**Санкт-Петербургский
государственный университет телекоммуникаций
им. проф. М. А. Бонч-Бруевича**

С. В. Козин, Н. А. Матиясевич

Программирование на языке высокого уровня

**САНКТ-ПЕТЕРБУРГ
2012**

Введение

Язык программирования Си разработан в начале 70-х годов Денисом Ритчи. Реализация языка относится к 1972г.

Язык Си традиционно используется для системного программирования. В частности, большая часть операционной системы UNIX была написана на этом языке. Однако язык Си находит широкое применение и в прикладном программировании.

Популярность языка Си обусловлена рядом факторов. Это эффективный, мощный язык программирования. Имеются два стандарта языка Си (C89 и C99). Оба стандарта содержат как определение языка, так и определение стандартной библиотеки. Для обеспечения переносимости программы следует использовать те его библиотечные функции, которые поддерживаются стандартом языка.

В основу создания языка были положены следующие принципы:

- Доверять программисту.
- Не мешать использовать программисту те средства, которые ему нужны.
- Сохранять язык простым и компактным.
- Обеспечить только один способ выполнения любой операции.
- Быстродействию следует отдать предпочтение по отношению к переносимости.

Некоторые авторы относят Си к языкам среднего уровня. В этом языке можно обнаружить как черты языка высокого, так и низкого уровня. С языками высокого уровня Си сближает наличие всех стандартных управляющих структур, которые характерны для структурного программирования. В языке Си используется развитый механизм подпрограмм. Используемые в языке подпрограммы называются функциями.

С языками низкого уровня Си сближает возможность работы с битами, байтами и адресами. Для языка Си характерна сравнительно низкая типизация, присущая языкам низкого уровня. В связи с этим на начальном этапе развития язык Си справедливо называли ассемблером высокого уровня. Одним из факторов исходного языка Си, определяющих низкий уровень типизации, являлось отсутствие контроля соответствия между параметрами определения функции и аргументами ее вызова. Это было обусловлено особенностями организации функций (старый стиль). В настоящее время в основном используется стиль организации функций, который называют новым. Последний заимствован Си из языка C++. Этот стиль основан на использовании так называемых прототипов. Прототип функции позволяет обеспечить корректную компиляцию вызова функции. Тем не менее, стандарт языка C89 допускал компиляцию вызова функции и при отсутствии прототипа. В этом случае компилятор создавал так называемое неявное объявление функции. Стандарт C99 неявное объявление не поддерживает.

Оценивая в целом язык Си, следует отметить его широкое применение, как для системного, так и для прикладного программирования. Следует

учитывать и наличие значительного количество языков, которые заимствовали базовый механизм организации типов, операторов и управляющих конструкций языка Си. К таким языкам относятся языки С++, C# и Java.

К недостаткам языка обычно относят сравнительно низкую типизацию и сложный синтаксис некоторых его конструкций.

Прежде чем переходить к последовательному изложению материала сделаем следующее замечание. Существуют различные варианты перевода на русский язык некоторых английских терминов, используемых в языке Си. В этом вопросе будем придерживаться принятым в [1] правилам перевода английских терминов на русский язык. Например, английский термин `operator` будем переводить как оператор, а термин `statement` будет переводиться как инструкция.

1. Структура программы на языке Си

Программа (Си - программа), написанная на языке Си, состоит из подпрограмм, которые в языке Си называются функциями. Точкой входа в Си – программу является функция `main()`. Отличие этой функции от других функций состоит в следующем:

- Выполнение Си - программы всегда начинается с первой инструкции функции `main()`.
- Завершение работы функции `main()` приводит к окончанию работы всей программы.

В отличие от некоторых других языков программа, написанная на языке Си, должна содержать, по крайней мере, одну подпрограмму (функцию). Такой функцией и является функция `main()`.

Функция в языке Си является основным строительным элементом, который используется при написании программ. На практике количество функций в программе может оказаться очень большим. В связи с этим возникает необходимость в объединении логически связанных функций. Такое объединение реализуется с помощью так называемых модулей. Под модулем в языке Си понимается один файл исходного текста с расширением .c.

В общем случае Си – программа имеет многомодульную структуру. На начальном этапе изучения языка Си ограничимся программами, исходный текст которых находится в одном файле. Такие программы являются одномодульными. Рассмотрим структуру простейших одномодульных программ.

2. Структура простейшей одномодульной программы

Как отмечалось выше, весь исходный текст одномодульной программы должен находиться в одном файле. Выделим среди одномодульных программ класс простейших одномодульных программ. К этому классу программ будем относить программы, состоящие только из одной функции `main()`.В

настоящей части пособия будут рассматриваться только такие программы. Отметим еще одно ограничение приводимых программ – в них будут отсутствовать так называемые внешние переменные. Внешними переменными называются переменные, которые объявляются вне функций. Программы, приводимые в настоящей части пособия, содержат три структурных компонента:

- Комментарий, относящийся ко всей программе в целом.
- Директивы препроцессора.
- Определение функции `main()`.

Остановимся на каждом из этих компонентов подробнее.

2.1. *Комментарии в программе на языке Си*

Комментарии используются для целей документирования программ. Комментарии не влияют на выполнение программы. При компиляции они удаляются из программы. Стандарт C99 позволяет использовать в программах комментарии двух видов:

- Комментарии, традиционные для языка Си.
- Однострочные комментарии, заимствованные из языка C++.

Традиционный комментарий в языке Си – это последовательность символов, входящих в его алфавит, которая начинается двумя символами “/*” и заканчивается этими же символами, следующими в обратной последовательности “*/.”. Такой комментарий может располагаться на нескольких строках. Приведем пример традиционного комментария.

```
/*
Лабораторная работа № 1
Программирование линейных вычислительных процессов

*/
```

Однострочный комментарий начинается с двух символов косая черта “//”. Такой комментарий включает в себя все символы текущей строки, следующие за этими символами. Ниже приводится пример применения однострочных комментариев.

```
//                 Лабораторная работа № 2
//    Программирование разветвляющихся вычислительных процессов
```

Следует учитывать, что не все компиляторы языка Си, используемые на практике, поддерживают однострочные комментарии.

2.2. *Начальные сведения о препроцессоре*

Препроцессор – это программа, которая выполняет предварительную обработку исходного текста программы, написанной на языке Си. Иногда препроцессор является самостоятельной системной программой, в других случаях он входит в состав компилятора. Препроцессор управляется так

называемыми директивами. В простейшем случае директива препроцессора имеет следующую структуру:

#имя_директивы `содержательная_часть`

Директива препроцессора всегда начинается с символа “#”, за которым следует имя директивы. Приведем примеры директив препроцессора:

```
#include<stdio.h>
#include<math.h>
#define MAXSIZE 100
```

В приведенном выше примере имеются три директивы препроцессора. Имя первой директивы include, а в ее содержательной части находится заключенное в угловые скобки имя системного заголовочного файла stdio.h. Эта директива предписывает препроцессору включить вместо рассматриваемой директивы содержимое системного заголовочного файла stdio.h. Наличие в модуле такой директивы необходимо в том случае, когда в нем содержатся вызовы функций стандартного ввода – вывода. В заголовочном файле stdio.h содержаться объявления функций стандартного ввода – вывода (о понятии объявления функции смотри немного ниже). Вторая директива с именем `include` выполняет такую же работу, что и первая директива препроцессора, но для математического заголовочного файла math.h. Этот заголовочный файл необходимо подключать в том случае, когда в исходном тексте модуля имеются вызовы библиотечных математических функций. Третья директива предписывает препроцессору заменить в рассматриваемом файле исходного текста все вхождения слова MAXSIZE константой `100`. Директива define используется для определения так называемых символических констант. В рассматриваемом случае такой константой является `MAXSIZE`. В языке Си существует традиция записывать заглавными буквами имя символической константы, определенной с помощью директивы `define`.

2.3. *Определение функции main().*

Определение функции main() строится по тем же правилам, что и определения других функций языка Си. Ее определение состоит из двух частей, которыми являются заголовок и тело. Существуют два вида допустимых заголовков функции `main()`. Первый вид заголовка используется в тех случаях, когда при запуске программы отсутствуют параметры, ей передаваемые. Такой вариант использования программы предусмотрен во всех задачах первой части настоящего пособия. В связи с этим ограничимся рассмотрением указанной разновидности заголовка функции main().

Заголовок функции main(), не принимающей параметров, имеет следующий вид.

```
int main(void)
```

Рассматриваемый заголовок содержит следующую информацию:

- Функция имеет имя `main`. На это указывают круглые скобки, следующие за этим именем.

● Функция main() должна вернуть в точку своего вызова целочисленное значение. Об этом свидетельствует зарезервированное слово **int**, с которого начинается заголовок, которое является спецификатором одного из целочисленных типов языка Си.

● У функции main() отсутствуют параметры. На это указывает зарезервированное слово **void**, стоящее в скобках.

Второй компонент определения функции main() – тело, содержащее алгоритм, ею реализуемый. В теле располагается последовательность предложений языка Си, которая заключена в фигурные скобки.

Ниже приводится простейшая программа. Это минимальная программа, которая может быть написана на языке Си. Она не выполняет никаких полезных действий, что дает основание считать ее пустой.

```
/*
Пустая программа.   Файл empty.c
*/
int main(void)
{
   return 0;
}
```

В рассматриваемой программе тело функции main() содержит единственное предложение, которым является инструкция **return**. Эта инструкция формирует результат, возвращаемый функцией main(). Этот результат функция main() передает операционной системе. Возврат значения, равного нулю, воспринимается операционной системой как нормальное завершение работы функции main().

2.4. *Пример простейшей программы*

Постановка задачи

Даны два целых числа a и b. Вычислить их сумму y.

Программа

Ниже приводится программа, предназначенная для решения данной задачи, а затем даются необходимые к ней пояснения.

```
/*
Программа, выполняющая сложение двух целых чисел
Файл   Summa.c
*/
#include<stdio.h>
#include<conio.h>

int main(void)
{
   int a, b, y;
   clrscr();
   printf("Первое число=");
   scanf("%d", &a);
   printf("Второе число=")
   scanf("%d", &b);
   y = a + b;
```

```
   printf("Сумма=%d\n", y);
   getch();
   return 0;
}
```

Протокол работы с программой.

Приведем протокол работы с рассматриваемой программой. Подчеркиванием выделен ввод пользователя.

```
Первое число=2<Enter>
Второе число=3<Enter>
```

Сумма=5

Анализ программы.

Заметим, что рассматриваемая программа содержит все три структурных компонента, характерные для простейших программ. Вначале приводится комментарий, относящийся ко всей программе в целом. Из этого комментария следует, что программа предназначена для сложения двух целых чисел, а ее исходный текст находится в файле `Summa.c`.

После комментария расположены две директивы препроцессора, содержащие указание для препроцессора подключить два заголовочных файла (`stdio.h` и `conio.h`). Первая директива предписывает выполнить эту работу для файла `stdio.h`, а вторая – для `conio.h`. Необходимость в подключении этих файлов связана с тем обстоятельством, что рассматриваемая программа работает с библиотечными функциями. Работа с функциями `printf()` и `scanf()` требует подключения файла `stdio.h`, а работа с функциями `clrscr()` и `getch()` - подключения файла `conio.h`.

Затем следует определение функции `main()`. Оно состоит из заголовка и тела. Заголовок имеет стандартный вид, совпадающий с заголовком пустой программы, текст которой приведен выше. Тело функции `main()` состоит из определения переменных и инструкций. В первой строке тела находится определение трех переменных `a`, `b` и `y`. Использование в определении спецификатора **`int`** является указанием на то, что все указанные переменные имеют целочисленный тип. Оставшаяся часть тела состоит из инструкций. Первой инструкцией является вызов функции `clrscr()`. Эта функция предназначена для очистки экрана. Заметим, что вызов любой функции, после которого поставлена точка с запятой, в программе, написанной на языке Си, становится инструкцией. Вызов функции языка Си состоит из ее имени и заключенного в круглые скобки списка аргументов. Вызов функции `clrscr()` не требует аргументов. Заметим, что в языке Си в вызове функции, не имеющей аргументов, нельзя опускать аргументные скобки. Это обусловлено тем обстоятельством, что круглые скобки заставляют компилятор рассматривать имя, после которого они находятся, как имя функции. Основная часть работы в рассматриваемой программе выполняется инструкциями, состоящими из вызова библиотечной функции, Все эти функции относятся к категории функций ввода – вывода. Остановимся на этих функциях более подробно.

