

САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
Кафедра динамического моделирования и биомедицинской инженерии

**А.С. КАРАБАЕВ**

# **ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ АППАРАТНЫХ СРЕДСТВ НА ЯЗЫКЕ СИ**

Учебно-методическое пособие

Саратов 2009

Караваев А.С. Введение в программирование аппаратных средств на языке Си. Учебно-методическое пособие для студентов факультета нано- и биомедицинских технологий Саратовского госуниверситета. Саратов, 2009. 104 с.

Караваев Анатолий Сергеевич, к.ф.-м.н., доцент базовой кафедры динамического моделирования и биомедицинской инженерии факультета нано- и биомедицинских технологий Саратовского государственного университета имени Н.Г. Чернышевского. Кафедра организована на базе Саратовского филиала Института радиотехники и электроники имени В.А. Котельникова РАН.  
e-mail: [karavaevas@gmail.com](mailto:karavaevas@gmail.com)

© А.С. Караваев, 2009

Учебно-методическое пособие оформлено в виде конспекта лекций. Пособие ориентировано на студентов и инженерно-технических работников, знакомых с языком Pascal и преследующих цель быстрого освоения основ популярного языка Си для программирования микропроцессоров ПК, устройств сопряжения, микроконтроллеров, цифровых сигнальных процессоров и периферийных устройств.

Основной упор сделан на вопросах, специфических для программирования аппаратных ресурсов: адресация памяти, ссылки и динамически создаваемые структуры данных, работа с отдельными битами и т.п. Затронуты некоторые теоретические вопросы взаимодействия АЛУ с периферийными устройствами, методика организации типичных структур данных (очередь, кольцевой буфер ввода, стек, связанный список). В последних лекциях дается краткое введение в язык ассемблера и затрагиваются вопросы совместного использования Си и ассемблера, в частности, встроенного ассемблера.

Учебное пособие снабжено значительным количеством иллюстраций, таблиц, схем и примеров программ. Тексты рассматриваемых в пособии примеров свободно доступны для скачивания с сайта [www.nonlinmod.sgu.ru](http://www.nonlinmod.sgu.ru).

Вопросы, замечания и предложения присылайте по адресу электронной почты.

## Оглавление

<b>Лекция 1 Введение. Простейшая программа на языке Си</b>	<b>5</b>
Введение	
Технические замечания	
Языки программирования	
Язык C	
Рекомендуемая литература	
Структура программы на языке C	
Комментарии	
Заголовочные файлы	
Объявление переменных	
Точка входа в программу, составной оператор	
Вывод на экран текстовых сообщений	
Общие замечания, хороший стиль программирования	
<b>Лекция 2 Интегрированная среда разработки Borland C++ 3.1.</b>	<b>14</b>
<b>Типы данных</b>	
Интегрированная среда разработки Borland C++	
Скалярные типы данных C	
Преобразования типов. Массивы	
Пользовательские типы данных	
Структуры	
Перечисления, объединения, битовые поля	
<b>Лекция 3 Основные операторы и конструкции языка</b>	<b>22</b>
Арифметические операторы	
Операторы инкремента и декремента	
Операторы условия if/else, switch	
Логические операции	
Циклы	
Безусловный переход	
Побитовые операторы	
Доступ к отдельным битам	
<b>Лекция 4 Адресация памяти и использование указателей</b>	<b>34</b>
Двоичная система счисления, бит, байт, слово	
Использование различных систем счисления	
Память ЭВМ	
Адресация и распределение памяти ЭВМ	
Использование ОП прикладной программой	
Объявление и использование указателей	
Разыменование указателей на структуры	
Арифметические действия с указателями	
Указатели и массивы	
<b>Лекция 5 Интерфейс с пользователем, работа с файлами, строки</b>	<b>45</b>
Форматированный вывод с помощью printf	
Ввод с помощью scanf	
Потоки	
Работа с текстовыми файлами	
Файлы произвольного доступа	
Обработка символов на ЭВМ	
Операции со строками в C	
<b>Лекция 6 Макросы, функции, библиотеки подпрограмм</b>	<b>54</b>
Подпрограммы, библиотеки подпрограмм	

Объявление функции в С	
Вызов функций	
Константы и макросы, директива #define	
Область видимости переменных	
Прототипы функций, библиотеки, рекурсия	
Создание “процедур”	
Функция main	
Заголовочные файлы, директива #include	
Стандартная библиотека С	
Указатели на функции	
Передача указателя на функцию, вызов по ссылке	
<b>Лекция 7 Организация обмена и хранения данных</b>	<b>64</b>
Динамические массивы	
Стек	
Очередь, сбор данных в многозадачной ОС	
Кольцевой буфер, линия задержки	
Связанный список	
<b>Лекция 8 Введение в организацию работы ЭВМ и язык ассемблера</b>	<b>78</b>
Язык ассемблера	
Структура ЭВМ	
Регистры процессора	
Сегментная адресация памяти	
Подготовка и создание программы на языке ассемблера	
Работа видеоадаптера ПЭВМ в текстовом режиме	
Простейшая программа на языке ассемблера	
<b>Лекция 9 Программирование на нескольких языках, встроенный ассемблер</b>	<b>90</b>
Программа на ассемблере с сегментами данных и стека	
Модели памяти	
Программирование на нескольких языках	
Использование встроенного ассемблера	
<b>Приложение 1: Сокращения и аббревиатуры</b>	<b>101</b>
<b>Приложение 2: Практические задания для самоконтроля</b>	<b>102</b>
<b>Рекомендуемая литература</b>	<b>105</b>

## **Лекция 1 Введение. Простейшая программа на языке Си**

Введение. Технические замечания. Языки программирования. Язык С. Рекомендуемая литература. Структура программы на языке С. Комментарии. Заголовочные файлы. Объявление переменных. Точка входа в программу, составной оператор. Вывод на экран текстовых сообщений. Общие замечания, хороший стиль программирования.

### **Введение**

Целью учебного курса является обучение студентов основам навыкам и приемам программирования аппаратных ресурсов персонального компьютера (ПК) и микропроцессорных систем. Материал представлен в форме, ориентированной на обучающихся, имеющих навыки программирования на языке Pascal в среде Borland Pascal.

Де-факто наиболее используемым в мире для программирования процессоров и периферийного оборудования ПК и микроконтроллеров в настоящее время является язык С (C++). Язык С предоставляет широкие возможности, хорошую кросс-платформенную переносимость и мощную информационную поддержку в силу широчайшего распространения в мировом сообществе программистов. Основная часть учебного курса содержит материалы, обучающие основам программирования на языке С, также рассмотрены примеры использования смешанного программирования, когда исходный текст, написанный на С, включает в себя ассемблерные вставки. Поэтому, для облегчения перехода на другой язык программирования, в тексте лекций часто проводятся аналогии между конструкциями языков С и Pascal. Из тех же соображений облегчения начального этапа адаптации к новому языку программирования, учебные задания, предлагаемые в рамках учебного курса, рекомендуется выполнять в среде Borland C++ 3.1. Это позволяет облегчить дополнительную нагрузку по привыканию к работе в незнакомой среде разработки, т.к. интегрированные среды разработки (IDE) Borland Pascal 7.0 и Borland C++ 3.1 в высокой степени унифицированы.

Основной упор сделан на вопросах, специфических для программирования аппаратных ресурсов: адресация памяти, ссылки и динамически создаваемые структуры данных, работа с отдельными битами и т.п. Затронуты некоторые теоретические вопросы взаимодействия АЛУ с периферийными устройствами, методика организации типичных структур данных (очередь, кольцевой буфер ввода, стек, связанный список).

### **Технические замечания**

В тексте лекций понятия и определения при их первом упоминании выделяются *курсивом*, участки листинга программы, переменные, и др. конструкции, относящиеся к программированию, выделены моноширинным шрифтом Courier New. Аббревиатуры расшифрованы при их первом использовании, кроме того, они сведены в списке сокращений и аббревиатур. По ходу изложения материала автор старался приводить для наиболее употребительных терминов английский перевод, т.к. почти вся техническая документация в мире, размещаемая в открытом доступе, представляется на английском языке.

### **Языки программирования**

Сотни разработанных к настоящему моменту языков программирования можно разделить на 3 большие класса:

1. машинные языки,
2. языки ассемблера,
3. языки высокого уровня.

Каждый процессор понимает только свой *машинный язык*. С точки зрения пользователя машинный язык представляет из себя набор целочисленных кодов операций (OpКод, англ. OpCode), значений и адресов операндов. Откройте с помощью текстового редактора

любой исполнимый файл – это программа на машинном языке. Для примера в листинге 1.1 приведен участок программы на машинном языке, складывающий целые числа 3 и 7 и помещающей результат сложения в 16 битную ячейку памяти:

```
0000 005A 880B 0000
E319 0000 0023 0100
004D 8806 0000 E119
1800 60A0 2100 0100
00C8 0200 0039 2600
0072 03E8 0000 C746
FE00 00B8 0300 0507
0089 46FE C9C3 8D9C
```

Листинг 1.1 Пример программы на машинном языке. Программа складывает целые числа 3 и 7 и помещает результат сложения в 16 битную ячейку памяти.

Понятно, что писать программы на машинном языке человеку очень неудобно, такие программы составляли лишь программисты на заре компьютерной эры, когда ЭВМ занимали несколько комнат. Тем не менее, многие современные языки программирования высокого уровня допускают использование т.н. inline-вставок, содержащих куски кода, написанные непосредственно на машинном языке (например, язык Pascal предлагает для этого директиву `inline`).

По мере развития компьютерной техники стало понятным, что программирование на машинных языках тормозит развитие компьютерной техники, является очень медленным и даже непосильным занятием для многих программистов. Вместо последовательности чисел непосредственно понятных процессору, стали использовать англоязычные аббревиатуры – появились *языки ассемблера*. Для преобразования таких программ в машинный язык используются специальные программы-трансляторы, называемые *ассемблерами* (англ. assembler – сборщик). Особенностью ассемблера является наличие взаимнооднозначного соответствия между текстом программы и машинным языком. То есть трансляция каждого ассемблерного предложения (строки, содержащей аббревиатуру команды и список операндов) осуществляется непосредственно в машинный код, поэтому, возможно и обратное преобразование машинного кода в текст – *дизассемблирование*. В листинге 1.2 приведен пример программы на языке ассемблера ПК, осуществляющей те же действия, что и программа в листинге 1.1:

```
MOV AX, 3
ADD AX, 7
MOV word ptr c, AX
```

Листинг 1.2 Пример программы на языке ассемблера ПК, осуществляющей сложение констант 3 и 5 и размещение результата сложения в двухбайтовую ячейку памяти.

С появлением языков ассемблера процесс программирования значительно упростился и ускорился, однако, даже для реализации простейших задач требовалось написание весьма громоздких программ. Для ускорения процесса программирования были разработаны языки высокого уровня, в которых для выполнения совокупности машинных инструкций достаточно написать один оператор (лист. 1.3).

```
c=3+7;
```

Листинг 1.3 На языке C для выполнения сложения, аналогичное приведенным выше примерам (лист. 1.1, 1.2), реализуется с помощью одного оператора.

Программы, преобразующие тексты программ, написанных на языках высокого уровня, в машинные языки называют *компиляторами* (англ. compiler). Очевидно, что с точки зрения программиста использование языков высокого предпочтительнее ассемблера. Языки высокого уровня обладают гораздо большей универсальностью по отношению к аппаратной части. Например, приведенная в листинге 1.3 команда сработает одинаково и на ЭВМ, построенной на базе процессоров Intel для ПК и на сигнальных процессорах Motorola, используемых, например, в игровых приставках, а ассемблерная программа из листинга 1.2 работать на Motorola не будет из-за отличий во внутреннем устройстве процессоров. Особенности системы команд конкретных процессоров учитываются компилятором при построении программы на машинном языке и участие в этом процессе программиста обычно не требуется. Вместе с тем, нужно понимать, что такое удобство и универсальность достигнуты ценой отказа от однозначного соответствия между исходным текстом и машинным кодом, поэтому, операция преобразования машинного языка в текст программы на языке высокого уровня невозможна. Типично, что размер машинного кода, полученного после компиляции программы с языка высокого уровня больше, а быстродействие такого кода ниже, чем для программы, написанной на языке ассемблера. Т.е. использование ассемблера позволяет, в общем случае, получить более *эффективный код*. Поэтому, для решения каждой конкретной задачи решение о выборе языка программирования должно приниматься индивидуально, исходя из соображений о том, что более важно, эффективный код или существенная экономия времени на этапе разработки и отладки программы.

### Язык С

Язык С и его расширения, например, С++, являются, в настоящее время, наиболее распространенными средствами для программирования периферийного оборудования, написания драйверов устройств и создания операционных систем.

Предшественником С был язык В, созданный Мартином Ричардсом в 1970 г. Язык В использовался для написания ранних версий UNIX. Широкого распространения В не получил в силу ряда недоработок, в частности, В не разделял типы данных.

Язык С был разработан сотрудником Bell Laboratories Деннисом Ричи в 1972 г. для компьютера DEC PDP-11 и получил известность, как язык операционной системы (ОС) UNIX. Сегодня практически все операционные системы написаны на С и С++: Microsoft Windows, UNIX, Linux, Mac OS и др. Язык С стал популярным в конце 70-х годов. Существенным толчком к развитию стала публикация в 1978 г. книги Кернигана и Ричи «Язык программирования С». Эта книга стала одним из самых удачных изданий по компьютерным дисциплинам, выпущенным в свет.

Применение С для программирования различных *аппаратных платформ* (компьютерных систем, построенных на базе различных типов процессоров) привело к созданию различных компиляторов и появлению разных вариантов языка, которые, не смотря на свою схожесть, часто оказывались несовместимыми. Это явилось серьезной проблемой разработчиков, ориентирующих свои программы для работы на разных платформах. Стало ясно, что нужна стандартизация языка. В 1989 г. такие требования по стандартизации были разработаны и утверждены под названием ANSI/ISO 9899:1990. Второе издание книги Кернигана Ричи, вышедшее в 1988 г., описывает эту стандартизированную версию С, под названием ANSI C. ANSI C получил широчайшее распространение в мире.

В 1986 г. сотрудник Bell Laboratories Бьерн Страуструп разработал надстройку над С, назвав новый язык С++. Принципиальным новшеством стало предоставление языком С++ *объектно-ориентированного подхода* к программированию (ООП), способного существенно упростить и ускорить создание масштабных программных проектов большими командами разработчиков.

В настоящее время на базе идей, лежащих в основе С и С++, выросли несколько языков программирования. В частности, широкое распространение получили кроссплатформенные языки С#, Java и Python.

Таким образом, изучение языков С и С++ оказывается полезным при освоении целого ряда популярных языков программирования. Однако, при программировании аппаратных ресурсов, часто оказывается достаточным использовать ANSI C. Поэтому, данный учебный курс ориентируется именно на этот стандарт, не затрагивая объектно-ориентированных возможностей С++.

Основные преимущества С и С++ по сравнению с другими языками программирования:

- хорошая кроссплатформенная переносимость программ;
- высокая эффективность кода (высокое быстродействие при относительно небольшом размере исполнимых файлов);
- обширная поддержка, включающая как большое количество литературы и справочных материалов, так и широкий выбор уже готовых библиотек подпрограмм;
- наличие свободно распространяемых бесплатных компиляторов (в том числе и для ОС MS Windows).

### **Рекомендуемая литература**

В качестве учебных и справочных материалов при освоении учебного курса можно рекомендовать книги, перечисленные ниже.

В качестве пособий для начинающих изучать язык С:

- Дейтел Х.М., Дейтел П.Дж. Как программировать на С. [1] – профессионально написанное объемное учебное пособие, снабженное значительным количеством примеров и контрольными вопросами, используется в качестве учебника во многих американских ВУЗах,
- а также Подбельский В.В. Язык С++: Учеб. пособие. [2] – материал излагается в легкой для усвоения форме.

В качестве справочника для изучающих С:

- Керниган Б., Ритчи Д. Язык программирования Си. Пер. с англ. –3-е издание испр. – СПб.: "Невский Диалект", 2001. –352 с. [3]

Для начинающих изучать язык ассемблера и архитектуру ПЭВМ:

- Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: «ДИАЛОГ-МИФИ», 2001. –640 с. – Авторами выпущена целая серия замечательных книг и справочников, которые подойдут для желающих изучать архитектуру ПЭВМ и освоить язык ассемблер “с нуля” в стиле подробного разбора последовательно усложняющихся примеров программ. [4]
- Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения. –2-е издание испр. и доп. –СПб.: «ООО ДиаСофтЮП», 2002. –656 с. – Хорошее учебное пособие с большим количеством примеров. [5]

Для желающих освоить язык ассемблера на профессиональном уровне два рекомендованных к использованию в ВУЗах учебных пособия, снабженных массой практически полезных примеров:

- Юров В.И. Assembler. Учебник для ВУЗов. –2-е издание. –СПб.: «ПИТЕР», 2003. – 637 с. [6]
- Юров В.И. Assembler. Практикум. –2-е издание. –СПб.: «ПИТЕР», 2006. –399 с. [7]

### **Структура программы на языке С**

Для знакомства с языком рассмотрим несколько простых программ. Первая (листинг 1.4) просто выводит на экран текстовое сообщение “Hello world!”.



```

/*    The simplest
      C program    */

#include <stdio.h>

void main(void){ //Entry point

    printf("Hello world!"); //Message

}

```

Листинг 1.4 Простая программа на языке C, реализующая вывод на экран текстового сообщения.

Более подробно рассмотрим представленный в листинге 1.5 текст еще одной простой программы, написанной на языке ANSI C. В программе объявляются 3 целочисленные переменные: a, b и c, переменным a и b присваиваются значения 2 и 3, соответственно, в переменную c сохраняется сумма a и b, значение, хранящееся в c, выводится на экран.

```

/*    Simple
      C program    */

#include <stdio.h>

signed int a,b,c;

void main(void){ //Entry point

    a=2;
    b=3;
    c=a+b;

    printf("\nc=%i", c);

}

```

Листинг 1.5 Простая программа на языке C, реализующая сложение 2 целых чисел и отображение результатов на экране.

В таблице 1.1 для пояснения работы программы (листинг 1.5) приводится аналогичная программа на языке Pascal.

No	Описание	Программа на C	Программа на Pascal
1.	Многострочный комментарий	/* Simple C program */	{ Simple Pascal program }
2.	Подключаемые модули и заголовочные файлы	#include <stdio.h>	Uses CRT;
3.	Объявление глобальных переменных	signed int a,b,c;	VAR a,b,c :Integer;
4.	Точка входа в программу	void main(void){	BEGIN

5.	Однострочный комментарий	//Entry point	{Entry point}
6.	Арифметические действия	a=2; b=3; c=a+b;	A:=2; b:=3; c:=a+b;
7.	Вывод на экран текстового сообщения и значения переменной	printf("\nc=%i", c);	WriteLn('c=', c);
8.	Точка выхода из программы	}	END.

Таблица 1.1 Сопоставление аналогичных программ на языках С и Pascal.

### Комментарии

Первые две строки программы представляют собой *многострочный комментарий* (англ. comment). Строка 5 таблицы 1.1 содержит пример *однострочного комментария*. Строки комментария, отмеченные указанными в таблице 1.2 парами символов, игнорируется при выполнении программы, поэтому, комментарий может содержать абсолютно любой текст. Обычно, комментарии используют для пояснения назначения участка программы.

Использование комментариев настоятельно рекомендуется, т.к. существенно улучшает читаемость программ и упрощает их отладку. Считается, что в профессионально написанных программах комментарии составляют до 30% текста.

/*	Пара символов, отмечающих начало многострочного комментария.
*/	Пара символов, отмечающих конец многострочного комментария.
//	Пара символов, отмечающих начало однострочного комментария, при выполнении программы игнорируется текст до конца строки, начиная с этих символов.

Таблица 1.2 Символы, определяющие комментарий в языке С.

### Заголовочные файлы

Заголовочные файлы С могут использоваться аналогично модулям в Pascal, для объявления глобальных переменных, констант, описания пользовательских типов данных и в качестве библиотек подпрограмм. Вместе с тем, в использовании библиотек подпрограмм и заголовочных файлов в С и модулей Pascal есть важные различия. Далее использование заголовочных файлов будет рассматриваться подробнее.

Заголовочный файл (h-файл, header-файл) подключается к программе директивой `#include` (в Pascal аналогично используется служебное слово `USES`) и размещается после этой директивы. Имя системного h-файла, входящего в комплект поставки компилятора, должно быть заключено между парами знаков “меньше”, “больше”, например: `#include <stdio.h>`, `#include <conio.h>`. Имя пользовательского h-файла, хранящегося в одной папке с исходным текстом программы, должно быть заключено между парами двойных кавычек, например: `#include <mysuper.h>`.

Объявление в листинге 1.5 заголовочного файла `stdio.h` (STanDart Input and Output, англ. Стандартный ввод-вывод) позволяет использовать в программе, в частности, подпрограмму для вывода текста на экран.

Директивы `#include` могут размещаться в любом месте программы до точки входа (функции `main`), но из соображений улучшения читаемости программы и уменьшения

вероятности ошибок программиста, рекомендуется размещать директивы единым блоком в начале программы.

### Объявление переменных

В С объявление переменной начинается с указания ее типа. Имя типа может состоять из одного (`char`), двух (`signed int`) и даже 3 слов (`unsigned long int`). Имена переменных следуют после указания типа через запятую, объявление типа неперенных заканчивается символом “;” (англ. semicolon). Более подробно типы данных в С будут рассматриваться далее.

В отличие от Pascal, где объявление глобальных переменных возможно только в специализированном разделе, помеченном служебным словом `VAR`, в С глобальные переменные могут объявляться в любом месте до точки входа в программу (до начала функции `main`), причем объявления могут разрываться директивами компилятора, описанием подпрограмм и т.п. Например, если начало программы (листинг 1.5) изменить на:

```
signed int a,b;
#include <stdio.h>
signed int c;
...
```

то программа скомпилируется без ошибок и результат ее работы не изменится по сравнению с исходным вариантом. Вместе с тем, такой стиль сильно ухудшает читаемость исходного текста и затрудняет отладку. Поэтому, рекомендуется, все глобальные переменные объявлять единым блоком, не разрывая их объявление другими конструкциями языка. Желательно код, содержащий объявления глобальных переменных, размещать сразу после объявления подключаемых заголовочных файлов.

### Точка входа в программу, составной оператор

*Точка входа в программу* (англ. entry point), т.е. начало основной программы в языке С оформлено, как функция, имеющая предопределенное имя `main` (англ. главная). Использование служебных слов `void` (англ. пустота, пустая операция) в строке 4 таблицы 1.1 означает, что эта функция не имеет аргументов и не возвращает никакого значения. (Вызов функции `main` с аргументами и возврат ей значения иногда используется для взаимодействия программы с операционной системой.) Использование подпрограмм в С более подробно будет рассмотрено далее.

*Тело программы* (англ. program body – последовательность команд и конструкций языка, собственно реализующая алгоритм) заключается между фигурными скобками.

Пара фигурных скобок “{ }” (англ. braces) с заключенными между ними командами и конструкциями языка образуют *составной оператор*. Правила и способы использования составного оператора полностью аналогичны конструкции “begin end” Pascal, где “{” соответствует begin, а “}” аналогично end. Пример использования составного оператора для выполнения в цикле нескольких команд представлен в листинге 1.6.

Программа на С	Программа на Pascal
<pre>for (i=1; i&lt;=50; i++) {     k=k+i*10;     j=j+i*100; };</pre>	<pre>For i:=1 to 50 do begin     k:=k+i*10;     j:=j+i*100; end;</pre>

Листинг 1.6 Использование составного оператора для выполнения в цикле нескольких команд. Приведены аналогичные по функциональности участки программы на языках С и Pascal.

## Вывод на экран текстовых сообщений

Для вывода на экран текстовых сообщений и содержимого переменных часто используется функция `printf`, описанная в заголовочном файле `stdio.h`.

Обратите внимание, что в C строковая константа обрамляется двойными кавычками (англ. double quotes) `"c="`, в отличие от Pascal, где строки заключаются в одинарные кавычки (англ. quotes): `'c='`.

Использование функции `printf` отлично от `Write` и `WriteLn`. `printf` использует т.н. *форматный вывод*. Это означает, что работа этой функции управляется *ESC-последовательностями* (читается: эскейп-последовательность, эск-последовательность) и *спецификаторами формата*.

В примере из листинга 1.5 пара символов `"\n"` образует ESC-последовательность, которая интерпретируется компилятором не как часть строковой константы, а как управляющая выводом текста команда "перейти на новую строку". Часть выводимого сообщения, расположенная правее этой ESC-последовательности будет перенесена на новую строку.

Пара символов `%i` также не будут отображаться на экране, т.к. они представляют собой спецификатор формата. Встретив такой спецификатор компилятор вместо этой пары символов отобразит на экране содержимое целочисленной переменной, следующей в списке параметров `printf` за строчной константой после запятой.

Форматный вывод в стиле C часто кажется людям, знакомых с другими языками программирования, одной из наиболее непривычных особенностей, но небольшая практика позволяет с легкостью его освоить.

Более подробное знакомство с форматным выводом, управляемым ESC-последовательностями и спецификаторами формата состоится в следующих лекциях.

## Общие замечания, хороший стиль программирования

Перед тем, как двинуться дальше важно отметить несколько технических моментов, касающихся программирования C.

Как уже отмечалось, текст заголовочных файлов обычно содержится в файлах с расширением `".h"`.

Исходные тексты программ сохраняются в файлах с расширениями `".c"` или `".cpp"` (от C plus plus – C++).

Важно помнить, что C различает регистр символов, т.е. можно объявить 2 переменные, например: `int k, K;` и эти переменные, `k` и `K` будут рассматриваться компилятором, именно как 2 независимые переменные. Согласно принятым соглашениям, имена всех стандартных (входящих в стандартный комплект поставки) функции C представляют собой наборы символов в **нижнем** регистре. Типичными ошибками программиста, переходящего на C, например, с языка Pascal, является объявление переменной в одном регистре, а попытка обращения к ней в программе в другом. Также типичной ошибкой является попытка использования при вызове стандартных функций языка символов в верхнем регистре (как это принято, например, в Pascal).

Начинающие программисты часто не уделяют должного внимания такому важному моменту, как стиль оформления исходного текста программы. Вместе с тем, использование единообразного стиля, согласующегося с общепринятыми тенденциями, отступов и комментариев способно существенно сократить время разработки программы за счет упрощения отладки. При разработке же проекта коллективом программистов, использование единого стиля является насущной необходимостью.

В любом случае, настоятельно рекомендуется писать программы в простой и четко манере KIS (англ. Keep It Simple – будь проще), придерживаясь при оформлении исходного текста нескольких общих правил:

1. Все команды и конструкции языка должны начинаться с новой строки, написание нескольких команд в одной строке крайне нежелательно;

2. При использовании таких конструкций языка, как условия, циклы и подпрограммы необходимо использовать отступы. В качестве отступа обычно используют от 2 до 4 символов “пробел” (англ. space), причем задавшись конкретным значением отступа, используйте его во всех своих программах.
3. Логически завершенные блоки исходного текста отделяются друг от друга 1-2 пустыми строками и сопровождаются комментариями.
4. Подпрограммы должны использоваться как можно чаще. Если вы можете выделить логически завершенный участок текста, оформляйте его в виде подпрограммы! Это улучшает читаемость программы, упрощает отладку и может существенно ускорить разработку последующих проектов.
5. Все подпрограммы должны сопровождаться комментариями, в которых кратко указано назначение подпрограммы, дается описание всех ее параметров и приводится список используемых подпрограммой глобальных переменных.
6. Старайтесь максимально использовать стандартные подпрограммы вместо написания собственных. Чтение документации по стандартным подпрограммам занимает гораздо меньше времени, чем отладка собственных аналогичных. Кроме того, как правило, стандартные подпрограммы обеспечивают лучшую переносимость, работают быстрее и надежнее, чем самодельные.
7. Названия переменных и подпрограмм должны быть осмысленными и отражать назначение этой переменной и подпрограммы, например, логично назвать переменную, хранящую длину некоего массива, “size”, “count”, или “razmer”, названия же типа “aaa02” не рекомендуются.
8. Глобальные переменные рекомендуется использовать как можно реже. Все переменные, которые по логике алгоритма могут быть локальными следует описывать локальными. Желательно все глобальные переменные сопровождать комментарием, описывающим их назначение и содержащим список использующих их подпрограмм.
9. Глобальные переменные, константы, объявления пользовательских типов и подпрограммы рекомендуется группировать в логические блоки в заголовочные файлы и библиотеки. Такие h-файлы и библиотеки должны содержать комментарий, включающий общее описание библиотеки (напр., “библиотека функций для расчета статистических характеристик”), дату ее последней модификации, имя и контактные данные автора исходного текста.
10. В среде С-программистов существуют негласные “правила хорошего тона” при именовании констант, переменных, пользовательских типов и подпрограмм. По ходу изложения некоторые из этих “правил” будут упоминаться.

## Лекция 2 Интегрированная среда разработки Borland C++ 3.1. Типы данных

Интегрированная среда разработки Borland C++. Скалярные типы данных C. Преобразования типов. Массивы. Пользовательские типы данных. Структуры. Перечисления, объединения, битовые поля.

### Интегрированная среда разработки Borland C++

Учебные задания, предлагаемые в рамках курса рекомендуется выполнять с использованием интегрированной среды разработки из пакета Borland C++ версии 3.1. Эта IDE в значительной степени унифицирована со средой из пакета Borland Pascal версии 7.0, что существенно облегчает на начальном этапе переход с языка программирования Pascal на C.

Как и IDE Borland Pascal 7.0, среда разработки Borland C++ 3.1 включает в себя:

1. текстовый редактор, позволяющий набирать исходный код программы и обеспечивающий автоматическую “подсветку” (выделение цветом) различных конструкций языка;
2. *компилятор* (англ. compiler), преобразующий исходный код программы в машинный язык;
3. *линковщик* или *линкер* (англ. link), позволяющий присоединять к исполняемому модулю библиотеки подпрограмм;
4. *отладчик* (англ. debugger), позволяющий выполнять программу по шагам и просматривать значения переменных на промежуточных этапах вычислений;
5. систему помощи;
6. ряд вспомогательных и дополнительных утилит и программ, например, ассемблер.

Основными достоинствами IDE Borland C++ 3.1 являются:

- удобство работы;
- высокая унификация (с точки зрения пользовательского интерфейса) со средой Borland Pascal 7.0;
- достаточно хороший компилятор, позволяющий создавать эффективный код;
- наличие в стандартном комплекте поставки дополнительных программ и утилит.

Основным недостатком является ориентированность среды на разработку программ для ОС MS DOS. Это, в частности, накладывает ограничения на размер обрабатываемых файлов и объем доступной программе оперативной памяти.

Вид рабочего стола интегрированной среды BC++ 3.1 представлен на рис. 2.1.

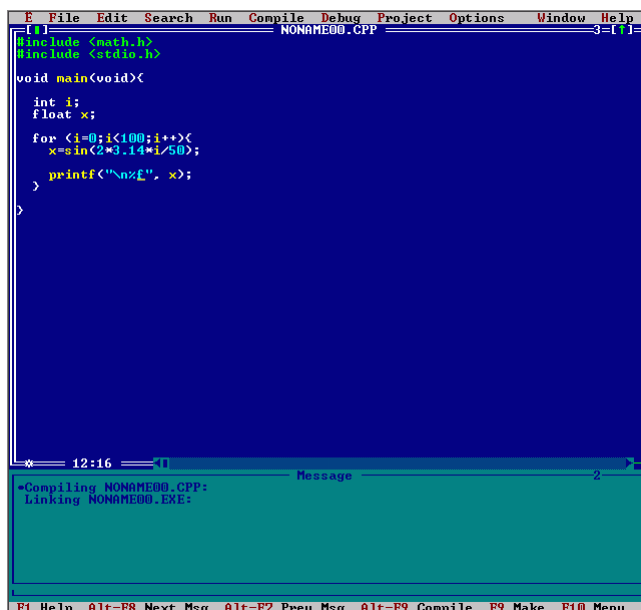


Рис. 2.1 Окно с исходным текстом программы в интегрированной среде Borland C++ версии 3.1.

В верхней строке рабочего стола расположено главное меню, нижняя строка содержит список текстовых контекстно-зависимых подсказок. Основное поле рабочего стола программы занимает окно редактора исходного текста программы и другие окна, например, окно сообщений компилятора “Message”, окно просмотра значений переменных “Watch”, диалоговые и другие окна.

Человека, знакомого с работой в среде Borland Pascal 7.0 освоение IDE Borland C++ 3.1 не затруднит. Для эффективной и комфортной работы в большинстве случаев достаточно запомнить лишь несколько сочетаний горячих клавиш, сведенных в таблице 2.1.

Комбинация клавиш	Назначение
<b>Редактор исходного кода</b>	
[F3]	Открыть файл с исходным текстом программы
[F2]	Сохранить в файл исходный текст программы
[Ctrl+Ins]	Скопировать выделенный участок текста в буфер
[Shift+Del]	Вырезать выделенный участок текста в буфер
[Shift+Ins]	Вставить текста из буфера
[Ctrl+Del]	Удалить выделенный участок текста
<b>Запуск и отладка программы</b>	
[Ctrl+F9]	Создать и запустить исполнимый (“.exe”) файл
[Alt+F5]	Просмотреть экран с результатами работы программы
[Ctrl+F2]	Прервать выполнение программы (например, при повисании)
[Ctrl+F8]	Установить/снять точку останова
[Ctrl+F7]	Добавить переменную в окно “Watch” для просмотра ее содержимого при отладке
[F8]	При пошаговом выполнении программы выполнить команды в текущей строке без захода в пользовательские подпрограммы.
[F7]	При пошаговом выполнении программы выполнить команды в текущей строке с заходом в пользовательские подпрограммы.

