internet-домену. Сокеты, размещённые в этом домене, могут использоваться для работы в любой IP-сети. Существуют и другие домены (AF_IPX для протоколов Novell, AF_INET6 для новой модификации протокола IP - IPv6 и т. д.).

Тип сокета определяет способ передачи данных по сети. Чаще других применяются:

SOCK_STREAM. Передача потока данных с предварительной установкой соединения. Обеспечивается надёжный канал передачи данных, при котором фрагменты отправленного блока не теряются, не переупорядочиваются и не дублируются. Совместное использование с параметром AF_INET связывает сокет с протоколом TCP.

SOCK_DGRAM. Передача данных в виде отдельных сообщений (датаграмм). Предварительная установка соединения не требуется. Обмен данными происходит быстрее, но является ненадёжным: сообщения могут теряться в пути, дублироваться и переупорядочиваться. Допускается передача сообщения нескольким получателям (multicasting) и широковещательная передача (broadcasting). Совместное использование с параметром AF_INET связывает сокет с протоколом UDP

SOCK_RAW. Этот тип присваивается низкоуровневым (т. н. "сырым") сокетам. Их отличие от обычных сокетов состоит в том, что с их помощью программа может взять на себя формирование некоторых заголовков, добавляемых к сообщению.

Обратите внимание, что не все домены допускают задание произвольного типа сокета. Например, совместно с доменом Unix используется только тип SOCK_STREAM. С другой стороны, для Internet-домена можно задавать любой из перечисленных типов. В этом случае для реализации SOCK_STREAM используется протокол TCP, для реализации SOCK_DGRAM - протокол UDP, а тип SOCK_RAW используется для низкоуровневой работы с протоколами IP, ICMP и т. д.

Наконец, последний атрибут определяет *протокол*, используемый для передачи данных. Как мы только что видели, часто протокол однозначно определяется по домену и типу сокета. В этом случае в качестве третьего параметра функции socket можно передать 0, что соответствует протоколу по умолчанию. Тем не менее, иногда (например, при работе с низкоуровневыми сокетами) требуется задать протокол явно.

Адреса

Прежде чем передавать данные через сокет, его необходимо связать с адресом в выбранном домене (эту процедуру называют именованием сокета). Иногда связывание осуществляется неявно (внутри функций connect и accept), но выполнять его необходимо во всех случаях. Вид адреса зависит от выбранного вами домена. В Unix-домене это текстовая строка - имя файла, через который происходит обмен данными. В Internet-домене адрес задаётся комбинацией IP-адреса и 16-битного номера порта. IP-адрес определяет хост в сети, а порт - конкретный сокет на этом хосте. Протоколы TCP и UDP используют различные пространства портов.

Для явного связывания сокета с некоторым адресом используется функция bind. Её прототип имеет вид:

```
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

В качестве первого параметра передаётся дескриптор сокета, который мы хотим привязать к заданному адресу. Второй параметр, addr, содержит указатель на структуру с адресом, а третий - длину этой структуры. Посмотрим, что она собой представляет.

```
struct sockaddr {
    unsigned short    sa_family;    // Семейство адресов, AF_xxx
    char              sa_data[14];  // 14 байтов для хранения адреса
};
```

Поле sa_family содержит идентификатор домена, тот же, что и первый параметр функции socket. В зависимости от значения этого поля по-разному интерпретируется содержимое массива sa_data. Разумеется,

работать с этим массивом напрямую не очень удобно, поэтому вы можете использовать вместо sockaddr одну из альтернативных структур вида sockaddr_XX (XX - суффикс, обозначающий домен: "un" - Unix, "in" - Internet и т. д.). При передаче в функцию bind указатель на эту структуру приводится к указателю на sockaddr. Рассмотрим для примера структуру sockaddr_in.

```
struct sockaddr_in {
    short int          sin_family;  // Семейство адресов
    unsigned short int sin_port;    // Номер порта
    struct in_addr     sin_addr;    // IP-адрес
    unsigned char          sin_zero[8]; // "Дополнение" до размера структуры
sockaddr
};
```

Здесь поле sin_family соответствует полю sa_family в sockaddr, в sin_port записывается номер порта, а в sin_addr - IP-адрес хоста. Поле sin_addr само является структурой, которая имеет вид:

```
struct in_addr {
    unsigned long s_addr;
};
```

Зачем понадобилось заключать всего одно поле в структуру? Дело в том, что раньше in_addr представляла собой объединение (union), содержащее гораздо большее число полей. Сейчас, когда в ней осталось всего одно поле, она продолжает использоваться для обратной совместимости.

И ещё одно важное замечание. Существует два порядка хранения байтов в слове и двойном слове. Один из них называется порядком хоста (host byte order), другой - сетевым порядком (network byte order) хранения байтов. При указании IP-адреса и номера порта необходимо преобразовать число из порядка хоста в сетевой. Для этого используются функции htons (Host TO Network Short) и htonl (Host TO Network Long). Обратное преобразование выполняют функции ntohs и ntohl.

Установка соединения (клиент)(только TCP/IP)

На стороне клиента для установления соединения используется функция connect, которая имеет следующий прототип.

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Здесь sockfd - сокет, который будет использоваться для обмена данными с сервером, serv_addr содержит указатель на структуру с адресом сервера, а addrlen - длину этой структуры. Обычно сокет не требуется предварительно привязывать к локальному адресу, так как функция connect сделает это за вас, подобрав подходящий свободный порт. Вы можете принудительно назначить клиентскому сокету некоторый номер порта, используя bind перед вызовом connect. Однако проще и надёжнее предоставить системе выбрать порт за вас.

Обмен данными

После того как соединение установлено, можно начинать обмен данными. Для этого используются функции send и recv. Функция send используется для отправки данных и имеет следующий прототип.

```
int send(int sockfd, const void *msg, int len, int flags);
```

Здесь sockfd - это, как всегда, дескриптор сокета, через который мы отправляем данные, msg - указатель на буфер с данными, len - длина буфера в байтах, а flags - набор битовых флагов, управляющих работой функции (если флаги не используются, передайте функции 0). Вот некоторые из них (полный список можно найти в документации):

MSG_OOB. Предписывает отправить данные как срочные (out of band data, OOB). Концепция срочных данных позволяет иметь два параллельных канала данных в одном соединении. Иногда это бывает удобно. Например, Telnet использует срочные данные для передачи команд типа Ctrl+C. В настоящее время использовать их не рекомендуется из-за проблем с совместимостью (существует два разных стандарта их использования, описанные в RFC793 и RFC1122). Безопаснее просто создать для срочных данных отдельное соединение.

MSG_DONTROUTE. Запрещает маршрутизацию пакетов. Нижележащие транспортные слои могут проигнорировать этот флаг.

Функция send возвращает число байтов, которое на самом деле было отправлено (или -1 в случае ошибки). Это число может быть меньше указанного размера буфера. Если вы хотите отправить весь буфер целиком, вам придётся написать свою функцию и вызывать в ней send, пока все данные не будут отправлены.

Для чтения данных из сокета используется функция recv.

```
int recv(int sockfd, void *buf, int len, int flags);
```

В целом её использование аналогично send. Она точно так же принимает дескриптор сокета, указатель на буфер и набор флагов. Флаг MSG_OOB используется для приёма срочных данных, а MSG_PEEK позволяет "подсмотреть" данные, полученные от удалённого хоста, не удаляя их из системного буфера (это означает, что при следующем обращении к recv вы получите те же самые данные). Полный список флагов можно найти в документации. По аналогии с send функция recv возвращает количество прочитанных байтов, которое может быть меньше размера буфера.

Обмен датаграммами

Как уже говорилось, датаграммы используются в программах довольно редко. В большинстве случаев надёжность передачи критична для приложения, и вместо изобретения собственного надёжного протокола поверх UDP программисты предпочитают использовать TCP. Тем не менее, иногда датаграммы оказываются полезны. Например, их удобно использовать при транслировании звука или видео по сети в реальном времени, особенно при широковещательном транслировании.

Поскольку для обмена датаграммами не нужно устанавливать соединение, использовать их гораздо проще. Создав сокет с помощью socket и bind, вы можете тут же использовать его для отправки или получения данных. Для этого вам понадобятся функции sendto и recvfrom.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
                const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
                  struct sockaddr *from, int *fromlen);
```

Функция sendto очень похожа на send. Два дополнительных параметра to и tolen используются для указания адреса получателя. Для задания адреса используется структура sockaddr, как и в случае с функцией connect. Функция recvfrom работает аналогично recv. Получив очередное сообщение, она записывает его адрес в структуру, на которую ссылается from, а записанное количество байт - в переменную, адресуемую указателем fromlen. Как мы знаем, аналогичным образом работает функция accept.

Некоторую путаницу вносят *присоединённые датаграммные сокеты* (connected datagram sockets). Дело в том, что для сокета с типом SOCK_DGRAM тоже можно вызвать функцию connect, а затем использовать send и recv для обмена данными. Нужно понимать, что никакого соединения при этом не устанавливается. Операционная система просто запоминает адрес, который вы передали функции connect, а затем использует его при отправке данных. Обратите внимание, что присоединённый сокет может получать данные только от сокета, с которым он соединён.