К функциям ввода – вывода в рассматриваемой программе относятся функции printf(), scanf() и getch(). Основную работу по вводу – выводу выполняют первые две функции. Последняя функция (getch()) может оказаться необходимой для задержки экрана. В некоторых системах программирования необходимость в ее использовании может отсутствовать. Эта функция ожидает от пользователя нажатия на любую клавишу. Перейдем к рассмотрению функций printf() и scanf().

Функции printf() и scanf() относятся к так называемым функциям форматированного ввода – вывода. Функция printf() применяется для вывода на дисплей, а функция scanf() – для ввода с клавиатуры. В этих функциях предусмотрены специальные средства для управления процессом ввода – вывода. Эти средства называются спецификациями преобразования. В функции scanf() спецификации преобразовании используются всегда. В простейших вариантах использования функции printf() спецификации преобразования могут отсутствовать. Спецификации преобразования, если они имеются, всегда находятся в первом аргументе вызова рассматриваемых функций. Для этого аргумента используется специальное название: форматная строка. Каждая такая спецификация начинается с символа процент (%) и заканчивается символом, определяющим тип преобразования. В связи с тем, что рассматриваемая программа работает только с данными целого типа, в функциях ввода – вывода используется тип d, предназначенный для преобразования данных целого типа.

Рассмотрим объем работы, которую необходимо было выполнить в настоящей программе по вводу – выводу. Во-первых, необходимо было обеспечить ввод двух исходных чисел: a и b. Для этого следует использовать функцию scanf(). Оба эти числа можно ввести с помощью одного вызова этой функции. Однако предпочтительнее вводить каждое число с использованием отдельного вызова этой функции. Представляется, что такая организация ввода упрощает работу пользователя по вводу данных и уменьшает количество ошибок, которые может допустить программист при его организации. Этот способ ввода и принят в рассматриваемой программе. Вызов функции scanf() в программе должен быть предварен выводом наводящего сообщения, которое уведомляет пользователя, о том какая информация от него запрашивается. Эти наводящие сообщения выводятся в программе с помощью вызова функции printf().

Остановимся подробнее на организации ввода переменной a. В программе для этой цели служит следующий фрагмент кода:

```
printf("a=");
scanf("%d", &a);
```

Функция printf() здесь вызывается с одним аргументом – форматной строкой, однако спецификации преобразования в ней отсутствуют. Такая форма вызова функции printf() используется для вывода на экран сообщений для пользователя. Текст, находящийся между кавычками, выводится на экран дисплея. Вызов функции scanf() имеет два аргумента.

Первым аргументом является форматная строка, содержащая одну спецификацию преобразования (%d), которая уведомляет компьютер о том, что вводимая с клавиатуры порция информации должна рассматриваться как целое число. Второй аргумент определяет адрес памяти, в которую следует направить введенное с помощью функции scanf() число. Для получения этого адреса образуется выражение &a. Здесь символ & (оператор взятия адреса) предписывает компилятору вычислить адрес переменной a. Выражение &a является так называемым указателем (см. п. 1. 17). Значением указателя является адрес того программного компонента, на который установлен указатель. Ввод переменной b организован аналогично вводу переменной a.

Вычисления суммы представлены в программе инструкцией

```
y = a + b;
```

Эта инструкция предписывает вычислить выражение, стоящее справа от символа “=”, и результат вычисления сохранить в переменной y. Символ "=" в языке Си называется оператором присваивания.

Перейдем к рассмотрению вопросов, связанных с выводом результатов вычислений. Вызов функции printf(), выполняющей эту работу, имеет следующий вид:

```
printf("Сумма=%d\n", y);
```

В рассматриваемом вызове функции printf() форматная строка кроме уже знакомой спецификации преобразования %d содержит некоторые другие символы. К их числу относятся символы, предшествующие спецификации преобразования. Особенность этих символов, состоит в том, что каждый из них имеет графическое изображение. Эти символы образуют текст (Сумма =), назначение которого служить пояснением к выводимой числовой информации. В конце форматной строки находятся два символа (\n), составляющие так называемую управляющую последовательность. Рассматриваемая управляющая последовательность носит название “новая строка”. Ее назначение в приведенном вызове состоит в переводе курсора на новую строку экрана. Кроме форматной строки в рассматриваемом вызове присутствует второй аргумент, представленный переменной y, значение которой требуется вывести на экран. Спецификация преобразования, содержащаяся в форматной строке (%d), предписывает компилятору выполнить преобразования выводимого численного значения переменной y. Дело заключается в том, что информация, хранящаяся в памяти компьютера, связанная с этой переменной, имеет машинную форму. Информация, появляющаяся на экране в результате работы вызова функции printf(), представляет собой последовательность символов. Руководствуясь спецификацией %d, компьютер должен построить строковое изображение числа, хранящегося в переменной y и поместить его в выводимой строке вместо спецификации преобразования. Таким образом, одни элементы форматной строки выводятся один к одному на экран дисплея (например, Сумма=), другие выполняют управляющее действие (например, \n), наконец,

элементы спецификации (например, %d) определяют преобразования, которые можно выполнить при выводе. Представляется, что сведения, изложенные выше, достаточны для того, чтобы убедиться в правильности приведенного выше протокола работы программы.

3. Алфавит, синтаксис и семантика

Любой язык программирования состоит из трех частей:

- Алфавит.
- Синтаксис.
- Семантика

Алфавит языка определяет перечень символов, которые допустимо использовать в программе. Синтаксис определяет правила построения сложных конструкций языка из более простых конструкций. Семантика определяет смысл, который можно связать с синтаксическими конструкциями. Не все синтаксически допустимые конструкции языка имеют семантику.

4. Алфавит языка Си

В исходных текстах программ, написанных на языке Си, допускается использование следующих символов:

1. Латинские строчные и прописные буквы:
   a, b, …, z и A, B, …, Z
2. Цифры: 0, 1, …, 9
3. Символ подчеркивания: _
4. Пробел (код 32)
5. Специальные символы: +, -, * и др.
6. Составные символы, воспринимаемые как один символ:
   <=, >= и др.
7. В комментариях и строковых константах допустимо использование кириллицы.

5. Понятие о типе

Понятие о типе относится к числу базовых понятий в языках программирования. Это понятие может быть введено следующим образом:

- Тип определяет множество значений, которые может принимать данное.
- Тип определяет перечень операций, которые можно применять к данным.
- Каждое значение может относиться только к одному типу.

Различают сильно типизированные и слабо типизированные языки. В сильно типизированном языке запрещается в одной операции использовать данные разных типов, что позволяет компилятору выявлять ошибки, которые программист мог допустить во время написания программы. Однако сильная типизация имеет и свои недостатки. Дело в том, что некоторые типы имеют

близкие характеристики и представляется естественным допускать их совместное применение в одной операции, позволяя компилятору выполнять автоматическое преобразование одного типа в другой. Однако наличие чрезмерно большого количества автоматических преобразований типов, характерное для языка Си, может усложнять использование языка.

6. Система типов языка Си

Система типов языка Си

В соответствии с одной из систем классификации все типы языка Си делят на следующие четыре категории:

- function,
- **void**,
- скалярные,
- агрегатные.

Остановимся вначале кратко на каждой из четырех категорий типов.

Понятие типа обычно связывают с данными. Особенностью системы типов языка Си является то обстоятельство, что объявление функций строится по тем же правилам, что и объявления данных. Объявление функции используется компилятором и программистом. Компилятору объявление позволяет корректно выполнить компиляцию вызова функции. Программисту объявление функции позволяет правильно обратиться к функции, т. е. правильно написать вызов функции. Примеры объявлений функций можно найти в справочных материалах к используемому вами компилятору.

Следует отметить, что **void** является зарезервированным словом. Оно используется для следующих целей:

- В качестве типа значения, возвращаемого функцией. Если из заголовка функции следует, что типом значения, которое она возвращает, является **void**, то это означает, что функция не возвращает значения. В этом случае она может использоваться подобно процедуре такого языка, как Паскаль.
- Зарезервированное слово **void** в языке используется для указания на то обстоятельство, что функция не имеет параметров.

В качестве примера применения типа **void** может служить объявление библиотечной функции clrscr(), которая не имеет возвращаемого значения и не принимает параметров:

```
void clrscr(void);
```

Каждое данное, относящееся к категории скалярных типов, характеризуется только одним значением. Данные, относящиеся к категории агрегатных типов, характеризуется некоторой совокупностью значений. Остановимся вначале на категории скалярных типов.

В стандарте языка C99 к скалярным типам относятся следующие виды типов:

- арифметические,

● логический,
● указатели,
● перечисления.

Логический тип появился только в стандарте С99. В связи с тем, что многие компиляторы его еще не поддерживают, этот тип не будет рассматриваться в настоящем пособии.

Арифметические типы составляют большую группу типов. В ней можно выделить три части:

● целые,
● вещественные,
● комплексные.

Комплексные типы появились только в стандарте С99. В связи с тем, что многие компиляторы его еще не поддерживают, этот тип не будет рассматриваться в настоящем пособии.

Каждая из оставшихся разновидностей арифметических типов включает в себя достаточно большое количество отдельных типов. К числу основных (базовых) арифметических типов относят следующие типы. Для целочисленных типов базовыми типами считаются типы **char** и **int**, а для вещественных типов – **float** и **double**. Обращает на себя внимание тот факт, что символьный тип (тип **char**) в языке Си рассматривается как разновидность целочисленного типа. Это обстоятельство можно рассматривать как проявление ослабленной типизации, которое присуще языку Си.

Базовые арифметические типы могут быть модифицированы. Для целей модификации служит ряд зарезервированных слов, к числу которых относятся:

● **signed** (знаковый),
● **unsigned** (беззнаковый),
● **long** (длинный),
● **short** (короткий).

С учетом модификаторов можно написать 17 разновидностей арифметических типов.

Основным целочисленным типом в языке Си является тип **int**. Это аппаратно-зависимый тип. Размер, который занимает данное, относящееся к этому типу, зависит от размера машинного слова конкретной среды программирования.

Указатели являются специальным видом данных. Особенность указателя состоит в то, что его значением является адрес памяти, начиная с которого хранится данное. Указатели будут подробно рассматриваться во второй части пособия. В этой части пособия указатели рассматриваются только в объеме, необходимом для грамотной работы с библиотечными функциями.

Перечислимый тип является взаимосвязанным набором целочисленных констант. Перечислимые типы будут рассматриваться во второй части пособия.

К категории агрегатных типов, с которыми можно работать в языке Си, относятся следующие типы:

- массивы,
- структуры,
- объединения.

7. Понятие об объекте

Объект – это термин, ориентированный на работу с оперативной памятью компьютера. Под объектом в языке Си понимается участок оперативной памяти компьютера, в котором хранится некоторое значение. Одни данные в языке Си являются объектами, другие – объектами не являются. К объектам относятся переменные (кроме регистровых переменных). Литералы (константы) к объектам не относятся. К объектам также не относятся символические константы, объявляемые с помощью директивы define

Замечание. Возникает следующий вопрос, каким образом можно отличить данные, являющиеся объектом от данных таковыми не являющихся? Для ответа на этот вопрос к этому данному следует применить оператор взятия адреса (&). Такая операция может использоваться только для объектов, а для данных, не являющихся объектами, эта операция оказывается недопустимой.

8. Лексемы

Программа, написанная на языке Си, состоит из отдельных предложений. Предложения в свою очередь состоят из лексем. Лексемы играют роль “кирпичиков”, из которых строятся предложения языка. Лексемы неделимы и сами по себе определяют некоторое содержание. В языке Си различают следующие виды лексем:

- Зарезервированные слова.
- Идентификаторы.
- Литералы
- Разделители (знаки пунктуации).
- Операторы.

Зарезервированные слова и идентификаторы играют роль слов в предложениях программы, написанной на языке Си. Литералы применяются для обозначения фиксированных значений данных, используемых в программе.

Операторы применяются для обозначения операций, предусмотренных языком программирования. Например, оператор “+” используется для обозначения операции сложения двух чисел.

К разделителям относятся следующие лексемы: “[“, “]”, “(“, “)” и другие.

9. Зарезервированные слова

Зарезервированные слова имеют фиксированный смысл, который закреплен за ними определением языка, Этот смысл не может быть изменен

программистом. В связи с этим зарезервированные слова не могут использоваться в качестве слов пользователя, к которым относятся идентификаторы (смотри ниже). Не будем перечислять все зарезервированные слова, предусмотренные языком Си. Будем вводить эти слова по мере необходимости.

10. Идентификаторы

Идентификатор – это конструкция языка, используемая для целей наименования. Идентификаторы выбираются программистом по своему усмотрению, но с учетом синтаксиса языка. В языке Си в качестве идентификатора может использоваться любая последовательность цифр и букв, которая начинается с буквы. Символ подчеркивания считается буквой. Следует учитывать, что идентификаторы языка Си чувствительны к регистру. Стандарт языка Си не ограничивает длину идентификатора. Ограничивается количество значащих символов в идентификаторе. В соответствии со стандартом С89 количество значащих символов в идентификаторе равно 31, а в стандарте С99 – 63 символа.