Таблица 2.1 Назначение горячих клавиш в среде Borland C++ 3.1. Клавиши, которые должны быть нажаты одновременно заключены между символами “[ ]”.

### Скалярные типы данных С

Язык С имеет систему скалярных типов близкую к языку Pascal. Соответствие между скалярными типами данных Pascal и С приводится в таблице 2.2.

Тип Pascal	Тип С	Размер	Диапазон значений
Byte	unsigned char	8 бит	0 до 255
Char	unsigned char	8 бит	0 до 255
ShortInt	signed char	8 бит	-128 до 127
Word	unsigned int	16 бит	0 до 65535
Integer	signed int	16 бит	-32768 до 32767
LongInt	signed long	32 бит	-2147483648 до 2147483647
-	unsigned long	32 бит	0 до 4294967295
Single	float	32 бит	3.4E-38 до 3.4E+38
Double	double	64 бит	1.7E-308 до 1.7E+308
Extended	long double	80 бит	3.4E-4932 до 1.1E+4932

Таблица 2.2 Соответствие между скалярными типами данных Pascal и С.

Таблица 2.2 описывает типы, используемые VC++ 3.1. Обратите внимание, что различные компиляторы C интерпретируют типы с одинаковыми названиями по-разному. Например, переменная типа `char` (без модификаторов) может интерпретироваться некоторыми компиляторами, как знаковое число, тип `int` (без модификаторов) чаще всего описывает 2 байтную переменную, но некоторые компиляторы, выделяют для переменной `int` 1 или 4 байта.

Переменная типа `char` в C допускает инициализацию как целым числом, так и литеральной (символьной) константой, например, допустимо:

```
char ch;  
ch=100;  
ch='A';
```

при этом переменной будет присвоен целочисленный код литерального символа.

C позволяет также использовать более короткую запись для определения целочисленных типов, опуская модификатор `signed`, т.е. описание типов `char`, `int` и `long` эквивалентно `signed char`, `signed int` и `signed long`, соответственно. Эквивалентны также описания типов `long` и `signed long int`. Какую форму записи выбрать, более короткую или более длинную, но понятную решает программист (таблица. 2.3).

No	Объявление переменной
1.	<code>long a, b;</code>
2.	<code>signed long a, b;</code>
3.	<code>signed long int a, b;</code>

Таблица 2.3 3 альтернативных способа объявления целочисленных знаковых 4-байтовых переменных `a` и `b`.

Таким образом, логика формирования названия целочисленных типов в C такова: есть 3 базовых типа: `char`, размером 1 байт, `int`, размером 2 байта и `long` (`long int`), размером 4 байта. Добавляя перед именами этих базовых типов модификаторы `signed` (англ. число со знаком) или `unsigned` (англ. число без знака) программист объясняет компилятору как надо интерпретировать переменную (как знаковую или беззнаковую, соответственно).

В C отсутствует в явном виде эквивалент логического типа данных Pascal `Boolean`. В качестве булевой переменной в C может выступать любая целочисленная (знаковая и беззнаковая) переменная, причем ее значение трактуется, как `False` (ложь) в случае, если она равна 0 и как `True` (истина) в противном случае.

Обратите внимание, что в Pascal независимо используются типы `Byte` – беззнаковая переменная, хранящая целое число от 0 до 255 и тип `Char` – “символьный” тип. Фактически внутреннее представление этих типов в ЭВМ идентично, т.к. в переменной типа `Char` фактически хранится целочисленный код символа – также число от 0 до 255. C не разделяет эти типы, и предлагает `unsigned char` в качестве альтернативы.

### Преобразования типов

Нередко в арифметических выражениях и операторах присваивания одновременно используются переменные различных типов данных. В таких случаях типы данных преобразуются в соответствии с несколькими правилами.

- При присваивании целочисленные типы данных автоматически преобразуются без потери информации в случаях, если переменной, занимающей больше байт в памяти, присваивается переменная, занимающая в памяти меньше байт (например, переменная типа `char` присваивается переменной типа `int`).



- Вещественные типы данных автоматически преобразуются без потери информации в случаях, если переменной, обеспечивающей большую точность, присваивается переменная, обеспечивающая меньшую точность (например, переменная типа `double` присваивается переменной типа `long double`).
- Автоматическое преобразование без потери информации осуществляется, если вещественной переменной присваивается переменная целого типа (например, переменная типа `int` присваивается переменной типа `double`).
- В случае, если переменная, занимающая больше памяти или имеющая большую точность присваивается переменной, занимающей меньше памяти или имеющей меньшую точность, а также, если в присваивании участвуют одновременно знаковая и беззнаковая переменные, возможна потеря или искажение информации.
- В некоторых случаях требуется осуществлять “принудительное” преобразование, *приведение типов*. Синтаксис операции: `a=(тип_приведения)b;`. При этом переменной `a` присваивается переменная типа “тип\_приведения”, содержащая значение `b`.
- Правила автоматического преобразования типов могут немного отличаться в зависимости от компилятора и платформы.

Приведение типов нужно использовать обдуманно, т.к. оно может привести к искажению данных. Например, текст:

```
unsigned int i;
unsigned char c;
i=0x0102; //i=258
c=(unsigned char)i; // c=2 (02h)
```

приведет к искажению значения, т.к. переменная типа `unsigned char` позволяет хранить целое число от 0 до 255, а в приведенном примере переменной `c` присваивается значение 258. В результате выполнения программы `c` сохранит младший байт `i` – 02h.

### Массивы

Для объявления одномерного массива в С используется синтаксис: тип элементов массива, имя массива, количество элементов массива, заключенное в квадратных скобках (таблица 2.4).

Объявление массива С	Объявление массива Pascal
<code>signed int P[100];</code>	<code>P :Array [0..99] Of Integer;</code>

Таблица 2.4 Объявление одномерного массива из 100 элементов в С и Pascal.

Индексация элементов массива в С **всегда** начинается с 0, поэтому первый элемент всегда имеет номер 0, а последний – номер на 1 меньше, чем указанное при объявлении массива количество элементов (листинг 2.1).

```

...
signed int swap, data[200], index;
...
data[0]=33;
...
index=199;
swap=data[index];
...

```

Листинг 2.1 Пример работы с одномерным массивом в С. Объявляются скалярные переменные `swap` и `index` и массив `data` типа `signed int`. Первому элементу массива `data` присваивается значение 33, значение последнего элемента массива сохраняется в переменной `swap`.

Обратите внимание, что объявление массива возможно в одной строке со скалярными переменными. В той же строке можно объявить несколько массивов, элементы которых имеют одинаковый тип.

Для объявления многомерных массивов используется аналогичная запись, в которой количество элементов в каждой размерности указывается в квадратных скобках. Объявление и пример работы с многомерным массивом С, а также аналогичные примеры на языке Pascal приведены в таблице 2.4.

язык С	язык Pascal	Комментарий
<code>double Wave[5][10];</code>	<code>Wave :Array[0..4,0..9] Of Double;</code>	Объявление 2-мерного массива
<code>Wave[2][9]=3.3;</code>	<code>Wave[2,9]:=3.3;</code>	Элементу массива присваивается значение
<code>char F[10][10][20];</code>	<code>F :Array[0..9,0..9,0..19] Of ShortInt;</code>	Объявление 3-мерного массива
<code>val=F[i][j][k];</code>	<code>val:=F[i,j,k];</code>	Переменной присваивается значение, хранящееся в массиве

Таблица 2.4 Примеры объявления многомерных массивов и работы с ними.

Стандарт ANSI C обязует разработчиков компиляторов обеспечить возможность задания максимальной размерности массива не ниже 12.

Строки в С представляют собой одномерные массивы элементов `unsigned char`. Работа со строками будет рассмотрена далее.

### Пользовательские типы данных

Pascal предоставляет программисту возможность определения “пользовательских типов данных”, представляющих собой *псевдонимы* (или *синонимы*) встроенных типов данных. В С аналогичные возможности реализуются с помощью ключевого слова `typedef` (от англ. *type definition* – задание типа). Синтаксис использования:

`Typedef имя_типа Имя_пользовательского_типа;`

Например: `typedef signed int Integer;` определяет тип `Integer`, который полностью аналогичен `signed int`.

Определение типов часто используется для того, чтобы дать укороченное имя известному типу данных. В таблице 2.5 приведены примеры определения синонимов имен типов на языках С и Pascal.

Язык C	Язык Pascal
<pre>typedef unsigned char U8; typedef signed int S16; typedef signed long int S32; typedef double TRealType;  S32 Count; RealType Mean;</pre>	<pre>TYPE     U8   = Byte;     S16  = Integer;     S32  = LongInt;     TRealType = Double;  VAR     Count :S32;     Mean  :TRealType;</pre>

Таблица 2.5 Определение пользовательских синонимов имен типов в C и Pascal.

Синонимы типа U8 – от unsigned 8 bit или S32 – от signed 32 bit обладают несколько худшей информативностью, зато гораздо компактнее.

Правилом хорошего тона при использовании typedef считается использование в качестве имен пользовательских типов идентификаторов, начинающихся с заглавной буквы. Часто в качестве первой буквы пользовательского типа данных используют T от Type – тип.

Понятно, что использование typedef никак не влияет на результирующий код исполняемого модуля или библиотеки. Вместе с тем, грамотное использование typedef может заметно облегчить работу программиста.

### Структуры

*Структурой (агрегатом)* (англ. structure) в языке C называют конструкцию языка, объединяющую (*инкапсулирующую*) под одним именем переменные разных типов. Эквивалентом структуры C является конструкция Record (запись) в Pascal см. таблицу 2.6.

Структуры часто используют для:

- чтения/записи разнородных данных из/в файл;
- взаимодействия с периферийным оборудованием (например, передача блока разнородных конфигурационных параметров периферийному устройству);
- построения *связанных списков*,
- логической группировки переменных разных типов и др.

Язык C	Язык Pascal
<pre>typedef struct {     unsigned char channel;     unsigned char mode;     float amplifier; } ADCChannel;  typedef struct {     ADCChannel channels[16];     unsigned int count;     unsigned char enabled[16]; } ADCChannels;  ... ADCChannel ADC0; ADCChannel ADC[16]; ADCChannels channels; ... void main(void){     ADC0.Channel=0;</pre>	<pre>TYPE     ADCChannel = Record         Channel   :Byte;         Mode      :Byte;         Amplifier  :Single;     end;      ADCChannels = Record         Channels   : Array [0..15] Of ADCChannel;         Count      : Word;         Enabled    : Array [0..15] Of Boolean;     end;  ... VAR     ADC0      :ADCChannel;     ADC       :Array[0..15] Of ADCChannel;     Channels  :ADCChannels;  ... BEGIN     ADC0.Channel:=0;</pre>

<pre> ADC0.Mode=1; ADC0.Amplifier=10.0; ... ADC[5]=ADC0; ADC[k].Mode=ADC[0].Mode; channels.channels[2]=ADC[15]; ... </pre>	<pre> ADC0.Mode:=1; ADC0.Amplifier:=10.0; ... ADC[5]:=ADC0; ADC[k].Mode:=ADC[0].Mode; Channels.Channels[2]:=ADC[15]; ... </pre>
--	---

Таблица 2.6 Объявление и использование переменной типа “структура” в С и Pascal.

Идентичные структуры могут быть объявлены несколькими немного отличными способами (таблица 2.7).

1	2	3
<pre> typedef struct {     unsigned char channel;     unsigned char mode;     float amplifier; } ADCChannel;  ... ADCChannel ADC0; ADCChannel ADC[16]; ... </pre>	<pre> struct ADCChannel {     unsigned char channel;     unsigned char mode;     float amplifier; };  ... struct ADCChannel ADC0,                         ADC[16]; ... </pre>	<pre> struct {     unsigned char channel;     unsigned char mode;     float amplifier; } ADC0, ADC[16]; </pre>

Таблица 2.7 Варианты описания структуры и объявления переменных типа структура.

В таблице 2.7 приведены 3 варианта описания одной и той же структуры, хранящей параметры одного канала многоканального аналого-цифрового преобразователя (АЦП, англ. ADC от Analog to Digital Converter). При использовании варианта объявления 2 “ADCChannel” – имя структуры часто называют *именем-этикеткой*.

На практике могут использоваться все 3 варианта описания, но чаще всего используют вариант 1 (см. пример в таблице 2.6). Второй вариант описания более громоздкий, третий вариант описания самый компактный, но при его использовании не создается имя-этикетка структуры, что часто неудобно.

Хорошим тоном при задании имени-этикетки считается использование первой заглавной буквы в имени этого идентификатора.

Переменные, входящие в состав структуры, часто называют *полями* (англ. field) *структуры*. В качестве полей могут выступать переменные всех основных типов, используемых в С, переменные типов, определенных пользователем, массивы и другие структуры.

В примере, представленном в таблице 2.6, описываются 2 структуры. ADCChannel, содержащая переменные для описания параметров одного канала АЦП (channel – номер канала, mode – режим работы канала, amplifier – коэффициент усиления предусилителя данного канала). Структура ADCChannels содержит описывание всех каналов данного АЦП (count – количество используемых каналов, каждый *i*-ый элемент массива enabled содержит логическое значение “истина” в случае, если канал *i* включен, “ложь” – в противном случае, каждый *i*-ый элемент массива channels представляет собой структуру типа ADCChannel, содержащую конфигурационные параметры *i*-го канала АЦП).

Доступ к переменной, включенной в структуру осуществляется по имени этой переменной. Для доступа используется оператор “точка”: “.”, которая ставится между именем переменной типа структура и именем поля этой переменной. Примеры обращения к различным полям структур рассмотрены в примере из табл. 2.6.

### **Перечисления, объединения, битовые поля**

В С введены такие конструкции языка, как *перечисления* `enum`, *объединения* `union` и *битовые поля* (англ. `bits field`).

Перечисления (*перечислимые константы*) являются в некотором смысле упрощенным аналогом конструкции “множество”, “`set`” в Pascal. Обычно их используют для определения массивов целочисленных констант.

Объединения можно рассматривать, как записи, в которых под хранение различных переменных отводится в памяти компьютера одно и то же место. Объединения используются, в основном, для экономии памяти.

Основное назначение битовых полей – также экономия памяти ЭВМ. Битовые поля представляют собой структуры с явно указанным количеством битов, выделяемым для хранения переменной. Например, в листинге из таблицы 2.6 рассматриваются структуры, описывающие параметры АЦП. Т.к. число каналов АЦП – 16 известно заранее, то понятно, что для хранения номера канала можно отвести 4 бита, а не 8 (переменная `channel` структуры `ADCCChannel` имеет тип `unsigned char`). Используя конструкцию “битовое поле” и такую априорно известную информацию о количестве каналов устройства, можно хранить информацию о конфигурации канала АЦП более компактно. Вместе с тем, злоупотребление использованием битовых полей не рекомендуется, т.к. их реализация аппаратно-зависима и отличается для разных процессоров.

### Лекция 3 Основные операторы и конструкции языка

Арифметические операторы. Операторы инкремента и декремента. Операторы условия if/else, switch. Логические операции. Циклы. Безусловный переход. Побитовые операторы. Доступ к отдельным битам.

#### Арифметические операторы

С оператором присваивания С “=”. Знакомство уже состоялось в предыдущих лекциях. Основные арифметические операции С сведены в таблице 3.1.

Название операции	Операция Pascal	Операция С	Пример С	Тип результата и аргументов С
Плюс	+	+	c=a+b	Целые или вещественные числа
Минус	-	-	c=a-b	Целые или вещественные числа
Унарный минус	-	-	c=-a	Целые или вещественные числа
Умножение	*	*	c=a*b	Целые или вещественные числа
Вещественное деление	/	/	c=a/b	Хотя бы один из аргументов (a или b) вещественное число
Целочисленное деление	div	/	c=a/b	Оба аргумента (a и b) целые числа. Дробная часть результата отбрасывается без округления.
Взятие по модулю (остаток от деления)	mod	%	c=a%b	Оба аргумента (a и b) целые числа. Возвращает остаток от деления a на b.

Таблица 3.1 Основные арифметические операции, определенные в С.

Видно, что система базовых арифметических операций С близка в Pascal. Нужно обратить внимание, что в С, в отличие от Pascal, нет специализированной операции целочисленного деления (аналога “div”). Усечение результата операции “/” до целого производится в случае, если оба аргумента – целые числа. Этот нюанс может вызвать появление в программе логических ошибок и должен быть учтен.

При вычислении арифметических выражений действуют правила приоритета операций аналогичные Pascal. Для изменения приоритета операций используются скобки “( )” (англ. parenthesis), в т.ч. вложенные (англ. nesting parenthesis). Наивысшим приоритетом обладают скобки (наиболее глубоко вложенные), затем унарный “-”, “\*”, “/”, “%”, затем бинарный “-” и “+”. Рассмотрим для примера вычисление арифметического выражения:

$$c = (1+2) * (3 + (6+4) / 2).$$

Логика его синтаксического разбора будет следующая:

1.  $c = (1+2) * (3 + 10 / 2)$  – вычислили выражение в наиболее глубоко вложенных скобках;
2.  $c = 3 * (3 + 5)$  – 2 пары скобок имеют одинаковый уровень вложенности, поэтому имеют одинаковый приоритет – вычисляем выражения в них слева направо (операция деления имеет высший приоритет по сравнению со сложением);
3.  $c = 3 * 8$  – скобки имеют наивысший приоритет;
4.  $c = 24$  – выражение вычислено.

Если в выражении участвует несколько операций с одинаковым приоритетом, то операции выполняются слева направо. Неправильный учет правил приоритета может стать причиной ошибок. Например, нужно вычислить значение выражения:

$$df = \frac{Fs}{2N},$$

если  $F_s=1000$ ,  $N=500$ , то  $df=1$ . Если попытаться запрограммировать вычисление этого выражения, как:  $df=F_s/2*N$ ; то результат вычисления будет 250000, т.к. деление и умножение имеют равные приоритеты, и операция будет выполнена слева направо. Корректный результат будет получен, если записать:  $df=F_s/(2*N)$ .

Общим правилом для уменьшения вероятности появления подобных ошибок является использование скобок во всех “сомнительных случаях”.

Создатели языка С были сторонниками лаконичного стиля программирования. Язык содержит дополнительные операторы присваивания, ускоряющие набор исходного текста. Такие операторы укороченной записи и полные варианты записи выражений приведены в таблице 3.2.

Оператор укороченной записи	Пример использования	Обычная запись
$+=$	$a+=b$	$a=a+b$
$-=$	$a-=b$	$a=a-b$
$*=$	$a*=b$	$a=a*b$
$/=$	$a/=b$	$a=a/b$
$\%=$	$a\%=b$	$a=a\%b$

Таблица 3.2 Операторы укороченной записи арифметических выражений.

Операторы в укороченной записи рекомендуются к использованию, т.к. для некоторых систем их использование может повысить быстродействие работы программы.

### Операторы инкремента и декремента

Широко используются в С унарные операторы инкремента и декремента (см. таблицу 3.3). Чаще всего их используют со счетчиками циклов и с аргументами функций.

Название оператора	Пример использования	Комментарий
Инкремент в префиксной форме	$++a$	Увеличивает целочисленную переменную $a$ на 1, затем использует результат в выражении
Инкремент в постфиксной форме	$a++$	Использует целочисленную $a$ в выражении, затем увеличивает $a$ на 1
Декремент в префиксной форме	$--a$	Уменьшает целочисленную переменную $a$ на 1, затем использует результат в выражении
Декремент в постфиксной форме	$a--$	Использует целочисленную $a$ в выражении, затем уменьшает $a$ на 1

Таблица 3.3 Унарные операторы инкремента и декремента.

Другое название 4 этих операций: *преинкремент*, *постинкремент*, *предекремент*, *постдекремент*, соответственно. Аналогом операций преинкремента “++” и предекремента “--” в Pascal являются функции “Inc” и “Dec”, соответственно.

Результат применения операторов в префиксной и постфиксной формах записи будет отличаться только при использовании этих операторов в выражениях. Преинкремент переменной увеличит ее значение перед ее использованием в выражении, постинкремент – сразу после выражения. Для примера в таблице 3.4 приводятся участки листингов программы, использующей операции пре- и постинкремента.

Нужно помнить, что использовать операторы “++” и “--” можно только с простыми переменными целых типов. Попытка скомпилировать текст, содержащий, например: “c=++ (a-b) \*c+5;” вызовет ошибку.

Постинкремент	Преинкремент
<pre>... int a,b;  a=0; a++; b=a; //b=1 printf("%i", a++); //На экране “1” b=a; //b=2 ...</pre>	<pre>... int a,b;  a=0; ++a; b=a; //b=1 printf("%i", ++a); // На экране “2” b=a; //b=2 ...</pre>

Таблица 3.4 Примеры использования пре- и постинкремента.

На многих системах, в частности, на большинстве IBM-совместимых ПК, операции инкремента и декремента работают быстрее, чем выражения типа a=a+1 или b=b-1 благодаря встроенной в соответствующие процессоры аппаратной поддержке операций инкремента и декремента. Поэтому, везде, где возможно, рекомендуется использовать именно эти операции.

### Операторы условия if/else, switch

C предлагает несколько конструкций, управляющих ходом выполнения алгоритма: конструкция “if else”, “else if” и оператор множественного выбора switch (аналог “case of” Pascal).

Формат использования оператора условия if в простейшем случае: if (УСЛОВИЕ) КОМАНДА;. Где УСЛОВИЕ – целочисленная константа, переменная или арифметическое выражение, которое будет интерпретироваться оператором if, как логическая переменная (подробнее логические операции в C рассматриваются в следующем разделе). В случае если нужно выполнить несколько команд, используют составной оператор “{ }”. Оператор “if-else” в общем случае имеет вид:

```
if (УСЛОВИЕ) {
    КОМАНДА1;
    ...
    КОМАНДА10;
}
else {
    КОМАНДА30;
    ...
    КОМАНДАН;
}
```

Примеры использования условных операторов в C и Pascal приведены в таблице 3.5. Обратите внимание, что в рассмотренном примере в третьем условном операторе условие записано, как if (c). Действительно, т.к. C интерпретирует ненулевые целочисленные выражения в качестве логической “Истины”, то такая запись означает “выполнить, если значение переменной c отлично от 0”.

Нужно также обратить внимание на типичную ошибку программистов, переходящих на C с языка Pascal в конструкции вида:

```
if (УСЛОВИЕ)
    КОМАНДА1;
else
```



КОМАНДА2;

перед else необходимо ставить “;”. В Pascal постановка “;” перед else в аналогичной конструкции вызывает ошибку компиляции.

Язык C	Язык Pascal
<pre>signed int a,b,c; ... if(a&gt;0) b=a; ... if(b&lt;0) {     a=5+c;     c=10; } ... if(c) {     a=b/10;     c=0; } else {     a=b*5;     c=a; }</pre>	<pre>a, b, c :Integer; ... If a&gt;0 then b=a; ... If b&lt;0 then begin     a:=5+c;     c:=10; end; ... If c&lt;&gt;0 then begin     a:=b div 10;     c:=0; end else begin     a:=b*5;     c:=a; end;</pre>

Таблица 3.5 Примеры использования операторов условия в C и Pascal.

Условный оператор “else if” используется в конструкциях множественного выбора – выбора одного варианта из многих. Формат использования оператора:

```
if (УСЛОВИЕ1) {
    КОМАНДА1;
    ...
}
else if (УСЛОВИЕ2) {
    КОМАНДА2;
    ...
}
...
else {
    КОМАНДАН;
    ...
}
```

Т.е. если УСЛОВИЕ 1 “Ложь”, то проверяется УСЛОВИЕ 2 в первой конструкции “else if”, если и оно “Ложь”, то УСЛОВИЕ 3 во второй конструкции “else if” (если она предусмотрена) и т.д. Если одно из условий в конструкции if или одной из конструкций “else if” окажется “Истина”, начнется выполнение блока команд для этой конструкции, если же все условия окажутся “Ложь”, то будут выполнены команды раздела else.

Pascal не имеет конструкции, эквивалентной “else if” в C, вместе с тем, используя несколько последовательно записанных конструкций “if-else” можно запрограммировать аналогичный алгоритм, хотя он будет более громоздким. Примеры реализации множественного выбора в C и Pascal приведены в таблице 3.6.

Для улучшения читаемости программ в соответствии с “правилами хорошего тона” рекомендуется вставлять в текст программы по 1 пустой строке перед и после условного оператора, кроме того, настоятельно рекомендуется использовать отступы шириной 2-4 символа перед всеми командами и вложенными конструкциями языка, содержащимися в разделе команд условных операторов.

Язык С	Язык Pascal
<pre>signed int a,b,c; ... if(a&lt;0) {     b=2*a;     c+=2; } else if(a&lt;=10) { //0&lt;a&lt;=10     b=10*a;     c+=10; } else if(a&lt;=20) { //10&lt;a&lt;=20     b=20*a;     c+=20; } else { //a&gt;20     b=0;     c=0; }</pre>	<pre>a, b, c :Integer; ... If a&lt;0 then begin     b:=2*a;     c:=c+2; end Else     If a&lt;=10 then begin {0&lt;a&lt;=10}         b:=10*a;         c:=c+10;     end     Else         If a&lt;=20 then begin {10&lt;a&lt;=20}             b:=20*a;             c:=c+20;         end         Else begin {a&gt;20}             b:=0;             c:=0;         end end;</pre>

Таблица 3.6 Пример использования “else-if” и аналогичный текст на Pascal.

Оператор switch С является аналогом “Case Of” Pascal. Формат использования:

```
switch (ВЫРАЖЕНИЕ) {
    case ЗНАЧЕНИЕ_ВЫРАЖЕНИЯ1: КОМАНДА1; break;
    case ЗНАЧЕНИЕ_ВЫРАЖЕНИЯ2: КОМАНДА2; break;
    ...
    case ЗНАЧЕНИЕ_ВЫРАЖЕНИЯN: КОМАНДАН; break;
    default: КОМАНДАL;
}
```

Пример использования этого оператора представлен в таблице 3.7.

Язык С	Язык Pascal
<pre>signed int a,b,c; ... switch (a){     case -1: b=1; break;     case 0:         b=0;         c*=100;         break;     case 1: b=2; break;     default:         b=-1; }</pre>	<pre>a, b, c :Integer; ... Case a Of     -1: b:=1;     0: begin         b:=0;         c:=100*c;     end;     1: b:=2;     Else         b:=-1; end;</pre>

Таблица 3.7 Пример использования оператора switch и аналогичный пример с “Case Of”.

Оператор break останавливает выполнение команд внутри конструкции switch и переходит на выполнение команд следующих в программе за этой конструкцией. Если break не использовать, то будут выполнены все команды, следующие за точкой входа в оператор switch. Пропуск в операторе switch команды break является типичной ошибкой программистов.

## Логические операции

Основное назначение логических выражений – управление циклами и ветвлениями алгоритма (например, в конструкциях “if-else”).

Как уже упоминалось, язык C, в отличие от Pascal, не имеет специального логического (“булевского”) типа данных. В качестве логических выражений для управления циклами и условными операторами используются обычные целочисленные константы, переменные или арифметические выражения. При этом если значение такого целочисленного выражения равно 0, то с точки зрения логических операций ему соответствует логическое значение “Ложь” (“False”), если значение ненулевое – то “Истина” (“True”).

Построение логических выражений осуществляется с помощью операторов отношения и логических операторов. Список этих операторов и их соответствие операторам Pascal приводится в таблице 3.8.

Название оператора C	Язык Pascal	Язык C	Пример C
<b>Операторы отношения</b>			
Равенство	=	==	if (a==b)
Неравенство	<>	!=	if (a!=b)
Больше	>	>	if (a>b)
Больше или равно	>=	>=	if (a>=b)
Меньше	<	<	if (a<b)
Меньше или равно	<=	<=	if (a<=b)
<b>Логические операторы</b>			
Логическое отрицание, логическая инверсия	NOT	!	if (!a)
Логическое “И”, логическая конъюнкция, логическое умножение	AND	&&	if (a&&b)
Логическое “ИЛИ”, логическая дизъюнкция, логическое сложение	OR		if (a  b)

Таблица 3.8 Операторы, использующиеся при построении логических выражений для управления условными циклами и ветвлениями.

Рассмотрим типичный пример использования логических выражений (таблица 3.9). Пусть имеется массив x, содержащий N элементов. Нужно запрограммировать условие, которое, работая в цикле, будет проверять, не выходит ли значение элемента за диапазон допустимых значений от 80 до 90.

Язык C	Язык Pascal
<pre> unsigned int i, N; float x[1000]; ... if ((i&lt;N) &amp;&amp; ((x[i]&lt;80)    (x[i]&gt;90))) {     printf("Out of band!"); } ... </pre>	<pre> i, N :Integer; x      :Array [0..999] Of Single; ... If (i&lt;N) AND ((x[i]&lt;80) OR (x[i]&gt;90)) then begin     Write('Out of band!'); end; ... </pre>

Таблица 3.9 Пример использования логических выражений. Проверка выхода элемента массива за допустимый диапазон значений.

Логическое выражение обрабатывается слева направо, причем, если результат обработки станет известен до его полной обработки, то процесс обработки останавливается. Например, в приведенном в таблице 3.9 алгоритме первой в логическом выражении стоит проверка превышения значением счетчика i максимального индекса

массива  $x$ :  $N-1$ . Если выражение  $i < N$  ложно, то операция логического “И” заведомо обратит в “Ложь” все выражение независимо от того, “Истина” или “Ложь” вторая половина условия. Поэтому, вычисление второй половины выражения не произойдет. В данном примере этот факт учтен при построении логического условия, т.к. если бы вторая половина выражения вычислялась, то при  $i \geq N$  могла произойти ошибка выхода за границы массива  $x[i]$ .

Логические выражения в С рассматриваются как арифметические операции с целыми числами. Например, допустимы непосредственные присваивания вида:

```
signed int a, b;
...
a=1>2;    //a=0
a=3<=10;  //a=1
a=5;
b=0;
a=a&&(b||4); //a=1
...
```

Эта особенность может оказаться удобной при отладке сложных логических выражений, управляющих условными операторами или циклами.

Т.к. логические выражения С рассматриваются компилятором, как арифметические, то, в частности, допустимо использования присваивания непосредственно в логическом выражении. Приведенный выше пример проверяет, делится ли нацело (остаток от деления  $=0$ ) число  $x$  на 5. Если число не кратно 5, то на экран выводится остаток от деления числа на 5, если кратно, то информация не выводится:

```
signed int x, m;
...
if (m=x%5) printf("mod=%i", m);
```

Важно отметить, что т.к. операция присваивания в С в логических выражениях разрешена, то частой ошибкой программистов является использование “=” вместо “==”.

### Циклы

С предоставляет программисту 3 типа циклов: while, “do-while” и for. Цикл while эквивалентен конструкции “while-do” языка Pascal. Формат использования: while (УСЛОВИЕ) КОМАНДА; , где КОМАНДА может быть набором команд внутри составного оператора “{ }”.

Пример использования цикла для инициализации массива  $x$ , содержащего  $N$  элементов, приведен в таблице 3.10.

Язык С	Язык Pascal
<pre>... i=0; while (i&lt;=N-1) {     x[i]=0;     i++; } ...</pre>	<pre>... i=0; While i&lt;=N-1 do begin     x[i]:=0;     Inc(i); end; ...</pre>

Таблица 3.10 Пример использования цикла while для инициализации массива.

Команды, размещенные в теле цикла “do-while” всегда выполняется хотя бы один раз. Формат этой конструкции языка:

```
do {
    КОМАНДА1;
    ...
```

```

    КОМАНДАН;
} while (УСЛОВИЕ);

```

Цикл “do-while” эквивалентен циклу “REPEAT-UNTIL” в Pascal в записи:

```

REPEAT
    КОМАНДА1;
    ...
    КОМАНДАН;
UNTIL NOT УСЛОВИЕ;

```

Цикл “do-while” используется на практике гораздо реже while и for, хотя иногда может быть удобен.

Цикл for в C позволяет реализовывать конструкции, аналогичные циклу For Pascal и предоставляет дополнительную функциональность. В общем случае формат использования for:

```

for (ИНИЦИАЛИЗАЦИЯ_СЧЕТЧИКА; УСЛОВИЕ; ИНКРЕМЕНТ_СЧЕТЧИКА) {
    КОМАНДА1;
    ...
    КОМАНДАН;
}

```

В таблице 3.11 представлен пример использования цикла for для решения задачи инициализации массива, рассмотренной в примере из таб. 3.10.

Язык C	Язык Pascal
<pre> ... for (i=0; i&lt;=N-1; i++) {     x[i]=0; } ... </pre>	<pre> ... For i:=0 to N-1 do begin     x[i]:=0; end; ... </pre>

Таблица 3.11 Пример использования цикла for для инициализации массива.

В языке C for обладает важными дополнительными возможностями относительно аналогичной конструкции Pascal:

- в разделе управления циклом ИНКРЕМЕНТ\_СЧЕТЧИКА счетчик может меняться произвольно (например, увеличиваться на произвольный шаг), Pascal предоставляет возможность увеличения (уменьшения) значения счетчика только на 1 за одну итерацию цикла;
- значение счетчика может быть изменено в теле цикла;
- значение счетчика не теряется после завершения цикла.

Циклы for допускают использования в разделах ИНИЦИАЛИЗАЦИЯ\_СЧЕТЧИКА и ИНКРЕМЕНТ\_СЧЕТЧИКА оператора “,” (запятая, англ. comma) для множественной индексации. Например, если имеются массивы данных x и y, содержащие N и M значений, соответственно:

```

for (i=0, j=N-1; (i<=N-1) && (j>=0); i++, j--) {
    z[i]=y[j]-x[i];
}

```

Для улучшения читаемости программ злоупотребление множественной индексацией не рекомендуется, вместе с тем, в ряде задач такая возможность оказывается полезной.