Закрытие сокета

Закончив обмен данными, закройте сокет с помощью функции close (под Windows closesocket).Это приведёт к разрыву соединения.

```
closesocket(tcp_socket);
close(tcp_socket);
```

Особенности работы под Windows

Для инициализации подсистемы Сокетов нужно вызвать функцию WSAStartup.

```
WSADATA wsaData;
WSAStartup(MAKEWORD(2,2), &wsaData);
```

В конце программы нужно вызвать WSACleanup();

Структура приложения-клиента TCP

WSAStartup
socket
connect
send
recv
closesocket
WSACleanup

Структура приложения-клиента UDP

WSAStartup
socket
sendto
recvftom
closesocket
WSACleanup

Установка соединения (сервер).

Установка соединения на стороне сервера состоит из четырёх этапов, ни один из которых не может быть опущен. Сначала сокет создаётся и привязывается к локальному адресу. Если компьютер имеет несколько сетевых интерфейсов с различными IP-адресами, вы можете принимать соединения только с одного из них, передав его адрес функции bind. Если же вы готовы соединяться с клиентами через любой интерфейс, задайте в качестве адреса константу INADDR_ANY. Что касается номера порта, вы можете задать конкретный номер или 0 (в этом случае система сама выберет произвольный неиспользуемый в данный момент номер порта).

На следующем шаге создаётся очередь запросов на соединение. При этом сокет переводится в режим ожидания запросов со стороны клиентов. Всё это выполняет функция listen.

```
int listen(int sockfd, int backlog);
```

Первый параметр - дескриптор сокета, а второй задаёт размер очереди запросов. Каждый раз, когда очередной клиент пытается соединиться с сервером, его запрос ставится в очередь, так как сервер может быть занят обработкой других запросов. Если очередь заполнена, все последующие запросы будут игнорироваться. Когда сервер готов обслужить очередной запрос, он использует функцию accept.

```
int accept(int sockfd, void *addr, int *addrlen);
```

Функция accept создаёт для общения с клиентом *новый* сокет и возвращает его дескриптор. Параметр sockfd задаёт слушающий сокет. После вызова он остаётся в слушающем состоянии и может принимать другие соединения. В структуру, на которую ссылается addr, записывается адрес сокета клиента, который установил соединение с сервером. В переменную, адресуемую указателем addrlen, изначально записывается размер структуры; функция accept записывает туда длину, которая реально была использована. Если вас не интересует адрес клиента, вы можете просто передать NULL в качестве второго и третьего параметров.

Обратите внимание, что полученный от accept новый сокет связан с тем же самым адресом, что и слушающий сокет. Сначала это может показаться странным. Но дело в том, что адрес TCP-сокета не обязан быть уникальным в Internet-домене. Уникальными должны быть только *соединения*, для идентификации которых используются *два* адреса сокетов, между которыми происходит обмен данными.

Структура приложения-сервера TCP

WSAStartup
socket
bind
listen
accept
send
recv
closesocket
closesocket
WSACleanup

ЗАДАНИЯ

Разбиться на пары: один участник пары пишет сервер, второй – клиент.

1. Написать клиентское приложение протокола echo.
2. Написать клиентское приложение протокола daytime.
3. Написать сервер протокола echo под TCP.
4. Написать сервер протокола daytime под TCP.

Функции обратного вызова

Callback (обратный вызов) — передача исполняемого кода в качестве одного из параметров другого кода. Обратный вызов позволяет в функции исполнять код, который задается в аргументах при её вызове. Этот код может быть определён в других контекстах программного кода и быть недоступным для прямого вызова из этой функции. Некоторые алгоритмические задачи в качестве своих входных данных имеют не только числа или объекты, но и действия (алгоритмы), которые естественным образом задаются как обратные вызовы.

Об обратном вызове можно думать как о действии, передаваемом некоторой основной процедуре в качестве аргумента. И это действие может рассматриваться как

- подзадача и использоваться для обработки данных внутри этой процедуры,
- «телефонная связь», используемая для того, чтобы «связываться» с тем, кто вызвал процедуру, при наступлении какого-то события (англ. callback дословно переводится как «звонок обратно»).