В языке Си существует традиция записывать имена переменных строчными символами, а имена символических констант, которые определены с помощью директивы define, – заглавными.

Во избежание возможных конфликтов с именами, зарезервированными для системных целей, рекомендуется не начинать пользовательские имена с двух подчёркиваний и подчёркивание, за которым следует заглавная буква.

Примеры корректных идентификаторов:

```
Number
number              /* Идентификаторы Number и number – это два
                    разных идентификатора */
_number
count_words         /* Идентификатор состоит из двух смысловых
                    частей (слов), разделенных символом
                    подчеркивания*/
number_
```

Примеры некорректных идентификаторов:

```
for                 /* совпадает с зарезервированным словом */
1_st_slovo          /* начинается с цифры */
super+              /* содержит недопустимый символ*/
```

Примеры идентификаторов, которые могут конфликтовать с системными идентификаторами

```
_Name               /* Начинается с символа подчеркивания,
                     за которым следует заглавная буква */
__file              /* Начинается с двух подряд следующих
                     символов подчеркивания */
```

11. Литералы

Литералы или константы предназначены для представления фиксированных значений. В языке Си различают две категории констант:

- Явные константы или литералы
- Символические константы.

Явная константа – это константа, тип и значение которой определяются ее записью. Явные константы в языке Си часто называют литералами. Литерал относится к категории лексем. Особенность литералов состоит в том, что они не являются объектами. Например, 2 – это явная константа (литерал) целого типа.

Символическая константа в программе представлена своим именем. При выборе этого имени следует руководствоваться соображениями повышения читабельности программы. С учетом этого имя символической константы должно раскрыть ее назначение. Символическую константу в языке Си можно определить, например, с помощью директивы препроцессора define. В данном разделе рассматриваются только явные константы. Рассмотрим константы, относящиеся к различным типам данных. Начнем с целочисленных констант.

11.1. *Целочисленные литералы*

Целочисленные литералы служат для представления привычных из математики целых чисел. Прежде всего, следует учитывать, что в языке Си имеются три разновидности целочисленных литералов, которые различаются используемой системой счисления. Программист имеет возможность при записи целочисленных литералов воспользоваться следующими системами счисления:

- Восьмеричной,
- Десятичной,
- Шестнадцатеричной.

По умолчанию используется десятичная система счисления. Для указания на систему счисления, отличную от десятичной системы, литерал должен быть снабжен префиксом. Для указания на восьмеричную систему счисления следует в качестве такого префикса использовать цифру нуль, а для работы с шестнадцатеричной системой следует использовать один из двух возможных префиксов: 0x или 0X.

В связи с тем, что существует значительное количество разновидностей данных целого типа, целочисленный литерал может содержать суффикс для указания на тип значения, им определяемого. В языке Си используется следующие суффиксы:

- Для литералов, относящихся к типу **long**, - символ l или L,
- Для литералов, относящихся к типу **long long**, – символы ll или LL,
- Для литералов, относящихся к беззнаковым типам – символ u или U.

Суффиксы, относящиеся к беззнаковым типам, могут комбинироваться с суффиксами, используемыми при записи литералов типа **long** и **long long**.

Литералами считаются только положительные числа. Наличие перед числовым литералом знака минус рассматривается как применение унарного оператора минус “-”.

Ниже приводятся примеры целочисленных констант.

Константа	Пояснение
020	Восьмеричная константа. Десятичное значение равно 16
0x25	Шестнадцатеричная константа. Десятичное значение равно 37.
2U	Десятичная константа типа **unsigned int** (или просто **unsigned**)
0L	Десятичная константа типа **long int** (или просто **long**)

11.2. *Литерал вещественного типа*

В общем случае константа вещественного типа состоит из следующих структурных частей:

- мантисса,
- экспоненциальная часть,
- суффикс, определяющий тип константы.

Если экспоненциальная часть отсутствует, мантисса должна содержать десятичную точку. Экспоненциальная часть, если она присутствует в вещественной константе, должна начинаться либо с символа “e”, либо с символа “E”, за которым должен следовать целочисленный порядок. Экспоненциальная часть может содержать знак. Суффикс F (или f) указывает, что константа имеет тип **float**; использование суффикса L (или l) свидетельствует о том, что константа имеет тип **long double**. При отсутствии суффикса константа имеет тип **double**. Ниже приводятся примеры вещественных констант. Численное значение вещественной константы при наличии экспоненциальной части определяется умножением мантиссы на десять в степени, определяемой величиной порядка.

Константа	Тип
3.14159	**double**
.25	**double**
23.	**double**
4e2	**double** Значение константы равно 4 * 10^2
.5E-3	**double** Значение константы равно 0.5 * 10^{-3}
2.4f	**float**
5.5L	**long double**

11.3. *Символьные литералы*

Символьный литерал - это лексема, состоящая из графически воспроизводимого (печатного) символа или управляющей

последовательности, (escape - последовательность) заключенных в одинарные кавычки. Признаком начала управляющей последовательности, является символ обратная косая черта “\”. Понятие управляющей последовательности можно трактовать следующим образом: символ \ “управляет” интерпретацией последующих за ним символов. Например, n в управляющей последовательности \n воспринимается не как символ ‘n’, а как управляющий символ, предписывающий компилятору перевести курсор на новую строку экрана. Примеры управляющих последовательностей приведены ниже.

Управляющая последовательность	Значение
`\n`	Новая строка
`\r`	Возврат каретки
`\\`	Обратная косая черта
`\'`	Одиночная кавычка
`\''`	Двойная кавычка
\ooo	o – восьмеричная цифра
`\xhh`	h – шестнадцатеричная цифра

Примеры символьных литералов с использованием управляющих последовательностей:

```
'\n'
'\7'
'\x41'
```

Следует отметить, что символьный литерал в языке Си имеет тип **int**. Такое положение является признаком ослабленной типизации, имеющей место в этом языке.

11.4. *Строковый литерал*

Строковым литералом в языке Си называется последовательность символов, заключенная в двойные кавычки. Например:

```
"Hello"
```

В строковый литерал можно включать управляющие последовательности. Например:

```
"Hello\n"
```

Два строковых литерала, разделенные одним или несколькими пробелами, воспринимаются как один строковый литерал. Например, следующая последовательность строковых литералов:

```
"Hello,"    " world"
```

воспринимается компилятором как единый строковый литерал следующего вида:

```
"Hello, world"
```

Каждому строковому литералу, состоящему из “n” символов, во время выполнения программы выделяется блок памяти объемом n + 1 байт. Дополнительный байт выделяется для хранения нуль – символа. Типом

массива, предназначенного для хранения строковой константы, является **char**[n + 1].

12. Переменные

Понятие переменной в языке Си введено для упрощения работы с оперативной памятью компьютера. В программе переменная представлена своим именем (идентификатором). Имя переменной используется для обращения к участку оперативной памяти, предназначенному для хранения ее значения. Переменная в языке Си используется в качестве синонима понятия объекта. Особенность переменной как объекта состоит в том, что для переменной может быть вычислен адрес. Переменные применяются для хранения данных, с которыми работает программа. В переменных могут запоминаться исходные данные, результаты промежуточных вычислений и выходные данные программы.

Все переменные до их использования в программе должны быть объявлены. Обращение к переменной, для которой отсутствует объявление, приводит на стадии компиляции к появлению сообщения об ошибке (error). Это обстоятельство позволяет выявлять с помощью компилятора наличие, так называемых, “типографских” ошибок, которые могут появиться в процессе ввода текста программы.

Особой разновидностью объявлений переменных являются их определения. Определением переменной называется такое объявление, использование которого позволяет компьютеру выделить память для ее хранения. В связи с этим будем рассматривать две категории объявлений переменных:

- Определяющее объявление или просто определение. Это объявление переменной, одновременно являющееся определением.
- Ссылочное объявление. Это объявление, не являющееся определением.

Определяющее объявление позволяет компилятору создать переменную. Ссылочное объявление позволяет использовать переменную, созданную в другой области программного кода.

В программе для каждой переменной должно содержаться только одно определяющее объявление. Любая переменная может иметь произвольное количество ссылочных объявлений.

В зависимости от контекста, в котором находится определение, различают две категории переменных:

- локальные,
- внешние.

Локальной переменной называется переменная, определение которой находится в теле функции. Определение внешней переменной записывается вне функций.

Объявление локальной переменной одновременно является и ее определением. Это упрощает работу с такими переменными.

При работе с внешними переменными возникают определенные сложности. Дело в том, что программный код, в котором создается такая переменная, может быть далеко расположен от ее использующего программного кода. В этом случае в программном коде, в котором эта переменная применяется, может потребоваться объявление переменной. В ссылочном объявлении переменной обычно используется зарезервированное слово **extern**.

В стандарте C89 появилась возможность использования в определении переменных зарезервированного слова **const**. В связи с этим переменные, с которыми работает программа на языке Си, можно разделить на две категории:

- изменяемые переменные – переменные, в определениях которых не используется зарезервированное слово **const**,
- неизменяемые переменные – переменные, в определениях которых используется зарезервированное слово **const**.

Переменная характеризуется следующими атрибутами:

- Имя.
- Тип.
- Адрес.
- Область действия.
- Время жизни.
- Связывание.

Остановимся кратко на новых понятиях, к которым относятся: область действия, время жизни и связывание.

Область действия переменной определяет участок программного кода, в пределах которого на переменную можно ссылаться, используя для этого ее имя. Для переменных стандарт языка Си определяет две разновидности времени жизни:

- Блок (последовательность объявлений и инструкций, заключенная в фигурные скобки); блоком является тело функции, но тело может содержать вложенные в него блоки.
- Файл.

Время жизни определяет промежуток времени, в пределах которого за переменной сохраняется выделенная для нее память. Стандарт определяет две разновидности времени жизни:

- Автоматическое.
- Статическое.

Автоматическое время жизни по умолчанию имеют переменные, созданные в блоке (локальные переменные). Память, выделенная для такой переменной, освобождается в момент времени, когда управление покидает блок, в котором эта переменная создавалась. Освобождение памяти для таких переменных происходит автоматически без участия программиста.

Статическое время жизни совпадает со временем выполнения программы. Такое время жизни имеют все внешние переменные. Локальная

переменная будет иметь статическое время жизни в том случае, если в ее объявлении использовать зарезервированное слово **static**.

Связывание устанавливает взаимосвязь (если она существует) между различными “появлениями” одного и того же имени переменной. Стандарт языка Си определяет три вида связывания:

- Внешняя,
- Внутренняя
- Отсутствие связывания.

Внешнее и внутреннее связывание могут иметь только внешние переменные. Последний вид связывания (отсутствие связывания) имеют локальные переменные. Только переменные, имеющие внешнее связывание, доступны компоновщику. Это позволяет переменную (например, m), созданную в одном файле (например, f1.c), сделать доступной в другом файле (например, f2.c). Для этого в файле f2.c необходимо написать ссылочное объявление переменной m. Внутреннее связывание позволяет переменную “запрятать” в том файле, в котором она создана. Все внешние переменные по умолчанию имеют внешнее связывание. Для переменной можно установить внутреннее связывание, воспользовавшись в ее определении зарезервированным словом static.

Переменная может быть инициализирована во время своего объявления. Инициализация переменной состоит в назначении ей начального значения. Для этого объявление переменной должно содержать специальный элемент, который называют инициализатором. Объявление переменной, содержащее инициализатор, всегда является одновременно и ее определением. С переменными можно выполнять только две операции:

- Чтение,
- Запись.

Переменные, с которыми можно выполнять обе операции, называются изменяемыми. Операцию записи для переменной можно запретить. Для этого в ее определении следует использовать зарезервированное слово **const**

Переменные, которые можно использовать только в режиме чтения, называют константными. Такие переменные могут использоваться для создания символических констант.

Общий формат определения переменных имеет сложную структуру. В определение могут входить необязательные элементы. Начнем рассмотрение конкретных примеров с более простого случая, когда в объявление входят только обязательные элементы. В этом случае объявление будет иметь следующий формат:

спецификатор_типа список_переменных

В этом случае определение явно устанавливает только тип переменных с помощью элемента *спецификатор_типа*. Определение распространяется на все переменные, перечисленные в элементе *список_переменных*. Значения остальных атрибутов объявляемые переменные получают в соответствии с принципом умолчания.