Любой из 3 управляющих разделов цикла (и даже все 3 сразу) может быть опущен (операторы “;” опускать нельзя), например, в форме:

```

for (; УСЛОВИЕ;) {
    КОМАНДА1;
    ...
}

```

```

    КОМАНДАН;
}
цикл for полностью эквивалентен while:
while (УСЛОВИЕ) {
    КОМАНДА1;
    ...
    КОМАНДАН;
}

```

Если в for не заполнен управляющий раздел УСЛОВИЕ, то компилятор предполагает, что УСЛОВИЕ всегда равно “Истина”. Таким образом, “вечный” цикл в С можно создать несколькими способами:

No	Язык С	Язык Pascal
1.	while(1) { ... }	While True do begin ... end;
2.	for(;;){ ... }	-
3.	for(;;){ ... }	-

Таблица 3.12 Примеры организации “вечного” цикла в С.

Для управления циклами С имеет 2 дополнительных оператора: break и continue.

Оператор break вызывает немедленный выход из цикла, в котором он был вызван, т.е. переход на выполнение оператора следующим непосредственно после конца цикла. Таким образом, если есть  $N$  вложенных друг в друга циклов, то для выхода из всей этой конструкции нужно вызвать break  $N$  раз (по одному разу в каждом из циклов, начиная с наиболее глубоко вложенного).

Оператор continue обеспечивает немедленный переход к началу цикла, например, вечным будет цикл:

```

i=0;
while(i<=10){
    continue;
    i++;
}

```

т.к. инкремент выполниться не сможет.

### Безусловный переход

*Безусловный переход (переход на метку, англ. jump) оформляется, как:*

```

...
goto МЕТКА;
...
МЕТКА:
...

```

где МЕТКА представляет собой идентификатор, записанный по обычным правилам С (должен начинаться с латинской буквы, может содержать цифры, не содержит символов пробела и т.п.).

Встретив оператор goto МЕТКА программа немедленно переходит на выполнение команды, следующей за меткой (англ. label) МЕТКА:. Переход может осуществляться в пределах одной функции.

Доказано, что оператор безусловного перехода в языках высокого уровня является избыточным, т.е. не существует алгоритмов при реализации которых без него нельзя было бы обойтись. “Правила хорошего тона” рекомендуют его избегать, для улучшения читаемости программ, вместе с тем, goto может оказаться полезен, например, для быстрого выхода из нескольких глубоко вложенных друг в друга циклов. Кроме того, оператор goto работает очень с большинством процессоров, поэтому, его применение может оказаться целесообразно для оптимизации кода по быстродействию для решения критических по времени задач, например, обработки данных в реальном времени на базе маломощного микропроцессора.

### Побитовые операторы

Язык C предоставляет программисту набор операторов, позволяющих работать с ячейками памяти, как с наборами отдельных битов. Такие операторы, называемые *побитовыми* (англ. bitwise), сведены в таблице 3.13.

№	Название оператора C	Язык Pascal	Язык C	Пример C	a=
1.	Побитовая инверсия	NOT	~	a=~a;	126 01111110b
2.	Побитовое “И”, побитовое умножение	AND	&, &=	a=a&1; или a&=1;	1 00000001b
3.	Побитовое “ИЛИ”, побитовое сложение	OR	,  =	a=a 2; или a =2;	131 10000011b
4.	Побитовое “исключающее ИЛИ”	XOR	^, ^=	a=a^1; или a^=1;	128 10000000b
5.	Сдвиг влево	SHL	<<, <<=	a=a<<2; или a<<=2;	4 00000100b
6.	Сдвиг вправо	SHR	>>, >>=	a=a>>3; или a>>=3;	16 00010000b

Таблица 3.13 Побитовые операторы в полной и укороченной записи и результат их применения для a=129 (10000001b).

1. Побитовая инверсия является унарной операцией. Она заменяет в двоичном представлении числа все 0 на 1 и все 1 на 0, т.е. *инвертирует* (англ. invert) биты числа.
2. Побитовое “И” – это бинарная операция. Бит результирующей переменной будет содержать 1 тогда и только тогда, когда единицы содержатся в обоих соответствующих битах операндов.
3. Побитовое “ИЛИ” – это бинарная операция. Бит результирующей переменной будет содержать 1 в случае, если 1 равен соответствующий бит хотя бы в одном из операндов. Отличие побитового “ИЛИ” от арифметического состоит в том, что при побитовом “ИЛИ” перенос бита в старший разряд никогда не осуществляется.
4. Побитовое “исключающее ИЛИ” – бинарная операция. Бит результирующей переменной будет содержать 1 только в случае, если соответствующие биты операндов различные, т.е. один из них равен 0, а другой 1.
5. Сдвиг влево a=x<<N; осуществляет побитовый сдвиг целочисленной переменной x на N бит. При этом, значения N старших битов числа теряются, а N младших битов числа становятся равными 0.
6. Сдвиг вправо a=x>>N; работает аналогично сдвигу влево, за исключением направления сдвига.

Важно понимать отличия побитовых операций от логических. Можно представлять, что перед применением логических операций операнды становятся равными 1, в случае, если их значение было отличным от 0 и равными 0 в противном случае, а логические

операции действуют аналогично побитовым, влияющим только на младший бит см. таблицу 3.14.

Логические операции	Результат	Побитовые операции	Результат
$\bar{A}=!128;$	0	$A=\sim 128;$	127
$a=128\&\&8;$	1	$a=128\&8;$	0
$a=128  0;$	1	$a=128 0;$	128

Таблица 3.14 Сопоставление работы логических и побитовых операций.

### Доступ к отдельным битам

В большинстве случаев процессоры обеспечивают побайтовую адресацию памяти. Однако, нередки случаи, когда необходимо проанализировать или изменить состояние отдельных битов. В этих случаях обращаются к ячейке памяти, размер которой кратен байту, а доступ к отдельным битам обеспечивается с помощью использования побитовых операторов.

Типичными задачами, требующими работы с отдельными байтами, являются, например, взаимодействие с периферийными устройствами через их регистры посредством анализа и изменения состояния битовых флагов и упаковка данных для их компактного хранения и быстрой передачи по каналам связи.

Для организации полного доступа к отдельным битам числа достаточно обеспечить выполнение с битом, номер которого задан, трех операций: проверка (англ. check) состояния бита, установка (англ. set) бита, при которой заданный бит становится равным 1 независимо от исходного состояния и сброс (англ. clear) или маскировка (англ. masking) бита, при которой заданный бит становится равным 0 независимо от исходного состояния.

Три перечисленные операции могут быть выполнены путем сравнения анализируемого числа с маской (англ. mask) – целочисленной константой, значение которой специально подобрано для работы именно с нужным битом.

Например, для установки бита 3 переменной unsigned char a; нужно выполнить:  $a=a|00001000b$ , т.е.  $a=a|8;$  или  $a|=8;$ , т.к. C не имеет встроенной поддержки двоичных констант. Для установки бита 7 нужно выполнить:  $a=a|10000000b$ , т.е.  $a|=128;$  и т.п.

Таким образом, для независимой от остальных битов установки бита с номером N, нужно применить операцию побитового “ИЛИ” с маской, двоичное представление которой содержит 1 только в N-ном бите.

Для независимого от остальных битов сброса (обнуления) бита с номером N, нужно применить операцию побитового “И” с маской, двоичное представление которой содержит 0 только в N-ном бите, например, для сброса бита 3 нужно выполнить:  $a=a\&11110111b$ , т.е.  $a\&=247;$ .

Для проверки состояния бита N можно использовать проверку условия с предварительной маскировкой. Например, для проверки состояния бита 3:

```
a&=8; //a=a&00001000 – предварительная маскировка
if (a!=0){ //Если бит был установлен
    ...
}
else{ //Если бит был сброшен
    ...
}
```

Задание маски в виде константы неудобно, т.к. для работы с разными битами нужно каждый раз рассчитывать соответствующие маски. Поэтому на практике маски



формируют автоматически с помощью операций сдвига. Соответствующие примеры сведены в таблице 3.15:

Действие	Текст программы
Проверить состояние бита N	<pre> x&amp;= (1&lt;&lt;N) ; if (x!=0) { //Если бит был установлен     ... } else{ //Если бит был сброшен     ... } </pre>
Установить бит N	<pre>x  = (1&lt;&lt;N) ;</pre>
Сбросить бит N	<pre>x&amp;=~ (1&lt;&lt;N) ;</pre>

Таблица 3.15 Организация доступа к отдельным битам ячейки памяти x.

#### Лекция 4 Адресация памяти и использование указателей

Двоичная система счисления, бит, байт, слово. Использование различных систем счисления. Память ЭВМ. Адресация и распределение памяти ЭВМ. Использование ОП прикладной программой. Объявление и использование указателей. Разыменование указателей на структуры. Арифметические действия с указателями. Указатели и массивы.

В главе кратко рассматриваются общие вопросы организации оперативной памяти ЭВМ и некоторые распространенные правила использования ОЗУ операционной системой и прикладными программами. В ходе изложения материала допущены некоторые упрощения, в частности, рассматривается только линейная адресация памяти. Несмотря на сделанные упрощения, на практике во многих случаях рассматриваемый материал достаточен для понимания организации работы с памятью ЭВМ.

##### Двоичная система счисления, бит, байт, слово

Элементарные физические ячейки памяти – *триггеры*, используемые в современных ЭВМ, могут иметь только 2 устойчивых состояния: включен и выключен. Если хотят абстрагироваться от физических особенностей устройства ЭВМ и сконцентрироваться на решении логических задач программирования вместо “триггер включен” и “триггер выключен” используют термины “истина” и “ложь”, соответственно. Также для краткости записи используют символические обозначения “1” и “0”, соответственно. Использование записи состояния ячеек памяти с помощью цифр позволяет применять к содержимому памяти ЭВМ арифметические и логические операции, используя привычный математический формализм.

Т.к. для записи состояния элементарной ячейки памяти оказывается достаточным использованием 2 цифр: “0” и “1”, то для внутреннего представления данных в ЭВМ используется *двоичная система счисления* (англ. *binary numeration*). Одна цифра в двоичном представлении называется “бит” (англ. *bit* от *BI*nary *di*giT).

По историческим причинам, минимально адресуемой единицей памяти ЭВМ является *байт* (англ. *byte*), состоящий, обычно, из 8 битов. Поэтому, даже если требуется опросить всего 1 бит данных, физически по шине ЭВМ все равно будет передан целый байт (8 бит), содержащий, в частности и интересующий бит.

Биты в байте принято нумеровать от 0 до 7 и отображать их при табличной записи справа налево. Т.е. *младший бит* (*наименее значимый бит*, англ. *LSB* от *least significant bit*) – бит номер 0 изображают справа, *старший бит* (*наиболее значимый бит*, англ. *MSB* от *most significant bit*) – бит номер 7 – слева (рис. 4.1).

Номер бита	MSB 7	6	5	4	3	2	1	LSB 0
Значение бита	0	1	0	0	0	1	0	1

Рис. 4.1 Графическое изображение байта данных, содержащего число 69. Биты нумеруются справа налево.

При работе со структурами данных в памяти часто используется термин *машинное слово* или *слово процессора* (англ. *CPU word*), под которым понимают максимальное количество битов данных, которые данный процессор может обработать одной инструкцией. Для современных ПК размер машинного слова обычно составляет 32 или 64 бита.

Первые IBM-совместимые ПК снабжались 16 разрядным процессором (т.е. размер слова такого CPU составлял 16 бит), поэтому, термин “слово” (“word”) также употребляется в литературе по отношению к целочисленным переменным, занимающим в памяти 2 байта, безотносительно разрядности процессора, например, язык Pascal имеет одноименный тип данных “Word”.

### Использование различных систем счисления

При работе с аппаратными ресурсами ЭВМ, организации доступа к отдельным битам данных и решении ряда других задач, программистам, обычно, приходится иметь дело с *десятичной* (англ. *decimal*) системой счисления, наиболее удобной для восприятия человеком, *двоичной* (англ. *binary*), использование которой имеет смысл при операциях с отдельными битами данных и *шестнадцатеричной* (англ. *hexadecimal*), удобной для компактной записи двоичных данных.

В литературе часто используют специальные обозначения для того, чтобы подчеркнуть, что константа является двоичным или шестнадцатеричным числом. Двоичные константы помечают префиксом или суффиксом “b”, например: “b1100100” или “1100100b”, с шестнадцатеричными числами используют суффиксы “h” или “H”, например: “64h” или “64H”.

Язык C позволяет в явном виде использовать шестнадцатеричные константы. Для указания на то, что константа является шестнадцатеричным числом, перед ней ставится префикс “0x”, например, инициализация переменной “a” числом 100 в десятичном виде записывается как: “a=100;”, идентичный текст для числа, записанного в шестнадцатеричном виде: “a=0x64;” (т.к. 64h в шестнадцатеричной системе счисления равно 100 в десятичной системе счисления). К сожалению, C не имеет встроенной возможности использования двоичных констант, однако, частично это ограничение удастся обойти. Подробнее, такая возможность будет рассмотрена в практической части курса.

Подробное рассмотрение способов преобразования чисел в разные системы счисления выходит за рамки данного учебного курса. Однако для справки можно использовать таблицу 4.1, которой вполне достаточно для преобразования целых беззнаковых чисел из шестнадцатеричной системы счисления в двоичную и обратно.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Таблица 4.1 Числа от 0 до 15 в десятичной, двоичной и шестнадцатеричной системах счисления.

Из представленной таблицы видны преимущества использования шестнадцатеричной системы счисления: двоичные числа длиной до 4 цифр (полубайт) в шестнадцатеричной записи отображаются всего одной цифрой от 0h до Fh.

## Память ЭВМ

Современные ПК снабжаются несколькими типами памяти, каждый из которых имеет свои особенности организации и использования. Исполняемая программа и константы постоянно хранятся в файле в *энергонезависимой памяти* ЭВМ, например, на накопителе на жестких магнитных дисках (HDD), Flash-накопителе (Flash-drive или SSD), оптическом накопителе (CD-ROM, DVD-ROM) и т.п. Энергонезависимая память обеспечивает надежное хранение информации длительное время (многие годы) независимо от того включен компьютер, или выключен.

Непосредственно после запуска программы она загружается в *оперативную память* (ОП, англ. *RAM, random access memory*), физически реализованную в виде *оперативного запоминающего устройства* (ОЗУ). Далее при выполнении программы процессор считывает последовательность инструкций именно из ОЗУ. В оперативной памяти также могут временно храниться данные, создаваемые программой во время ее работы или загружаемые программой из файлов, хранящихся в энергонезависимой памяти. Оперативная память не может хранить данные после выключения питания ЭВМ, зато ее быстродействие при чтении/записи данных может на 2-3 порядка превышает быстродействие накопителей энергонезависимой памяти. При выполнении наиболее критичных по времени арифметических операций данные из ОЗУ могут быть перенесены в *регистры процессора* (англ. *registers*). Регистры представляют собой ячейки оперативной памяти, расположенные непосредственно на кристалле центрального процессора и их чтение/запись осуществляется еще быстрее, чем при использовании ОЗУ. На основе описанной выше концепции, работают ЭВМ, имеющие архитектуру фон Неймана (по имени математика, разработавшего теоретическую модель функционирования ЭВМ). На основе фон неймановской архитектуры построены практически все современные ПЭВМ.

Наряду с архитектурой фон Неймана на практике используется и другая парадигма, получившая название Гарвардской архитектуры. Гарвардская архитектура предполагает разделение памяти команд и памяти данных. Например, некоторые современные микроконтроллеры, построенные по схеме Гарвардской архитектуры, хранят команды и константы в энергонезависимой памяти (обычно Flash – она допускает относительно высокую скорость чтения), а в оперативной памяти находятся только переменные. При таком подходе программа не загружается в ОЗУ из энергонезависимой памяти, поэтому оперативную память удастся сэкономить. Кроме того, в некоторых случаях применение гарвардской архитектуры позволяет увеличить быстродействие программы за счет возможности одновременной загрузки в ядро процессора команд из Flash-памяти и данных из ОЗУ. Отличия двух упомянутых процессорных архитектур схематически представлены на рис. 4.2.

Кроме указанных типов памяти практически любая ПЭВМ содержит еще несколько уровней сверхбыстрой *кэш-памяти процессора* (англ. *CPU cache memory*), *память видеоадаптера*, *кэш-память накопителя HDD* и др. Однако, в подавляющем большинстве случаев, программист явно взаимодействует только с 3 типами памяти: энергонезависимая память (файловые операции), ОЗУ общего назначения (работа с переменными), регистры процессора (программирование аппаратных ресурсов, оптимизация программы по быстродействию).

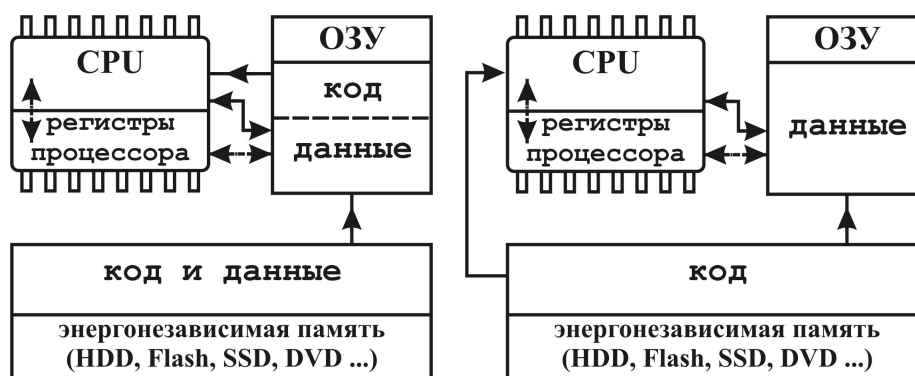


Рис. 4.2 Схематическое сопоставление особенности архитектуры фон Неймана (слева) и гарвардской архитектуры (справа).

### Адресация и распределение памяти ЭВМ

На практике во многих случаях программист может рассматривать ОП как массив байтов: `unsigned char Memory[MemorySize]`, где `MemorySize` – размер ОП в байтах. Говоря о размещении байта в памяти, используют термин *смещение* (англ. *offset*). Смещение представляет собой номер байта в ОП в сплошной нумерации. Смещение байта, можно рассматривать как индекс элемента массива `Memory`. Размещаемые в оперативной памяти переменные могут занимать в ОЗУ несколько последовательных байтов (например, переменные типа `int` занимают 2 последовательных байта), при этом обращения к отдельным байтам таких переменных допустимо. Часто смещение указывают относительно начала некоторой структуры памяти, например, относительно первого байта программы или начала некоторого массива и т.п. В таком случае явно оговаривают, относительно чего записано смещение.

*Адресом* (англ. *address*) переменной (и вообще любой структуры данных) в ОП называют смещение первого байта этой переменной.

Различные операционные системы хранят в ОП свои служебные данные: таблицы векторов прерываний, системный стек, переменные, необходимые для работы ОС и др. Область памяти, предназначенная для хранения такой информации, называется *системной памятью* (англ. *system memory*) и доступ к этой области памяти для прикладных программ ограничен или запрещен. Системная память обычно располагается в начале ОЗУ, начиная с адреса `0h` (рис. 4.3).

Область ОЗУ не занятая системной памятью и загруженными прикладными программами называется *кучей* (англ. *heap*) (рис. 4.3). Куча может использоваться прикладными программами в ходе их выполнения для хранения временных данных.

Операционные системы обычно активно участвуют в распределении памяти для прикладных программ. Для этого они предоставляют специализированное системное программное обеспечение – *менеджер памяти* (англ. *memory manager*).

Менеджер памяти (МП) хранит таблицу с информацией о местоположении и размере в ОП занятых и свободных участков памяти (рис. 4.3). При запуске прикладной программы МП определяет на основе имеющейся таблицы использования памяти адрес в ОЗУ, куда будет загружена программа. Если в ходе выполнения программе требуется память ( $N$  байт), например, для создания массива, она запрашивает МП, МП проверяет, имеется ли в данный момент непрерывный незанятый участок кучи размером не менее  $N$  байт, если имеется, то МП передает программе его адрес  $m$  и отмечает в таблице, что  $N$  байт по адресу  $m$  занято.

Хорошим тоном программирования является освобождение программой памяти, занятой некоторой созданной временной структурой данных, после того, как она станет не нужна. При этом программа должна явно сообщить МП с помощью специальной команды, что  $N$  байт данных по адресу  $m$  больше ей не нужны, МП помечает

соответствующий участок кучи, как свободный и освобожденная память может использоваться другими программами.

Типичной ошибкой программистов является неаккуратное освобождение памяти, занятой временными структурами данных, *динамически создаваемыми* во время выполнения программы. При этом возникает т.н. *утечка памяти* (англ. *memory leakage*), которая приводит к перерасходу памяти из кучи.

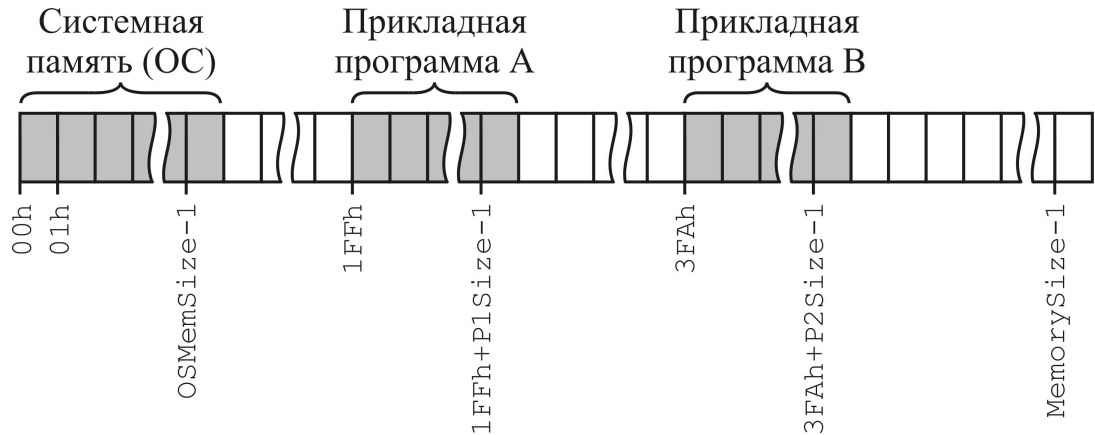


Рис. 4.3 Распределение памяти ЭВМ. OSMemSize – объем оперативной памяти (в байтах), занимаемый операционной системой. P1Size и P2Size – размер (в байтах) загруженных прикладных программ, MemorySize – объем всей имеющейся оперативной памяти (в байтах). Белым цветом отмечены области памяти, использующиеся в качестве кучи.

### Использование ОП прикладной программой

Рассмотрим более подробно, схему использования оперативной памяти прикладной программой. Рассмотрим пример программы на языке Pascal (листинг 4.1). При этом вся информация, приведенная в данном разделе, остается справедливой и для программ, написанных на языке С. Схематически распределение ОП, занимаемой этой программой, представлено на рис. 4.4.

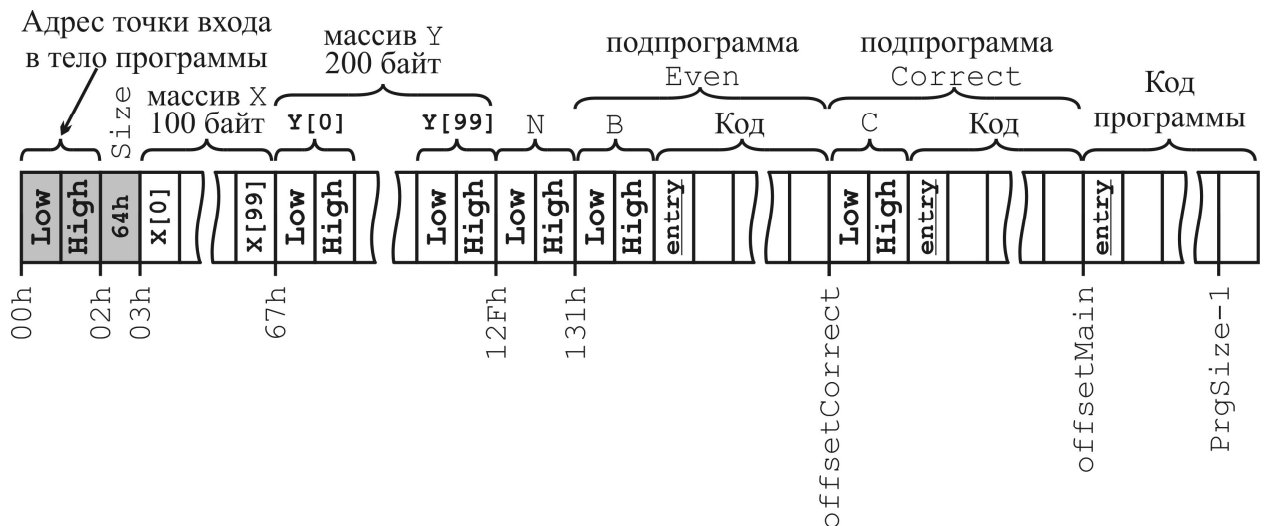


Рис. 4.4 Схема распределения ОП, занятой программой из листинга 4.1. Все смещения указаны относительно начала программы в памяти ЭВМ. offsetCorrect – смещение Correct, offsetMain – смещение начала кода тела программы, PrgSize – размер программы. На рисунке все смещения указаны относительно начала программы в ОП ЭВМ. Словом entry отмечены точки входа в подпрограммы и основную программу. Первое слово (2 байта) в занятой программой ОП содержит адрес точки входа (значение offsetMain), т.е. адрес с которой процессор начнет интерпретировать содержимое ОП как набор инструкций.

Обратите внимание, что сразу после загрузки программы в память будет определено лишь значение константы `Size` (отмечена на рис. 4.4 серым цветом), остальные переменные (локальные и глобальные) будут содержать случайные значения, сохранившиеся в соответствующей области памяти, например, после временного использования этой памяти другой прикладной программой. Поэтому, перед первым использованием переменных их необходимо явно *инициализировать* (англ. *initialize*), явно присвоив им какие-нибудь начальные значения.

При программировании первых ЭВМ программист должен был явно в виде численных констант указывать адреса ячеек ОП, к которым обращалась программа. Это существенно затрудняло и замедляло разработку, приводило к появлению большого количества ошибок. Поэтому, уже в первых компиляторах ассемблера появилась возможность вместо численных констант, определяющих адрес ячейки памяти, использовать символические обозначения имен переменных. На этапе компиляции такие символические имена переменных заменяются адресами ячеек памяти автоматически, причем конкретные значения смещений компилятор может задавать по-разному, пытаясь оптимизировать быстродействие и/или размер кода.

Переменные, память для которых выделяется на этапе загрузки программы, называют *статическими переменными* (англ. *static variable*). Переменные, временно создаваемые во время выполнения программы, называют *динамическими переменными* (англ. *dynamic variable*, *heap variable*). Память для хранения последних выделяется из кучи.

В свете вышесказанного: команда присваивания `N:=300` с точки зрения памяти будет выполняться так. На этапе компиляции компилятор задал адрес статической переменной `N`: `12Fh` (рис. 4.4) относительно начала программы. Число 300 в двоичном виде записывается, как `b00000000100101100`, занимая в памяти 2 байта. Поэтому, присваивание `N:=300` означает, что *младший* (англ. *low*) байт числа 300: `b00101100` будет размещен по адресу `12Fh`, а *старший* байт (англ. *high*): `b00000001` – по адресу `130h`.

```
CONST
    Size = 100;

VAR
    X :Array [0..Size-1] Of Byte;
    Y :Array [0..Size-1] Of Integer;
    N :Word;

Function Even(A:Integer):Boolean;
    {Возвращает True, если A четно}
    Var
        B :Integer;
    Begin
        B:=A;
        If A mod 2=0 then
            Even:=True
        Else
            Even:=False;
    end;

Procedure Correct(VAR A:Integer);
    {Если A четно, вернет 2A, иначе дополнит до четного}
    Var
        C :Integer;
    Begin
        C:=A;
```

```

    If Even(C) then
        A:=2*C
    Else
        A:=C+1;
end;

BEGIN
    N:=300;
    ...
END.

```

Листинг 4.1 Программа на Pascal, использующая подпрограммы, глобальные и локальные переменные.

Представленная в разделе информация, описывает простейший для понимания способ размещения прикладной программы в ОП. Такой способ реализуется в некоторых микроконтроллерах, может применяться для размещения в ОП ПЭВМ компьютерной программы небольшого размера.

При создании прикладных программ для ПЭВМ глобальные переменные и код программы чаще размещаются в разных *сегментах* (англ. *segments*) памяти. С точки зрения программиста это означает, что данные и код могут размещаться в разных местах ОП, не образуя непрерывного блока.

Различные компиляторы могут задавать *выравнивание* (англ. *alignment*) переменных в ОП. Например, *выравнивание по границе слова* означает, что статически создаваемая переменная типа `char` будет занимать в ОП слово (2 байта), причем один из этих байтов, естественно, использоваться никогда не будет. Очевидно, что наиболее экономичное с точки зрения использования памяти выравнивание должно осуществляться по границе байта, как в примере на рис. 4.4, однако выравнивание по границе слова может повысить скорость работы программы. Многие компиляторы С предоставляют возможность управлять выравниванием, при этом, программист сам должен решать возникающие вопросы совместимости форматов структур данных и оптимизации программного кода. Особое внимание на существующие правила выравнивания нужно обращать при использовании структур. Многие компиляторы допускают настройку правил выравнивания элементов структуры. В любом случае, для эффективной и безошибочной работы, разрабатываемой программы необходимо изучить техническую документацию, описывающую архитектуру процессоров для которых создается программа, а также возможности и особенности используемого компилятора.

### Объявление и использование указателей

*Указатель* (англ. *pointer*) это переменная, хранящая адрес переменной. Часть задач не может быть решена без использования указателей, некоторые задачи могут быть решены с помощью указателей более эффективно. В частности, указатели необходимы для создания динамических переменных, построения специальных структур данных, например, связанных списков, эффективной обработки больших блоков данных (передача указателей на массивы в подпрограммы), взаимодействия с периферийным оборудованием и других задач.

ЭВМ обрабатывает указатель как целочисленную беззнаковую переменную (обычно, 2 или 4 байтную).

Указатели в С объявляются с помощью символа “\*”, который ставят перед именем переменной. Например, указатель на переменную типа `char`: `char *pc;`, на переменную типа `int`: `int *pi;`. Используют также нетипизированный указатель `void *p;`, который хранит адрес блока данных произвольного размера в ОЗУ. В конечном счете, обычно, с нетипизированным указателем используют операцию явного приведения типа (см. листинг 4.2).



Для работы с указателями язык C предлагает 2 унарных оператора: “&” – *оператор взятия адреса* и “\*” – *оператор косвенной адресации* или *оператор разыменованье* (англ. *dereferencing*). Пример использования типизированных указателей представлен на рис. 4.5.

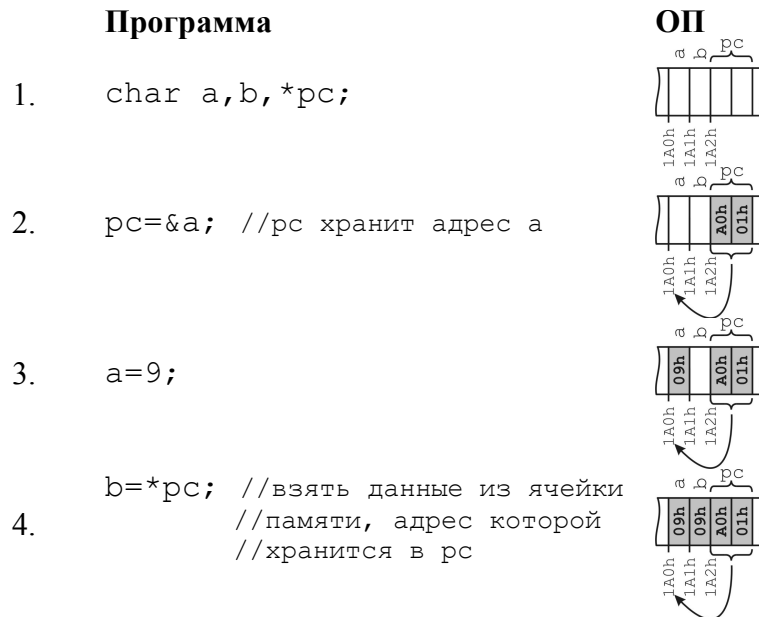


Рис 4.5 Работа с типизированным указателем. Серым цветом отмечены ячейки памяти, значение которых определено.

В 1 строке на рис. 4.5 содержится объявление 2 переменных типа `char` и одной переменной – указателя на `char`. Эти 3 переменные статически размещаются в последовательных ячейках памяти с адресами 1A0h, 1A1h и 1A2h, соответственно (рис. 4.5). Переменная-указатель занимает в памяти 2 байта.

В строке 2 (рис. 4.5) с помощью унарной операции взятия адреса “&”, адрес переменной `a` – 01A0h сохраняется в `pc`. Теперь `pc` *указывает* на `a`.

Команда, записанная в строке 4, читается так: “взять данные, хранящиеся в ячейке памяти, адрес которой содержится в `pc` и сохранить эти данные в `b`”. Операция `*pc` – “взять данные по адресу” – называется *разыменованием* переменной-указателя.

Обращение к ячейке памяти по имени (непосредственное обращение к ячейке памяти с заданным адресом), как в 3 строке рис. 4.5, называют *непосредственной (прямой) адресацией* (англ. *immediate addressing*), обращение к данным, хранящимся в той же ячейке, с помощью указателя называют *косвенной адресацией* (англ. *indirect addressing*).