Для лучшего понимания причин использования обратного вызова рассмотрим простую задачу выполнения следующих операций над списком чисел: напечатать все числа, возвести все числа в квадрат, увеличить все числа на 1, обнулить все элементы. Ясно что алгоритмы выполнения этих четырёх операций схожи — это циклы обход всех элементов списка с некоторым действием в теле цикла, применяемым к каждому элементу. Это несложный код, и в принципе можно написать его 4 раза. Но давайте рассмотрим более сложный случай, когда список у нас хранится не в памяти, а на диске, и со списком могут работать несколько процессов одновременно и необходимо решать проблемы синхронизации доступа к элементам (несколько процессов могут выполнять разные задачи — удаления некоторых элементов из списка, добавления новых, изменение существующих элементов в списке). В этом случае задача обхода всех элементов списка будет довольно сложным кодом, который не хотелось бы переписывать несколько раз. Правильнее создать функцию общего назначения для обхода элементов списка и дать

возможность программистам абстрагироваться от того, как именно устроен алгоритм обхода и писать лишь функцию обратного вызова для обработки отдельного элемента списка.

Рассмотрим использование функций обратного вызова на примере.

```
typedef int (__stdcall *CompareFunction)(const byte*, const byte*);
```

Ниже описана функция, принимающая в качестве параметров указатель на массив, число элементов массива, размер каждого элемента и указатель на метод сравнения элементов массива. Метод сравнения, в данном случае, есть функция обратного вызова.

```
void DLLDIR __stdcall Bubblesort(byte* array,
                                 int size,
                                 int elem_size,
                                 CompareFunction cmpFunc);
```

Использование callback функции позволяет использовать одну и ту же функцию (Bubblesort) для сортировки, к примеру, как массивов целочисленных данных, так и массивов, элементами которых являются строки.

```
void DLLDIR __stdcall Bubblesort(byte* array,
                                 int size,
                                 int elem_size,
                                 CompareFunction cmpFunc)
{
   for(int i=0; i < size; i++)
   {
      for(int j=0; j < size-1; j++)
      {
         // make the callback to the comparison function
         if(1 == (*cmpFunc)(array+j*elem_size,
                  array+(j+1)*elem_size))
         {
            // the two compared elements must be interchanged
            byte* temp = new byte[elem_size];
            memcpy(temp, array+j*elem_size, elem_size);
            memcpy(array+j*elem_size,
                   array+(j+1)*elem_size,
                   elem_size);
            memcpy(array+(j+1)*elem_size, temp, elem_size);
            delete [] temp;
         }
      }
   }
}
```

Для примера приведем код функций сравнений двух элементов типа int и char*. Обе функции в качестве параметров принимают два сравниваемых аргумента.

```
int __stdcall CompareInts(const byte* velem1, const byte* velem2)
{
   int elem1 = *(int*)velem1;
   int elem2 = *(int*)velem2;

   if(elem1 < elem2)
      return -1;
   if(elem1 > elem2)
      return 1;

   return 0;
}

int __stdcall CompareStrings(const byte* velem1, const byte* velem2)
{
   const char* elem1 = (char*)velem1;
   const char* elem2 = (char*)velem2;

   return strcmp(elem1, elem2);
```

```
}
```

В коде основной программы нужно заметить, что функции сортировки передаются в виде адреса на начало функции (&CompareInts).

```
int main(int argc, char* argv[])
{
   int i;
   int array[] = {5432, 4321, 3210, 2109, 1098};

   cout << "Before sorting ints with Bubblesort\n";
   for(i=0; i < 5; i++)
      cout << array[i] << '\n';

   Bubblesort((byte*)array, 5, sizeof(array[0]), &CompareInts);

   cout << "After the sorting\n";
   for(i=0; i < 5; i++)
      cout << array[i] << '\n';

   const char str[5][10] = {"estella",
                            "danielle",
                            "crissy",
                            "bo",
                            "angie"};

   cout << "Before sorting strings with Bubblesort\n";
   for(i=0; i < 5; i++)
      cout << str[i] << '\n';

   Bubblesort((byte*)str, 5, 10, &CompareStrings);

   cout << "After the sorting\n";
   for(i=0; i < 5; i++)
      cout << str[i] << '\n';

   return 0;
}
```

В нашем случае в callback функцию можно вынести работа сервера с клиентом, после чего вызывать эту функцию в отдельном **потоке** с помощью WinAPI функции CreateThread.