Пример 1.

```
/*     Файл f1.c                        */
int m;
static int n;
int main(void)
{
   double x, y;
   /*    Переменные m и n доступны      */
   /*                                 */
   return 0;
}

/*    Файл f2.c                        */
    extern int m; /* Ссылочное объявление переменной m*/
                 /* Переменная m доступна            */
```

В этом программном коде имеются пять объявлений. Четыре из них находятся в файле f1.c. Все эти объявления относятся к категории определяющих объявлений. С помощью этих объявлений устанавливаются типы, и выделяется память для четырех переменных: m, n, x и y. В начале определены две целочисленные переменные m и n. Обе переменные являются внешними. Их область действия начинается от точки определения и распространяется до конца файла f1.c. В связи с этим переменные m и n доступны в теле функции main(). Переменная m имеет внешнее связывание. Это означает, что ее можно сделать доступной в любом другом файле программы. Например, она будет доступна в файле f2.c. Доступной ее в файле f2.c делает наличие в нем ссылочного объявления этой переменной. Время жизни этих переменных совпадает со временем выполнения программы.

Переменная n имеет внутреннее связывание. Это обусловлено наличием в ее объявлении зарезервированного слова **static**. В файле f2.c эта переменная не доступна.

В теле функции main() объявлены две локальные переменные x и y, имеющие тип **double**. Область действия этих переменных начинается от точки объявления и заканчивается закрывающей скобкой тела функции main(). В связи с этим рассматриваемые переменные нельзя использовать в любой другой функции программы, кроме функции main(). Время жизни этих переменных заканчивается в момент завершения работы вызова функции main().

Рассмотрим теперь более детально вопрос об инициализации переменных во время их объявления. Следует отметить, что эта операция должна для каждой переменной выполняться индивидуально. Для инициализации некоторой переменной необходимо записать оператор присваивания “=”, затем записывается инициализатор. Следует отметить, что для внешних переменных в качестве инициализатора можно использовать только так называемые константные выражения, а при инициализации локальных переменных допустимо использовать выражения общего вида. Приведем пример.

Пример 2.

```
int m = 10;
int main(void)
{
   double x = 1;
   double y = sin(x);
   /*    ...      */
   return 0;
}
```

В этом примере инициализированы во время объявления три переменные: внешняя переменная m и две локальные переменные x и y.

Наконец приведем пример объявления константной переменной. Как указывалось ранее, в объявлении такой переменной следует использовать зарезервированное слово const.

Пример 3.

```
const int MaxSize = 100;
int main(void)
{
   MaxSize = 200; /* Ошибка. Значение MaxSize изменять
                     нельзя */
   /*  */
}
```

В этом примере объявлена неизменяемая переменная MaxSize. Значение этой переменной зафиксировано в момент ее объявления и не может быть изменено в процессе выполнения программы.

13. Символические константы в языке Си

Символическая константа – это константа, которой в программе назначено некоторое имя. Применение символических констант вместо явных констант повышает читабельность программы и облегчает внесение в нее изменений. Одно из правил хорошего стиля программирования требует, чтобы в программе не было так называемых “магических чисел”. Под магическими числами понимаются конкретные числовые значения. Сложность работы с “магическими числами” заключается в отсутствии у них семантики. Символическая константа наделяет конкретное значение определенной семантикой, что и способствует повышению читабельности программы. В языке Си имеются три способа представления символических констант, в которых предусматривается использование соответственно:

- директивы препроцессора **define**.
- зарезервированного слова **const**.
- констант перечисления.

Приведем пример объявления символических констант.

```
#define PI 3.14159
const int BirthDay = 1964;
enum {MaxSize = 100};
```

Более предпочтительным способом определения символических констант является способ, основанный на применении зарезервированного

слова **const**. Это обусловлено тем обстоятельством, что при использовании этого способа работает аппарат контроля типов.

Имеется один недостаток, связанный с использование определения символических констант с помощью зарезервированного слова **const**. Дело в том, что символическую константу, определенную таким образом нельзя использовать при определении размера массивов. В этом случае можно воспользоваться константами перечисления (массивы будут рассматриваться во второй части пособия).

14. Операторы, выражения и инструкции. Общие сведения.

При программировании на языке Си необходимо различать следующие программные элементы:

- операторы,
- выражения,
- инструкции.

Операторы (operators) предназначены для выполнения некоторых элементарных действий. Например, в языке Си имеется оператор сложения (+), предназначенный для сложения двух чисел.

Выражение (expression) – конструкция языка, используемая для вычисления одного значения или (и) достижения побочного эффекта (side effect). О побочных эффектах при вычислении выражений см. п. 1.14.3. Выражения строятся из операндов, операторов и круглых скобок. В качестве операндов могут выступать константы, переменные, вызовы функций и выражения в круглых скобках. Литерал и переменную можно рассматривать как частный случай выражения. Пусть, например, имеется следующая синтаксическая конструкция:

```
3 + 5
```

Это пример простейшего выражения, в котором используется упоминавшийся выше оператор сложения.

Инструкция (statement) – это отдельное предложение языка Си, предписывающее компилятору выполнить некоторые действия. К числу инструкций относятся такие управляющие конструкции, как **if**, **for** и т. д. Важно отметить, что в языке Си имеется тесная связь между выражениями и инструкциями. Дело в том, что любое выражение, после которого поставлена точка с запятой, становится инструкцией. Такой вид инструкции называется инструкцией – выражением. Приведем пример. Пусть a, b и y – переменные типа **double**. Тогда следующая синтаксическая конструкция является выражением:

```
y = a * b
```

В этом выражении используются два оператора. Первым из них является оператор присваивания (=), а вторым - оператор умножения (*). Если теперь в конце этого выражения поставить точку с запятой, то получим инструкцию присваивания:

```
y = a * b;
```

В настоящем разделе основное внимание будет уделено операторам и выражениям.

14.1. *Классификация операторов*

Операторы языка Си делятся на три категории:

- унарные,
- бинарные,
- тернарные.

В основу этой классификации положено количество операндов, с которыми работает оператор. Унарные операторы имеют только один операнд, бинарные операторы работают с двумя операндами и у единственного в языке Си тернарного оператора – три операнда (? :). Например:

- -a (выражение содержит унарный минус),
- a – b (выражение содержит бинарный минус),
- a > b ? a : b (выражение содержит тернарный оператор).

14.2. *Приоритет и ассоциативность операторов.*

Важнейшими характеристиками операторов являются их приоритет и ассоциативность. Использование этих характеристик позволяет определить смысл выражения. Если отсутствуют скобки, то операнды сильнее связываются с операторами, имеющими более высокий приоритет. Можно говорить о том, что операторы с более высоким уровнем приоритета сильнее “притягивают” операнды. Если два оператора имеют одинаковый приоритет, то связывание определяется ассоциативностью. Имеются две разновидности ассоциативности операторов: левосторонняя и правосторонняя. При левосторонней ассоциативности процесс связывания начинается слева, а при правосторонней ассоциативности – справа. Приоритет и ассоциативность позволяют выполнить группировку операторов и операндов в выражении, содержащем более одного оператора.

Приоритеты операторов языка Си и их ассоциативность представлены в таблице, приведенной ниже.

Приоритет	Название оператора	Знак операции	Ассоциативность
1 (высший)	Функция	()	Левосторонняя
	Индексирование	[]	
	Селектор поля структуры	->	
	Селектор поля структуры	.	
	Инкремент постфиксный	++	
	Декремент постфиксный	--	
2	Логическое отрицание	!	Правосторонняя
	Побитовое отрицание	~	
	Инкремент префиксный	++	

	Декремент префиксный	`--`	
	Унарное сложение	`+`	
	Унарное вычитание	`-`	
	Разыменование	`*`	
	Взятие адреса	`&`	
	Приведение типа	(тип)	
	Размер объекта	`sizeof`	
3	Умножение	`*`	
	Деление	`/`	
	Вычисление остатка деления	`%`	
4	Сложение бинарное	`+`	
	Вычитание бинарное	`-`	
5	Сдвиг влево	`<<`	
	Сдвиг вправо	`>>`	
6	Меньше	`<`	
	Неменьше	`>=`	Левосторонняя
	Больше	`>`	
	Небольше	`<=`	
7	Сравнение на равенство	`==`	
	Сравнение на неравенство	`!=`	
8	Побитовое И	`&`	
9	Побитовое исключающее ИЛИ	`^`	
10	Побитовое ИЛИ	`|`	
11	Логическое И	`&&`	
12	Логическое ИЛИ	`||`	
13	Вычисление выражения по условию	`?:`	
14	Присваивания:	`= += -= *= /= %= >>= <<= &= ^= &= ^= |=`	Правосторонняя
15 (низший)	Запятая	`,`	Левосторонняя

В таблице операторы, имеющие одинаковый приоритет, объединены в группы. За каждой группой операторов закреплен номер, определяющий ее приоритет. Наибольший приоритет (1) имеет группа, расположенная вначале таблицы, а наименьший (15) – имеет оператор “запятая”, который находится в конце таблицы.

Приведем примеры выражений, при вычислении которых необходимо учитывать приоритет и ассоциативность операторов.

Пример 1. Рассмотрим следующее выражение:

```
a + b * c
```

В этом выражении используются два оператора: + (сложить) и * (умножить). Оператор умножить имеет более высокий приоритет по сравнению с оператором сложить. Это следует из таблицы приоритетов, приведенных выше. В рассматриваемом примере вначале будет выполнено умножение переменных b и c, а затем - сложение полученного результата со значением переменной a и полученное значение станет значением всего рассматриваемого выражения.

Пример 2. Рассмотрим выражение, в котором операторы имеют одинаковый приоритет:

```
a * b / c
```

Операторы * и / (делить) имеют одинаковый приоритет. В этом случае следует учитывать ассоциативность операторов. Обратившись к таблице, приведенной выше, находим, что операторы * и / имеют одинаковый приоритет и являются левоассоциативными. При левосторонней ассоциативности операторы выполняются в порядке их следования слева направо. Поэтому вначале будет выполнено умножение, а затем деление.

14.3. *Побочные эффекты и вычисления выражений*

Следует учитывать, что при вычислении выражений языка Си возможно появление побочных эффектов. Побочные эффекты могут иметь различную природу. Например, имеются операторы языка Си (присваивание, инкремент и декремент), при выполнении которых изменяется значение операнда. Побочные эффекты могут иметь место при выполнении вызовов функций.

Обратимся к приведенной выше инструкции присваивания

```
y = a * b;
```

Побочный эффект здесь состоит в изменении операнда “y”, которому присваивается новое значение, равное значению выражения a * b.

Результат вычислений, предусмотренный выражением, может использоваться двумя способами:

- в виде значения выражения,
- с помощью побочного эффекта.

Основным способом является использование значения выражения. В процессе выполнения программы выражение замещается вычисленным значением.

Операторы инкремента и декремента рассматриваются в п. 1.15.1. Там же обсуждаются и побочные эффекты при их выполнении.

Функции языка Си имеют много общего с математическими функциями. Существенное отличие функций языка Си от математических функций состоит в том, что их выполнение может сопровождаться появлением побочных эффектов. В этом отношении язык Си не является уникальным. Побочные эффекты имеют место при работе функций многих языков программирования.

Перейдем к выяснению существа рассматриваемой проблемы. Пусть имеется некоторая произвольная математическая функция y = f(x).

Многократные вычисления этой функции для фиксированного значение аргумента x дают всегда один и тот же результат. Причем на этот результат не влияют вычисления других математических функций. Иначе обстоит дело при использовании функций языка Си. Для иллюстрации сказанного приведем пример, в котором используются две библиотечные функции: `srand()` и `rand()`, прототипы которых имеют следующий вид:

```
#include<stdlib.h>
void srand(unsigned seed);
int rand(void);
```

Функции srand() и `rand()` применяются для получения так называемой псевдослучайной последовательности чисел. Функция `rand()` выполняет основную работу по формированию такой последовательности. Для получения последовательности, состоящей из `n` чисел, необходимо `n` раз вызвать эту функцию. Функцию `srand()` следует предварительно вызвать для формирования каждой новой последовательности псевдослучайных чисел. Вызов этой функции используется для инициализации генератора псевдослучайных чисел. Инициализация генератора одним и тем же значением аргумента функции `srand()` приводит к созданию идентичных псевдослучайных последовательностей. Последовательность чисел, сгенерированная функцией `rand()` без инициализации генератора, идентична полученной при инициализации с помощью функции `srand()` со значением аргумента, равным 1. Ниже приведен программный код, с помощью которого формируются две числовые последовательности, состоящие каждая из трех чисел.

```
srand(10);
printf("%-20s%5d", "The first series: ", rand() % 100);
printf("%5d", rand() % 100);
printf("%5d\n", rand() % 100);
srand(50);
printf("%-20s%5d", "The second series: ", rand() % 100);
printf("%5d", rand() % 100);
printf("%5d\n", rand() % 100);
```

Замечание 1. В выражении rand() % 100 используется оператор “вычисления остатка” от деления, использующий знак `%`. Этот оператор вычисляет остаток от деления левого операнда на правый.