Несколько примеров работы с указателями приведены в листинге 4.2.

```
unsigned int i,k;
unsigned int *pi; // Объявление указателя на unsigned int
unsigned char c;
unsigned char *pc; // Объявление указателя на unsigned char
void *p; // Объявление нетипизированного указателя

i=0x0103; //b00000000100000011
c=5;
pc=&c; //pc хранит адрес переменной c
pi=&i; //pi хранит адрес переменной i

k=*pi; //k хранит значение 259 (103h)-2 байта данных по адресу pi
k=*pc; //k хранит значение 5-1 байт данных по адресу pc
```

```

pc=(unsigned char *)pi; //pc хранит тот же адрес, что и pi
k=*pc; // k хранит значение 3 - байт данных по адресу pc (младший байт i)

p=&i; // p хранит адрес i. Приведение типов не требуется!
pc=(unsigned char *)p; // pc хранит тот же адрес, что и p

```

Листинг 4.2 Примеры использования указателей в С.

Хорошим тоном программирования считается использование в качестве первого символа идентификатора переменной-указателя литеры “p” или “P” (от англ. pointer).

### Разыменование указателей на структуры

При разыменовании указателей на структуру для компактной записи можно использовать особый оператор “->”. Если определена структура, объявлен и инициализирован указатель на нее, то разыменовывать указатель на структуру можно 2 способами:

```

typedef struct {
    unsigned char channel;
    unsigned char mode;
    float amplifier;
} ADCChannel;

...
ADCChannel ADC0,*pADC0;
pADC0=&ADC0;

...
(*pADC0).mode=1; //Разыменование указателя обычным способом
pADC0->mode=1;   //Разыменование с помощью специального оператора “->”

```

Т.е. специализированный оператор “->” позволяет одновременно разыменовывать указатель и получить доступ к полю данных. Использование оператора “->” предпочтительно.

### Арифметические действия с указателями

Указатель можно инициализировать, присвоив ему адрес (например, как в строке 2 рис. 4.5).

Если инициализировать указатель, используя предопределенную константу “NULL” (описанную в заголовочном файле `stdlib.h`) или численную константу 0, то компилятор считает, что такой указатель никуда не указывает. 0 это единственная численная константа, которую ANSI C разрешает явно присваивать указателю. Хотя в большинстве случаев инициализация с помощью “NULL” и 0 идентична, рекомендуется использовать “NULL”.

К указателям могут применяться операторы равенства и отношения (>, >=, <, <=, ==, !=). Также к указателям может быть применен ограниченный набор арифметических операций:

1. инкремент (++);
2. декремент (--);
3. добавление целого числа (+, +=);
4. вычитание целого числа (-, -=);
5. вычисление разности 2 указателей.

Результат арифметических действий зависит от типа указателя: инкремент и декремент адреса осуществляется на число, равное произведению добавляемой константы на размер типа переменной, на который ссылается указатель. Поясняющий пример приведен на рисунке 4.6.

<b>char * (1 байт в памяти)</b>	<b>int * (2 байта в памяти)</b>
<pre>char *pc, c; c=3; pc=&amp;c; //pc=0131h       //*pc=03h  pc=pc+1; //pc=0132h         //*pc=FFh  pc++; // pc=0133h       //*pc=A0h</pre>	<pre>int *pi, i; i=0xFF03; pi=&amp;i; //pi=0131h       //*pi=FF03h  pi=pi+1; //pi=0133h         //*pi=01A0h  pi++; // pi=0135h       //*pi=0000h</pre>

Рис 4.6 Арифметические операции с указателями. Прибавление 1 к указателю на char увеличивает pc на 1 (т.к. char занимает в ОП 1 байт), инкремент указателя на int увеличивает смещение, хранящееся в pi на 2 (т.к. int занимает в ОП 2 байта).

### Указатели и массивы

Указатели в С могут использоваться эквивалентно массивам и даже предоставляют более широкую функциональность. В С имя массива является указателем-константой на первую ячейку занятую им памяти, а указатели допускают задание смещения в стиле индексации элементов массива.

Например, пусть имело место объявление `int *pi, x[size];`, тогда команды: `pi=x;` и `pi=&x[0];` эквивалентны, т.к. идентификатор массива – `x` является указателем на первый элемент массива. С указателями можно работать как с массивами, например, если `pi` хранит адрес массива `x`, то доступ к `i`-му элементу массива можно получить, записав `x[i]` или `pi[i]`.

На рис. 4.7 приведены примеры доступа к ячейкам памяти, как к элементам массива с помощью указателей.

```
int *pi, i, x[5];
```

```
for(i=0; i<=4; i++)
    x[i]=i;
```

```
pi=x;
```

```
i=x[2]; //i=2
i=(pi+2); //i=2
i=pi[2]; //i=2
```

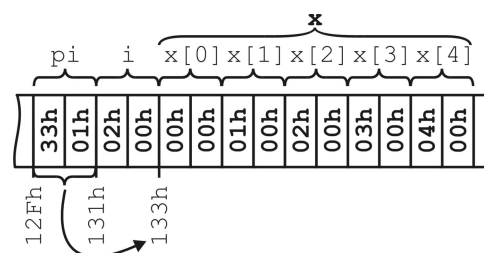


Рис 4.7 Примеры доступа к элементу массива `x` с индексом 2 с помощью указателей. Все 3 представленные способа эквивалентны.

Имя статического массива является указателем-константой, попытка ее изменения вызовет ошибку.

Использование указателей позволяет по-разному интерпретировать одну и ту же область памяти. В качестве примера в листинге 4.3 приведен участок исходного текста программы, где один и тот же массив данных интерпретируется одновременно как массив `unsigned int` и `unsigned char`.

```

unsigned char *pc;
unsigned int i, x[10];

... //Массив x заполняется данными

pc=(unsigned char *)x; //pc содержит адрес x

for (i=1;i<=19;i+=2){ //Размер массива x 20 байт
    pc[i]=1; //Все нечетные байты pc[] устанавливаем равными 01h
}

```

Листинг 4.3 Пример использования указателей. Все старшие байты слов в массиве x устанавливаются равными 01h с помощью “наложения” массива unsigned char \*pc на область памяти, занимаемую массивом unsigned int x[10] и коррекции всех нечетных элементов pc[].

## Лекция 5 Интерфейс с пользователем, работа с файлами, строки

Форматированный вывод с помощью `printf`. Ввод с помощью `scanf`. Поток. Работа с текстовыми файлами. Файлы произвольного доступа. Обработка символов на ЭВМ. Операции со строками в C.

### Форматированный вывод с помощью `printf`

Функция `printf` из заголовочного файла `stdio.h` предоставляет широкие возможности для т.н. *форматированного вывода* информации.

С помощью функции можно осуществлять: округление вещественных значений до указанного числа десятичных знаков, выравнивание столбца по положению десятичной точки, по правому или левому краю, вывод литеральных символов для организации дружелюбного диалога с пользователем, задание ширины поля для всех типов данных и др.

Формат вызова функции:

```
printf("Строка_управления_форматом", a1, a2,..., an);
```

где `a1, a2,..., an` – переменные или константы, значения которых должны быть выведены (необязательны). Строка управления форматом содержит *литеральные символы, символы ESC-последовательности, спецификаторы преобразования, флаги, информацию о ширине полей и точности преобразования*.

Спецификаторы преобразования начинаются с символа “%”, их список приведен в таблице 5.1.

Спецификатор	Описание
%d или %i	Выводит целое число со знаком
%hd или %hi	Выводит целое short со знаком
%ld или %li	Выводит целое long со знаком
%o	Выводит восьмеричное целое число без знака
%u	Выводит десятичное целое число без знака
%x и %X	Выводит шестнадцатеричное целое число без знака (в нижнем или верхнем регистре, соответственно)
%#x и %#X	Шестнадцатеричное число выводится с префиксом “0x”
%e или %E	Выводит вещественное число в экспоненциальной форме
%f	Выводит вещественное число в форме с плавающей точкой
%g или %G	При выводе вещественного числа автовыбор между %f или %e (%E).
%L	Выводит вещественное число long double
%c	Вывод символа
%s	Вывод строки
%p	Вывод адреса, который хранит переменная-указатель
%%	Вывод символа “%”
Префикс +	Например, “%+i”. Ставится перед спецификатором вывода числа. Заставит выводить “+” перед положительными числами.
Префикс пробел	Например, “% i”. Ставится перед спецификатором вывода числа. Заставит выводить “ ” (пробел) перед положительными числами.
Префикс Целое число	Например, “%10i”. Задаёт ширину поля, которая отводится под число, по умолчанию используется выравнивание по правому краю. Если количество цифр в числе превышает ширину поля, то поле игнорируется.
Префикс точка и целое число	Например, “%.2f”. Задаёт количество знаков после десятичного разделителя, отображаемых при выводе вещественного числа.

Префикс -	Например, "%-i". Задаёт выравнивание по левому краю.
Префикс 0	Например, "%0i". Дополняет число слева нулями до ширины поля.

Таблица 5.1 Основные спецификаторы преобразования строки управления форматом printf.

Префиксы: #, +, -, 0, пробел, еще называют *флагами*.

ESC-последовательности начинаются с символа "\". Они позволяют вывести на экран некоторые спецсимволы и осуществить управляющие выводом действия. Список основных ESC-последовательностей приведен в таблице 5.2.

ESC-последовательность	Описание
\n	Помещает курсор на начало следующей строки
\t	Горизонтальная табуляция
\'	Вывод символа одинарной кавычки
\"	Вывод символа двойной кавычки
\?	Вывод символа "?"
\\	Вывод символа "\"

Таблица 5.2 Основные ESC-последовательности, используемые в строке управления форматом printf.

В листинге 5.1 приведены примеры использования форматного вывода:

```
signed int a, b;
float r;

a=100;
b=200;
r=12345.6789;

printf("Hello world!");
//На экране:
//Hello world!
printf("\na=%i b=%i", a, b);
//На экране:
//a=100 b=200
printf("\n  \'dec\'      \'hex\' \\'na=%-#8xa=%+06i", a, a );
//На экране:
//  \'dec\'      \'hex\'
//a=0x64      a=+00100
printf("\nf=%4.2f", r);
//На экране:
//f=12345.68
```

Листинг 5.1 Примеры использования форматного вывода с помощью printf.

### Ввод с помощью scanf

Функция scanf из заголовочного файла `stdio.h` позволяет осуществлять ввод информации, в частности, с клавиатуры.

Формат вызова функции:

```
scanf("Строка_управления_форматом", &a1, &a2,..., &an);
```

где  $a_1, a_2, \dots, a_n$  – переменные или константы, значения которых должны быть введены.

Строка управления форматом содержит только спецификаторы формата (из таб. 5.1) без префиксов.

Хороший тон программирования рекомендует за одно обращение вводить значение одной переменной, сопровождая ввод выводом поясняющих сообщений для пользователей:

```
signed int a, b, z[10];

//Ввод скалярных величин
printf("\na=");
scanf("%i", &a);
printf("\nb=");
scanf("%i", &b);
...
//Ввод массива
for (a=0;a<=9;a++){
    printf("\nz[%i]=", a);
    scanf("%i", &z[a]);
}

//Вывод массива
printf("\nz["]=");
for (a=0;a<=9;a++){
    printf("%4i", z[a]);
}
```

Листинг 5.2 Пример ввода с помощью scanf.

## Потоки

Большинство современных прикладных компьютерных программ достаточно сложны и имеют большой размер. Поэтому, в практику программирования активно внедряются технологии, направленные на упрощение труда программистов и ускорение разработки конечной программы даже в случае, если использование таких технологий снижает эффективность результирующего кода (уменьшает быстродействие, увеличивает размер кода). Результатом развития такого подхода являются, в частности, *потоки* (англ. *streams*). Потоки – это технология программирования, направленная на упрощение процесса создания программ и повышение читаемости исходных текстов. Эта технология подразумевает использование единообразного, с точки зрения разработчика, программирования ввода-вывода данных при обмене с ОЗУ, файлами, консольными устройствами, периферийным оборудованием. (*Консольными* называются устройства, предназначенные для непосредственного взаимодействия с пользователем: монитор, клавиатура, принтер и т.п.). Технология потоков активно используются в C++. Язык C задействует их, в частности, при организации ввода-вывода. Например, вызов функции `fprintf(stdout, "Hello!");` приведет к выводу сообщения на экран аналогично `printf("Hello!");`. Т.е. вывод на консольное устройство – экран монитора осуществляется единообразно с выводом в файл (`stdout` – предопределенный указатель, указывающий на стандартное устройство вывода).

При создании на C небольших приложений, ориентированных на работу с оборудованием, активное использование потоков далеко не всегда целесообразно. Вместе с тем, знать о существовании таких технологий необходимо, т.к. в специальной литературе термин “потоки” используется широко.

### Работа с текстовыми файлами

ANSI C реализует работу с файлами двумя основными способами. Первый подход обеспечивает последовательный доступ к данным в файле и удобен для работы с текстовыми файлами. Второй позволяет осуществлять чтение или запись в произвольное место файла.

Организация последовательного доступа к файлам в C во многом схожа с текстовыми файлами Pascal (таб. 5.3).

№	Запись в текстовый файл C	Запись в текстовый файл Pascal
1.	#include <stdio.h>	
2.	FILE *F; int i;	VAR F :Text; i :Integer;
3.	if ((F=fopen("C:/test.txt", "w"))==NULL) { ... //Не удалось открыть файл }	Assign(F, 'C:\test.txt'); Rewrite(F);
4.	for (i=0; i<=9; i++) fprintf(F, "%4i %4i\n", i, 2*i);	For i:=0 to 9 do WriteLn(F, i:4, 2*i:4);
5.	fclose(F);	Close(F);

Таблица 5.3 Создание текстового файла средствами ANSI C и Pascal.

В 1 строке таблицы подключается заголовочный файл "stdio.h", это необходимо для использования подпрограмм работы с файлами.

Во 2 строке таблицы объявляется переменная-указатель на структуру FILE. Структура хранит информацию о файле с которым она связана и нужна для взаимодействия программы с операционной системой. Количество и состав полей FILE различается для различных ОС и при использовании разных файловых систем. Стандарт ANSI C предписывает разработчикам компиляторов языка C учитывать такие технические особенности и программист в большинстве случаев может не заботиться о внутреннем устройстве этой структуры. Поэтому, непосредственный доступ к полям структуры FILE обычно не производится. Обычно эта структура хранит такие данные, как: дату и время создания файла, дескриптор (англ. *handle*) – уникальный целочисленный номер файла, с которым оперирует ОС, список атрибутов файла, информацию о правах доступа к нему и т.п. Переменная типа Text в Pascal также является указателем на аналогичную запись.

В 3 строке таблицы файл *открывается для записи*. В C это делается одной командой: FILE \*fopen("Путь\_к\_файлу", "Режим\_доступа"). Функция fopen возвращает указатель на структуру FILE, поля которой хранят информацию о свойствах файла Путь\_к\_файлу. Если ОС не может предоставить доступ к интересующему файлу, например, он уже используется другой программой, то fopen вернет NULL. При открытии файла можно выбрать один из 6 режимов доступа (таб. 5.4).

Режим доступа	Описание
"r"	Открыть для чтения.
"w"	Открыть для записи. Если файл не существует, то он создается, если существует, его содержимое уничтожается.
"a"	Открывает (или создает) файл для дозаписи в конец.
"r+"	Открыть для чтения и записи.
"w+"	Открыть для чтения и записи. Если файл не существует, то он создается, если существует, его содержимое уничтожается.
"a+"	Открыть для чтения и записи, дозапись в конец.

Таблица 5.4 Возможные режимы доступа при открытии файла с помощью fopen.



В 4 строке таблицы 5.3 в текстовый файл выводятся числа в 2 столбца. Числа в обоих столбцах выравниваются по правому краю 4 символьного поля. Для этого в программе используется функция `fprintf`. Она является вариантом функции `printf` для работы с файлами отличается от `printf` только наличием дополнительного аргумента. Первым аргументом `fprintf` является указатель на структуру `FILE`, связанную с файлом, с которым осуществляется работа.

В 5 строке таблицы с помощью функции `fclose` файл закрывается. При этом все хранящиеся в оперативной памяти временные файловые буфера физически записываются на диск, операционная система информируется, что закрываемый файл больше не используется программой, связь между `F` и структурой `FILE` разрушается.

Для чтения данных из файла, созданного с помощью программы из таблицы 5.3 можно использовать программу, приведенную в листинге 5.3. В ней с помощью функции `fscanf` осуществляется чтение в переменные `a` и `b` значений из двух столбцов текущей строки загружаемого файла. В цикле `while` осуществляется проверка на достижение конца файла с помощью `feof(FILE *)`, `feof` возвращает ненулевое значение (“Истина”) в случае, если достигнут конец файла.

```
FILE *F;
int a, b;
...
F=fopen("col2.txt", "r");
if (F==NULL) {
    printf("Error!");
}
else{
    while(!feof(F)) {
        fscanf(F,"%i %i", &a, &b);
        ...
    }
    fclose(F);
}
```

Листинг 5.3 Пример чтения значение из текстового файла, хранящего 2 столбца данных.

### Файлы произвольного доступа

Представленные в предыдущем разделе примеры были ориентированы на работу с текстовыми файлами. С предоставляет программисту мощные средства для работы с файлами данных произвольного доступа (аналогично *обобщенным файлам* `File` в Pascal). Для организации такого доступа обычно оказывается достаточно 3 дополнительных функций, описанных в заголовочном файле `stdio.h`: `fwrite`, `fread`, `fseek`.

Функция `int fseek(FILE *F, long Offset, int Whence)` устанавливает смещение виртуального указателя чтения/записи `Offset`, байт относительно `Whence`. `Whence` задается константой: `SEEK_SET` – высчитывать смещение относительно начала файла, `SEEK_CUR` – относительно текущей позиции указателя или `SEEK_END` – относительно конца файла. В случае ошибки `fseek` возвращает ненулевое значение.

`fwrite(void *P, int Size, int N, FILE *F)` записывает в файл `F` `Size` байтов данных из буфера, адрес которого хранится в `P`. Количество записываемых блоков данных `N`. Запись осуществляется в позицию, определяемую значением виртуального указателя чтения/записи. После записи виртуальный указатель смещается на число записанных байтов.

fread(void \*P, int Size, int N, FILE \*F) читает из файла F Size байтов данных в буфер, адрес которого хранится в P. Количество читаемых блоков данных N. После записи виртуальный указатель смещается на число прочитанных байтов.

Пример создания файла с использованием fwrite приведен в листинге 5.4.

```
FILE *F;
int i, x;

if ((F=fopen("test.dat", "w"))==NULL)
    printf("Error!");
else{
    for (i=0;i<=9;i++){
        fwrite(&i, sizeof(int), 1, F);
        x=2*i;
        fwrite(&x, sizeof(int), 1, F);
    }

    fclose(F);
}
```

Листинг 5.4 Пример создания файла с помощью fwrite.

В листинге 5.5 рассмотрен пример организации произвольного доступа к файлу.

```
FILE *F;
int x;

if ((F=fopen("test.dat", "r+"))==NULL)
    printf("Error!");
else{
    fseek(F,5*sizeof(int), SEEK_SET); //Задать смещение указателя чтения
    fread(&x,sizeof(int), 1, F); //Прочитать 2 байта в переменную int
    fseek(F,5*sizeof(int), SEEK_SET); //Задать смещение указателя записи
    if (x==0) {
        x=0xFFFF;
        fwrite(&x,sizeof(int), 1, F);
        fwrite(&x,sizeof(int), 1, F);
    }
    else{
        x--;
        fwrite(&x,sizeof(int), 1, F);
    }

    fclose(F);
}
```

Листинг 5.5 Пример чтения и записи файла с произвольным доступом.

В представленном выше примере предполагается работа с файлом, представляющим собой массив 2 байтовых целых значений (File Of Integer в Pascal).

Содержимое байтов со смещениями 10 и 11 читается в 2 байтовую переменную типа int. Далее анализируется значение этой переменной:

- если она равна 0 (т.е. оба прочитанных байта равны 0), то в файл со смещением 10 последовательно записывается 2 раза содержимое двухбайтной целой переменной `x`: `FFFFh`;
- если не равна 0, то это значение переменной `x` уменьшается на 1, и записывается обратно в файл в исходную позицию (смещение 10).

### Обработка символов на ЭВМ

Как уже упоминалось, *литералы* (символы) в С обрабатываются, как беззнаковые целочисленные переменные. Обычно, символ занимает в памяти ЭВМ 8 бит, однако для ускорения передачи информации по каналам связи иногда используют 5-7 битные символы, а стандарт UNICODE подразумевает использование 16 бит. Язык С предоставляет для хранения символов тип `char`, который, для большинства компиляторов, эквивалентен типу `byte Pascal`, т.е. это целое 8 битовое число без знака.

Для корректного отображения символа на устройствах вывода информации используются стандартизированные таблицы кодировки, при этом число, хранящееся в переменной `char`, представляет собой просто номер элемента в такой таблице. Наиболее широко в России используются кодовые таблицы ASCII (символы латиницы, некоторые спецсимволы), а также КОИ-8r и Windows-1251, включающие символы кириллицы. 8 байтовое число без знака может кодировать элементы таблицы, содержащей не более 256 элементов. С принятием 16-битного стандарта кодирования символов UNICODE стало возможным представлять алфавиты всех существующих в мире языков в одной таблице. Выбор текущей кодовой таблицы обычно является задачей операционной системы, но иногда таблица навязывается компилятором. Например, при работе с компилятором, входящим в состав среды BC 3.1, можно использовать лишь символы ASCII.

Таким образом, т.к. индекс 100 в таблице ASCII закреплен за символом “d”, то приведенные в листинге 5.6 присваивания эквивалентны:

```
char ch;
...
ch='d'; //ch хранит значение 100
ch=100; //ch хранит значение 100
//Арифметические операции с переменной char
ch-=10; //ch хранит значение 90
ch/=2;  //ch хранит значение 45
ch++;   //ch хранит значение 46
printf("%c %i", ch, ch);
//На экране
//. 46
```

Листинг 5.6 Использование символьных переменных.

При обработке строки `ch='d'`; ЭВМ обращается в таблице кодировки и находит в ней код, соответствующий присваиваемому символу, который и заносится в целочисленную переменную.

При выводе символа на экран монитора, происходит обратная операция: ЭВМ находит в таблице кодировки символ, соответствующий коду, хранящемуся в целочисленной переменной. Затем ОС ЭВМ обращается к таблице, хранящий *шрифт* – правила начертания данного символа на экране монитора и отображает символ на экране в соответствии с этими правилами.

Таким образом, между операциями присваивания символьной константы и операциями вывода символа на экран к переменной можно применять любые операции отношения, арифметические и логические операции. Таким образом, термин “символьная

переменная” достаточно условен. Многие компиляторы допускают использование в качестве “символьных” любых целочисленных переменных, например, BC 3.1 корректно отработает:

```
int i;
...
i='d';
...
printf("%c", i);,
```

конечно, при выводе значения переменной *i* на экран, как символа, *i* будет принудительно приведена к типу *char* по известным правилам.

Термин используется, в первую очередь, чтобы подчеркнуть, что данная целочисленная переменная участвует в организации интерфейса с пользователем.

Символы с кодами от 0 до 31 в таблице ASCII не отображаются на экране непосредственно, и называются *управляющими символами*. При инициализации символьной переменной для их записи можно непосредственно использовать их числовой код или т.н. ESC-последовательность. Например, эквивалентны записи: *ch=0;* и *ch='\0';*. Т.е. *'\0'* является ESC-последовательностью, задающей символ с кодом 0. Более подробно ESC-последовательности рассмотрены в разделе “Форматированный вывод с помощью *printf*”, полный их список можно найти в справочниках и службе помощи используемого компилятора.

Некоторые полезные функций для работы с символами описаны в заголовочном файле *ctype.h*.

**Операции со строками в C**

Язык C подразумевает хранение строк в памяти ЭВМ как непрерывный массив элементов типа *char*, обязательно завершающийся символом с кодом 0. Такое представление называется нуль-терминированной строкой (англ. null-terminated character string), в литературе также используется обозначение ASCIIZ. “Переменная типа строка” представляет собой указатель на первый элемент такого массива – *char \**.

В таблице 5.5 сопоставляются способы представления строк, используемые в Pascal и в C.

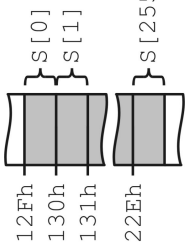
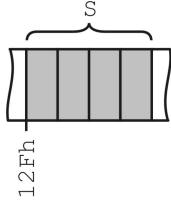
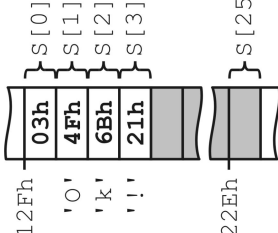
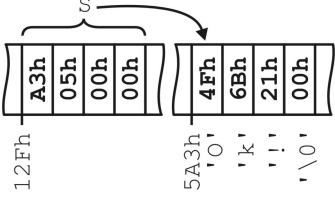
No	Язык Pascal	Данные в памяти	Язык C	Данные в памяти
1.	<i>S :String;</i>		<i>char *S;</i>	
2.	<i>S:='Ok!';</i>		<i>S="Ok!";</i>	

Таблица 5.5 Представление строк в C и Pascal.

Из таблицы 5.5 видно, что ASCIIZ формат более компактный, чем формат, используемый Pascal, который резервирует для строки 256 байт памяти независимо от ее фактической длины. Кроме того, в отличие от Pascal, строки ASCIIZ ограничены по длине только объемом доступной памяти.

Обратите внимание, что для определения в С строчной константы используются двойные кавычки, а для задания символьной константы – одинарные. Записи 'd' и "d" совершенно различны: первая представляет собой однобайтную константу, содержащую число 100 (ASCII-код символа "d"), вторая запись имеет смысл указателя на массив в ОП, состоящий из двух 8-битных элементов (ASCII кода символа "d" – числа 100 и терминирующего нуля).

Некоторые полезные функции преобразования строки в число и обратно описаны в заголовочном файле `stdlib.h`, например: `itoa`, `ltoa`, `ultoa` – преобразуют в строку, соответственно, целое, длинное целое и длинное целое без знака, `atoi`, `atol`, `atof` – преобразуют строку в, соответственно, целое, длинное целое и вещественное `float`.

Основные функции для работы с 0-терминированными строками описаны в заголовочном файле `string.h`. Некоторые примеры работы со строками сведены в таблице 5.6.

No	Язык C	Язык Pascal	Описание
1.	<code>char *S1,*S2,*S3;</code> <code>char ch;</code> <code>int N,i;</code>	<code>S1,S2,S3 :String;</code> <code>Ch :Char;</code> <code>N,i :Integer;</code>	Объявление 3 переменных типа строка, одной "символьной" и одной целочисленной переменных
2.	<code>S1=NULL;</code>	<code>S1:=' ';</code>	Инициализация "пустой строкой"
3.	<code>S1="Yes";</code> <code>S2="No";</code>	<code>S1:=' Yes' ;</code> <code>S2:=' No' ;</code>	Инициализация переменных
4.	<code>S3=S1;</code>	–	S3 и S1 указывает на одну область памяти
5.	<code>strcpy(S3,S1);</code>	<code>S3:=S1;</code>	Копирует данные из строки S1 в строку S3
6.	<code>S3[0]='d';</code>	<code>S3[1]:='d';</code>	Первая буква строки S3 заменяется на "d"
7.	<code>N=strlen(S2);</code>	<code>N:=Length(S2);</code>	Определение длины строки S2
8.	<code>ch=S2[N-1];</code>	<code>ch:=S2[N];</code>	В символьную переменную ch сохраняется последний символ строки
9.	<code>S3=strcat(S1,S2);</code>	<code>S3:=S1+S2;</code>	Присоединяет строку S2 к S1 (S3="YesNo")
10.	<code>if(!strcmp(S1,S2)){</code> ... <code>}</code>	<code>If S1=S2 then begin</code> ... <code>end;</code>	Сравнение строк. <code>strcmp(S1,S2)</code> возвращает 0 если строки S2 и S1 совпадают.

Таблица 5.6 Примеры работы со строками. Сопоставление Pascal и C.

## Лекция 6 Макросы, функции, библиотеки подпрограмм

Подпрограммы, библиотеки подпрограмм. Объявление функции в С. Вызов функций. Константы и макросы, директива `#define`. Область видимости переменных. Прототипы функций, библиотеки, рекурсия. Создание “процедур”. Функция `main`. Заголовочные файлы, директива `#include`. Стандартная библиотека С. Указатели на функции. Передача указателя на функцию, вызов по ссылке.

### Подпрограммы, библиотеки подпрограмм

Современные концепции структурного программирования предполагают активное использование *подпрограмм* (англ. *subroutine, subprogram*) при разработке прикладного программного обеспечения (ПО). Программный код, который был однократно аккуратно оформлен в виде подпрограммы и отлажен, может затем многократно использоваться в различных разработках, в том числе многими разработчиками. Такой подход может существенно уменьшить время разработки ПО и снизить вероятность ошибок в нем.

Логически сгруппированные подпрограммы часто объединяют в модули (англ. *unit*) и библиотеки (англ. *library*). Библиотеки и модули обычно представляют собой скомпилированный объектный код набора подпрограмм, который может быть добавлен к основной программе. Часто в целях защиты авторских прав разработчика такие модули распространяются без исходных текстов, сопровождаясь лишь технической документацией, содержащей список подпрограмм модуля и описание параметров этих подпрограмм.

### Объявление функции в С

В языке С все подпрограммы принято называть *функциями* (англ. *function*), в отличие от Pascal, где подпрограммы подразделяются на функции и *процедуры* (англ. *procedure*). Синтаксис объявления функции С:

```
ТИП0 Имя_функции (ТИП1 Параметр1, ТИП2 Параметр2, ..., ТИПN ПараметрN) {
    //Объявление локальных переменных
    int a,b;
    ...
    //Тело функции
    ...
    return ЗНАЧЕНИЕ;
}
```

ТИП0 – тип значения, возвращаемого функцией, ТИП1, ... ТИПN – типы параметров функции, Параметр1, ..., ПараметрN – имена формальных параметров функции, используемые внутри ее тела. Для того, чтобы функция возвращала значение используется оператор `return ЗНАЧЕНИЕ;`. Оператор `return` приводит к немедленному завершению работы текущей функции, команды, расположенные после него, игнорируются. Функция может содержать несколько `return`, например, в операторах ветвления (`if, switch`).

Язык С	Язык Pascal
<pre>int AbsAdd(int a, int b){     int c;     c=a+b;     if (c&gt;=0)         return c;     else         return -c; }</pre>	<pre>function AbsAdd(a,b:Integer):Integer; Var     c :Integer; begin     c:=a+b;     If c&gt;=0 then         AbsAdd:=c     Else         AbsAdd:=-c; end;</pre>

Таблица 6.1 Примеры функций, возвращающих модуль суммы 2 чисел на С и Pascal.

В таблице 6.1 приведен пример функции, выполняющей простые арифметические операции.

Функция может не иметь параметров, в этом случае, вместо списка параметров указывается служебное слово `void` (англ. пустота), а при вызове функции после имени обязательно ставятся парные круглые скобки, например:

```
void init(void){ //Объявление функции без параметров
    char cod;
    int x;
    ...
}
...
void main(void){
    char rs;
    int x;
    ...
    init(); //Вызов функции без параметров
    if (rs!=0) ...;
    ...
}
```

Листинг 6.1 Вызов функции без параметров.

### Вызов функций

Рассмотренный в данном разделе пример представлен для общего ознакомления с материалом. Механизмы организации вызова подпрограмм и возврата в вызывающую функцию приведены здесь в несколько упрощенном виде, особенности использования различных процессоров и компиляторов не учитываются.

На рисунке 6.1 схематически изображен участок ОП ЭВМ, в котором размещена программа из листинга 6.1.

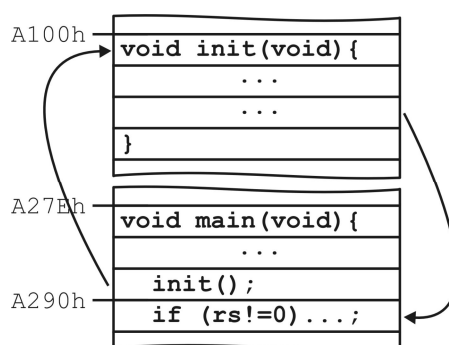


Рис. 6.1 Размещение в ОП программы из листинга 6.1.

Все процессоры имеют специальный регистр (быстродействующую ячейку памяти, размещенную физически на кристалле микросхемы процессора), хранящий адрес команды, которая начнет выполняться процессором следом за текущей. Часто этот регистр называют IP (от англ. instruction pointer – указатель команд).

После старта программы IP содержит адрес точки входа в программу – функции `main()` – A27Eh. По мере выполнения функции `main()`, IP увеличивается после выполнения каждой команды (IP может уменьшаться только наличием в функции циклов

или безусловном переходе на предыдущую команду). Наконец IP достигает команды вызова функции `init()`, при этом:

1. в специально зарезервированной для программы области памяти – стеке программы сохраняется *адрес возврата* (англ. `return address`) – адрес команды, следующей за командой вызова функции (A290h);
2. в случае, если функция имеет параметры, они также сохраняются в стеке;
3. в памяти, отведенной программе, обычно, в программном стеке выделяется память для хранения локальных переменных программы;
4. указателю команд IP присваивается адрес точки входа в вызываемую функцию (A100h).

Далее, продолжается выполнение процессором команд, содержащихся в теле вызванной функции.

При достижении конца функции, или оператора `return` в ней, происходят следующие действия:

1. из стека извлекается адрес возврата (A290h) и загружается в IP;
2. освобождается память, занятая в стеке программы параметрами и локальными переменными отработавшей функции;
3. процессор продолжает выполнение команд, начиная с адреса IP.