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId);
```

Параметры функции CreateThread.
LpThreadAttributes - является указателем на структуру LPSECURITY_ATTRIBUTES. Для присвоения атрибутов защиты по умолчанию, передавайте в этом параметре NULL.
DwStackSize - параметр определяет размер стека, выделяемый для потока из общего адресного пространства процесса. При передаче 0 - размер устанавливается в значение по умолчанию.
LpStartAddress - указатель на адрес входной функции потока.
LpParameter - параметр, который будет передан внутрь функции потока.
DwCreationFlags - принимает одно из двух значений: 0 - исполнение начинается немедленно, или CREATE_SUSPENDED - исполнение приостанавливается до последующих указаний.
LpThreadId - Адрес переменной типа DWORD в который функция возвращает идентификатор, приписанный системой новому потоку.

Понятие потока.

Под словом «поток» имеется в виду «поток команд», то есть последовательность инструкций, которые считывает и исполняет процессор. С каждым из потоков связан свой набор значений регистров процессора, называемый контекстом потока (thread context). Но кроме этого система создает еще ряд других объектов, однозначно связанных с каждым потоком. Важнейший из них (после контекста) – конечно же, стек (создание своего стека для каждого потока возможно благодаря тому, что в контексте потока сохраняется, в том числе, и регистр указателя стека, ESP). Для нас это важно, поскольку именно в стеке создаются и хранятся все локальные или автоматические переменные. Поскольку у каждого потока свой стек, то у каждого потока имеется свой собственный экземпляр локальной переменной, доступный только ему. Процесс – это объединение нескольких потоков. А объединяет эти потоки единое виртуальное адресное пространство. В этом пространстве размещаются код и данные приложения (обычно это один exe- и несколько dll-модулей). Именно единство этого пространства и делает обмен данными между потоками приложения предельно простым.

При запуске программы ОС всегда создает один поток – главный. Именно в нем программа начинает свое выполнение. В зависимости от типа программы и настроек компилятора в главном потоке может выполняться функция main, WinMain, _tmain или функция, заданная пользователем. Главный поток живет до тех пор, пока самая первая функция в стеке не завершила свое выполнение. В то ж время, главная функция может создавать дополнительные потоки, которые будут выполняться одновременно в основным потоком. Для нашей задачи нужно, чтобы главная функция сервера main создавала отдельный поток для работы с каждым подключившимся клиентом. Для этого, в качестве адреса входной функции потока (параметр LpStartAddress) нужно будет указать адрес нашей функции работы с клиентом. Для этого сначала описываем прототип функции, обеспечивающей взаимодействие с клиентом:

```
DWORD WINAPI WorkWithClient(LPVOID client_socket);
```

Далее в коде основной функции севера, вызываем создание отдельного потока с указанием адрема функции WorkWithClient:

```
DWORD thID;
CreateThread(NULL,NULL,&WorkWithClient,&client_socket,NULL,&thID);
```

ЗАДАНИЕ НА ДОМ

Реализовать простой консольный чат: один участник пишет – все читают.
(Сервер запоминает соединившихся к нему клиентов и рассылает принятое от одного из клиентов сообщение всем остальным. Для организации работы с клиентами использовать callback функцию)

СПИСОК ЛИТЕРАТУРЫ

1. Дейтел Х. М. Как программировать на C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2000 г. – 1024 с.: ил.
2. Страуструп Б. Язык программирования C++. Специальное издание: ++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2008 г. – 1104 с. :ил.
3. Шилдт Г. Полный справочник по C++: ++: Пер. с англ. – М.: Изд-во Вильямс, 2007 г. – 800 с.: ил.
4. Шилдт Г. C++: Базовый курс: ++: Пер. с англ. – М.: Изд-во Вильямс, 2008 г. – 624 с.: ил.
5. Джонсон Б., Скибо К., Янг М.: Основы Microsoft Visual Studio .NET 2003: - М.: Изд-вр Русская редакция, 2003 г. – 464 с.: ил.
6. Г. Шилдт Теория и практика C++: пер. с анг. – СПб.: BHV – Санкт-Петербург, 1996ю – 416 с., ил.
7. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.
8. Дейтел Х. М. Как программировать на C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2000 г. – 1024 с.: ил.
9. Браунси К. Основные концепции структур данных и реализация в C++: Пер. с англ. – М.: Изд-во Вильямс, 2002 г. – 320 с.: ил.
10. Топп У., Форд У. Структуры данных в C++: ++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2006 г. – 816 с. :ил.
11. Дейтел Х. М. Как программировать на C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 2000 г. – 1024 с.: ил.
12. Ватолин Д. С. Алгоритмы сжатия изображений. Методическое пособие: М.: Издательский отдел факультета Вычислительной Математики и Кибернетики МГУ им. М.В.Ломоносова, 1999 г. — 76 с.
13. Ахо, Альфред, В., Хопкрофт, Джон, Ульман, Джеффри, Д. Структуры данных и алгоритмы. — Издательский дом «Вильямс», 2000. — С. 384.
14. Стивенс У., UNIX: Разработка сетевых приложений. - СПб.: Питер, 2004
15. Шмидт Д., Хьюстон С. Программирование сетевых приложений на C++. Том1. — Бином-Пресс, 2003. — С. 304.