Замечание 2. Текстовые литералы: “The first series ” и “The second series” выводятся с использованием выравнивания по левой границе отведенного для каждого поля шириной в 20 позиций. Для этого в спецификации им соответствующей используется модификатор минус (-). При выводе чисел используется принятое по умолчанию выравнивание по правой границе поля.

В результате выполнения этого фрагмента кода в системе программирования `Dev C++ version 4` были получены две последовательности псевдослучайных чисел:

```
The first series:  72   99   71
The second series:  1   51   10
```

Анализ полученных результатов позволяет сделать следующие выводы:

- Значения, возвращаемые функцией `rand()`, в каждой из последовательностей зависят от аргумента, передаваемого функции `srand()`.
- Функция `rand()` не имеет параметров. Несмотря на это при каждом ее очередном вызове возвращается новое значение.

Оба выявленных выше факта являются следствием побочных эффектов, имеющих место при выполнении рассматриваемых функций.

Побочный эффект при выполнении функции может состоять в выполнении дополнительных действий. К таким дополнительным действиям, например, может относиться вывод на экран дисплея, очистка экрана. Другим видом побочного эффекта при выполнении функции может являться изменение значений переменных, действующих в точке вызова. Рассмотрим следующий программный код.

```
int n;
printf("n=");
scanf("%d", &n);
```

Побочный эффект при выполнении вызова функции printf(), состоит в выводе на экран дисплея наводящего сообщения (n =), а побочный эффект при выполнении функции scanf() состоит в изменении значения переменной n.

При анализе побочных эффектов важно знать, когда они завершаются. Программные точки, в которых гарантированно заканчиваются побочные эффекты, называются точками следования (sequence point). Точки следования имеют место:

- в конце каждого выражения, не являющегося подвыражением,
- после вычисления первого операнда следующих операторов: “&&” (Логическое И), “||” (Логическое ИЛИ), “?: “ (Вычисление выражения по условию), “,” (оператор запятая).
- после вычисления всех параметров функции перед началом выполнения тела функции.
- в конце выражений, управляющих работой инструкций: if, switch, while, do..while и в конце каждого из трех выражений в инструкции for.

14.4. *Порядок вычисления выражений*

Под порядком вычисления (`order of evaluation`) понимается порядок, в котором вычисляются значения отдельных членов выражения. В языке Си имеется небольшое количество операторов, для которых определен порядок вычисления операндов. К числу таких операторов относятся:

- логическое И (`&&`),
- логическое ИЛИ (`||`),
- тернарный оператор (`?:`),
- оператор запятая (`,`).

Для других операторов очередность вычисления операндов не определена. Например, при анализе выражения `sin(x) + cos(x)` нельзя

ожидать, что вызов функции `sin(x)` будет выполнен до вызова функции cos(x).

15. Арифметические операторы и выражения

В языке Си имеется шесть унарных и пять бинарных арифметических операторов, показанных в приводимой ниже таблице. В таблице предполагается, что переменные `i` и `n` имеют целый тип, а переменные a и b – любой числовой тип.

Оператор	Назначение	Пример
`+`	Унарный плюс	+a
`-`	Унарный минус	-a
`++`	Две разновидности инкремента	++a (префиксный инкремент) a++ (постфиксный инкремент)
`--`	Две разновидности декремента	--a (префиксный декремент) a-- (постфиксный декремент)
`+`	Бинарное сложение	`a + b`
`-`	Бинарное вычитание	`a - b`
`*`	Умножение	`a * b`
`/`	Деление	`a / b`
`%`	Остаток от деления нацело	`i % n`

15.1. *Унарные операторы*

Остановимся на отдельных операторах. Отметим, что унарный оператор “+” относится к категории программистских недоразумений. Этот оператор никаких полезных действий не выполняет. Дело в том, что значение выражения, содержащего этот оператор, совпадает со значением его операнда. Выражение, содержащее унарный оператор “-”, возвращает значение, отличающееся от операнда только знаком.

Особое место среди унарных операторов занимают операторы инкремента (++) и декремента (--). Операнд, к которому применяются эти операторы, должен относиться к категории lvalue. Особенность этих операторов связана с наличием при их выполнении побочного эффекта. Побочный эффект состоит в изменении значения операнда на 1. Причем для инкремента значение его операнда увеличивается на 1, а для декремента - уменьшается на 1. Оба оператора существуют в двух формах: префиксной и постфиксной формах. Форма оператора не влияет на побочный эффект, имеющий место при выполнении оператора, а определяет возвращаемое оператором значение. При префиксной форме оператор возвращает измененное значение операнда, а при использовании постфиксной формы возвращается первоначальное значение операнда.

Приведем пример. Пусть имеется следующий фрагмент программы.

```
/*    ..........................*/
int n = 5;
int m = 5;
printf("++n=%d\n", ++n);
printf("=%d\n", n);
printf("m++=%d\n", m++);
printf("m=%d\n", m);

/*    ...........................*/
```

В результате выполнения рассматриваемого фрагмента программы вывод на экран дисплея будет иметь следующий вид:

```
++n=6
n=6
m++=5
m=6
```

В рассматриваемом примере к переменным “n” и “m”, имеющим равные значения, были применены различные формы инкремента. К переменной “n” была применена префиксная форма инкремента, а к переменной “m” – постфиксная форма. Окончательные значения рассматриваемых переменных оказались одинаковыми. Значения, возвращаемые в результате вычисления выражений ++n и m++, оказались разными. Выражение ++n вернуло новое значение своего операнда (с учетом побочного эффекта), а выражение m++ вернуло исходное значение своего операнда.

15.2. *Бинарные операторы*

Перейдем к рассмотрению бинарных арифметических операторов. Операторы “+” (сложить), “-” (вычесть), “*” (умножить) имеют обычный арифметический смысл. При использовании бинарных операторов в языке Си допускается использование операндов, имеющих разные арифметические типы. При этом компилятор выполняет автоматическое преобразование типа. Следует учитывать особенность, имеющую место в языке Си, при выполнении операции деления (оператор /) для данных целого типа. Результат выполнения операции деления в этом случае имеет целый тип. Стандарт языка Си строго определяет эту операцию только для случая, когда оба операнда положительны. В этом случае дробная часть отбрасывается. В том случае, когда только один операнд отрицателен, результат зависит от реализации. Приведем пример. Рассмотрим результаты выполнения следующего вызова функции printf()

```
printf("23 / 4 = %d  -23 / 4= %d\n", 23 / 4, -23 / 4);
```

При выполнении программы, содержащий этот вызов функции, в среде Builder v. 6 был получены следующие результаты:

```
23 / 4= 5  -23 / 4= - 5.
```

Отметим, что при выполнении в другой среде при вычислении выражения -23 / 4 может быть получено число -6.

Бинарный оператор “%” выполняет вычисление остатка от деления левого операнда на правый операнд. Каждый из операндов может иметь

любой целочисленный тип. Результат выполнения рассматриваемого оператора, если один из операторов отрицателен, зависит от реализации.

Бинарные операторы “ – ” и “+” имеют одинаковый приоритет, который ниже приоритета операторов “*”, “/” и “%”. Приоритет унарных операторов выше приоритета бинарных операторов. Унарные операторы имеют правую ассоциативность, а бинарные операнды – левую ассоциативность.

15.3. *Преобразования типа при выполнении бинарных операторов*

Вычислять значение выражения можно только в том случае, если его операнды имеют одинаковые типы. Компилятор сильно типизированного языка должен “отказываться” обрабатывать выражения, операнды которых имеют разные типы. Однако такой подход не всегда является оправданным. Дело в том, что существуют близкие по своему назначению типы. К таким, например, типам относятся арифметические типы. В распоряжение программиста язык Си предоставляет большое количество разновидностей арифметических типов. Представляется целесообразным допускать использование некоторых категорий смешанных выражений, операнды которых имели бы разные, но близкие по своему назначению типы. Такой подход реализован в языке Си.

В языке Си предусмотрена возможность выполнения преобразования типов при вычислении смешанных арифметических выражений. Назначение преобразования типа состоит в том, чтобы привести операнды к одному общему типу. Язык предусматривает две разновидности преобразования типов;

- Неявное (автоматическое).
- Явное.

Остановимся вначале на автоматическом преобразовании типов.

Автоматическое преобразование типов

Правила выполнения автоматических преобразований типов таковы, что обычно не приводят к потере информации. Дело в том, что при их использовании операнды с меньшим диапазоном значений преобразуются в операнды с большим диапазоном значений, как, например, преобразование целого числа в вещественное число в выражении 2 + 5.3.

При выполнении автоматических преобразований типов все арифметические типы разделены на две категории:

- Короткие типы
- Длинные типы.

В основу этой классификации положен размер памяти, которую занимают данные.

К коротким типам относятся следующие типы:

- **char**,
- **signed char**,

▪ **unsigned char**,
▪ **short**,
▪ **unsigned short**.

Для всех коротких типов при преобразованиях типов вначале выполняется операция, которая называется целочисленным повышением: все короткие типы преобразуются к типу **int**. Для этого правила имеется одно уточнение. Тип **unsigned short** преобразуется в тип **int** только в том случае, когда этот тип (**int**) достаточен для представления всего диапазона значений типа **unsigned short** (обычно это имеет место в тех системах, где для представления данных типа **short** отводится половина машинного слова и полное слово – для данных типа **int**). В последнем случае тип **unsigned short** преобразуется в **unsigned int**. Рассмотрим программный код, приведенный ниже:

```
#include<stdio.h>
#include<conio.h>
int main(void)
{
   short n = 5;
   char  ch = ‘A’;
   printf(“n + ch = %d  sizeof(n + ch)=\n”, n + ch,
          sizeof(n + ch));
   getch();
   return 0;
}
```

В вызове функции printf() вычисляется значение выражения n + ch и размер памяти, который занимает результат вычисления этого значения. При выполнении этой программы в среде Builder v . 6 на экране дисплея получен следующий результат:

```
n + ch = 70  sizeof(n + ch)=4
```

Значение переменной ch, преобразованное к типу **int**, будет равно 65. Сложение этого числа со значением переменной n и дает в качестве результата вычисления выражения n + ch число 70. Завершающая часть вывода (sizeof(n + ch) = 4) может рассматриваться как доказательство того положения, что вычисления действительно выполнялись с операндами, преобразованными к типу **int**.

При преобразовании длинных типов используется принцип ранжирования. С каждым из таких типов связывается его ранг. Ранжированная последовательность длинных типов, начиная с типа, имеющего наивысший ранг, имеет следующий вид:

- **long double**,
- **double**,
- **float**,
- **unsigned long long**,
- **long long,**
- **unsigned long,**
- **long,**
- **unsigned,**

- **int.**

При использовании в выражении операндов, имеющих разные ранги, тип операнда, имеющий меньший ранг, приводится к типу операнда, имеющего больший ранг. Например, если в выражении один операнд имеет тип **double**, а второй – **long**, то тип второго операнда должен быть преобразован к типу **double**.

Из этого правила существует одно исключение. Если один из операндов имеет тип **long**, а второй - **unsigned**, причем не все значения **unsigned int** могут быть представлены типом **long**, то оба операнда преобразуются к типу **unsigned long**.

Явное преобразование типа

Явное преобразование типа основано на применении оператора приведения типов. Общий вид применения оператора приведения типа имеет следующий вид:

```
(тип) выражение
```

Здесь *тип* – это любой тип, который поддерживается языком Си. Например, следующая запись позволяет преобразовать значение выражения x * y / 5 к типу **float**:

(**float**) (x * y / 5)

В отличие от автоматических явные преобразования типов могут быть как безопасными, так и опасными.

Оператор приведения типа является унарным оператором и имеет тот же приоритет, что и другие унарные операторы.

15.4. *Математические функции*

При написании арифметических выражений часто приходиться использовать стандартные математические функции, прототипы которых находятся в заголовочном файле math.h. Вызов функции состоит из ее имени и заключенного в круглые скобки списка фактических параметров. В качестве фактического параметра математических функций можно использовать арифметическое выражение. Ниже в таблице приведены прототипы некоторых математических функций.