Организации вызова подпрограммы и обмена с ней параметрами является обширным вопросом, который решается по-разному различными компиляторами. Например, используется 3 основных способа передачи данных в подпрограмму:

- через стек передается значение параметра (в большинстве случаев, C использует именно такой способ),
- через стек передается адрес ячейки памяти, хранящей значение передаваемого параметра,
- параметры передаются в подпрограмму через регистры процессора,

а возможны также комбинированные варианты, обмен информацией через глобальные переменные программы, через области памяти в системной куче и др. Подробное рассмотрение вопросов организации взаимодействия с подпрограммами выходит за рамки данного курса.

### **Константы и макросы, директива `#define`**

C, в отличие от Pascal, позволяет использовать макросы. Для определения макроса в C используется *команда препроцессора* (англ. `preprocessor`) `#define`. Команды препроцессора – служебные слова, предваряемые префиксом `#` – не компилируются в машинные коды. Они предназначены лишь для совершения некоторых действий перед компиляцией исходного текста.

Например, команда препроцессора, `#define NAME expression` указывает компилятору, что ПЕРЕД компиляцией нужно заменить по всему тексту программы идентификатор `NAME` на выражение `expression`. Чаще всего в таком виде *макроопределение* (англ. `macrodefinition`, `macros`) `#define` используется для определения констант, например: `#define PI 3.1415` определяет константу PI, которую можно использовать везде, где допускается использование числовой константы 3.1415.

Макросы допускают использование параметров, при использовании в программе такие макросы *расширяются* на этапе предобработки, т.е. в их текст подставляются заданные в программе значения параметров, и этим текстом заменяется идентификатор макроса в тексте программы. Параметры перечисляются при объявлении макроса через запятую в скобках после его идентификатора.

Пример объявления и использования макроса приведен в листинге 6.2.



```
#include <stdio.h>

#define BIT_SET(X,N) (X) |=1<<(N) //Объявление макроса с параметрами

void main(void){
    int M=0, Flags=0;

    BIT_SET(Flags, M+2); //Установить 2-ой бит переменной Flags

    printf("\nFlags=%i", Flags);
}
```

Листинг 6.2 Объявление и использование макроса с параметрами. Макрос BIT\_SET устанавливает в переменной X бит номер N в 1 независимо от его начального состояния.

Нужно обратить внимание, что в теле макроса его параметры заключены в скобки. Это рекомендуется делать всегда во избежание логических арифметических ошибок. Например, если бы скобок не было, то макрос из листинга 6.2 расширился бы в `Flags|=1<<M+2 (M=0)`. Т.е. приоритет выполнения арифметических операций был бы нарушен, что привело бы к логической ошибке – результат был бы неверным.

Таким образом, несмотря на внешнее сходство с функциями, макросы являются не более чем средством автоматической, обработки исходного текста программы перед компиляцией. Например, допустимо, хотя и бессмысленно, даже такое использование макроопределения:

```
#include <stdio.h>

#define ENTRY void main(void){

int a,b;

ENTRY
    a=5;
    b=2;
    printf("\na+b=%i", a+b);
}
```

Листинг 6.3 Пример использования макроопределения.

Чаще всего макросы используют для реализации компактных арифметических выражений, аналогично примеру из листинга 6.2. По сравнению с функциями макросы могут дать некоторый выигрыш в быстродействии программ, т.к. при вызове функции расходуется процессорной время на сохранение в стеке параметров и адреса возврата, а также переходы на точку входа в функцию и адрес возврата. Очевидно также, что при оптимизации программы по размеру макросы проигрывают функциям.

Рекомендуется использовать в именах макросов все заглавные литеры, это позволит “с первого взгляда” отличить макрос от функции или переменной.

### **Область видимости переменных**

Правила распространения области видимости переменных и имен функций в C близки к правилам Pascal. Переменные, объявленные до объявления функций, считаются глобальными, и их областью видимости распространяется на все функции программы, включая `main()`. Переменные, объявленные внутри тела функции считаются локальными

для этой функции и их область видимости распространяется только внутри функции, в теле которой они были объявлены. Поэтому, локальные переменные, объявленные в нескольких функциях, могут иметь одинаковые имена. Если внутри тела функции объявлена локальная переменная, имеющая такое же имя, как и глобальная переменная, то обращения к переменной внутри такой функции будут направлены именно к локальной переменной. Т.е. внутри такой функции глобальная переменная модифицироваться и читаться не будет.

Правила “хорошего тона” рекомендуют минимизировать использование в программе глобальных переменных. Считается, что злоупотребление ими ухудшает читаемость программы и увеличивает вероятность логических ошибок программистов. Также настоятельно рекомендуется всегда объявлять как локальные переменные счетчики циклов, переменные для промежуточных вычислений и т.п.

### **Прототипы функций, библиотеки, рекурсия**

Функции в программе C могут быть описаны в любом месте программы, как до, так и после функции `main()`. Допустим рекурсивный вызов функцией себя собой:

```
void func1(void){  
    ...  
    func1();  
    ...  
}
```

Листинг 6.4 Пример рекурсивного вызова функции.

В отличие от Pascal, вложенные описания функций не допускаются.

Если в программе описаны функции `f1()` и `f2()`, то для того, чтобы `f2()` могла вызвать `f1()`, `f2()` должна быть описана после `f1()`.

C позволяет использовать библиотеки подпрограмм – куски скомпилированного кода, содержащие тела нескольких функций. Использование библиотек с одной стороны позволяет многократно использовать однократно написанные и отлаженные подпрограммы, с другой стороны, защищает разработчика от несанкционированного использования и модификации его разработки, т.к. модули могут поставляться без исходных текстов. Например, в виде модулей поставляется стандартная библиотека языка C.

При построении программы, включающей несколько сторонних модулей, после компиляции теста, подготовленного программистом, осуществляется *компоновка программы*, т.е. соединение скомпилированных модулей и установка связей между ними. Компоновка осуществляется специальной утилитой, поставляемой обычно вместе с компилятором – *компоновщиком* или *линкером* (англ. linker). В C тела модулей подключаются после программы, написанной программистом. Поэтому, если не предпринять специальных действий, функции из подключенных модулей использовать не удастся. Действительно, если в подключенном модуле, описана функция `f3()`, которую программист пытается вызвать в `main()`, произойдет ошибка и компиляция будет прервана, т.к. в момент вызова `f3()` в `main()` компилятор еще не знает о существовании этой функции, т.к. ее находится после `main()`. Для решения этой проблемы C предлагает использовать *прототипы* (англ. prototype) функций. Прототип представляет из себя описание, сохраняющее заголовок функции (ее имя и список параметров). Прототип функции может быть описан в любом месте программы до первого обращения к этой функции. Прототип нужен, чтобы сказать компилятору, встретившему в процессе компиляции вызов функции, тело которой еще не было скомпилировано: “не ругайся, компилятор, функция будет описана ниже”.

Примеры использования прототипа функции приведены в таблице 6.2.

(a)	(б)	(в)
<pre>int init(int a, char c);  int init(int a, char c){     ... }  void main(void){     ...     rs=init(val, 0); }</pre>	<pre>int init(int a, char c);  void main(void){     ...     rs=init(val, 0); }  int init(int a, char c){     ... }</pre>	<pre>int init(int, char);  void main(void){     ...     rs=init(val, 0); }  int init(int a, char c){     ... }</pre>

Таблица 6.2 Различные случаи использования прототипов: (а)– прототип необязателен, т.к. функция `init()` вызывается после объявления, (б)– прототип обязателен, т.к. функция `init()` вызывается в `main()` до ее объявления, (в) –при описании прототипа указывать имена формальных параметров функции необязательно, главное перечислить в правильном порядке типы всех формальных параметров и указать тип возвращаемого значения.

Если в программе описано много функций, вызывающих друг-друга, то удобно в начале программы описать прототипы всех функций, чтобы далее не задумываться о том, какая функция была описана раньше, а какая позже. Часто описание прототипов функций выносят в заголовочные `h`-файлы. Поэтому, например, перед использованием функции `printf` нужно подключить заголовочный файл `stdio.h`, где описан прототип этой функции, иначе произойдет ошибка компиляции, т.к. тело самой функции содержится в файле библиотеки, подключаемой после функции `main()`.

### Создание “процедур”

Функция `C` может возвращать только одно значение. Однако часто желательно, чтобы одна функция возвращала несколько значений, или имелась бы возможность модификации входных параметров функции. В языке `Pascal`, например, для этих целей используются процедуры (`procedure`) с директивой `VAR` при описании параметров. Аналогичные возможности можно получить в `C`, если передавать в функцию не значение переменной, а ее адрес. Пример создания на `C` функции, меняющей местами значение 2 переменных, приведен в таблице 6.3.

Язык C	Язык Pascal
<pre>void Swap(int *a, int *b){     int sw;     sw=*a;     *a=*b;     *b=sw; }  void main(void){     int data[100], i, j;      ...     for (i=0;i&lt;=98;i++){         for (j=i+1;j&lt;=99;j++){             if (data[j]&lt;data[i])                 Swap(&amp;data[i], &amp;data[j]);         }     } }</pre>	<pre>Procedure Swap(VAR a,b: Integer); Var     sw :Integer; Begin     sw:=a;     a:=b;     b:=sw; End;  VAR     i,j    :Integer;     data  :Array [0..99] Of Integer; BEGIN     ...     For i:=0 to 98 do begin         For j:=i+1 to 99 do begin             If data[j]&lt;data[i] then                 Swap(data[i], data[j]);         end;     end; END.</pre>

Таблица 6.3 Пример функции, меняющей местами значения 2 переменных.

### Функция `main`

В С, в отличие от Pascal, тело программы оформлено в виде функции с предопределенным именем, которая является точкой входа в программу. Такое оформление тела программы является лишь данью стилю С. Функция `main()` не является полноценной функцией С и не может быть вызвана ни рекурсивно, ни другой функцией.

Параметры `main()` используются для передачи программе параметров командной строки, и их список должен быть описан строго определенным образом. Возвращаемое `main()` значение, обычно, целое число, используется для сообщения ОС кода завершения программы. Возврат функцией `main()` кода 0 сигнализирует ОС о завершении программы без ошибок, возврат ненулевого значения означает, что при выполнении программы произошла ошибка.

### Заголовочные файлы, директива `#include`

Рассмотрим еще одну широко используемую директиву – `#include`, которая нужна для подключения препроцессором заголовочных файлов. Синтаксис ее использования: `#include <имя_файла>` или `#include "имя_файла"`. Если с `#include` используются треугольные скобки `<>`, то поиск файла `имя_файла` осуществляется препроцессором в системных директориях, двойные кавычки указывают, что файл расположен в той же папке, что и исходный текст программы. Поэтому, обычно, треугольные скобки используются при подключении заголовочных файлов, входящих в стандартную поставку компилятора, а кавычки – для подключения дополнительных заголовочных файлов, например, разработанных самим программистом.

Механизм работы директивы прост: встретив `#include`, препроцессор подставляет вместо этой директивы текст, хранящийся в текстовом файле `имя_файла`. `имя_файла` представляет собой простой текстовый файл, содержащий любой кусок текста программы.

Например, пусть созданы заголовочный файл `sample.h` и файл с исходным текстом `sample.c`:

<pre>#include &lt;stdio.h&gt;  int a;  void main(void){</pre>	(a)	<pre>#include &lt;stdio.h&gt;  int a;  void main(void){     int b,c;     ...     b=7;     c=5;     a=b+c;     ... }</pre>	(b)
<pre>#include "sample.h"      int b,c;     ...     b=7;     c=5;     a=b+c;     ... }</pre>	(б)		

Листинг 6.5 Иллюстрация работы директивы `#include`. (a) – Текст заголовочного файла `sample.h`. (б) – Текст исходного файла программы `sample.c`, подключающий заголовочный файл `sample.h`. (б) – Текст программы `sample.c` после работы препроцессора: текст, содержащийся в `sample.h` вставляется вместо строки `#include "sample.h"`. Именно текст (в) будет обрабатываться компилятором.

В листинге 6.5 проиллюстрирован механизм работы `#include`. На практике, конечно, `h`-файлы не используются для разрыва тела программы, т.к. это крайне негативно

сказывается на читаемости. На практике, заголовочные файлы чаще всего используются для описания прототипов функций, тела которых хранятся в подключаемых к программе библиотеках, объявления констант, глобальных переменных и описания макросов.

Именно так используются заголовочные файлы стандартной библиотеки C, поставляемой с компилятором языка. Тела функций хранятся в виде объектного кода в библиотеках, которые подключаются к пользовательской программе по умолчанию, но чтобы использовать эти функции, программист должен подключить соответствующий заголовочный файл, содержащий объявление прототипа функции, необходимые константы и макросы.

Заголовочные файлы могут использоваться и для хранения текстов функций, хотя хранение функций в скомпилированном виде в библиотеке предпочтительнее.

### Стандартная библиотека C

Стандартная библиотека C содержит 15 библиотек, прототипы функций, макросы и глобальные переменные которых описаны в заголовочных файлах (таблица 6.4).

h-файл	комментарий
<b>assert.h</b>	Представляет дополнительные инструменты для отладки программ.
<b>ctype.h</b>	Прототипы некоторых функций обработки символов (char).
<b>errno.h</b>	Макросы, для сообщения о типе ошибке.
<b>float.h</b>	Пределы для чисел с плавающей точкой.
<b>limits.h</b>	Пределы для целых чисел.
<b>locale.h</b>	Для локализации программы с учетом соглашений о представлении даты, времени, валют и т.п. в различных форматах, принятых в разных странах мира.
<b>Math.h</b>	Функции и макросы для математических вычислений.
<b>setjmp.h</b>	Для организации глобальных переходов (goto) между функциями.
<b>signal.h</b>	Обработка прерываний.
<b>stdarg.h</b>	Поддержка функций с переменным числом параметров.
<b>stddef.h</b>	Определение некоторых дополнительных типов.
<b>stdio.h</b>	Прототипы функций стандартной библиотеки ввода/вывода.
<b>stdlib.h</b>	Различные функции.
<b>string.h</b>	Работа со строками.
<b>Time.h</b>	Работа с системными временем и датой.

Таблица 6.4 Заголовочные файлы стандартной библиотеки C.

Перечисленные 15 заголовочных файлов должен содержать любой компилятор, соответствующий стандарту ANSI C. Кроме того, различные компиляторы могут содержать в своей стандартной поставке дополнительные библиотеки и заголовочные файлы.

Одним из преимуществ языка C является наличие значительного количества дополнительных библиотек различного назначения, распространяемых, в частности, посредством сети Internet под различными (в т.ч. и бесплатными) лицензиями.

### Указатели на функции

Указатель на функцию представляет собой адрес точки входа в эту функцию. Например, указатель на функцию `init()` рис. 6.1 будет содержать значения `A100h`. Получение указателя на функцию, создание таблицы указателей на функцию, передача указателя на функцию подпрограммам, вызов функции по ссылке являются важнейшими инструментами взаимодействия прикладной программы с операционной системой и драйверами периферийного оборудования.

Например, системная таблица векторов прерываний является ни чем иным, как массивом указателей на функции. Для создания собственного обработчика прерываний программист должен написать функцию, кодирующую необходимые операции и сохранить ее адрес в соответствующую ячейку этого массива, хранящегося в ОП ОС.

Широко используются указатели на функции при обработке сигналов в реальном времени с помощью периферийного оборудования. Например, устройства сбора данных (АЦП) часто взаимодействуют с ПК с помощью технологии прямого доступа к памяти – ПДП (англ. direct memory access – DMA). При обработке получаемых данных в режиме реального времени программист передает программному драйверу (англ. driver) АЦП адрес созданной им *функции-обработчика сигнала* (англ. callback function, handler), кодирующей специализированные для решаемой задачи действия, например, фильтрацию сигнала. Драйвер АЦП, в свою очередь, периодически вызывает пользовательский обработчик, передавая ему адрес буфера в ОП, куда сохраняются полученные данные, а также количество фактически полученных от АЦП и еще не обработанных выборок.

Программным драйвером, в данном, случае называют специализированное фирменное ПО, обеспечивающее непосредственное управление взаимодействием с оборудованием на низком уровне и предоставляющее программисту-разработчику прикладного ПО удобные средства для взаимодействия с этим оборудованием, например, посредством создания собственного callback-обработчика. Использование такого драйвера позволяет прикладному программисту сосредоточиться на решении конкретной прикладной задачи, освобождая его от необходимости вникать в особенности аппаратного устройства конкретного оборудования.

### **Передача указателя на функцию, вызов по ссылке**

Указатель на функцию может быть получен с помощью обращения к имени функции, записанному без круглых скобок (см. пример в листинге 6.6). Для хранения указателя на функцию обычно используют переменную `void *`.

Указатель на функцию может быть присвоен другой переменной, передан подпрограмме в качестве параметра, может использоваться для вызова функции по ссылке.

Пример, приведенный в листинге 6.6, иллюстрирует различные аспекты использования указателей на функции в С.

```
#include<stdio.h>

int Processor (int c, int (*Calculate) (int, int)){
    //Вызывает по ссылке Calculate(c,c+1)
    //использует формальный параметр-функцию Calculate
    return (*Calculate)(c,c+1);
}

int Processor2(int c, void *P){
    //Вызывает по ссылке функцию, адрес которой хранится в P
    //Processor2 полностью идентична по функциональности Processor
    return (*(int (*)(int, int))P)(c,c+1);
}

int Adder(int a, int b){
    //Возвращает a+b
    return a+b;
}

int Multiplier(int a, int b){
    //Возвращает a*b
    return a*b;
}

void *P[2]; //Объявление массива нетипизированных указателей

void main(void){
```

```

P[0]=Adder; //Получаем адрес функции Adder и сохраняем его в массиве
P[1]=Multiplier; //Получаем адрес Multiplier и сохраняем его в массиве
//Обычный вызов Adder по имени
printf("\n%i", Adder(1,2));
//Вызов Adder по ссылке (адрес Adder хранится в P[0])
printf("\n%i", (*(int (*)(int, int))P[0])(1,2));
//Вызывает Processor, передавая ей указатель на Adder
printf("\n%i", Processor(2, (int (*)(int, int))P[0]));
//Вызывает Processor, передавая ей указатель на Multiplier
printf("\n%i", Processor(2, Multiplier));
//Вызывает Processor2, передавая ей указатель на Adder
printf("\n%i", Processor2(2,P[0]));
//Вызывает Processor2, передавая ей указатель на Multiplier
printf("\n%i", Processor2(2, Multiplier));
}

```

Листинг 6.6 Примеры получения адреса функции, создания массива (таблицы) указателей на функции, вызова функции по ссылке, различных способов передачи указателя на функцию в подпрограммы. После выполнения программы на экран в столбик будут выведены значения: 3, 3, 5, 6, 5, 6.

В представленном примере описаны, в частности, 2 функции: Adder и Multiplier, выполняющие простые арифметические действия. Принципиально, что функции имеют одинаковые списки параметров (2 переменные типа int) и одинаковые типы возвращаемых значений.

В начале программы создается таблица указателей на функции: адреса Adder и Multiplier сохраняются в массиве нетипизированных указателей P[].

Команда `(*(int (*)(int, int))P[0])(1,2)` в функции printf обеспечивает вызов по ссылке функции, адрес которой хранится в P[0] (в нашем случае, Adder), с параметрами (1, 2). Рассмотрим эту команду более подробно. При ее записи использовался синтаксис: `(*(тип_приведения)P[0])(параметры)`. Читать эту команду следует так:

1. нетипизированный указатель P[0] явно приводится к типу указателя на функцию,
2. приведенный указатель разыменовывается, что эквивалентно вызову функции,
3. функция вызывается с параметрами, перечисленными в круглых скобках справа.

При приведении типов, тип указателя на функцию (в круглых скобках перед P[0]) записан как `(int (*)(int, int))`. Такая запись читается так: “указатель на функцию, имеющую 2 параметра типа int и возвращающую значение типа int”. Символ “\*” необходимо заключать в круглые скобки, т.к. без них запись `(int * (int, int))` читалась бы: “прототип функции, имеющей 2 параметра типа int и возвращающей указатель на значение типа int”. Последнее описание не соответствует прототипу используемых нами арифметических функций Adder и Multiplier и приведет к ошибке. Таким образом, синтаксис описания явного приведения типа нетипизированного указателя к указателю на функцию следующий: `(возвращаемый_тип (*)(типы_параметров))`.

Ниже в листинге 6.6 рассматриваются примеры передачи указателя на функцию в качестве параметра другой подпрограммы и вызова переданной функции по ссылке внутри подпрограмм.

Описанные в программе функции Processor и Processor2 абсолютно идентичны по функциональности и отличаются только тем, что у Processor приведение типа указателя к типу функции осуществляется при описании формальных параметров, а у Processor2 те же действия производятся внутри тела функции.

На практике широко используются все перечисленные приемы работы с указателями на функции.

## Лекция 7 Организация обмена и хранения данных

Динамические массивы. Стек. Очередь, сбор данных в многозадачной ОС. Кольцевой буфер, линия задержки. Связанный список.

Существует несколько способов организации данных, широко используемых в практике программирования. К ним относятся, например, динамические массивы, стеки, связанные списки и т.п. В этой лекции рассматриваются вопросы организации средствами С некоторых таких наиболее употребительных конструкций.

### Динамические массивы

Размер статических массивов в С задается на этапе компиляции. Изменить его во время выполнения программы невозможно. Это может привести к крайне неэффективному расходованию памяти при работе с данными переменной длины (например, при обработке файлов). Эффективным решением является создание временных динамических массивов, память для которых выделяется во время выполнения программы из кучи после того, как станет точно известен требуемый размер массива (например, после определения размера загружаемого файла). Пример создания динамического массива рассмотрен в листинге 7.1.

```
#include <stdlib.h>
...
signed int *pi;
unsigned int i,N;
void *p;
... // определено значение N
p=malloc(N*sizeof(signed int)); //Запрашиваем память из кучи

if (p!=NULL){ //Если память выделена
    pi=(signed int*)p; //явное приведение типа указателя
    for (i=0;i<=N-1;i++){
        pi[i]=i;
    }
    ... //работа с массивом pi[]
    free(p); //массив больше не нужен, освободим занятую им память
}
```

Листинг 7.1 Создание динамического массива `signed int pi[N]`, где `N` определяется во время выполнения программы.

Прототипы функций для выделения и освобождения памяти из кучи описаны в заголовочных файлах `stdlib.h` и `alloc.h`. В приведенном примере (листинг 7.1) программа запрашивает у менеджера памяти память для хранения `N` элементов типа `signed int` из кучи. Для этого вызывается функция `malloc`. Аргументом функции является количество байтов, которое запрашивается у менеджера памяти. Если менеджер памяти может предоставить непрерывный участок ОП требуемого размера, то `malloc` возвращает его адрес в виде нетипизированного указателя (`void *`), в противном случае, `malloc` вернет `NULL`. Для вычисления требуемого количества памяти в байтах рекомендуется использовать конструкцию вида `N*sizeof(signed int)`, где `N` – количество элементов массива типа `signed int`. Макрос `sizeof` возвращает размер в байтах переменной или типа данных, указанных ему в качестве аргумента.

Для того чтобы использовать выделенную область памяти в качестве массива, ее адрес присваивается указателю на `signed int`, при этом используется явное приведение



типов: `pi=(signed int*)p;`. Далее, указатель `pi` можно использовать, как массив, задавая смещение в виде индекса `pi[i]=i;`.

После того, как динамический массив стал ненужным, занимаемую им ОП нужно освободить, для предотвращения утечки памяти. Для этого используется функция `free`. В качестве аргумента функция принимает указатель на ранее выделенную с помощью `malloc` область памяти.

Удобной для создания целочисленных массивов может оказаться еще одна функция выделения памяти – `calloc`. Она выполняет действия аналогичные `malloc`, но вдобавок принудительно заполняет выделенный буфер ОП нулями.

Для изменения размера уже созданного динамического массива в ходе выполнения программы используется функция `void * realloc(void *P, size_t Size)`. `P` – указатель на ранее выделенную в куче область, `Size` – новый размер выделяемой памяти в байтах, в случае, если память выделить удалось, `realloc` возвращает указатель на нее, в противном случае, возвращается `NULL`. Если `realloc` не может выделить непрерывный участок памяти по адресу `P`, при увеличении размера буфера, то буфер будет выделен там, где такой участок достаточного размера имеется. Данные из старого буфера при этом будут автоматически скопированы на новое место. Операция копирования в памяти больших объемов данных может занимать значительное время, поэтому использовать `realloc` в цикле для увеличения размера буфера не рекомендуется.

### Стек

*Стеком* (англ. stack) – называют способ организации доступа к данным в соответствии с принципом “последним вошел, первым вышел” – LIFO (англ. Last Input, First Output). К стеку обращаются с помощью 2 команд: сохранить значение в стеке и извлечь значение из стека. При этом значение, сохраненное последним, будет извлечено из стека первым. Следом за ним будет извлечено значение, сохраненное предпоследним и т.д. В любой момент времени для чтения доступен только верхний элемент стека, т.е. значение, сохраненное последним.

В качестве механического аналога стека можно представить работу подавателя пистолетной обоймы (рис. 7.1).

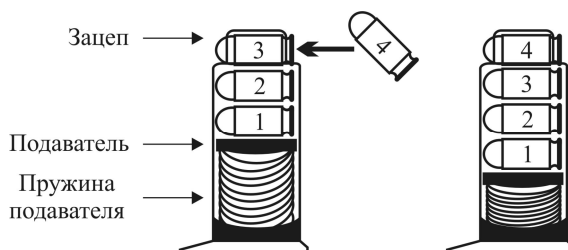


Рис. 7.1 Пистолетная обойма. Пружина подавателя стремится вытолкнуть патроны из обоймы, патроны удерживаются зацепом в верхней части обоймы. При стрельбе патроны отстреливаются в порядке противоположном снаряжению. Т.е. патрон, снаряженный последним, будет выпущен первым.

Программисты часто используют стек для временного хранения данных, например большинство языков программирования передают через стек значения параметров подпрограммам.

Пример организации стека приведен в листинге 7.2.

```

/* --- Текст "модуля" stack.h --- */
typedef      signed int      TStackElement; //Определить тип элементов стека
typedef      signed int *    PStackElement; //и тип-указатель на элемент

#define      STACK_ERROR_OK      0 //Нет ошибки
#define      STACK_ERROR_OVF     1 //Код ошибки "стек переполнен"
#define      STACK_ERROR_EMPTY   2 //Код ошибки "стек пуст"

unsigned int stack_TOP,      //Указатель вершины стека
            stack_BOTTOM,   //Указатель дна стека
            stack_SIZE;     //Количество элементов в стеке
char        stack_ERR;      //Код последней ошибки
int         *stack;

PStackElement stack_init(unsigned int Size){
    //Инициализирует стек и
    //выделяет память для хранения Size элементов типа int
    //Возвращает указатель на буфер в случае успешной инициализации
    stack_TOP=0;
    stack_BOTTOM=0;
    stack_SIZE=0;
    stack_ERR=0;

    stack=(PStackElement)malloc(Size*sizeof(TStackElement));

    if (stack!=NULL) stack_SIZE=Size;

    return stack;
}

void stack_free(void){
    //Освобождает занимаемую стеком память
    stack_TOP=0;
    stack_BOTTOM=0;
    stack_SIZE=0;
    stack_ERR=0;

    free(stack);
}

char stack_error(void){
    //Возвращает код последней ошибки при операции со стеком
    signed char err;

    err=stack_ERR;
    stack_ERR=0;

    return err;
}

void stack_push(TStackElement X){
    //Сохраняет элемент в стеке
    if (stack_TOP<stack_SIZE){
        stack[stack_TOP]=X;
        stack_TOP++;
    }
    else stack_ERR=STACK_ERROR_OVF;
}

TStackElement stack_pop(void){
    //Выталкивает (читает) элемент из стека
    if (stack_TOP!=stack_BOTTOM){
        stack_TOP--;
        return stack[stack_TOP];
    }
}

```

```

    }
    else stack_ERR=STACK_ERROR_EMPTY;
}

/* --- Текст программы stack.c, подключающей stack.h --- */
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

void main(void){
    int a, i;
    char error;

    if (stack_init(5)!=NULL){ //Создание стека, содержащего 5 элементов
        for (i=0;i<6;i++){
            stack_push(i);
            error=stack_error();
            if (error) printf("\nStack error #%i", error);
        }
        for (i=0;i<6;i++){
            a=stack_pop();
            error=stack_error();
            if (error)
                printf("\nStack error #%i", error);
            else
                printf("\n%i",a);
        }

        stack_free();
    }
}

```

Листинг 7.2 Пример организации стека и работы с ним.

Работа со стеком организована посредством 5 функций, использующих несколько глобальных переменных:

`stack_init` – выделяет из кучи память для хранения данных в стеке, инициализирует глобальные переменные;

- `stack_free` – освобождает занимаемую стеком память в куче;
- `stack_error` – возвращает код последней ошибки работы со стеком;
- `stack_push` – сохраняет значение в стеке;
- `stack_pop` – извлекает значение из стека.

Приведенный в листинге 7.2 исходный текст учитывает большинство рекомендаций хорошего тона программирования:

- специфическая функциональность реализована в виде нескольких небольших функций, каждая из которых выполняет одну логически завершенную операцию;
- специфические для организации работы со стеком функции, глобальные переменные, константы, определения типов и т.п. инкапсулированы в отдельном модуле (в данном случае, просто сведены в отдельном заголовочном файле);
- в `stack.h` контролируются специфические ошибки работы со стеком;
- коды специфических ошибок описаны в виде мнемонических констант;
- доступ к глобальным переменным осуществляется посредством функций (т.к. рекомендуется избегать непосредственного обращения к глобальным переменным) и т.п.

В последующих примерах для экономии места и концентрации внимания на специфической функциональности некоторые рекомендации “хорошего тона” не будут

учитываться, однако, при написании собственных проектов пренебрегать такими правилами не следует.

Для пояснения работы `stack.c` на рис. 7.2 приведены кадры стека в разные моменты работы программы. *Кадром стека* называют содержимое области данных стека и значения смещений указателей стека в некоторый момент времени.

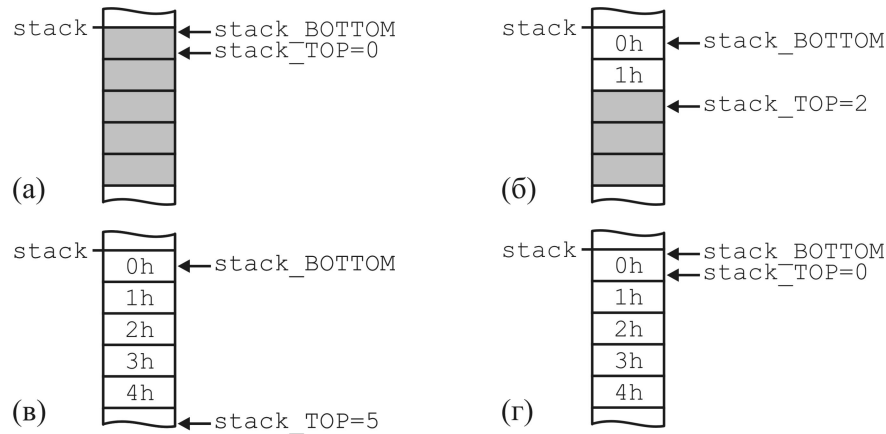


Рис. 7.2 Кадры стека: (а)– после выполнения `init_stack`, (б)– после выполнения 2 итерации первого цикла `for` (т.е. после выполнения 2 команд `stack_push`), (в)– после выполнения 5 итерации первого цикла `for`, (г)– после выполнения 5 итерации второго цикла `for`. Серым отмечены ячейки памяти, значение которых не определено. В приведенном примере, все ячейки памяти 2 байтовые.

После успешной инициализации `stack` хранит адрес выделенной области памяти (5 двухбайтовых ячеек), глобальные переменные `stack_BOTTOM` и `stack_TOP`, хранят значения 0 (рис. 7.2а).

Переменная `stack_TOP` содержит смещение (в единицах размера 1 ячейки памяти) вершины стека, т.е. индекс (относительно `stack`) двухбайтовой ячейки памяти в которую будет сохранено значение следующей командой `stack_push`.

Переменная `stack_BOTTOM` хранит смещение дна стека, в приведенном примере она всегда 0 и ее использование необязательно, однако, ее использование может повысить гибкость управления размером выделяемой стеку памяти.

При вызове `stack_push` значение `X` сохраняется в двухбайтовую ячейку, номер которой (относительно `stack`) хранится в `stack_TOP`, значение `stack_TOP` после сохранения инкрементируется. Кадр стека после 2 итераций первого цикла `for` в `stack.c` представлен на рис. 7.2б. Кадр стека после 5 итераций этого цикла приведен на рис. 7.2в, видно, что `stack_TOP` указывает на ячейку памяти вне стека. После 6 вызова `stack_push` (цикл осуществляет 6 итераций) происходит ошибка переполнения стека – `STACK_ERROR_OVF`, т.к. производится попытка записи в ячейку за пределами стека.

Аналогично, каждый вызов `stack_pop` уменьшает на 1 значение `stack_TOP` и `stack_pop` возвращает в программу значение, хранящееся в соответствующей ячейке памяти. После 5 вызовов `stack_pop` во 2 цикле `for` программы `stack.h` кадр стека имеет вид, представленный на рис. 7.2г, т.е. стек пуст и его состояние аналогично состоянию сразу после инициализации (рис. 7.2а). На 6 итерации цикла при попытке вызвать `stack_pop` для пустого стека, возникает ошибка `STACK_ERROR_EMPTY`.

### Очередь, сбор данных в многозадачной ОС

*Очередью* (англ. queue) – называют способ организации доступа к данным в соответствии с принципом “первым вошел, первым вышел” – FIFO (англ. First Input, First Output). Элемент, размещенный в очереди первым, будет извлечен из нее первым. Следующим будет извлечен элемент, размещенный в очереди вторым и т.д.

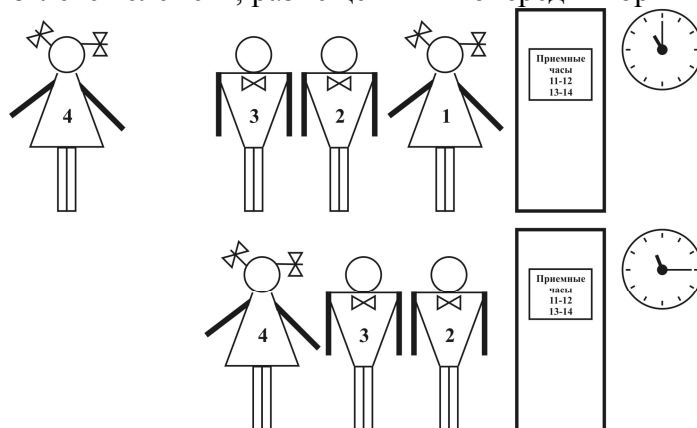


Рис. 7.3 Иллюстрация работы структуры очередь. Элемент, размещенный в очереди первым, будет извлечен из нее первым. Следующим будет извлечен элемент, размещенный в очереди вторым и т.д. Операции добавления элемента в очередь и извлечения элемента из очереди могут быть асинхронны.

Важно, что операции добавления элемента в очередь и извлечения элемента из очереди могут быть асинхронны. Благодаря этому свойству, структуры типа очередь наиболее часто используются в качестве буфера при обмене данными в режиме ПДП между периферийным оборудованием и ЭВМ, работающей под управлением многозадачной ОС (рис. 7.4).

Многозадачные ОС эмитируют для пользователя одновременное выполнение нескольких приложений. Для этого процессор поочередно циклично переключается между задачами, затрачивая на выполнение каждой некоторое время. Большинство ОС (Windows, большинство Linux, MAC OS и др.) не могут гарантированно зафиксировать время, которое будет проходить между двумя обращениями к одной задаче, осуществляющей сбор данных с периферийного устройства. (ОС, которые могут гарантировать время, затрачиваемое на выполнение отдельной пользовательской задачи, называют ОС жесткого времени.)

Синхронизацию периферийного устройства, например, АЦП и ПК для обработки данных в реальном времени осуществляют следующим образом. АЦП сохраняет выборки, получаемые от исследуемого объекта через равные интервалы времени, в ОП ЭВМ, используя канал ПДП. Прямой доступ к памяти реализуется контроллером шины и не нагружает ядро CPU, занятое выполнением прикладных и системных задач. Область ОП, куда АЦП сохраняет данные, организуется в виде структуры очередь. Когда процессор переключается для выполнения задачи обработки полученных данных, программа обработки определяет количество выборок данных, полученных за то время, пока процессор выполнял другие задачи и обрабатывает полученные данные.

Для исключения потери данных при организации кольцевого буфера FIFO должны выполняться 2 условия: среднее время, затрачиваемое процессором на обработку одного отсчета АЦП меньше времени выборки и длина буфера FIFO достаточно велика для компенсации времени, затрачиваемого процессором на решение всех запущенных задач.

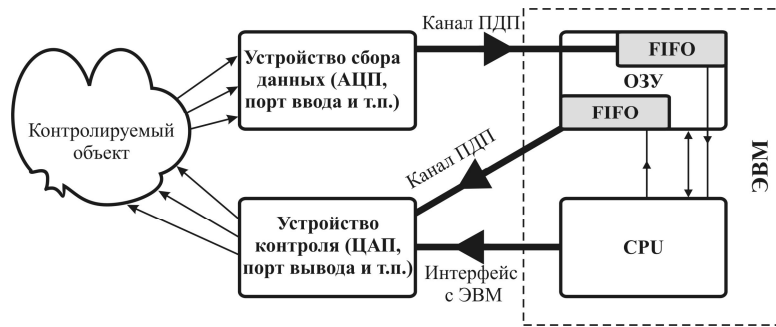


Рис. 7.4 Организация системы сбора данных и контроля, работающей в реальном времени. Центральный процессор ЭВМ, работающей под управлением, многозадачной ОС, синхронизует свою работу с периферийными устройствами сбора данных и контроля посредством буферов FIFO в ОЗУ. Периферийные устройства обмениваются данными с ОЗУ ЭВМ через каналы прямого доступа к памяти. Если устройство контроля не должно обеспечивать отклика, синхронного с устройством сбора данных, то его управление может осуществляться центральным процессором асинхронно, например, через какой-нибудь стандартный интерфейс взаимодействия ЭВМ с внешними устройствами.

### Кольцевой буфер, линия задержки

В примере, приведенном на рис. 7.3, стоящие в очереди люди (элементы буфера FIFO) перемещаются после того, как очередной человек пройдет на прием. При практической реализации структуры FIFO перемещение элементов в памяти компьютера является для процессора достаточно трудоемкой задачей. Поэтому, как и при организации стека, обычно элементы очереди не перемещаются, а изменяются только значения указателей чтения и записи, хранящих, соответственно, смещение элемента, который будут считан последующей операцией чтения и смещение ячейки памяти, в которой будет размещен элемент последующей операцией записи. Обычно, FIFO организуют в виде *кольцевого буфера* (англ. circular buffer). Пример организации такой структуры приведен в листинге 7.3.

```
#include <stdio.h>

typedef int    TQueueElement; //Обозначаем тип - элемент очереди
typedef int*   PQueueElement; //и тип - указатель на элемент очереди

unsigned int   queue_Write, //Смещение ячейки для записи
               queue_Read,  //Смещение элемента для чтения
               queue_Size ; //queue_Size-1 - количество элементов в очереди
PQueueElement queue_P;      //Указатель на область ОП, где размещается буфер

void queue_init(PQueueElement P, unsigned int Size){
    //Инициализирует очередь размером Size-1 элементов
    //Буфер будет размещен по адресу P
    queue_Write=0;
    queue_Read=0;
    queue_Size=Size;
    queue_P=P;
}

void queue_put(TQueueElement X){
    //Сохраняет элемент в очереди
    queue_P[queue_Write]=X;
    queue_Write++;
    if (queue_Write==queue_Size) queue_Write=0;
}

TQueueElement queue_get(void){
    //Извлекает элемент из очереди
    TQueueElement X;
```

```

X=queue_P[queue_Read];
queue_Read++;
if (queue_Read==queue_Size) queue_Read=0;
return X;
}

unsigned int queue_count(void){
    //Возвращает количество элементов в очереди
    if (queue_Write>=queue_Read)
        return queue_Write-queue_Read;
    else
        return queue_Size-queue_Read+queue_Write;
}

void main(void){
    #define SIZE 5
    TQueueElement Data[SIZE];
    int i, X;
    //Инициализируем очередь
    queue_init(Data, SIZE); //Буфер размещается по адресу Data
                             //и может хранить до SIZE-1, т.е. 4 элементов

    //Добавляем в очередь 3 элемента
    queue_put(1); printf("\nCount=%i", queue_count());
    queue_put(2); printf("\nCount=%i", queue_count());
    queue_put(3); printf("\nCount=%i", queue_count());
    //Извлекаем из очереди 2 элемента
    X=queue_get(); printf("\nX=%i, Count=%i", X, queue_count());
    X=queue_get(); printf("\nX=%i, Count=%i", X, queue_count());

    //Добавляем в очередь 6 элементов
    for (i=1;i<=6;i++){
        queue_put(i+3); printf("\nCount=%i", queue_count());
    }
}

```

Листинг 7.3 Пример организации на языке C структуры “очередь” в виде кольцевого буфера FIFO.

Для организации работы с кольцевым буфером в рассмотренном примере используется 4 функции: инициализации буфера: `queue_init`, добавления элемента в очередь: `queue_put`, извлечения элемента из очереди: `queue_get` и проверки количества элементов в очереди: `queue_count`. Как и в случае организации стека при добавлении/извлечении элемента перемещения данных в памяти не происходит, а изменяются лишь значения указателей чтения и записи очереди. При этом, достигая конца выделенного для буфера участка памяти – “хвоста” очереди, указатели “перепрыгивают” на начало массива данных. Графически можно представить организацию такого буфера в виде массива, конец которого замкнут с его началом (рис. 7.5). При этом указатели чтения и записи “вращаются” вокруг этой структуры в одном направлении. Когда указатель записи совершит “полный оборот” вокруг кольцевого буфера, наиболее старые данные постепенно будут перетираться новыми данными. Если эти данные еще не были прочитаны, то они будут потеряны.

Из листинга 7.3 видно, что в цикле происходит переполнение кольцевого буфера. В англоязычной литературе для обозначения ситуации не приводящего к ошибке переполнения кольцевого буфера часто используют термин *overrun*, в отличие от переполнения линейного буфера (например, стека) – *overflow*. Аналогично, чтение пустого кольцевого буфера является ошибкой *overread* или *underflowing*, а линейного буфера – *emptying*.

Для экономии места пример, приведенный в листинге 7.3, не содержит средств выявления и учета ошибок десинхронизации, имеющих, например, в листинге 7.2, однако, на практике, такие средства контроля желательно обеспечить.

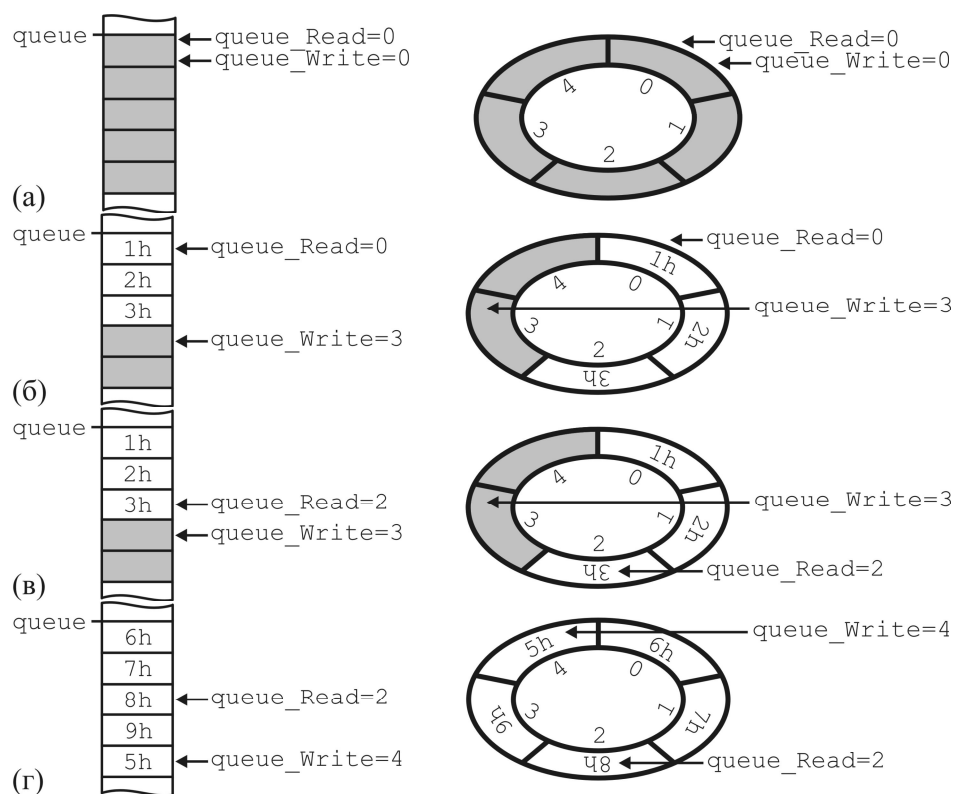


Рис. 7.5 Кадры кольцевого буфера FIFO из листинга 7.3. (а)– Сразу после инициализации буфера. (б)–После размещения в буфере 3 элементов. (в)–После извлечения из буфера 2 элементов. (г)–После добавления в буфер еще 6 элементов. Все ячейки памяти двухбайтовые (int). Слева схематически изображена линейная структура участка памяти, справа – графическое представление кольцевого буфера: конец буфера “замкнут” с его началом. Серым цветом отмечены ячейки памяти, значения в которых не определены.

В задачах обработки и генерации данных иногда нужно организовывать задержку сигнала на несколько выборок. Например, задержка бывает нужной для организации синхронной обработки 2 сигналов, один из которых заведомо запаздывает на фиксированное время относительно второго. Синхронизация возможна с помощью программной задержки второго сигнала. Линия задержки необходима для программной реализации широкого класса колебательных систем – автогенераторов с запаздывающей обратной связью.

Для организации линии задержки обычно используют кольцевой буфер, у которого указатель записи дополнительно смещен вперед, относительно указателя чтения, на константу, равную времени запаздывания (в единицах количества выборок).

### Связанный список

*Связанные списки* (англ. list) широко используются в практике программирования для хранения различных данных. Основным преимуществом связанных списков перед массивами является гибкость при использовании ОП: ставшие ненужными элементы немедленно удаляются из памяти, новые элементы списка могут храниться в произвольных местах ОП, в отличие от массивов, требующих выделения в куче непрерывного свободного участка. Размер связанного списка ограничен только объемом свободной ОП. Использование списков дает также преимущества при решении многих задач, сводящихся к графам, задач, подразумевающих иерархическую организацию данных и т.п. Недостатки связанных списков:



- использование дополнительной памяти для хранения указателей на *дочерние* (англ. child) и возможно, *родительские элементы* или *элементы-владельцы* (англ. owner);
- работа со списком, обычно, медленнее, чем с массивом;
- связанные списки сегментируют кучу.

*Односвязным* (англ. simply connected) списком называют список, каждый элемент которого хранит единственный указатель на дочерний элемент. Навигация по такому списку возможна только в одну сторону от переменной-указателя на *корневой* (англ. root) или *головной* (англ. head) элемент.

Более гибкая и быстрая навигация возможна по двусвязному (англ. doubly-connected) списку, каждый элемент которого хранит адреса и дочернего и родительского элементов.

Связанный список обычно организуется в виде иерархии структур (struct), каждая из которых имеет поле-указатель на дочернюю структуру. Например, односвязный список из структур TList:

```
typedef struct{
    int y;           //Переменная для хранения данных
    void *Child;     //Указатель на дочернюю структуру
} TList;
```

будет храниться в памяти, примерно так, как схематически изображено на рис 7.6.

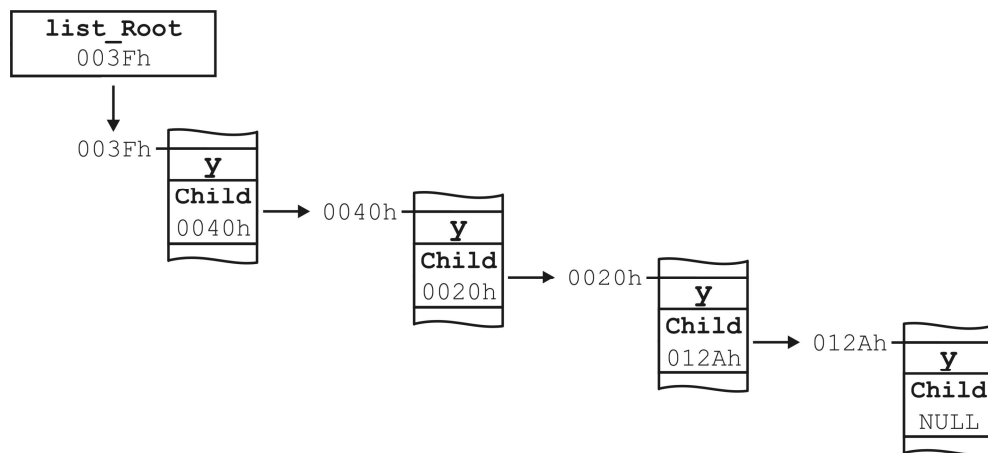


Рис. 7.6 Размещение в ОП данных, организованных в виде односвязного списка.

Пример организации двусвязного списка схематически представлен на рис. 7.7. Исходный текст программы, организующей двусвязный список приведен в листинге 7.4. Нужно отметить, что код функций, реализованных в приведенном примере для работы со списком “оптимизирован для понятности” исходного текста. При его разработке автор старался улучшить читаемость, иногда, в ущерб эффективности.

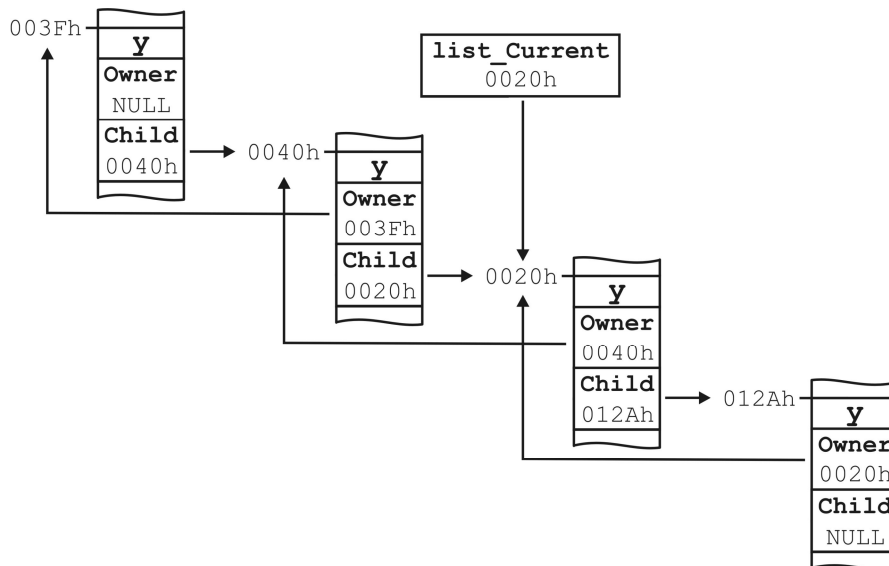


Рис. 7.7 Вариант организации связанного списка – двусвязный список. Кроме адреса дочернего элемента такой список хранит указатель на элемент-владелец.

Функции: `list_index_get`, `list_index_put` и `list_index_delete` в представленном примере избыточны. Они предназначены для эмуляции работы со связанным списком, как с массивом, элементы которого можно гибко удалять, освобождая память и добавлять в процессе работы.

```

#include <stdio.h>
#include <alloc.h>

typedef int      TListData; //Объявление типа для хранения данных в списке

typedef struct{ //Структура представляет собой элемент двусвязного списка
    TListData y;      //Переменная для хранения данных в списке

    void      *Owner, //Указатель на структуру-владельца
    *Child; //Указатель на дочернюю структуру
} TList;

TList      *list_Current=NULL; //Указатель на текущий элемент списка
unsigned int list_Count=0,      //Количество элементов в списке
            list_Current_Number=0; //Индекс текущего элемента в списке

void list_last(void){
    //Переместить указатель list_Current на последний элемент списка
    if (list_Current!=NULL){
        while (list_Current->Child!=NULL)
            list_Current=(TList*)(list_Current->Child);
        list_Current_Number=list_Count-1;
    }
}

void list_first(void){
    //Переместить указатель list_Current на первый элемент списка
    if (list_Current!=NULL){
        while (list_Current->Owner!=NULL)
            list_Current=(TList*)(list_Current->Owner);
        list_Current_Number=0;
    }
}

void list_next(void){
    //Переместить указатель list_Current на следующий элемент списка

```

```

    if ((list_Current!=NULL)&&((list_Current->Child!=NULL))) {
        list_Current=(TList*)(list_Current->Child);
        list_Current_Number++;
    }
}

void list_previous(void) {
    //Переместить указатель list_Current на предыдущий элемент списка
    if ((list_Current!=NULL)&&((list_Current->Owner!=NULL))) {
        list_Current=(TList*)(list_Current->Owner);
        list_Current_Number--;
    }
}

void *list_add(TListData x) {
    //Добавить элемент в конец списка
    TList *P=NULL; //Указатель на новый элемент

    if ((P=(TList*)malloc(sizeof(TList)))!=NULL) { //Выделить память
                                                //для нового элемента
        //Инициализируем поля созданной структуры:
        P->y=x; //Инициализируем поле данных
        P->Owner=NULL; //Владелец (если будет) будет определен ниже
        P->Child=NULL; //Т.к. элемент добавляется в конец списка,
                        //то он не будет иметь дочернего

        list_last(); //Переместить list_Current на последний элемент списка

        if (list_Current!=NULL) { //Если список не был пуст
            //Владельцем нового элемента станет последний элемент списка:
            P->Owner=list_Current;
            //Новый элемент станет дочерним для последнего элемента списка:
            list_Current->Child=P;
        }
        list_Current=P; //list_Current указывает на добавленный элемент списка

        list_Count++; //Инкремент общего количества элементов в списке
        //Индекс текущей (последней) записи станет list_Count-1:
        list_Current_Number=list_Count-1;
    }
    //Вернуть указатель на добавленную структуру или NULL, если нет памяти
    return P;
}

void list_delete(void) {
    //Удалить текущий элемент из списка и освободить занимаемую им ОП
    TList *P; //Переменная для временного хранения адреса текущей структуры

    if (list_Current!=NULL) { //Если список не пуст
        P=list_Current; //Сохраним адрес текущей структуры
        //Если текущий элемент единственный (нет ни владельца, ни дочернего):
        if ((list_Current->Owner==NULL)&&(list_Current->Child==NULL))
            list_Current=NULL;
        //Если текущий элемент первый, но не единственный (нет владельца):
        else if (list_Current->Owner==NULL) {
            //Текущим станет следующий элемент:
            list_Current=(TList*)(list_Current->Child);
            //Этот элемент отмечается как первый (без владельца)
            list_Current->Owner=NULL;
        }
        //Если текущий элемент последний, но не единственный (нет дочернего):
        else if (list_Current->Child==NULL) {
            //Делаем текущим предыдущий элемент:
            list_previous();
        }
    }
}

```

```

        //и помечаем его, как последний
        list_Current->Child=NULL;
    }
    else{ //Если удаляется элемент в середине списка
        //Дочерней структурой предыдущего элемента
        list_previous();
        //станет дочерний элемент P
        list_Current->Child=P->Child;
        //Владельцем элемента, следующего за удаляемым, станет
        list_next();
        //бывший владелец P
        list_Current->Owner=P->Owner;
    }

    free(P); //Освобождаем память в куче, занимаемую удаляемым элементом
    list_Count--; //Декремент общего количества элементов в списке
}

TListData list_get(void){
    //Возвращает значение поля данных текущей записи
    return list_Current->y;
}

void list_put(TListData x){
    //Изменяет значение поля данных текущей записи на x
    list_Current->y=x;
}

TListData list_index_get(unsigned int index){
    //Возвращает значение поля данных элемента с индексом index
    unsigned int i;

    if (index>=list_Count) return 0;

    list_first();
    for (i=0;i<index;i++) list_next();

    return list_get();
}

void list_index_put(unsigned int index, TListData x){
    //Изменяет значение поля данных элемента с индексом index на x
    unsigned int i;

    if (index<list_Count){
        list_first();
        for (i=0;i<index;i++) list_next();
        list_put(x);
    }
}

void list_index_delete(unsigned int index){
    //Удаляет элемент с индексом index
    unsigned int i;

    if (index<list_Count){
        list_first();
        for (i=0;i<index;i++) list_next();
        list_delete();
    }
}

void main(void){

```

```

    //В программе эмулируется работа со связанным списком, как с массивом
#define SIZE 5 //Размер "массива"
int i, j, swap;

for (i=0;i<SIZE;i++){
    list_add(i*2); //Создать связанный список, содержащий (0,2,4,6,8)
}

list_index_delete(3); //Удалить из списка элемент с индексом 3

//Сортировка данных в списке - "массива" - по убыванию
for (j=0;j<SIZE;j++){
    for (i=0;i<SIZE-1;i++){
        if (list_index_get(i+1)>list_index_get(i)){
            swap=list_index_get(i);
            list_index_put(i,list_index_get(i+1));
            list_index_put(i+1,swap);
        }
    }
}

//Вывести значения элементов связанного списка на экран:
for (i=0;i<list_Count;i++){
    printf("\nIndex=%i, Element=%i", i, list_index_get(i));
    //Выведет значения 8, 4, 2, 0
}
}

```

Листинг 7.4 Пример организации двусвязного списка и эмуляции работы с ним, как с массивом.

## Лекция 8 Введение в организацию работы ЭВМ и язык ассемблера

Язык ассемблера. Структура ЭВМ. Регистры процессора. Сегментная адресация памяти. Подготовка и создание программы на языке ассемблера. Работа видеоадаптера ПЭВМ в текстовом режиме. Простейшая программа на языке ассемблера.

### Язык ассемблера

Первые ЭВМ появились в 40-х годах XX века. Эти устройства занимали несколько больших комнат, по вычислительной мощности были сопоставимы с микроЭВМ, встраиваемых в современные сотовые телефоны и программировались путем непосредственного указания числовых кодов операций и номеров ячеек памяти. Вскоре появились более совершенные ЭВМ, “понимающие” десятки команд, программы усложнились и их размер превысил сотню команд. Для упрощения труда программистов был разработан язык *ассемблера* (от англ. assembler – сборщик), позволяющий заменить числовые коды команд символьными сокращениями или аббревиатурами, а номера ячеек памяти мнемониками имен переменных.

Транслятор ассемблера при генерации машинного кода заменяет эти аббревиатуры и мнемонические обозначения численными кодами операций и номерами (адресами) ячеек памяти. Однозначность такой трансляции символьных обозначений в коды допускает обратную операцию – дизассемблирование – преобразование машинных кодов в мнемонические обозначения команд. Понятно, что при таком подходе программа оказывается привязанной к системе команд конкретной ЭВМ или процессора. Однако, несмотря на громоздкий исходный текст с множеством различных команд и плохую переносимость между различными семействами ЭВМ, ассемблер оставался практически единственным языком программирования до 60-х годов XX века. Такую живучесть обеспечили прямота и четкость этого языка, и эффективность результирующего машинного кода. В конце 50-х годов размер типичной программы перерос 1000 строк, кроме того, появилось большое количество семейств ЭВМ, и разработчики предложили процедурно-ориентированные языки программирования – языки “высокого уровня”, наиболее известные из которых Fortran, Algol, Pascal и C. Эти языки обеспечивали существенно лучшую переносимость программ и предоставили ряд удобств программистам, однако, их преимущества были достигнуты ценой снижения эффективности машинного кода и потери возможности однозначной трансляции-дизассемблирования программ.

В настоящее время, скорости разработки прикладных программ уделяется большое внимание. Поэтому, основными инструментами разработки стали языки высокого уровня и даже интерпретируемые языки (C#, Python и др.). Однако, “низкоуровневое программирование” на языке ассемблера до сих пор применяется для решения некоторых типов задач, и в обозримом будущем полный отказ от использования ассемблера не ожидается.

Задачами, требующими использования ассемблера являются:

- разработка системного программного обеспечения (операционных систем и их компонент);
- разработка программных драйверов периферийных устройств;
- анализ и коррекция кода программы при недоступном исходном тексте (типичные задачи хакинга, изучение компьютерных вирусов, полицейские мероприятия и т.п.);
- анализ и математическая обработка сигналов в реальном времени и др.

Таким образом, ассемблер часто используют для решения задач в областях биомедицинской инженерии, промышленной автоматизации и др.

Программы, написанные на языке ассемблера полностью, сейчас редкость даже в области программирования микроконтроллеров и сигнальных процессоров. Современные языки программирования, например, C, позволяют использовать отдельные функции, написанные на ассемблере и даже ассемблерные вставки, как часть кода функции,

написанной на языке высокого уровня. Такой подход позволяет сочетать простоту разработки и отладки языка высокого уровня с высокой эффективностью наиболее критичных участков кода, написанных на ассемблере.

Образно, использование языка ассемблера можно сравнить с попыткой наладить деловое общение с китайцем. Вы можете общаться с китайцем на универсальном английском – это программирование на С, можете говорить через переводчика – эквивалент использования интерпретируемого языка, например, Python, но наиболее эффективная коммуникация получится, в случае, если вы освоите сложный китайский язык и заговорите с китайцем на родном для него языке.

Еще одной сильной стороной ассемблера являются его методические качества. Это родной язык ЭВМ, поэтому, понимание ассемблера существенно помогает в профессиональном освоении любого языка высокого уровня, позволяя понять внутренние механизмы функционирования этого языка.

### Структура ЭВМ

Одной из основных сложностей, с которой сталкивается начинающий программист на ассемблере, является необходимость понимания некоторых особенностей устройства ЭВМ на аппаратном уровне. Базовые сведения, необходимые для понимания основных принципов организации работы любой ЭВМ сведены в этом разделе. Нужно понимать, что приведенные здесь сведения очерчивают собирательный образ, и не являются универсальными для всех ЭВМ. Например, далеко не все современные процессоры имеют каналы прямого доступа к памяти, ряд современных процессоров имеют шину для быстрого доступа к ОЗУ встроенную непосредственно в кристалл и т.п. Блок схема, содержащая, с учетом сделанных оговорок, основные структурные элементы ЭВМ приведена на рис. 8.1.

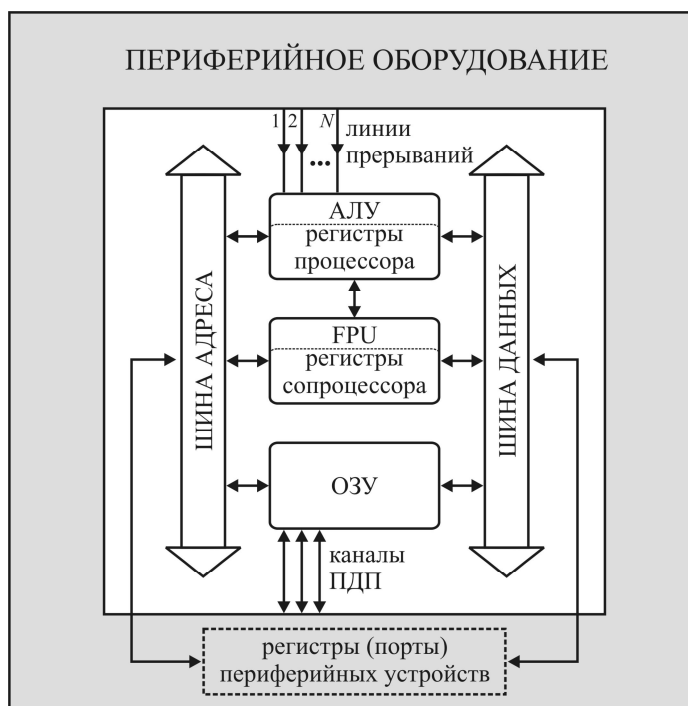


Рис. 8.1 основные структурные элементы ЭВМ.

- Центральным элементом ЭВМ является *арифметико-логическое устройство* (АЛУ, англ. arithmetic and logic unit, ALU), обеспечивающее выполнение программы и осуществляющее управление остальными блоками. Современные процессоры могут иметь несколько блоков АЛУ, работающих параллельно.

- Основной обмен данными между АЛУ, ОЗУ, математическим сопроцессором и периферийными устройствами осуществляется через *шину адреса* (англ. address bus) и *шину данных* (англ. data bus). Шина предназначена для обмена цифровой информацией между несколькими устройствами. Физически, она представляет собой набор *линий* (англ. lines) – проводников и контролирующую логику. К настоящему моменту разработаны и используются в различных системах десятки стандартов шин, как последовательных, так и параллельных (передающих несколько битов в один момент времени). Некоторые из этих стандартов подразумевают передачу и данных и адреса по одним и тем же линиям. Многие ЭВМ, например, современные ПК могут иметь несколько шин, используемых для обмена данными между различными блоками и элементами.
- Оперативное запоминающее устройство (ОЗУ, англ. random-access memory, RAM) предназначено для временного хранения данных, используемых программой. Основными характеристиками ОЗУ является его объем и скорость с которой оно может обмениваться данными с АЛУ.
- Обычно, АЛУ может выполнять инструкции только целочисленной арифметики. Работу с вещественными числами оно *эмулирует* (англ. emulate), т.е. симулирует. Эмуляция существенно замедляет вычисления, поэтому, уже в середине 80-х годов XX века, были разработаны в виде отдельных устройств т.н. *математические сопроцессоры* (англ. mathematical coprocessor), позволяющие быстро осуществлять арифметические действия с вещественными числами. Вещественные числа представляются в ЭВМ, как числа с плавающей точкой (англ. floating point), поэтому, по-английски такие устройства называют Floating Point Unit (FPU). Сейчас многие процессоры имеют по несколько блоков FPU, размещенных на одном кристалле с АЛУ. Однако, микроконтроллеры и сигнальные процессоры обычно имеют только блоки для целочисленной арифметики, а FPU не имеют.
- Регистры процессора представляют из себя быстродействующие ячейки памяти, расположенные непосредственно в АЛУ. Регистры используются для хранения операндов, параметров, промежуточных и окончательных результатов арифметических и логических операций, для обмена данными с ОЗУ, для управления работой процессора и контроля его состояния. В зависимости от архитектуры процессора и назначения регистра, регистры могут иметь разрядность 8, 16, 32, 64 или 80 бит. С точки зрения программиста, программирование процессора фактически сводится к программированию его регистров. В различных системах программный доступ к регистрам может осуществляться по имени регистра, по его номеру (в этом случае, говорят: “по адресу регистра в адресном пространстве ввода-вывода”), адресное пространство регистров может встраиваться в адресное пространство ОЗУ (в этом случае, к регистру обращаются, как к ячейке ОЗУ с фиксированным адресом). Набор регистров конкретного процессора строго определен и подробно описан в технической документации процессора, часто этот набор называют *регистровым файлом* (англ. register file) данного процессора. Современные процессоры имеют более сотни регистров различного назначения.
- Регистры (порты) периферийного устройства представляют из себя ячейки памяти, расположенные в микросхемах управляющей логики этого устройства. Их назначение аналогично назначению регистров процессора: управление и контроль состояния устройства, обмен данными с АЛУ. Список регистров устройства, правила их адресации и использования подробно описываются в технической документации (англ. manual, developer’s manual, programmers’s guide) на данное устройство.
- Механизм обработки *прерываний* (англ. interrupt) широко используется на практике для взаимодействия с периферийным оборудованием. ЭВМ, обычно, имеет несколько десятков линий прерываний, представляющих собой отдельные проводники, соединяющие периферийное устройство с контроллером прерываний



АЛУ. При посылке сигнала запроса на прерывание (англ. *Interrupt reQuest, IRQ*) периферийным устройством контроллеру прерываний по линии  $k$ , АЛУ приостанавливает выполнение текущей программы, запоминает адрес следующей команды  $NI$  текущей программы и немедленно начинает выполнение процедуры-обработчика прерывания номер  $k$  (говорят: осуществляется переход на вектор прерывания  $k$ ). Естественно, адрес процедуры обработчика прерывания  $k$ , должен быть предварительно загружен в специальную системную область памяти – таблицу векторов прерываний программистом, пишущим прикладную программу. После выполнения функции-обработчика, контроллер прерываний продолжает выполнять задачу, прерванную возбуждением прерывания (англ. *interrupt execution*), начиная с запомненного адреса возврата  $NI$ . Взаимодействие с периферийными устройствами посредством механизма обработки прерываний используется в случаях, требующих немедленной реакции процессора на событие периферийного устройства, например, в системах контроля, при синхронном сборе данных с помощью внешнего АЦП и т.п.