Имя функции	Прототип	Описание
abs	**int** abs(**int** num);	Вычисление модуля аргумента num
ceil	**double** ceil(**double** num);	Возвращает наименьшее целое, которое удовлетворяет условию >= num. Обратите внимание на тип возвращаемого значения (**double**).
cos	**double** cos(**double** num);	Вычисляет значение косинуса от аргумента num. Значение аргумента должно быть задано в радианах.
fabs	**double** fabs(**double** num);	Вычисление значение модуля аргумента num
floor	**double** floor(**double** num);	Возвращает наибольшее целое,

		которое удовлетворяет условию <= num.
exp	**double** exp(**double** num)	Вычисляет значение экспоненты от аргумента num
log	**double** log(**double** num);	Вычисляет значение натурального логарифма от аргумента num.
log10	**double** log10(**double** num);	Вычисляет значение логарифма по основанию 10 от аргумента num.
pow	**double** pow(**double** base, **double** x);	Вычисляет значение аргумента base, возведенное в степень x.
sin	**double** sin(**double** num);	Вычисляет значение синуса от аргумента num. Значение аргумента должно быть задано в радианах.
sqrt	**double** sqrt(**double** num);	Вычисляет значение корня квадратного от аргумента num.
tan	**double** tan(**double** num);	Вычисляет значение тангенса от аргумента num. Значение аргумента должно быть задано в радианах.

16. Оператор присваивания и инструкция присваивания

В языке Си различают две формы оператора присваивания:

- простой оператор присваивания,
- составной оператор присваивания.

Составной оператор отличается наличием дополнительной операции, которая выполняется до присваивания. Рассмотрим вначале простой оператор присваивания.

Любой оператор присваивания, не входящий в состав другого выражения, может выступать в качестве инструкции присваивания, если после него поставить точку с запятой. Например, n = 5 – это оператор присваивания, а n = 5; - это инструкция присваивания.

16.1. *Простой оператор присваивания*

Основное назначение оператора присваивания в языках программирования состоит в выполнении следующих операций:

- в запоминании в памяти входных данных и результатов вычислений некоторых выражений.
- в перемещении данных из одной области памяти в другую.

В языке Си присваивание имеет расширенное по сравнению с другими языками (Паскаль и Бейсик) назначение. Дело в том, что в языке Си может использоваться возвращаемое оператором присваивания значение. Это связано с тем обстоятельством, что присваивание в языке Си относится к категории операторов (operator), а не инструкций (statement), как это имеет место в других языках. Приведем пример, в котором используется значение, возвращаемое оператором присваивания.

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

```
int main(void)
{
  int x = 10;
  int y = 20;
  printf("%d\n", x = y);
  /*  ...  */
  getch();
  return 0;
}
```

В этом программном коде вызов функции printf() используется для вывода на дисплей результата выполнения оператора присваивания.

Оператор присваивания относится к категории бинарных операторов. В качестве знака присваивания в языке Си используется математический знак равенства “=”. При выполнении оператора присваивания вначале вычисляется его правый операнд, затем вычисленное значение записывается в область памяти, адрес которой определяется его левым операндом. Требование знания адреса накладывает ограничения на выражение, которое может использоваться в качестве левого операнда присваивания. К левому операнду должна быть допустима операция взятия адреса. Выражения, допускающие применение этой операции, принято называть lvalue выражениями. Различают две категории lvalue выражений:

- изменяемые,
- неизменяемые.

В качестве левого операнда присваивания следует применять модифицируемое lvalue выражение.

Возвращаемым значением оператора присваивания является значение, записанное в его левый операнд. В книгах по языку Си и в сообщениях компилятора за левым операндом оператора присваивания закрепился термин lvalue (left side value), а за правым - rvalue (right side value) Таким образом, в общем случае простой оператор присваивания имеет следующий формат:

```
модифицируемое_lvalue_выражение = rvalue_выражение
```

Простейшим видом *модифицируемого_lvalue_выражения* является простая переменная, в объявлении которой отсутствует зарезервированное слово **const**. К числу других допустимых выражений относятся выражения, используемые для доступа к элементам массивов и структур. В качестве правого операнда оператора присваивания может использоваться произвольное выражение, тип которого совместим по присваиванию с типом операнда, стоящего слева от знака присваивания. Все арифметические типы в языке Си совместимы по присваиванию.

Приведем примеры операторов присваивания. Пусть x и y – переменные, имеющие тип **double**. Тогда допустимы следующие операторы присваивания.

```
x = 4.5
y = x - 3
```

Отметим, что приведенные выше операторы присваивания становятся инструкциями присваивания, если в конце каждого из них поставить точку с запятой:

```
x = 4.5;
y = x - 3;
```

Приведем примеры некорректных присваиваний.

```
define MAX 100
const int n = 10;
int main(void)
{
   n = 1;           /* Ошибка */
   MAX = 200;       /* Ошибка */
   return 0;
}
```

При компиляции этого программного кода в среде Builder v. 6 появились два сообщения об ошибках (error). Первое из них относилось к инструкции “n = 1;” и содержало следующее сообщение “Cannot modify const object” (Нельзя модифицировать константный объект). Второе сообщение было связано с инструкцией “MAX = 200;” и содержало сообщение “Lvalue required” (Lvalue требуется).

16.2. .Множественное присваивание

Язык Си позволяет в одном выражении присвоить значение нескольким переменным. Такое присваивание называется множественным. Пусть имеются три переменные k, n, и m, имеющие тип int. Тогда допустима следующая инструкция присваивания:

```
k = n = m = 4;
```

В результате вычисления приведенного выше выражения все три переменные получат значение, равное 4.

16.3. Составной оператор присваивания

Составной оператор присваивания имеет следующий формат

```
a @= b
```

Здесь @ – один из десяти операторов: +, -, *, /, %, <<, >>, &, |, ^.

Программная конструкция

```
a @= b
```

полностью эквивалентна программной конструкции

```
a = a @ b
```

Приведем ряд примеров. Пусть имеются следующие операторы присваивания, записанные в форме простого присваивания, для которых имеется возможность записать их в форме составного присваивания:

```
x = x + dx
p = p * i
```

Эквивалентные этим простым присваиваниям составные присваивания будут иметь следующий вид:

```
x += dx
p *= i
```

16.4. *Преобразование типа при присваивании*

Если при выполнении оператора присваивания тип переменной, стоящей слева от оператора присваивания, не совпадает с типом значения выражения, находящегося справа, то выполняется преобразование типа в соответствии со следующим правилом: тип значения выражения приводится к типу переменной.

Следует учитывать, что при присваивании возможна потеря информации. Это может иметь место в том случае, когда диапазон значений типа значения выражения превосходит диапазон типа переменной. Это, например, возможно при выполнении следующего фрагмента программы.

```
int main(void)
{
  short small;
  long big;
  float f;
  double d;
  /*   …   */
  small = big;
  f = d;
  big = d;
  /*   …   */
 }
```

Все приведенные выше присваивания относятся к разряду потенциально опасных операций, при которых возможна потеря информации. Рассмотрим первое из этих присваиваний: small = big. В этом случае обе переменные имеют целочисленные значения. Однако диапазон значений типа переменной big превосходит диапазон значений переменной small. Действительно, переменная small имеет тип **short**, а переменная big – тип **long**. Диапазон значений типа **long** превосходит диапазон значений типа **short**. Такие присваивания в языке Си выполняются путем отбрасывания старших разрядов записываемого в переменную значения. Во втором присваивании: f = d переменной типа **float** присваивается значение типа **double**. При таком присваивании возможна потеря информации

17. Начальные сведения об указателях. Выходные параметры функции

Указатели чрезвычайно широко применяются при программировании на языке Си. Например, указатели используются в вызове функции scanf() (см. программный код, приведенный в пункте. 1.2.4). Понять работу этой функции, не располагая хотя бы самыми общими сведениями об указателях, весьма затруднительно. В связи с этим ниже приводятся начальные сведения об указателях. Представляется, что приводимые сведения позволят уяснить механизм возврата результатов вычислений, полученных в функции, через ее параметры.

Указатели предназначены для работы с адресами. Существуют две разновидности указателей. К первой из них относятся указатели–переменные. Такие указатели предназначены для хранения адресов. Ко второй разновидности относятся указатели–выражения, служащие для вычисления (получения) адресов.

Ограничимся рассмотрением одной категории указателей–выражений, которая используется для получения адреса объекта. Такой вид указателя встречается при работе с функцией scanf(). Пусть x – переменная одного из арифметических типов. Тогда применение к этой переменной унарного оператора взятия адреса & позволяет образовать указатель–выражение следующего вида &x.

Простейший формат объявления указателя–переменной имеет следующий вид

```
тип_объекта *имя_переменной_указателя
```

Приведем пример объявления указателя, предназначенного для хранения адресов объектов типа **double**

```
double *px;
```

Указатель – переменную можно инициализировать во время объявления. Такая инициализация выполнена с помощью указателя–выражения в приводимом ниже коде.

```
double x  = 10.5;
double *px = &x;
```

Здесь &x – указатель – выражение, которое позволяет получить адрес объекта x.

Основным оператором при работе с указателями является унарный оператор разыменования (*). С помощью операции разыменования можно обратиться к объекту, с которым работает указатель (на который установлен указатель.) Приведем пример

```
double x  = 10.5;
double *px = &x;
*px = 10;
printf("*px=%0.4g\n", *px);
```

В этом примере выражение *px используется дважды. Один раз для записи в объект, на который установлен указатель, а второй раз – для чтения.

Полезной является точка зрения, в соответствии с которой указатель – это программный компонент, управляющий доступом к другому программному компоненту. В этом примере с помощью указателя px, получаем доступ к памяти, отведенной для переменной x.

Теперь представим, что указатель px и объект x, с которым указатель работает, принадлежат различным функциям. В этом случае одна функция получает доступ к памяти, выделенной в другой функции. Это дает возможность функции возвращать в точку вызова результаты, полученные в теле вызываемой функции, используя аппарат параметров. Для иллюстрации этой возможности рассмотрим библиотечную функцию modf(), которая используется для извлечения из действительного числа целой и дробной части.

Прототип рассматриваемой функции имеет следующий вид:

```
#include<math.h>
double modf(double value, double *iptr);
```

Функция modf() имеет два параметра. С помощью параметра value в функцию передается исходное действительное число. Функция извлекает из переданного числа дробную и целую части. Функция modf() возвращает дробную часть, а целая часть сохраняется в переменной, на которую будет установлен указатель iptr.

Ниже приводится программный код, иллюстрирующий работу рассматриваемой функции.

```
#include <stdio.h>
#include<math.h>
#include<conio.h>
int main(void)
{
   double n, x;
   x = modf(10.2, &n);
   printf("n= %0.4g x= %0.4g", n, x);
   getch();
   return 0;
}
```

В приведенном программном коде, дробная часть исходного числа, которую вернула функция modf(), сохранена функцией main() в переменной x, а целая часть оказалась записанной функцией modf() в переменную n. Протокол работы с рассматриваемой программой, который приведен ниже, подтверждает правильность выполненного анализа:

n= 10 x= 0.2

18. Принятие решений и логические величины. Операторы отношения и сравнения на равенство

При разработке программ часто возникает необходимость в принятии решений. В строго типизированных языках для этих целей используются логические (булевские) типы данных. В классическом языке Си логический тип отсутствовал. Для принятия решений использовались некоторые соглашения относительно замены логических значений числами. В связи с тем, что эти соглашения должны обеспечивать взаимные переходы между числами и логическими значениями необходимо учитывать следующее:

- Принцип интерпретации числовых значений в качестве логических значений
- Интерпретация логических значений в качестве чисел.

В языке Си эти соглашения выглядят следующим образом:

- Любое число (целое или вещественное), отличное от нуля, может интерпретироваться в соответствующем контексте как логическое значение true, а число, равное нулю – как логическое значение false.
- Логическое значение true в контексте, требующем числовой интерпретации, воспринимается как целое число равное 1, логическое значение false – воспринимается как целое число равное 0.

При этом оказывается, что в качестве “источников” логических значений можно использовать следующие программные конструкции:

- Обычные арифметические выражения (нулевое значение такого выражения соответствует логическому значению false, а любое не нулевое значение - true).
- Отношения и сравнения на равенство.
- Выражения, использующие логические операторы, в качестве операндов которых служат арифметические выражения, отношения и сравнения на равенство.

Ближайшей нашей задачей является рассмотрение отношений и сравнений на равенство.

Как уже отмечалось ранее, отношения и сравнения на равенство используются для получения логических значений. Это обеспечивается путем оценки истинности отношений и сравнений на равенство. Причем в том случае, когда истинность имеет место, указанное выражение возвращает значение 1, в противном случае возвращается значение 0.