Прерывания могут возбуждаться самим АЛУ в случае возникновения исключительных ситуаций (англ. *exсerption*), например, при ошибках арифметических операций типа деления на 0.

Прерывания также могут возбуждаться программно по команде программиста. Программный вызов прерываний широко используется, например, в прикладных программах MS-DOS для организации взаимодействия с операционной системой.

- Каналы прямого доступа к памяти – ПДП предоставляют возможность периферийным устройствам писать данные в ОЗУ и читать из него информацию в обход АЛУ. Это позволяет осуществлять обработку непрерывно поступающих от периферийного устройства данных без потери и искажения информации даже если АЛУ работает в многозадачном режиме и его загрузка меняется во времени непредсказуемо. Типовой задачей, требующей использования ПДП, является ввод/вывод данных через АЦП/ЦАП для процессора, работающего в многозадачном режиме. Для ПЭВМ это, например, запись звука и видео, синтез звука и т.п.
- Центральным процессором – CPU называют микросхему, содержащую АЛУ. При этом современные процессоры могут иметь на своем кристалле несколько параллельно работающих АЛУ (многоядерный процессор), несколько FPU, контроллеры шин, и даже графический контроллер.

### Регистры процессора

Работа с регистрами процессора является одним из основных средств общения с ним программиста, поэтому, принципы их использования должен понимать каждый профессиональный программист. Традиционно, для иллюстрации используют систему регистров процессора i8086. С одной стороны, этот процессор имеет всего 13 программно доступных 16 битовых регистров, что упрощает изучение, с другой стороны, все производители процессоров для IBM-PC совместимых компьютеров неуклонно соблюдают принцип обратной программной совместимости, и все разработанные процессоры для PC могут работать в режиме совместимости с i8086, используя именно эти регистры. При работе на других системах, например, с микроконтроллерами, названия регистров, конечно, не сохраняются, однако, основные принципы организации работы с регистрами остаются неизменными.

Информация о регистрах процессора i8086 сведена в таблице 8.1.

Регистры общего назначения AX, BX, CX и DX могут использоваться программистом произвольным образом. Более того, старшие (англ. *High*) и младшие (англ. *Low*) байты этих регистров: AH, BH, CH, DH, AL, BL, CL, DL, соответственно, имеют собственные имена, к которым программист также может обращаться. Названия этих регистров указывают на специфические команды, с которыми может использоваться только

конкретный регистр. Например, при использовании команды организации цикла LOOP в качестве счетчика цикла может использоваться только регистр CX. Многие команды, однако, допускают использование в качестве операнда любого из регистров общего назначения.

Регистры общего назначения			Регистры-указатели	
<b>AX</b>	<b>AH</b>	<b>AL</b>	<b>SI</b>	Индекс источника
<b>BX</b>	<b>BH</b>	<b>BL</b>	<b>DI</b>	Индекс приемника
<b>CX</b>	<b>CH</b>	<b>CL</b>	<b>BP</b>	Указатель базы
<b>DX</b>	<b>DH</b>	<b>DL</b>	<b>SP</b>	Указатель стека
Сегментные регистры			Прочие регистры	
	<b>CS</b>	Регистр сегмента команд	<b>IP</b>	Указатель команд
	<b>DS</b>	Регистр сегмента данных	<b>FLAGS</b>	Регистр
	<b>ES</b>	Дополнительный сегментный регистр		
	<b>SS</b>	Регистр сегмента стека		

Таблица 8.1 Назначение регистров процессора i8086.

Индексные регистры SI, DI и в большинстве случаев, BP так же, как и регистры общего назначения могут использоваться произвольным образом, однако, основное назначение этих регистров – хранить смещения в обрабатываемых массивах данных и работать со стеками (BP). Указатель стека SP используется исключительно для хранения смещения вершины стека. Регистры-указатели, в отличие от регистров общего назначения, не допускают побайтовую адресацию.

Сегментные регистры предназначены для хранения адресов сегментов программы (о сегментах пойдет речь ниже) и не могут использоваться для каких иных целей.

Указатель IP хранит адрес команды, которая будет выполняться процессором. Значение, хранящееся в IP изменяется автоматически в соответствии с логикой программы. Этот регистр недоступен программно ни для чтения, ни для записи и предназначен для внутреннего использования АЛУ.

Регистр FLAGS хранит битовые коды состояния процессора, которые, обычно, называют флагами. Его структура приведена в таблице 8.2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				<b>OF</b>	<b>DF</b>	<b>IF</b>	<b>TF</b>	<b>SF</b>	<b>ZF</b>		<b>AF</b>		<b>PF</b>		<b>CF</b>

Таблица 8.2 Регистр состояния процессора FLAGS.

CF – флаг переноса (англ. carry flag) устанавливается в 1, если последняя арифметическая операция вызвала перенос.

PF – флаг четности (англ. parity flag) устанавливается в 1, если младший байт результата последней операции содержит четное число единиц в двоичной записи.

AF – дополнительный флаг переноса (англ. auxiliary flag) используется при операциях с двоично-десятичными числами.

ZF – флаг нуля (англ. zero flag) устанавливается в 1, если результат последней арифметической операции был 0.

SF – флаг знака (англ. sign flag) устанавливается в 1, если результат последней арифметической операции был отрицательным числом.

OF – флаг переполнения (англ. overflow flag) индицирует выход результата последней операции за пределы допустимого диапазона значений.

Значения перечисленных выше битовых флагов в регистре FLAGS автоматически обновляются процессором после выполнения каждой команды и программно доступны

для чтения. Проверка состояния битовых флагов необходима, в частности, для организации условных переходов – ветвлений.

Значения флагов TF, IF и DF доступны и для чтения и для записи и должны устанавливаться программистом. Эти 3 флага имеют следующие назначения:

TF – флаг трассировки (англ. trace flag) используется в режиме отладки программы.

IF – глобальный флаг прерываний (англ. interrupts flag) разрешает, если установлен, запускать обработчики прерываний периферийных устройств.

DF – флаг направления (англ. direction flag) используется для управления командами работы с массивами.

### Сегментная адресация памяти

“Проклятием” современных ЭВМ, особенно, ПК, является необходимость обеспечения производителями обратной программной совместимости с предыдущими семействами процессоров для удобства пользователей. Из-за этого, даже современные процессоры несут в своей архитектуре “детские болезни” еще со времен i8086. Одной из таких “детских болезней” является необходимость стартовать все процессоры для IBM-PC в однозадачном *режиме реального адреса* (реальный режим, англ. real mode), в котором обеспечивается полная программная совместимость с i8086 с сегментной адресацией памяти, обеспечивающей адресацию не более чем 1 Мб ОЗУ. В этом режиме функционируют ОС реального режима, например, MS-DOS. Для полного использования возможностей современного процессора он может быть программно переведен в *режим защищенного адреса* (защищенный режим, англ. protected mode), в котором аппаратно поддерживается многозадачность и реализуется существенно усовершенствованный вариант сегментной адресации, практически снимающий ограничения на объем адресуемой ОП.

Однако для понимания различных способов адресации необходимо понимать их основные принципы и исторические причины возникновения, которые, как уже упоминалось выше, во многом определили особенности современной архитектуры процессоров для IBM-PC совместимых ПЭВМ. Поэтому, ниже будет подробно рассмотрена сегментная адресация процессоров i8086.

Несколько упрощенная схема взаимодействия АЛУ и ОЗУ процессора i8086 представлена на рис. 8.2.

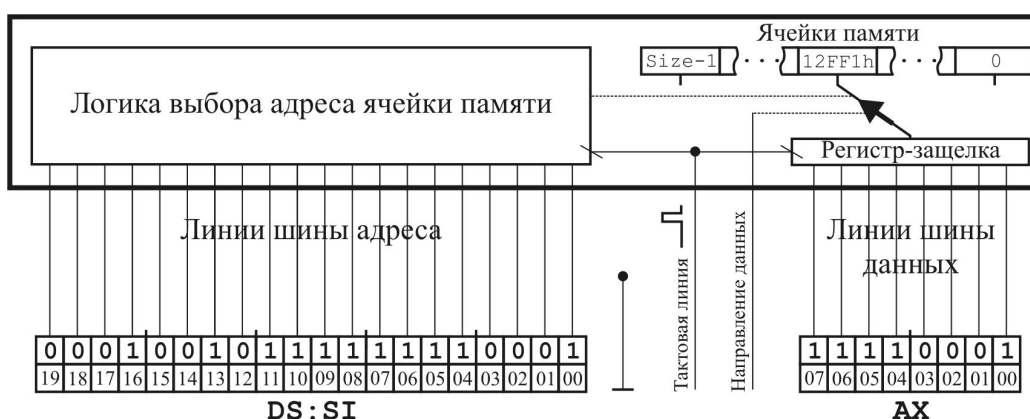


Рис. 8.2 Упрощенная схема для иллюстрации организации взаимодействия АЛУ и ОЗУ процессора i8086.

Для передачи/чтения байта в/из ОЗУ используется 8-битная шина данных. Физически, она организована в виде 8 проводников (линий), причем отсутствие напряжения на проводе (относительно некоторого общего электрода) интерпретируется логическими схемами, как логический 0, а присутствие напряжения (обычно, около 5 В) – как логическая 1. Адрес (номер) ячейки памяти байтового размера задается здесь путем выставления логических

уровней напряжения на линии 20 разрядной шины адреса. Легко подсчитать, что 20 разрядная (т.е. содержащая физически 20 проводников) шина адреса может непосредственно адресовать память объемом до  $2^{20} = 1048576 = 1 \text{ Мб}$ .

При передаче байта реализуется следующий алгоритм работы схем ОЗУ и АЛУ:

1. 8 бит передаваемого байта выставляются на 8 соответствующих линий шины данных в виде логических уровней напряжения.
2. 20 битный прямой адрес ячейки ОЗУ выставляется на 20 соответствующих линий шины адреса.
3. Линия выбора направления данных позволяет определить, чтение или запись ОЗУ будет производиться.
4. По линии тактирования шины (англ. bus clock line) АЛУ подает на входы-защелки адресной логики и временного регистра ОЗУ прямоугольный импульс, который вызывает 2 действия:
5. логическая схема адресации ОЗУ, управляемая параллельным кодом шины адреса, физически соединяет ячейку памяти, адрес которой задан двоичным кодом, выставленным на линиях адресной шины с регистром-защелкой временного хранения данных. Регистр-защелка непосредственно соединен также с линиями шины данных.
6. Данные с линий шины данных, сохраненные в регистре-защелке временного хранения, сохраняются в выбранной ячейке ОЗУ, либо, в зависимости от выбранного направления передачи данных, значения битов ячейки ОЗУ выставляются на линии шины данных.

Эта несложная схема обмена данными с ОЗУ не вызвала бы никаких проблем у программистов, если бы для адресации можно было использовать один регистр процессора, однако, i8086 содержит лишь 16 разрядные регистры, поэтому, для задания адреса ячейки памяти приходится использовать пару регистров.

Эта необходимость использования для адресации пары регистров и привела к введению в процессоре i8086 сегментной адресации. Такой метод адресации продемонстрировал ряд преимуществ и получил развитие.

Сегментная адресация для процессора i8086 реализуется так:

1. В одном из сегментных регистров процессора сохраняются старшие 16 бит линейного 20 разрядного адреса сегмента в ОЗУ, причем младшие 4 байта адреса считаются всегда равными 0. Т.е. сегментный адрес хранится в “упакованном” виде.
2. В регистре общего назначения или регистре-указателе сохраняется смещение в байтах относительно начала сегмента.
3. При расчете линейного адреса значение, хранящееся в сегментном регистре, смещается на 4 бита влево, при этом получается 20 битовый адрес сегмента в “неупакованном” виде. Младшие 4 бита “распакованного” линейного сегментного адреса всегда равны 0, по построению.
4. К 20 битному “распакованному” адресу сегмента прибавляется значение смещения, хранящееся в регистре-указателе или регистре общего назначения. Полученное 20 битное число, представляющее собой полный линейный адрес блока данных или переменной в ОЗУ выставляется в виде двоичного кода на линии шины адреса.

Работа описанной процедуры формирования линейного адреса иллюстрируется на рис. 8.3.

№	Операция	Двоичное представление	Комментарий
1.	DS=12BFh	00000001001010111111b	“Упакованный” 16 битный адрес сегмента
2.	SI=401h	00000000010000000001b	Смещение
3.	12BF0h	00010010101111111000b	“Распаковка” адреса сегмента до 20 бит сдвигом влево содержимого DS на 4 бита
4.	$  \begin{array}{r}  +12BF0h \\  +00401h \\  \hline  12FF1h  \end{array}  $	$  \begin{array}{r}  +00010010101111111000b \\  +00000000010000000001b \\  \hline  00010010111111111001b  \end{array}  $	Получение полного линейного адреса, как суммы “распакованного” адреса сегмента DS и содержимого SI

Рис. 8.3 Формирование полного линейного адреса из DS:SI.

Таким образом, сегментная адресация накладывает логическую структуру сегментов на линейно организованную шину адреса, причем программист работает именно с сегментами.

Объем ОП современных ПК в тысячи раз превышает лимит в 1 Мб, навязываемый особенностями архитектуры реального режима процессора. Для использования всей доступной ОП процессор программно переводят в защищенный режим. В этом режиме физические адреса сегментов хранятся в специальных областях памяти в виде 32 и более разрядных указателей, а в сегментных регистрах сохраняются не адреса регистров, а их номера (селекторы). Зная селектор, АЛУ может извлечь физический адрес сегмента из предварительно сформированной программистом структуры – т.н. таблицы дескрипторов сегментов.

Развитие идеи сегментной адресации в защищенном режиме позволяет при программировании современных процессоров определять правила доступа к конкретному сегменту.

Специфические вопросы организации сегментной адресации, затронутые в этом разделе, в основном относятся к программированию IBM-PC совместимых ЭВМ. Материал раздела можно рассматривать как один из примеров решения концептуального вопроса адресации ОП ядром процессора.

Кроме двух перечисленных способов адресации (сегментной и линейной) современные процессоры реализуют и другие парадигмы, подробное рассмотрение которых, однако, выходит за рамки данного обзорного курса.

### Подготовка и создание программы на языке ассемблера

Одной из трудностей, с которой сталкивается программист, использующий ассемблер, является плохая переносимость программ. В случае программ для IBM-PC-совместимых ПК обычно обеспечивается совместимость программ снизу-вверх. Т.е. программы, написанные для более старых семейств процессоров с высокой вероятностью могут быть откомпилированы и запущены на процессорах более новых семейств. Однако это не гарантируется, особенно, в случае использования специфических наборов инструкций. Например, несовместимы некоторые наборы инструкций процессоров фирм AMD<sup>TM</sup> и Intel<sup>TM</sup>. Ситуация дополнительно осложняется спецификой написания программ, ориентированных на работу под управлением различных операционных систем. Например, программы, созданные для работы с MS-DOS в большинстве случаев могут быть запущены в MS Windows, однако, программы для MS Windows в MS-DOS работать не будут.

Переносимость программ на языке ассемблера, написанных для различных семейств микроконтроллеров и сигнальных процессоров и вовсе является нетривиальной задачей.

Все рассмотренные ниже тексты на языке ассемблера ориентированы на работу программ под управлением ОС MS-DOS. В текстах используются лишь инструкции и обращения к регистрам для процессора i8086, что позволяет запускать их на любых IBM-PC-совместимых ПК. Программы тестировались в эмуляторе MS-DOS, встроенном в MS Windows XP SP3. Компиляция исходных текстов осуществлялась с помощью

бесплатного компилятора турбо ассемблера фирмы Borland<sup>TM</sup> – TASM v.3.1, линковка с помощью линкера TLINK v.5.1. Указанные компилятор и линкер входят в комплект поставки среды разработки Borland C++ 3.1.

Не вдаваясь в технические особенности турбо ассемблера, в простейшем случае, для создания исполнимых файлов из исходных текстов рассматриваемых примеров нужно выполнить следующую последовательность действий:

1. Выбрать в качестве текущей директории папку “BIN” в директории установки IDE Borland C++ 3.1.
2. Скопировать текстовый файл с программой на языке ассемблера имеющий расширение “.asm” в папку “BIN”.
3. В командной строке MS-DOS набрать и выполнить команду:  
`tasm text`
4. В командной строке MS-DOS набрать и выполнить команду:  
`tlink text`
5. В результате проделанных действий будет создан исполнимый файл `text.exe`, где `text` – имя файла с расширением “.asm”, содержащего исходный текст.

Созданные программы в среде MS Windows удобнее выполнять, запустив в режиме эмуляции MS-DOS командную оболочку, например, Norton Commander.

### **Работа видеоадаптера ПЭВМ в текстовом режиме**

Рассмотренные в следующих разделах программы на языке ассемблера активно используют для вывода информации на экран монитора прямой доступ к текстовому видеобуферу. Поэтому, перед началом обсуждения полезного кода программы необходимо в нескольких словах пояснить технические особенности организации текстового видеобуфера ПЭВМ и работы с ним видеоадаптера.

Текстовый видеобуфер может использоваться только для вывода на экран букв алфавитов и спецсимволов. Для его использования видеоадаптер должен быть программно переведен в текстовый режим или оставлен в нем, т.к. изначально видеоадаптер стартует в текстовом режиме.

Основными преимуществами текстового режима работы видеоадаптера по сравнению с графическим режимом являются:

- относительная простота программирования видеоадаптера,
- совместимость со старыми видеоадаптерами и
- высокая скорость вывода информации на экран.

В IBM-PC-совместимых ПЭВМ предполагается, что видеобуфер занимает непрерывный участок в ОЗУ по адресу `B8000h`. Размер занимаемой буфером ОП зависит от выбранного разрешения текстового видеорежима (т.е. от того, сколько символов могут быть отображены на экране одновременно в данном режиме работы видеоадаптера).

Для хранения информации об одном выводимом на экран символе в видеобуфере отводится 2 байта, т.е. слово. Формат слова видеобуфера рассмотрен на рис. 8.4. Байты видеобуфера с четными смещениями (0, 2, 4, ...) – младшие байты слов текстового видеобуфера – хранят ASCII-коды выводимых символов. Байты видеобуфера с нечетными смещениями (1, 3, 5, ...) – старшие байты слов текстового видеобуфера – хранят атрибуты выводимых символов. Байт атрибутов включает цвет символа, цвет фона под символом и битовый флаг мерцания.

Видеоадаптер ПЭВМ, работающий в текстовом режиме, непрерывно циклически с высокой скоростью выбирает слова из текстового видеобуфера ОЗУ и обеспечивает вывод изображения на экран монитора, причем цвет выводимых символов определяется соответствующими байтами атрибутов, а соответствие между ASCII-кодами и символами на экране определяется по таблице, предварительно загруженной в ОП видеоадаптера утилитами операционной системы. Например, в зависимости от загруженной кодовой таблицы, одному и тому же коду символа (для кодов, больших 127) может

соответствовать на экране спецсимвол для рисования таблиц или буква локального, например, кириллического алфавита.

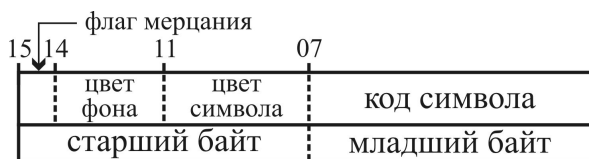


Рис. 8.4 Структура слова текстового видеобуфера ПЭВМ.

### Простейшая программа на языке ассемблера

Для знакомства с языком ассемблера подробно рассмотрим простейшую программу, осуществляющую вывод на экран монитора одного символа посредством прямого обращения к специализированной области ОЗУ – текстовому видеобуферу.

```

text    segment 'code'          ; (1) Объявление начала сегмента кода
        assume  CS:text         ; (2) В регистре сегмента кода
                                   ;    хранится адрес сегмента text

begin:  mov     AX, 0B800h       ; (3) Адрес текстового видеобуфера
        mov     ES, AX          ; (4) сохраним в ES через AX

        mov     DI, (80*20+1)*2 ; (5) Смещение в видеобуфере
        mov     AL, 'X'         ; (6) Выводимый символ
        mov     AH, 0Ah         ; (7) атрибут символа

        mov     ES:[DI], AX     ; (8) Выводим символ из AX в видеобуфер

stop:   jmp     stop            ; (9) "Вечный" цикл

text    ends                    ; (10) Конец сегмента команд
        end     begin           ; (11) Конец программы с указанием
                                   ;    метки-точки входа

```

Листинг 8.1 Пример программы на языке ассемблера, осуществляющей вывод на экран символа “X” посредством непосредственной записи в текстовый видеобуфер, располагающийся в ОЗУ по адресу B800h.

Программа в листинге 8.1 состоит из 11 строк, называемых *предложениями ассемблера*. Каждое предложение может содержать до 4 полей: поле метки или имени, оператора, операндов и комментария.

Метка в ассемблере представляет собой мнемонический идентификатор и имеет тот же смысл, что и метка оператора безусловного перехода `goto` в C и Pascal. После имени метки должен стоять символ двоеточие “:”. Имена сегментов, процедур и структур данных подчиняются обычным правилам записи идентификаторов языков программирования. Используемый нами ассемблер – TASM v.3.1 по умолчанию не чувствителен к регистру символов в идентификаторах.

Операторы могут быть командами языка (инструкциями процессора) – предложения (3-9) нашей программы, или могут представлять собой директивы ассемблера (псевдооператоры) – предложения (1), (2), (10), (11). Предложения, содержащие директивы ассемблера в код непосредственно не транслируются, а используется для управления работой компилятора, аналогично, например, макроопределениям `#define` в C.

В зависимости от оператора, предложение может содержать от 0 до 2 операндов. Операнды в предложении ассемблера отделяются друг от друга запятой “,”.

Комментарий ассемблера используется аналогично однострочному комментарию в С. Комментарий начинается символом “;” и может быть размещен либо в последнем поле предложения ассемблера, либо на отдельной строке.

Все программы на ассемблере состоят из сегментов. В самом простейшем случае должен быть объявлен хотя бы один сегмент команд, как в нашем листинге 8.1. В этом случае, программа имеет следующую структуру:

```
имя_сегмента      segment      'code'
                   assume       CS:имя_сегмента

точка_входа:
... ;Команды ассемблера
имя_сегмента      ends
                   end          точка_входа
```

Идентификатор `имя_сегмента` размещаемый перед служебным словом `segment` может использоваться в программе для получения адреса этого сегмента. Описатель `'code'` – т.н. *класс сегмента* – определяет правила работы процессора с сегментом, например, содержимое сегмента кода можно исполнять, а сегмента данных – нет и т.п.

Описание любого сегмента завершается директивой `ends` перед которой стоит имя этого сегмента.

Директива `assume` нужна для указания компилятору, в каких сегментах размещаются первый сегмент команд и первый сегмент данных. Строка `assume CS:имя_сегмента` заносит адрес первого (и единственного в нашем случае) сегмента команд `имя_сегмента` в сегментный регистр `CS`. Т.к. в нашей простейшей программе обращения к ячейкам ОП отсутствуют (что, конечно, нетипично), то упоминать в `assume` сегментный регистр данных `DS` не требуется.

Последнее предложение программы всегда содержит оператор `end` с указанием метки точки входа в программу – `точка_входа`. Такое явное указание точки входа необходимо, т.к. в общем случае после директивы `assume` до начала основного кода программы могут размещаться объявления структур данных, макросов, процедур, переменных (ячеек памяти) и т.п. Выполнение кода программы всегда начинается процессором с адреса, указанного меткой “`точка_входа:`”.

Полезный код программы составляют предложения (3-9), размещенные после точки входа “`begin:`”.

В предложениях (3) и (4) известный из технической документации адрес текстового видеобуфера `B8000h` заносится в сегментный регистр `ES`. Особенности архитектуры процессоров IBM-PC совместимых ПЭВМ не позволяют занести в сегментный регистр непосредственное значение или значение из регистра памяти. Данные в сегментный регистр могут быть загружены только из регистра общего назначения или из стека. Поэтому, в предложении (3) сдвинутый на 4 байта (на тетраду, англ. *tetrad*) вправо, в соответствии с правилами вычисления линейного адреса `i8086`, адрес видеобуфера загружается в регистр общего назначения `AX` с помощью команды `mov`. А в предложении (4) значение из `AX`, наконец, загружается в `ES`. Шестнадцатеричные константы отмечаются в ассемблере постфиксом “`h`” и должны начинаться с цифры. Поэтому, в предложении 3 второй операнд записан, как `0B800h`, а не `B800h`.

Формат использования команды `mov`:

`mov` приемник, источник

Команда копирует значение, из источник в приемник, при этом старое значение приемник теряется. В качестве источник может выступать регистр процессора, ячейка памяти или непосредственное значение, в качестве приемник – регистр или ячейка



памяти. Запрещается указывать в качестве обоих операндов ячейки памяти и загружать сегментный регистр как-либо, кроме регистра общего назначения.

В предложении (5) командой `mov` в регистр `DI` заносится смещение, относительно начала видеобуфера, куда будет выведен символ. Смещение 0 соответствует левой верхней позиции на экране. Типичное разрешение экрана в текстовом режиме по горизонтали составляет 80 символов, поэтому, смещение  $80*20$  означает первую слева позицию в 20 строке (считая сверху),  $(80*20+1)$  – вторая слева позиция. На 2 смещение необходимо умножить, т.к. одному символу на экране соответствуют 2 байта в видеобуфере.

В предложениях (6) и (7) в регистре `AX` формируется слово видеобуфера. Для этого, младший байт слова текстового буфера – код символа заносится в младшую половину `AX` – `AL`. В старшую половину `AX` – `AH` заносится байт атрибутов. Значение `0Ah` в байте атрибутов означает светло-зеленый символ на черном фоне.

В предложении (8) слово из регистра `AX` помещается по адресу, формируемому парой регистров `ES:DI`. `DI` содержит значение  $(80*20+1)*2=3202=0C82h$ , поэтому, соответствии с описанными выше правилами формирования полного линейного адреса, пара `ES:DI` будет указывать на ячейку `B8000h+0C82h=B8C82h`.

Запись `ES:[DI]` означает разыменованное – “взять данные по адресу `ES:DI`”. Таким образом, в предложении (8) слово из регистра `AX` копируется по адресу `B8C82h` в видеобуфер со смещением  $(80*20+1)*2$ . Видеоадаптер немедленно автоматически без дополнительных усилий со стороны программиста обновит изображения на экране, выведя во 2 (слева) позиции 20 (сверху) строки ярко-зеленую букву “X”.

В строке (9) организовано зависание программы в “вечном” цикле для того, чтобы можно было пронаблюдать результат ее работы. Цикл организован с помощью команды безусловного перехода `jmp имя_метки` (англ. `jump` – прыжок), осуществляющей немедленный переход на метку `имя_метки`. Конечно, обычно используется более корректный способ завершения работы программы, который будет рассмотрен в последующих примерах.

## Лекция 9 Программирование на нескольких языках, встроенный ассемблер

Программа на ассемблере с сегментами данных и стека. Модели памяти. Программирование на нескольких языках. Использование встроенного ассемблера.

### Программа на ассемблере с сегментами данных и стека

В предыдущем разделе была разобрана простейшая программа на ассемблере. Эта программа не содержит процедур и состоит только из одного сегмента – обязательного сегмента кода. Такая структура программы нетипична, при решении практических задач, обычно, программы содержат, как минимум 3 сегмента: команд, данных и стека. В листинге 9.1 рассмотрен усовершенствованный вариант программы из листинга 8.1, содержащий эти 3 сегмента. В этом усовершенствованном варианте программы вывод одного символа на экран монитора оформлен в виде процедуры. Используя эту процедуру, программа выводит на экран хранящуюся в ОП текстовую строку.

```

text          segment      'code'                ; (1)Объявление начала сегмента кода
              assume       CS:text, DS:data      ; (2)CS хранит адрес text,
                                                ; В DS будет загружен адрес data

OutChar        proc        ; (3)Объявление процедуры OutChar, выводящей в видеобуфер со
                        ; смещением DI символ, код которого хранится в AL
              push         AX                    ; (4)Временно сохраним AX в стеке
              mov          AX, 0B800h           ; (5)Используем AX для загрузки
              mov          ES, AX               ; (6)регистра ES адресом видеобуфера
              pop          AX                   ; (7)Извлечем из стека значение AX

              mov          AH, 0Ah              ; (8)Зададим цвет символа светло-зеленый

              mov          ES:[DI], AX          ; (9)Вывод в видеобуфер
              ret                    ; (10)Команда возврата из процедуры
OutChar        endp          ; (11)Конец процедуры OutChar

WaitESC        proc        ; (12)Объявление процедуры WaitESC, организующей задержку до
                        ; нажатия клавиши [ESC]
              push         AX                    ; (13)Временно сохраним AX в стеке

scan:
              in           AL, 60h              ; (14)Читаем содержимое порта 60h, т.е. из
                                                ; регистра контроллера клавиатуры, в AL
                                                ; скэн-код нажатой клавиши
              cmp          AL, 1                ; (15)Проверка условия: сравниваем AL с 1:
              je           exit                 ; (16)ЕСЛИ AL=1(скэн-код [ESC]), ТО
                                                ; переход на метку exit – выход из процедуры,
              jmp          scan                 ; (17)ИНАЧЕ – заикливание безусловным
                                                ; переходом на метку scan, пока не нажат [ESC]

exit:
              pop          AX                    ; (18)Извлечем из стека значение AX
              ret                    ; (19)Команда возврата из процедуры
WaitESC        endp          ; (20)Конец процедуры WaitESC

begin:
                                                ; (21)Точка входа в тело программы
              mov          AX, data             ; (22)Явно загружаем в сегментный регистр
              mov          DS, AX               ; (23)данных DS адрес сегмента data

              mov          DI, 80*20*2         ; (24)Начальное смещение в видеобуфере
              mov          CX, msgSize          ; (25)Количество символов в строке в CX
              mov          SI, 0                ; (26)Инициализируем SI нулем

cicle:
              mov          AL, msg[SI]          ; (27)В AL загружаются байт, из msg
                                                ; со смещением SI относительно начала msg
              call         OutChar              ; (28)Вызываем процедуру OutChar

              inc          SI                    ; (29)Инкремент SI

```

```

        add        DI, 2          ; (30) Увеличиваем DI на 2

        loop       cicle         ; (31) Организация цикла с переходом на
                                ; метку cicle. Цикл итерирует CX раз
        call       WaitESC       ; (32) Вызываем процедуру WaitESC –
                                ; программа ожидает нажатия [ESC]
        mov        AH, 4Ch        ; (33) Организация корректного завершения
        mov        AL, 00h        ; (34) программы и возврата в MS-DOS
        int        21h           ; (35) путем обращения к системной функции
                                ; DOS посредством вызова прерывания 21h
text     ends                    ; (36) Конец сегмента кода

data     segment                 ; (37) Объявление начала сегмента данных
msg      db      'Hello world!' ; (38) В сегменте кода размещен массив –
                                ; строка символов
msgSize=$-msg                    ; (39) Определяем размер строки в байтах
data     ends                    ; (40) Конец сегмента данных

stack    segment                stack ; (41) Объявление начала сегмента стека
        dw      16 dup (0)       ; (42) Резервируем и инициализируем
                                ; нулями 20 слов в стеке
stack    ends                    ; (43) Конец сегмента стека

        end        begin        ; (44) Конец программы с указанием
                                ; точки входа

```

Листинг 9.1 Пример программы на языке ассемблера, содержащей сегменты кода, данных и стека и осуществляющей вывод на экран строки символов, хранящейся в ОП. Вывод осуществляется посредством непосредственной записи в текстовый видеобуфер.

Первым отличием этой программы, по сравнению с предыдущим вариантом (листинг 8.1) является добавление в строке (2) директивы, указывающей, что регистр DS будет указывать на отдельный сегмент данных: DS:data. После этого указания, при обращениях к ячейкам памяти по умолчанию будет предполагаться, что сегментный адрес нужно извлекать из DS.

Точка входа в тело программы расположена в строке (21). Первыми командами является настройка DS на сегмент данных data – (22, 23). В строках (24-26) осуществляется необходимая начальная инициализация. Загружаемое в CX значение msgSize представляет собой константу – количество символов в выводимой строке. Это значение рассчитывается на этапе компиляции в соответствии с директивой, описанной в строке (39). Директива msgSize=\$-msg в этой строке читается так: “получить разность текущего смещения (\$) и смещения msg”. Т.е. msg указывает на начало области ОП, где хранится текст, а размещенная сразу после этого текста директива \$ позволяет определить адрес конечной ячейки памяти, разность \$-msg позволяет определить длину буфера в байтах, т.е. количество символов в строке.