В таблице приводятся названия рассматриваемых операций, математические знаки операций, операторы языка Си и примеры их применения

Название операции	Математический знак	Оператор Си	Пример
меньше	$<$	`<`	`a < b`
не меньше	$\geq$	`>=`	`a >= b`
больше	$>$	`>`	`a > b`
не больше	$\leq$	`<=`	`a <= b`
равно	$=$	`==`	`a == b`
не равно	$\neq$	`!=`	`a != b`

Первые четыре оператора, приведенные в таблице, имеют одинаковый приоритет, а операторы сравнения на равенство имеют приоритет на один уровень ниже. Рассматриваемые операторы имеют более низкий приоритет по сравнению с арифметическими операторами. Пусть имеется следующий фрагмент программного кода:

```
/* . . . . . . . . . . . . . . . . . . . . . . . . . . . . */
int n  = 3;
printf(“n > 2 = %d n < -1 = %d\n”, n > 2, n < -1);
/* . . . . . . . . . . . . . . . . . . . . . . . . . . . . */
```

В этом фрагменте кода оцениваются значения отношений $n > 2$ и $n < -1$ для случая, когда значение переменной n равно 3. Нетрудно видеть, что первое отношение должно принимать значение true (истина), а второе отношение – значение false (ложь). Обратимся теперь к выводу, полученному в результате выполнения данного фрагмента, который имеет следующий вид:

```
n > 2 = 1 n < -1 = 0
```

С учетом принятых в языке Си относительно кодирования булевских значений соглашений в качестве значения отношения $n > 2$ выведено число 1, в качестве значения второго отношения – число 0.

18.1. *Логические операторы*

Логические операторы используются для повышения выразительных средств языка. В языке Си имеются три логических оператора:

- ! – логическое НЕ,
- && - логическое И,
- || - логическое ИЛИ.

Оператор ! является унарным оператором, а операторы && и || - бинарными. В качестве операндов логических операторов могут использоваться арифметические выражения, отношения и сравнения на равенство. Логические операции задаются с помощью так называемых таблиц истинности. Предположим, что имеются две логические величины a, b. Тогда таблицы истинности для рассматриваемых операторов можно представить в следующем виде:

a	b	Логическая операция			
		!a	!b	a && b	a || b
false	false	true	true	false	false
false	true	true	false	false	true
true	false	false	true	false	true
true	true	false	false	true	true

Значение результата выполнения оператора ! является обратным по отношению к его операнду. Значение, полученное в результате выполнения оператора &&, истинно только в том случае, когда истинны его оба операнда. Значение результата выполнения оператора || истинно в том случае, когда истинен хотя бы один операнд. Отметим, что бинарные операторы относятся к немногочисленной категории операторов, для которых фиксирован порядок вычисления операндов. Этот порядок выполнения состоит в следующем. Вначале всегда вычисляется левый операнд. Если после его вычисления значение выражения становится очевидным, то правый операнд не вычисляется. Например, если при вычислении выражения, содержащего оператор &&, значение левого операнда оказалось равным false, то из этого заведомо следует, что значением всего выражения будет логическое значение false. При этом необходимость в вычислении правого операнда отпадает.

18.2. *Поразрядные операторы*

Эти операторы могут быть полезными в тех случаях, когда программисту требуется непосредственно взаимодействовать с аппаратурными компонентами компьютера. В языке Си имеется шесть поразрядных операторов:

- ~ поразрядное НЕ,
- & поразрядное И,
- | поразрядное ИЛИ,
- ^ поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ,
- << сдвиг влево,
- >> сдвиг вправо.

Отметим, что поразрядные операторы применимы только к целочисленным операндам.

Краткое описание поразрядных операторов содержится в таблице, приведенной ниже.

Оператор	Название	Описание
~	Поразрядное НЕ	Все биты операнда, равные 0, устанавливаются в 1, а все биты, значения которых равны 1, устанавливаются в 0.
&	Поразрядное И	Бит результата устанавливается в 1 только в том случае, когда соответствующие биты обоих операндов установлены в 1, во всех других случаях этот бит устанавливается в 0.
|	Поразрядное ИЛИ	Бит результата устанавливается в 1 в том случае, когда хотя бы один из соответствующих битов операндов установлен в 1 и устанавливается в 0, если оба соответствующих бита, установлены в 0.
^	Поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ	Бит результата устанавливается в 1, если соответствующий бит одного и только одного операнда установлен в 1, во всех других случаях этот бит устанавливается в 0.
<<	Сдвиг влево	Сдвигает влево биты левого операнда на количество позиций, которое определяется правым операндом. Правые освобождающиеся позиции заполняются нулями.
>>	Сдвиг вправо	Сдвигает вправо биты левого операнда на количество позиций, которое определяется правым операндом. Способ заполнения освобождающихся позиций является аппаратно – зависимым.

Пусть имеется следующий фрагмент программы, содержащий выражения, использующие поразрядные операции.

```
/* ....................*/
unsigned char n = 0x2A;
unsigned char m = 0x71;
unsigned char r;

r = ~n;
printf("~n = %X\n", r);

r = ~m;
printf("~m = %X\n", r);
```

```
    r = n & m;
    printf("n & m = %X\n", r);

    r = n | m;
    printf("n | m = %X\n", r);

    r = n << 2;
    printf("n << 2  = %X\n", r);

    r = m >> 1;
    printf("m >> 1  = %X\n", r);
    /* ........................*/
```

Результаты выполнения этого фрагмента программы имели следующий вид:

```
~n = D5
~m = 8E
n & m = 20
n | m = 7B
n << 2 = A8
m >> 1 = 38
```

19. Условные выражения (оператор ?:)

Условные выражения записываются с помощью тернарного оператора ?:. Этот оператор имеет следующий формат:

выр1 ? *выр2* : *выр3*

Здесь *выр1*, *выр2* и *выр3* – выражения.

Условное выражение вычисляется следующим образом. Вначале оценивается булевское значение выражения *выр1*. Если его значением является true, то дальше вычисляется значение выражения *выр2* и значение этого выражения становится значением всего условного выражения, в противном случае вычисляется значение выражения *выр3* и его значение становится значением всего условного выражения.

В качестве примера применения условного выражения рассмотрим задачу о нахождении наибольшего значения двух вещественных чисел a и b. Ниже приведена программа, предназначенная для решения этой задачи.

```
/*                  Файл max.c          */
#include<stdio.h>
int main(void)
{
   double a, b;
   printf("a=");
   scanf("%lf", &a);
   printf("b=");
   scanf("%lf", &b);
   printf("max=%8.3f\n", (a > b ? a : b));
   return 0;
}
```

Здесь использовалось то обстоятельство, что условное выражение действительно является выражением, и оно может быть использовано в качестве аргумента в вызове функции printf().

20. Оператор запятая

Это бинарный оператор, имеющий следующий формат:

```
выр1 , выр2
```

Здесь *выр1* и *выр2* – выражения. Вначале вычисляется левый операнд. Значение этого операнда после его вычисления отбрасывается (исключается из рассмотрения). Затем вычисляется правый операнд. Вычисленное значение этого операнда становятся типом и значением всего выражения в целом. В месте, где расположена запятая, находится точка следования (sequence point). Это означает, что до перехода к вычислению выражения *выр2* все побочные эффекты, которые могут иметь место при вычислении выражения *выр1* должны быть завершены. Оператор запятая в основном применяется в инструкции **for** в ее первом и третьем выражениях (см. 1.29).

21. Оператор sizeof

Этот оператор в качестве своего результата возвращает размер памяти, занимаемый его операндом. Оператор используется в двух формах:

- sizeof(*type*)
- sizeof *expr*

В первой форме в качестве операнда используется тип, а во втором – выражение. Во втором случае выражение часто заключается в круглые скобки, хотя необходимости в этом нет.

Приведем примеры использования рассматриваемого оператора:

```
printf("sizeof 2 = %d\n", sizeof 2);
printf("sizeof(double) = %d\n", sizeof(double));
```

22. Инструкции перехода

Инструкции перехода приводят к безусловной передаче управления в другую точку программы. К этой категории инструкций в языке Си относятся следующие инструкции:

- **break**,
- **continue**,
- **goto**,
- **return**.

22.1. *Инструкция break*

Эта инструкция используется только в теле инструкций **for**, **while**, **do …while** и **switch.** Ее назначение состоит в завершении выполнения инструкций, внутри которых она расположена. При использовании инструкции **break** во вложенных циклах завершит свою работу только текущий цикл. После этого управление передается циклу, находящемуся "на один уровень выше".

22.2. *Инструкция continue*

Эта инструкция используется только в теле циклических инструкций **for**, **while** и **do**…**while**. Ее выполнение приводит к завершению очередного шага цикла.

22.3. *Инструкция goto*

Выполнение этой инструкции состоит в прерывании естественного порядка выполнения программы и безусловной передаче управления. Рассматриваемая инструкция состоит из двух частей: зарезервированного слова **goto** и имени метки. В качестве имени метки используется идентификатор. Пример инструкции **goto**:

goto error;

Чтобы эта инструкция выполнялась правильно, в программе должна встретиться инструкция, помеченная меткой error. Приведем пример такой инструкции:

```
error : printf("Ошибка ввода\n");
```

22.4. *Инструкция return*

Эта инструкция возвращает управление в вызывающую функцию. Если функция должна иметь возвращаемое значение, то в теле должна встретиться инструкция **return**, имеющая следующий формат:

```
return [выр];
```

Рассматриваемая инструкция состоит из двух структурных частей: зарезервированного слова return и необязательного выражения *выр*. Необходимость в выражении *выр* отсутствует в том случае, когда функция не возвращает значение. В качестве примера приведем определение функции, предназначенной для определения максимального значения двух чисел.

```
double max(double a, double b)
{
     return ((a > b) ? a : b);
}
```

В теле рассматриваемой функции max() находится только одна инструкция – инструкция **return**. Эта инструкция выполняет два действия. Во-первых, инструкция формирует с помощью условного выражения возвращаемый результат. Во-вторых, она прекращает вычисления в функции max() и передает управление в точку вызова функции.

23. Составная инструкция

Составная инструкция – это последовательность объявлений, определений и инструкций, заключенных в фигурные скобки. Эта инструкция используется для группирования других инструкций. Дело заключается в том, что внутри некоторых управляющих конструкций допускается использование единственной инструкции. Составную инструкцию следует применять в тех же случаях, когда в соответствии с

алгоритмом необходимо поместить несколько инструкций, а синтаксис языка разрешает использовать только одну,

24. Инструкция if else

Эта инструкция используется для программирования разветвляющихся алгоритмов. Назначение этой инструкции состоит в моделировании на языке Си двух стандартных управляющих конструкций структурного программирования, которые применяются для организации разветвляющихся алгоритмов:

- Альтернатива.

- Действие или обход.

24.1. *Полная и сокращенная формы инструкции if*

Рассматриваемая инструкция применяется в двух формах:

- Полной
- Сокращенной

В полной форме инструкция **if** .. **else** моделирует стандартную управляющую структуру, которая называется альтернатива, а в сокращенной форме – стандартную управляющую структуру действие или обход.

Рассматриваемая инструкция имеет следующий формат:

```
if(выр)
    инструкция1
[else
    инструкция2]
```

Инструкция **if** в общем случае содержит следующие компоненты:

- Зарезервированные слова **if** и **else**.
- Внутренние инструкции *инструкция1* и *инструкция2*.
- Выражение *выр*, предназначенное для принятия решения об использовании внутренних инструкций.

В формате инструкции **if**, представленном выше, в квадратные скобки заключена необязательная часть инструкции **if**. В зависимости от наличия или отсутствия этой части говорят либо о полной форме инструкции **if**, либо о сокращенной форме. В полной форме необязательная часть присутствует, а в сокращенной – нет.

Полная форма инструкции if позволяет организовать разветвление на два направления. Первое направление (true – ветвь разветвления) представлено единственной инструкцией языка Си *инструкция1*, а второе направление (false - ветвь) – единственной инструкцией *инструкция2.*

Логика работы полной инструкции **if** такова. В зависимости от значения, которое имеет выражение *выр* (true или false) будет выполняться либо внутренняя инструкция *инструкция1*, либо внутренняя инструкция *инструкция2*. Внутренняя инструкция *инструкция1* выполняется только в том случае, когда значение выражения *выр* равно true, в противном случае будет выполняться внутренняя инструкция *инструкция2*. Нетрудно видеть,

что логика работы полной инструкции **if** соответствует стандартной управляющей конструкции “альтернатива”.

В сокращенной форме инструкции **if** имеется только одна внутренняя инструкция (*инструкция1*). Работа такой формы инструкции **if** отличается от работы полной формы инструкции **if** только для случая, когда значение выражения *выр* равно false. В этом случае при использовании сокращенной формы инструкции **if** не выполняется ни каких действий. Можно сказать, что в этом случае имеет место обход действия, предусмотренного единственной инструкцией *инструкция1*.