Команды с (27-31) будут выполняться в цикле, т.к. команда loop (31) обеспечивает переход на метку (cicle) CX раз. Поэтому, цикл итерирует столько раз, сколько символов оказалось в выводимой строке.

В строке (27) в AL загружается код очередного символа. Для этого осуществляется обращение к msg, как к массиву. Запись msg[SI] читается так: “взять данные по адресу, смещение которого рассчитывается, как msg+SI”. В общем случае нужно было бы записать: DS:msg[SI]. Однако, т.к. сегмент данных в нашей программе явно задан и DS инициализирован, то смещение высчитывается, относительно начала сегмента данных по умолчанию.

Строка (28) вызывает процедуру OutChar, описанную в строках (3-11). Синтаксис описания процедур в ассемблере таков:

```

Имя_проц    proc ;Точка входа в процедуру
              ;Тело процедуры
              ret ;Команда возврата из процедуры
Имя_проц    endp ;Конец описания процедуры

```

Для вызова процедуры Имя\_проц нужно написать `call Имя_проц`. Команда `call` сохраняет в стеке адрес возврата, т.е. смещение команды, следующей непосредственно за `call`, и осуществляет немедленный переход на точку входа в процедуру, занося в `IP` адрес процедуры. Команда `ret` в процедуре извлекает из вершины стека адрес возврата и загружает его в `IP`, осуществляя немедленный возврат в программу.

Процедура `OutChar` осуществляет вывод символа в видеобuffer со смещением, предварительно загруженным в `DI`. Код символа должен быть предварительно загружен в `AL`. Процедуры в ассемблере не поддерживают явно механизма передачи параметров, как в языках высокого уровня. Обмен параметрами и результатами с процедурами может осуществляться через глобальные переменные, регистры процессора (как в нашем случае) или через стек. Механизм вывода в текстовый видеобuffer и соответствующий код подробно рассмотрены в лекции 8. В комментариях могут нуждаться лишь команды сохранения значения в стеке – `push` (4) и извлечения из стека – `pop` (7). Их использование понадобилось в нашем случае, для сохранения содержимого `AL` (переданный процедуре код символа), т.к. в строках (5-6) регистр `AX` используется для временного хранения сегментного адреса видеобufferа.

После цикла, осуществляющего вывод на экран текстовой строки, расположен вызов функции `WaitESC` (32). Эта функция, описанная в строках (12-20), обеспечивает задержку (зацикливание) до нажатия пользователем клавиши `[ESC]`. Рассмотрим последовательно команды этой функции. В строках (13) и (18) осуществляется, соответственно, сохранение в стеке и восстановление значения `AX`. Для данной программы это не обязательно, однако, хороший тон программирования на ассемблере требует такого предохранения от изменения для всех регистров, изменяемых в процедуре.

В строке (14) с помощью команды `in` осуществляется чтение байта из порта `60h` – контроллера клавиатуры. Из официальной технической документации известно, что чтение из порта `60h` позволяет получить т.н. скэн-код нажатой клавиши.

Скэн-код – фактически представляет собой порядковый номер клавиши на клавиатуре, ASCII-код высчитывается программным обеспечением ОС из скэн-кода с учетом нажатия служебных клавиш. Например, нажатие алфавитно-цифровой клавиши при нажатой или отпущенной клавише `[Shift]` возвращает разные ASCII-коды, хотя скэн-код не изменяется.

За клавишей `[ESC]` закреплен скэн-код 1, и в строке (15) с помощью команды `cmp` осуществляется сравнение операндов. Команда `cmp` вычитает второй операнд из первого. Разность не сохраняется, однако, как и при любой арифметической операции, `cmp` (от англ. *compare* – сравнить) изменяет флаги процессора, которые могут быть проанализированы с помощью команд условного перехода. Например, если `[ESC]` была нажата, то после выполнения чтения порта `60h` (14) `AL` будет содержать код 1. В этом случае, разность `AL-1`, вычисленная в строке (15) даст 0, при этом (при нулевом результате предыдущей арифметической операции) установится флаг `ZF` регистра `FLAGS`. В строке (16) используется одна из команд условного перехода, анализирующая флаг `ZF`, – `je` (от англ. *Jump if Equal* – перейти, если равно). Команда `je exit` выполнит переход на метку `exit`, “перепрыгнув” строку (17), только в случае, если флаг `ZF` установлен, т.е. если `AL=1`, что произойдет при нажатии `[ESC]`.

В противном случае, при `AL` не равном 1, `ZF` не будет установлен и в строке (16) переход осуществлен не будет, поэтому, будет выполнена строка (17), где осуществляется

безусловный переход – `jmp` на метку `scan`, это приведет к заикливанию процедуры чтения скэн-кода и его проверки.

Строки (33-35) программы обеспечивают корректное ее завершение, выгрузку из памяти и возврат в DOS специфическим для системы MS-DOS способом. Для этого командой `int` возбуждается специальное сервисное прерывание MS-DOS – 21h, передающее управление этой ОС. Если при возбуждении этого прерывания АН содержит код т.н. *системной функции* завершения работы программы – 4Ch, то работа программы прерывается, она выгружается из памяти и ОС передается т.н. код завершения работы программы, хранящийся в AL. Этот код может быть впоследствии проанализирован.

Подробное описание особенностей функционирования программ под управлением ОС MS-DOS и перечень доступных системных функций можно найти в справочных руководствах по MS-DOS.

Строки (37-40) содержат описание сегмента данных `data`. При описании сегмента данных явного объявления класса сегмента не требуется. В строке (38) объявляется текстовая строка. Директива `db` указывает, что инициализация осуществляется побайтово. Имя ячейки памяти – `msg` “хранит” смещение текстовой строки относительно начала сегмента. Указание при объявлении ячеек памяти имени необязательно, но часто удобно.

В строках (41-43) объявляется сегмент стека, на что указывает директива `stack` (41). В отличие от сегмента данных, явная инициализация регистра `SS` не требуется. В `SS` автоматически загружается адрес сегмента, объявленного с директивой `stack`. В строке (42) для размещения стека резервируются 16 слов (с запасом). Директива `dw` означает пословную инициализацию. Директива инициализации `16 dup(0)` заполняет 16 последовательных слов памяти значениями 0.

В нашей программе стек используется для хранения адреса возврата при вызове процедур и для временного сохранения значений, хранящихся в регистре `AX` (4) и (13).

### Модели памяти

Из рассмотренного выше примера видно, что программы, написанные на языке ассемблера достаточно громоздки, причем существенный размер занимают описания сегментов команд, стека и данных. Вместе с тем, оказалось, что на практике наиболее часто встречается лишь несколько наиболее типичных способов организации сегментов, описание которых организуется единообразно. Такие типичные способы организации сегментов в программе получили название *моделей памяти* (англ. *memory models*). Большинство современных ассемблеров предоставляют возможность т.н. упрощенного описания сегментов, для чего программист с помощью специальной директивы указывает ассемблеру выбранную им для своей программы модель памяти, а подробное описание сегментов, аналогичное использованному в листинге 9.1, ассемблер формирует автоматически.

Язык C, как и все языки программирования высокого уровня, не имеет встроенных средств описания отдельных сегментов, однако, большинство компиляторов позволяют программисту выбирать для каждой программы модель памяти. Например, компилятор, Borland C++ 3.1 поддерживает 6 моделей памяти:

Модель памяти	Сегменты		
	Команд	Данных	Стека
Tiny	64 Кб		
Small	64 Кб	64 Кб	
Medium	1 Мб	64 Кб	
Compact	64 Кб	1 Мб	
Large	1 Мб	1 Мб	
Huge	1 Мб	несколько по 64 Кб	64 Кб

Таблица 9.1 Модели памяти, поддерживаемые Borland C++ 3.1.

Например, в соответствии с моделью `tiny` организована программа из листинга 8.1: программа имеет всего один сегмент, в котором хранятся и команды и данные и стек. При этом регистры CS, DS и SS хранят одинаковые адреса. Часто модель `tiny` используют для написания небольших служебных программ – утилит.

Программа, рассмотренная в листинге 9.1, удовлетворяет модели `huge`, т.к. только эта модель имеет выделенный сегмент стека. Организация в примере 9.1 отдельного сегмента стека потребовалась исключительно для учебных целей. На практике стек чаще размещается в сегменте данных, а модель `huge` используется только при создании больших программ.

Выбранная модель памяти определяет размер используемых переменных-указателей. Например, программы, созданные с использованием модели `tiny` содержат всего один сегмент. Для адресации данных и кода внутри него достаточно сохранять только двухбайтовое смещение относительно начала этого сегмента. Поэтому, все переменные-указатели в таких программах по умолчанию двухбайтовые.

Модель `huge`, напротив, допускает наличие нескольких сегментов команд и нескольких сегментов данных. Для обращения в процессе работы программы к разным сегментам требуется перенастройка адресов, хранящихся в сегментных регистрах CS и DS, поэтому, переменные-указатели, в общем случае, должны хранить и двухбайтовый сегментный адрес и двухбайтовое смещение внутри сегмента. Поэтому, по умолчанию для модели `huge` используются переменные-указатели размером 4 байта.

Согласно сложившейся терминологии, переменные-указатели, хранящие только смещение внутри сегмента принято называть ближними указателями (англ. `near pointer`), а указатели, допускающие адресацию различных сегментов – дальними указателями (англ. `far pointer`). В силу рассмотренной выше специфики сегментной адресации программирование IBM-PC для ОС MS-DOS подразумевает выделение 2 байт для ближних указателей и 4 байт для дальних.

Идея организации памяти программы с помощью выбора одной из нескольких предопределенных типовых моделей памяти широко используется, в т.ч. на базе различных аппаратных платформ, отличаясь иногда в специфике конкретной реализации.

### Программирование на нескольких языках

Грамотное программирование на языке ассемблера теоретически позволяет программисту написать программу любой сложности, получив в результате наиболее эффективный код. Однако, как видно на примерах 8.1 и 9.1 программы на ассемблере громоздки, требуют от программиста специфических знаний и соблюдения массы формальных правил, кроме того, эти программы, в общем случае, обладают относительно плохой переносимостью. Использование языков высокого уровня, например, C существенно облегчает работу программиста при написании крупных программ, но код на C менее эффективен. Кроме того, некоторые действия, например, прямое обращение к портам периферийных устройств, требует обязательного использования ассемблера. Современные компиляторы позволяют совместно использовать преимущества языка

высокого уровня и эффективность ассемблерных программ, допуская написание частей одной программы на разных языках программирования. В частности, большинство компиляторов С поддерживают три основных способа совместного использования кода на С и ассемблере и их комбинации:

№	Способ взаимодействия	Комментарий
1.	Экспорт объектного модуля, созданного на С, для использования ассемблерной программой	Используется редко. Обычно при таком подходе на С программируется интерфейс с пользователем.
2.	Импорт объектного модуля, созданного на ассемблере, для использования с программой на С	Используется часто. Этот подход позволяет наиболее полно использовать преимущества совместного программирования на С и ассемблере. Такая организация программы сложна с точки зрения программиста.
3.	Использование <i>встроенного ассемблера</i> – написание на ассемблере части исходного кода непосредственно внутри исходного текста программы на языке С	Используется часто. Подход имеет некоторые ограничения при использовании ассемблера. Такая организация программы наиболее проста с точки зрения программиста.

Таблица 9.2 Подходы к организации программ, разные части которых написаны на языках ассемблера и С.

Наиболее гибким с точки зрения предоставляемых возможностей является второй способ (или комбинация первого и второго способов), описанный в таблице 9.2. Однако, его использование сложно с точки зрения программиста. При его реализации критическая часть программы пишется на языке ассемблера и компилируется в объектный код. При написании могут использоваться все преимущества ассемблера: гибкое описание сегментов и структур данных, низкоуровневая оптимизация под конкретное семейство процессоров и т.п. Однако для того, чтобы результирующий объектный модуль мог быть подключен к модулю основной программы, скомпилированной из языка С, необходимо учесть массу формальностей, касающихся именования переменных, подпрограмм и ячеек памяти, правил стыковки сегментов и т.п., причем нужно принимать во внимание специфику использования различных компиляторов.

В силу перечисленных сложностей 1 и 2 подходов наиболее часто на практике применяют третий способ, программируя на встроенном ассемблере. Встроенный ассемблер большинства С-компиляторов не позволяет явно описывать сегменты и процедуры. Он имеет различные ограничения, в частности, при определении ячеек памяти и использовании регистров процессора, однако, с учетом ограничений, позволяет получать код близкий по эффективности к полноценному ассемблеру. Основными достоинствами использования встроенного ассемблера по сравнению с двумя другими подходами, являются простота использования с точки зрения программиста и хорошая переносимость исходных текстов программ между различными компиляторами.

Подробное описание особенностей работы и ограничений при использовании встроенного ассемблера нужно искать в руководствах программиста и технической документации на каждый конкретный компилятор.

Для написания кода на встроенном ассемблере в Borland C++ 3.1 используется конструкция:

```
asm{
    //Текст программы на языке ассемблера
}
```

Встроенный ассемблер ВС++ 3.1 не допускает модификации сегментных регистров, кроме ES, имеются существенные ограничения при объявлении поименованных ячеек памяти и вызове подпрограмм. При использовании команд перехода метки, на которые осуществляется переход, должны быть описаны вне конструкции `asm{ . . . }`, существуют и другие ограничения. Комментарии оформляются по правилам С. Однако, внутри

конструкций `asm{...}` допустимы обращения по именам к переменным, объявленным в программе по правилам языка C. Таким образом, встроенный ассемблер VC++ 3.1 целесообразно использовать для написания небольших участков программ внутри функции.

### Использование встроенного ассемблера

В качестве примера использования встроенного ассемблера рассмотрим программу, позволяющую осуществлять вывод на экран монитора в текстовом режиме строки текста заданным цветом в точку с указанными координатами. Такие функции не реализованы в стандартных библиотеках C и могут представлять самостоятельный интерес для начинающего программиста.

```
#define clBlack      0    //Описания констант, кодирующих цвет
#define clBlue      1
#define clGreen     2
#define clCyan      3
#define clRed       4
#define clMagenta   5
#define clBrown     6
#define clLightGray 7
#define clDarkGray  8
#define clLightBlue 9
#define clLightGreen 10
#define clLightCyan 11
#define clLightRed   12
#define clLightMagenta 13
#define clYellow     14
#define clWhite      15
#define clBlink      1    //и наличие
#define clNoBlink    0    //признака мерцания

//Для видеорежима 80x50
#define MAX_WIDTH    80 //Количество знакомест в строке
#define MAX_HEIGHT   50 //Количество строк на экране

unsigned char      text_attribute; //Глобальная переменная - атрибуты

void set_color(char color, char bgcolor, char blink){
    //Задаёт цвет текста, цвет фона и признак мерцания -
    //формирует байт атрибутов, модифицируя глобальную text_attribute
    asm{
        push    AX                //Сохраняем содержимое используемого регистра AX

        mov     AL, bgcolor        //Формируем атрибут цвета фона сдвигом 3 битного
        shl     AL, 4              //кода фона в старшую тетраду байта атрибутов

        mov     AH, blink          //Если blink
        cmp     AH, 0              //не равен 0
        je      no_blink           //то
        or      AL, 0x80           //установить старший бит - признак мерцания
    }

    no_blink:
    asm{
        mov     AH, color          //Устанавливаем атрибуты цвета текста,
        add     AL, AH              //добавляя значение color к содержимому AL
        mov     text_attribute, AL //Сформированный в AL байт атрибутов помещаем
                                   //в глобальную переменную text_attribute
        pop     AX                  //Восстанавливаем AX
    }
}
```



```

void string2screen(char x, char y, char *string){
    //Выводит на экран строку string, начиная с позиции с координатами
    //(x, y). При выводе символы отображаются с атрибутом text_attribute
    asm{
        push    AX                //Сохраняем
        push    BX                //значения
        push    DI                //регистров,
        push    SI                //используемых в
        push    ES                //функции

        mov     AX, 0xB800        //Загружаем в ES адрес
        mov     ES, AX            //текстового видеобуфера: B8000h
        //Рассчитаем начальное смещение в видеобуфере DI<-(MAX_WIDTH*y+x)*2
        mov     AL, y             //AL=y
        mov     AH, MAX_WIDTH     //AH=MAX_WIDTH
        mul     AH                //AX=AL*AH: (MAX_WIDTH*y)
        xor     BX, BX            //BX=0 (Результат команды эквивалентен mov BX, 0)
        mov     BL, x             //BL=x
        add     AX, BX            //AX=AX+BL: (MAX_WIDTH*y+x)
        shl     AX, 1             //AX=AX*2: (MAX_WIDTH*y+x)*2
        mov     DI, AX            //DI<-(MAX_WIDTH*y+x)*2

        mov     AH, text_attribute //Загружаем в AH байт атрибутов

        mov     SI, string        //SI хранит адрес начала строки string
    }

    cycle:
    asm{
        mov     AL, [SI]          //Разыменование: загрузка в AL байта по адресу SI,
                                //т.е. ASCII-код очередного символа строки
        cmp     AL, 0             //ЕСЛИ достигнут завершающий строку 0
        je      end_of_string     //ТО на выход

        mov     ES:[DI], AX       //ИНАЧЕ вывести очередной символ в видеобуфер

        inc     SI                //Инкремент смещения очередного символа строки
        add     DI, 2             //Увеличиваем смещение видеобуфера

        jmp     cycle            //Цикл итерирует до вывода всех символов строки
    }

    end_of_string:
    asm{
        pop     ES                //Восстанавливаем
        pop     SI                //значения
        pop     DI                //сохраненных
        pop     BX                //регистров
        pop     AX                //Команды pop вызываются в порядке обратном push
    }
}

void clear_screen(void){
    //Очищает экран, заполняя его пробелами с текущим цветом фона
    asm{
        push    AX                //Сохраняем
        push    DI                //значения
        push    ES                //используемых
        push    CX                //регистров

        mov     AX, 0xB800        //Загружаем адрес

```

```

mov     ES, AX          //текстового видеобуфера в ES

mov     AL, ' '          //В AL заносим код символа "пробел"
mov     AH, text_attribute //В AH сохраняем значение байта атрибутов

mov     DI, 0            //Начальное смещение в видеобуфере

mov     CX, MAX_WIDTH*MAX_HEIGHT //Количество символов, помещающихся
                                //на экране в текущем видеорежиме помещаем в
                                //счетчик цикла CX
}

cycle:
asm{
    mov     ES:[DI], AX    //Отправляем очередной "пробел" в видеобуфер

    add     DI, 2          //Увеличиваем смещение в видеобуфере

    loop    cycle         //Цикл итерирует CX раз
}

asm{
    pop     CX             //Восстанавливаем
    pop     ES             //значения
    pop     DI             //используемых
    pop     AX             //регистров
}
}

void main(void){
    set_color(clYellow, clBlue, clBlink ); //Установим цвет фона - синий,
                                           //цвет символа - желтый,
                                           //атрибут мерцания установлен
    clear_screen(); //Очистим экран
    string2screen(10,10,"Hello world!"); //Вывод на экран в заданную позицию
                                           //текстовой строки
}

```

Листинг 9.2 Программирование вывода информации на экран монитора с помощью непосредственного обращения к текстовому видеобуферу на встроенном ассемблере Borland C++ 3.1.

Функция `set_color` обеспечивает формирование байта атрибутов, структура которого приведена на рис. 8.4. Сформированный байт атрибутов сохраняется в глобальной переменной `text_attribute`, которую используют функции вывода на экран.

Функция `string2screen` выводит на экран текстовую строку. Вывод осуществляется начиная со знакоместа, имеющего координаты (x, y). Символ слева вверху экрана имеет координаты (0, 0). В примере предполагается, что разрешение текстового видеорежима 80x50, однако, изменяя константы `MAX_WIDTH` и `MAX_HEIGHT` можно адаптировать функции для использования в текстовом видеорежиме с любым другим разрешением. Рассмотрим эту функцию более подробно.

Функция начинается с сохранения в стеке значений всех модифицируемых ей регистров процессора и заканчивается восстановлением их значений. Нужно обратить внимание, что количество вызовов команд `push` и `pop` внутри подпрограмм должно быть одинаковым, т.к. адрес возврата из функции сохраняется в стеке.

Достаточно громоздким выглядит блок команд вычисления начального смещения в видеобуфере, которое сохраняется в `DI`. Оно рассчитывается по формуле:  $DI = (MAX\_WIDTH * y + x) * 2$ . Сначала вычисляется произведение:  $MAX\_WIDTH * y$ , для чего

`MAX_WIDTH` и `y` копируются в половинки регистра `AX`, которые затем перемножаются командой `mul` с сохранением результата в `AX`.

Далее вычисляется и сохраняется в `AX` значение  $(MAX\_WIDTH * y + x)$ . Для этого значение `x` временно сохраняется в `BX` и добавляется к содержимому `AX` командой `add`, причем результат также сохраняется в `AX`.

Обратите внимание, что для инициализации `BX` нулем использована команда `xor BX, BX`. Результат эквивалентен команде `mov BX, 0`, которая используется в программе в аналогичных случаях для улучшения читаемости программы. Команда `xor BX, BX` приведена здесь как пример возможной оптимизации кода, т.к. на большинстве процессоров IBM-PC она занимает меньше памяти и выполняется быстрее, чем `mov BX, 0`.

В регистр `AX` загружается сформированный заранее байт атрибутов. В индексный регистр `SI` сохраняется адрес 0-терминированной строки `string`, т.е. смещение ее первого символа.

Далее с помощью безусловного перехода организуется цикл, на каждой итерации которого в `AL` загружается значение байта по адресу, хранящемуся в `SI` (т.е. ASCII-код очередного символа из строки `string`), после чего значение `AX` заносится в видеобuffer и осуществляется инкремент смещения `SI` в строке `string` и смещения `DI` в видеобufferе. Выход из этого цикла произойдет только тогда, когда в `AL` окажется код 0 завершающего строку служебного символа. Для этого осуществляется проверка `AL` командой `cmp AL, 0` и предусмотрен условный переход на метку `end_of_string`, указывающую за границы главного цикла `cycle`. Следует обратить внимание, что в соответствии с правилами использования встроенного ассемблера Borland C++ 3.1, метки описаны в C-программе вне конструкций `asm{...}`.

Функция `clear_screen` очищает экран, выводя во все его знакоместа пробелы, причем цвет фона определяется значением, хранящимся в `text_attribute`. Функция аналогична `string2screen` и не нуждается в подробных комментариях. Здесь она приведена в качестве примера использования на встроенном ассемблере цикла с заданным числом итераций – `loop`.

## Приложение 1: Сокращения и аббревиатуры

### Лекция 1

ПК	– Персональный компьютер
PC	– англ. Personal Computer
IDE	– англ. Integrated Developers Environment
ОпКод	– Код Операции
OpCod	– англ. Operation Cod
ЭВМ	– Электронно-Вычислительная машина
ОС	– Операционная Система
OS	– Operational System
ООП	– Объектно-ориентированное программирование
DOS	– Disk Operating System

### Лекция 2

АЦП	– Аналого-цифровой преобразователь
ADC	– Analog to Digital Converter

### Лекция 4

bit	– BInary digiT
LSB	– Least significant bit
MSB	– Most significant bit
CPU	– Central processing unit
ПЭВМ	– Персональная электронно-вычислительная машина
HDD	– Hard disk drive
CD-ROM	– Compact disc read-only memory
DVD-ROM	– Digital versatile disk read-only memory
SSD	– Solid state drive
ОП	– Оперативная память
МП	– Менеджер памяти

### Лекция 6

ПО	– Программное обеспечение
ПДП	– Прямой доступ к памяти
DMA	– Direct memory access

### Лекция 6

LIFO	– Last input, first output
FIFO	– First input, first output

### Лекция 8

АЛУ	– Арифметико-логическое устройство
ALU	– Arithmetic and logic unit
ОЗУ	– Оперативное запоминающее устройство
RAM	– Random-access memory
FPU	– Floating Point Unit
IRQ	– Interrupt ReQuest

## Приложение 2: Практические задания для самоконтроля

Задания рекомендуется выполнять в среде Borland C++ 3.1, используя только средства языка C и стандартные библиотеки.

### Базовые навыки программирования, простейший интерфейс пользователя

1. Создать компьютерную программу, выводящую на экран монитора текст: “Hello, C!”
2. Создать компьютерную программу, предлагающую пользователю ввести с клавиатуры 2 знаковых целых числа: a и b и выводящую их сумму – c.
3. Создать компьютерную программу, вычисляющую значение выражения:  
$$\left( \left( \frac{x^2}{y^3} + \frac{1}{x} \right) : \left( \frac{x}{y^2} - \frac{1}{y} + \frac{1}{x} \right) \right) : \frac{(x-y)^2 + 4xy}{1 + yx^{-1}}.$$
 Известно, что при  $x = 0.2$  и  $y = 20$  значение выражения равно 0.25. При решении задачи используйте функции, прототипы которых описаны в заголовочном файле `math.h`. Используйте встроенную систему помощи Borland C++ 3.1.
4. Создать компьютерную программу, выводящую на экран минимальное и максимальное значения массива, состоящего из 5 целых знаковых чисел. Значения массива вводятся с клавиатуры. После ввода введенные значения из массива распечатываются на экране в одну строку.
5. Создать компьютерную программу, сортирующую массив из вещественных чисел по возрастанию, используя “пузырьковую” сортировку. Число элементов массива вводится пользователем перед вводом значений из массива. Исходный и отсортированный массивы выводятся на экран в 2 строках: один под другим. Под каждое выводимое значение отводится 10 знакомест, значения выравниваются по левому краю, знак числа (и “+” и “-”) выводится всегда, вещественные значения округляются при выводе на экран до 1 знака после десятичного разделителя.
6. Создать программу, которая читает с клавиатуры двухбайтовое беззнаковое целое число и распечатывает на экране ASCII-символ, соответствующий этому числу, а также само число в десятичной и шестнадцатеричной системах счисления. Число в шестнадцатеричной системе счисления дополняется постфиксом “h”.

### Отладка программы

7. На примере программы из п. 5 продемонстрировать навык осуществления отладки программы. Показать умения: постановки точки останова, выполнения программы по шагам, просмотра значений счетчика цикла и содержимого массива при пошаговом выполнении.

### Работа с файлами

8. Создать компьютерную программу, сохраняющую в ASCII-файл “sin.txt” 10 периодов функции синус, с амплитудой 10, средним -2, частотой, обеспечивающей 20 значений на период синусоиды и нулевой начальной фазой. Значения выводятся в файл в один столбец, выравниваются по правому краю поля шириной 10 знакомест. Значения выводятся в формате с фиксированной точкой. Вещественные числа двойной точности, хранящие значения этой функции синуса в дискретные моменты времени округляются при выводе до 5 знаков после десятичного разделителя.
9. Создать программу, которая читает данные из файла “sin.txt”, созданного при выполнении программы из п. 7, масштабирует эти данные и сохраняет их в массив знаковых двухбайтовых целых. Масштабирование осуществляется так, чтобы искажение формы сигнала было минимальным. Синусоидальный сигнал,

сохраненный таким образом в массив знаковых двухбайтовых целых сохраняется в типизированный файл "sin.dat".

10. Прочитать данные из типизированного файла "sin.dat", созданного при выполнении задания п. 8, сохранить их в массив, оценить и отобразить на экране значения матожидания, дисперсии, стандартного отклонения, максимальное и минимальное значения из файла.

#### **Работа с отдельными битами**

11. Создать программу, которая читает с клавиатуры однобайтовое беззнаковое целое и, используя побитовые операции, выводит на экран это число в двоичной системе счисления. После вывода двоичного числа выводить на экран постфикс "b".
12. Создать программу, которая читает с клавиатуры однобайтовое беззнаковое целое, выводит на экран это число в двоичной системе счисления, затем сбрасывает второй бит введенного числа и распечатывает на экране полученное значение в десятичной и двоичной системах счисления.
13. Создать программу, которая читает с клавиатуры однобайтовое беззнаковое целое, выводит на экран это число в двоичной системе счисления, затем устанавливает пятый бит введенного числа и распечатывает на экране полученное значение в десятичной и двоичной системах счисления.
14. Создать программу, которая читает с клавиатуры однобайтовое беззнаковое целое  $a$  и маску – последовательность из 8 символов "X", "C" или "S". Первым вводится младший бит маски, последним – старший. Число  $a$  изменяется в соответствии с маской: если в некотором бите маски стоит "X", то значение соответствующего бита  $a$  не изменяется, если маска содержит "C", соответствующий бит числа  $a$  сбрасывается, если маска содержит "S", соответствующий бит числа  $a$  устанавливается. На экран в столбец выводятся: введенное число  $a$  в десятичной системе счисления,  $a$  в двоичной системе счисления, маска, результирующее число в двоичной системе счисления и результирующее число в десятичной системе счисления.

#### **Функции и макросы**

15. Создать функцию `get_bit`, проверяющую, установлен ли бит  $N$  беззнакового однобайтового числа  $X$  и возвращающую значение логической истины в случае, если этот бит установлен.
16. Создать функцию `set_bit`, устанавливающую бит  $N$  беззнакового однобайтового числа  $X$ , доступ к  $X$  организуется по ссылке.
17. Создать функцию `cli_bit`, сбрасывающую бит  $N$  беззнакового однобайтового числа  $X$ , доступ к  $X$  организуется по ссылке.
18. Создать макрос `SET_BIT`, устанавливающий бит  $N$  беззнакового числа  $X$ .
19. Создать макрос `CLI_BIT`, сбрасывающий бит  $N$  беззнакового числа  $X$ .
20. Реализовать программу из п. 11 с помощью разработанной в п. 15 функции `get_bit`.
21. Реализовать программу из п. 14, используя функции `set_bit` и `cli_bit` разработанные при выполнении пп. 16 и 17.
22. Реализовать программу из п. 14, используя макросы `SET_BIT` и `CLI_BIT` разработанные при выполнении пп. 18 и 19.
23. Разработать программу, осуществляющую расчет кросскорреляционной функции 2 сигналов. Оформить расчет коэффициента корреляции для фиксированной задержки в виде функции, на вход которой в качестве параметров подаются: указатели на 2 массива данных:  $X$  и  $Y$ , количество элементов, используемых при расчете коэффициента корреляции, задержка  $\tau$  второго сигнала относительно

первого в отсчетах. Функция должна возвращать рассчитанное значение коэффициента корреляции. Тестирование программы производить, загрузив в анализируемые массивы: временной ряд из файла “sin.txt”, созданный при выполнении п. 8 – в X и его копию без 5 первых выборок – в Y. Результаты расчета ККФ вывести в текстовый файл, первый столбец которого содержит значение задержки, а второй столбец – значение соответствующего коэффициента корреляции. Расчет производить для значений задержки:  $\tau \in [-40, 40]$ .

24. Разработать программу, осуществляющую пузырьковую сортировку массива знаковых двухбайтовых целых. Пользователь вводит количество элементов массива, имя типизированного файла с данными, и символ “F” или “B” для указания, прямого или обратного порядка сортировки, соответственно. Массив создается динамически с выделением памяти из кучи. Вывести на экран в 2 столбца исходный и отсортированный массивы.

Программа должна содержать функции:

`bubble_sort`, `forward`, `backward` и `swap`.

- Аргументы функции `swap`: указатели на 2 переменные типа `int`. `swap` меняет местами содержимое переменных-указателей.
- Аргументы функции `forward`: указатель на массив данных и индекс текущего обрабатываемого элемента. Функция осуществляет сравнение 2 последовательных элементов массива и замену их местами с помощью вызова `swap` для сортировки массива по возрастанию.
- Набор аргументов функции `backward` идентичен набору аргументов `forward`. Функция осуществляет сравнение 2 последовательных элементов массива и замену их местами с помощью вызова `swap` для сортировки массива по убыванию.

Аргументы функции `bubble_sort`: указатель на массив данных, количество элементов массива, указатель на функцию, осуществляющую сравнение и при необходимости, перестановку 2 последовательных элементов массива. Функция осуществляет сортировку массива, вызывая по ссылке, `forward` или `backward` в зависимости от того, по возрастанию или убыванию ведется сортировка.

### Заголовочные файлы

25. Язык C не имеет встроенной поддержки констант в двоичной системе счисления, однако, это ограничение можно обойти. Для этого нужно создать и подключить к программам заголовочный файл, содержащий определения “двоичных” констант, например: `#define b00010011 19` для десятичного числа 19 и т.д. Напишите программу, создающую текстовый файл `binary.h`, содержащий определения таких “двоичных” констант для десятичных чисел от 0 до 255, т.е. для описания 1 байта.
26. Добавьте в заголовочный файл `binary.h`, созданный при выполнении п. 24 прототипы и описание функций `get_bit`, `set_bit` и `cli_bit`, а также определения макросов `SET_BIT` и `CLI_BIT`.

## Литература

1. Дейтел Х.М., Дейтел П.Дж. Как программировать на С. –М.: «Бином-Пресс», 2009. – 910 с.
2. Подбельский В.В. Язык С++: Учеб. пособие. –5-е издание. –М.: «Финансы и статистика», 2003. –560 с.
3. Керниган Б., Ритчи Д. Язык программирования Си. Пер. с англ. –3-е издание испр. – СПб.: "Невский Диалект", 2001. –352 с.
4. Рудаков П.И., Финогенов К.Г. Язык ассемблера: уроки программирования. – М.: «ДИАЛОГ-МИФИ», 2001. –640 с.
5. Голубь Н.Г. Искусство программирования на Ассемблере. Лекции и упражнения. –2-е издание испр. и доп. –СПб.: «ООО ДиаСофтЮП», 2002. –656 с.
6. Юров В.И. Assembler. Учебник для ВУЗов. –2-е издание. –СПб.: «ПИТЕР», 2003. – 637 с.
7. Юров В.И. Assembler. Практикум. –2-е издание. –СПб.: «ПИТЕР», 2006. –399 с.