Приведем примеры использования инструкции **if**.

Пример 1. Даны два числа a и b. Необходимо найти значение наибольшего из этих чисел.

```
/*
  В коде, приведенном ниже a, b max – переменные типа double.
  Переменные a и b содержат исходные данные, а переменная max –
  ожидаемый результат
*/
if(a > b)
    max = a;
else
    max = b;
/* Вызов функции printf()расположен в линейной части и
  выполняется сразу после завершения работы инструкции if */
printf(“Максимальное значение = %0.3f”, max);
/*              Конец фрагмента программного кода     */
```

В качестве примера применения сокращенной формы рассматриваемой инструкции рассмотрим усложненный вариант задачи, решение которой было выбрано в качестве иллюстрации применения полной инструкции if.

Пример 2. Даны три числа a, b и c. Необходимо найти значение наибольшего из чисел.

Решение. Предположим вначале, что имеется только два числа. Пусть это будут числа a и b. Такая задача уже решена. Ее решение приведено выше. Теперь эту задачу следует решать повторно, сравнивая уже значения переменных c и max. Оказывается, что такое сравнение можно выполнить с помощью сокращенной формы инструкции **if**. Ниже приведен программный код, содержащий решение рассматриваемой задачи.

```
/*
   В коде, приведенном ниже, a, b, c и max – переменные типа
   double.Переменные a, b и c содержат исходные данные, а
   переменная max – ожидаемый результат
*/
if(a > b)
     max = a;
else
     max = b;
if(c > max)
     max = c;
```

```
printf("Максимальное значение = %0.3f", max);
/*                 Конец фрагмента программного кода     */
```

24.2. *Вложенные инструкции if*

Внутри каждой из ветвей инструкции **if else** может быть расположена другая инструкция **if**. Такие инструкции называются вложенными. Следует отметить, что большая глубина вложенности может приводить к ухудшению читабельности программы. Кроме того, может появиться специальная проблема, называемая проблемой висячего else. Такая проблема имеется, например, в инструкции if, структура которой приведена ниже.

```
if(выр1)
   if(выр2)
     инструкция1
   else
     инструкция2
```

При чтении приведенной выше конструкции возникает проблема, состоящая в том, что необходимо определить к какой из двух инструкций **if** единственное зарезервированное слово **else**. Иными словами, необходимо выяснить внешняя или внутренняя инструкция if является сокращенной. Эта проблема в языке Си решается следующим образом. Компилятор всегда относит слово **else** к ближайшему слову **if**, для которого еще нет ветви **else**. Учитывая все изложенное, можно сделать вывод о том, что в приведенном примере внешняя инструкция if является сокращенной, а внутренняя – полной.

Для регулирования структуры вложенных инструкций **if** следует использовать фигурные скобки. Ниже приводится переработанный вариант использования вложенных инструкций if, приведенных выше.

```
if(выр1)
   {
    if(выр2)
      инструкция1
    }
else
     инструкция2
```

В новой инструкции **if** внешняя инструкция является полной, а внутренняя – сокращенной.

25. Инструкция switch

Инструкция **switch**, иногда называемая переключателем, предназначена для организации многовариантного разветвления.

25.1. *Синтаксис инструкции switch*

Инструкция switch может иметь сложную структуру. На самом верхнем уровне рассматриваемая инструкция состоит из двух конструктивных частей:

- Заголовок.
- Тело.

Заголовок инструкции **switch** имеет следующий формат:

```
switch(выр)
```

Здесь **switch** – зарезервированное слово, *выр* – выражение целого типа.

Телом может быть единственная инструкция языка Си, в качестве которой обычно используется составная инструкция.

Инструкции, входящие в состав тела переключателя **switch**, могут быть помеченными специальными метками. Метка отделяется от помечаемой ею инструкции двоеточием. В теле инструкции **switch** используются метки двух видов:

- label_case.
- label_default.

Метка вида label_case состоит из двух частей:

- зарезервированное слово **case.**
- Константное выражение целого типа.

Метка вида label_default состоит из одного зарезервированного слова **default**.

Выражение *выр*, входящее в состав заголовка инструкции **switch**, играет роль своеобразного селектора, выбирающего требуемую метку внутри тела переключателя. Дело заключается в том, что значение этого выражения определяет метку той инструкции, с которой должно начинаться выполнение тела переключателя.

Опишем более детально процесс выполнения инструкции **switch**.

1. Вычисляется значение выражения *выр*, входящего в состав заголовка.
2. Если значение выражения выр совпадает со значением константного выражения одной из меток **case**, то управление передается инструкции тела, которая помечена этой меткой, а затем будут выполняться последовательно все оставшиеся инструкции тела переключателя.
3. Если значение выражения *выр* не совпадает со значением константного выражения ни одной метки, но имеется инструкция, помеченная меткой **default**, то управление передается инструкции, помеченной этой меткой.
4. Если значение выражения *выр* не совпадает со значением константного выражения ни одной метки и отсутствует инструкция, помеченная меткой **default**, то выполнение инструкции **switch** на этом заканчивается и управление передается инструкции, расположенной непосредственно за инструкцией **switch**.

1.25.2. Использование инструкции switch

Как правило, тело инструкции **switch** – составная инструкция, внутренние инструкции которой помечены метками **case** и **default**. После перехода на выбранную инструкцию остальные метки не влияют на выполнение тела переключателя. Для прекращения выполнения тела переключателя следует использовать инструкции перехода (обычно инструкцию **break**). Посмотрим, к чему может привести их отсутствие.

Обратимся к программному коду, приведенному ниже, предполагая, что используемая в нем переменная n имеет тип **int**.

```
/*  Использование инструкции switch, в теле которой отсутствуют
                    инструкции перехода
     */
switch(n)
{
     case 1 : printf(".");
     case 2 : printf("..");
     case 3 : printf("...");
     case 4 : printf("....");
}
```

Если перед выполнением приведенного выше фрагмента программы значение переменной n было равно 3, то при выполнении инструкции **switch** будет выведено семь точек, т.к. будут выполнены два последних вызова функции printf().

Чтобы в приведенном фрагменте программы выполнялся только тот вызов функции printf(), которому передано управление, необходимо все помеченные инструкции разделить инструкциями **break**. В результате получим следующий программный код:

```
/*  Инструкция switch, в теле которой используются инструкции
    break                                      */
switch(n)
{
    case 1 : printf(".");
             break;
    case 2 : printf("..");
             break;
    case 3 : printf("...");
             break;
    case 4 : printf("....");
             break;
}
```

Теперь для любого значения величины n из отрезка [1, 4] будет выполняться только один вызов функции printf().

Заметим, что в ветви, соответствующей значению n == 4, использование инструкции **break** необязательно, но считается элементом хорошего стиля программирования.

26. Функциональные компоненты цикла

Алгоритм называется циклическим в том случае, когда некоторые его части могут выполняться неоднократно. В общем случае в циклическом алгоритме могут присутствовать следующие функциональные компоненты:

- Инициализация (подготовка к первому выполнению цикла, возобновление цикла).
- Проверка условия нахождения в цикле.
- Рабочая часть цикла.

- Подготовка к очередному выполнению рабочей части цикла (продвижение цикла).

Основу цикла составляют действия, многократное выполнение которых должно привести к решению поставленной задачи. Функциональный компонент цикла, реализующий такие действия, будем называть рабочей частью цикла. Например, рабочая часть цикла может содержать инструкции, обеспечивающие обработку текущей порции данных. Рабочая часть цикла является его первым функциональным компонентом.

Проверка условия нахождения в цикле предназначена для определения момента, когда работа цикла должна быть закончена. Это второй функциональный компонент, который должен присутствовать в любом цикле.

Назначением инициализации является создание корректных начальных условий для работы цикла. Например, здесь может выполняться инициализация счетчика – переменной целого типа, которая используется для управления работой цикла. Это третий обязательный функциональный компонент цикла.

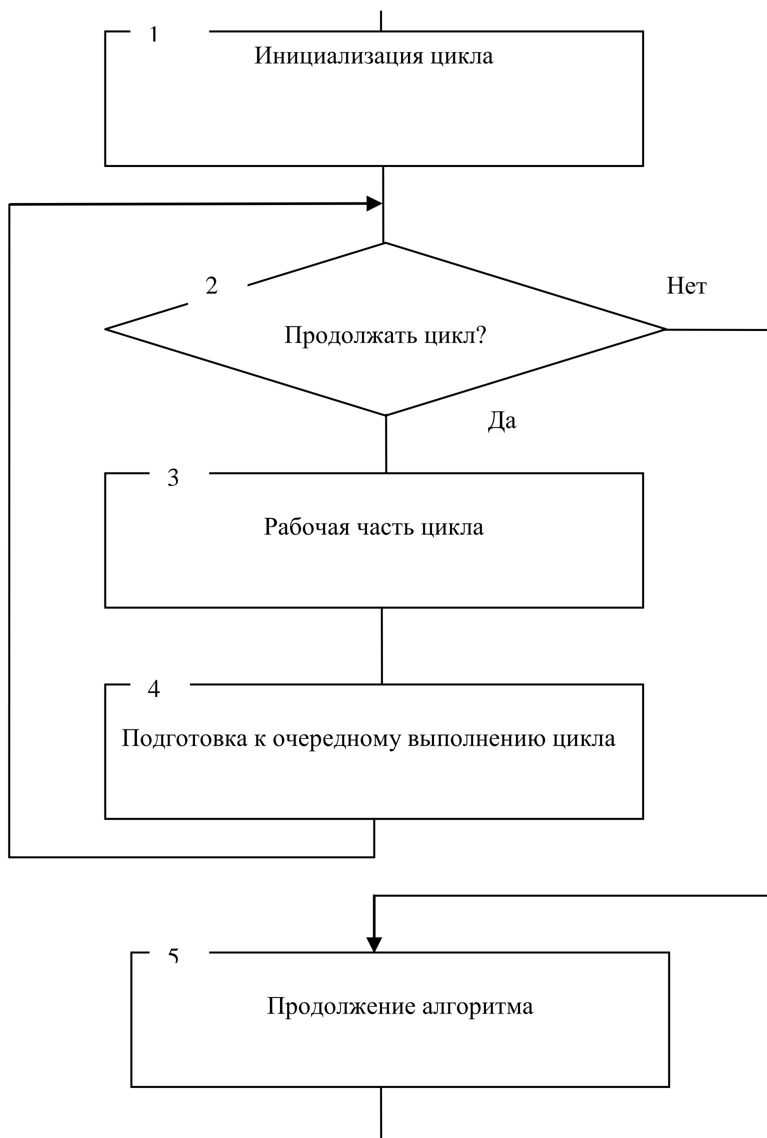
Наконец, последний из функциональных компонентов цикла предназначен для продвижения цикла. Здесь, например, могут располагаться инструкции, обеспечивающие получение новой порции обрабатываемых данных. Кроме того, здесь может изменяться значение счетчика, управляющего работой цикла.

Все компоненты, кроме инициализации могут выполняться неоднократно, образуя так называемый контур цикла. Однократный “проход” цикла будем называть шагом. В качестве синонимов используются следующий понятия: итерация и виток цикла.

Разработку цикла следует начинать с его рабочей части. Необходимо выяснить, что же должен делать цикл? Ответ на этот вопрос должен носить общий характер. Он должен быть пригоден для произвольного шага цикла.

Остальные компоненты цикла выполняют служебные функции. К их разработке следует переходить только после того, как станет ясным, какие действия необходимо выполнять в рабочей части цикла.

В конкретной реализации цикла взаимное расположение частей цикла, составляющих его контур, может быть различным. Для иллюстрации назначения отдельных функциональных частей цикла рассмотрим один из вариантов организации циклического алгоритма, который приведен на рисунке, представленном ниже. Особенностью этого варианта организации цикла является то обстоятельство, что выполнение первого шага цикла начинается с проверки нахождения в цикле. Такие циклы называются циклами с предусловием.



Рассмотрим пример организации циклического алгоритма. Пусть необходимо вычислить сумму квадратов первых “n” натуральных чисел. Введем следующие обозначения: i – очередное натуральное число, s – искомая сумма. Для нахождения суммы s необходимо выполнить “n” сложений. Действительно, $s = 1^2 + 2^2 + .. + i^2 +.. + n^2$. Указанные действия можно осуществить за “n” шагов цикла, используя принцип накопления суммы. Для этого под s следует понимать текущее значение суммы, которое только после завершения работы цикла оказывается равным искомому значению суммы квадратов. Используя такой подход, в рабочей части цикла (символ 3) следует поместить инструкцию – присваивание языка Си следующего вида:

```
s += i * i; /* s = s + i * i ;  */
```

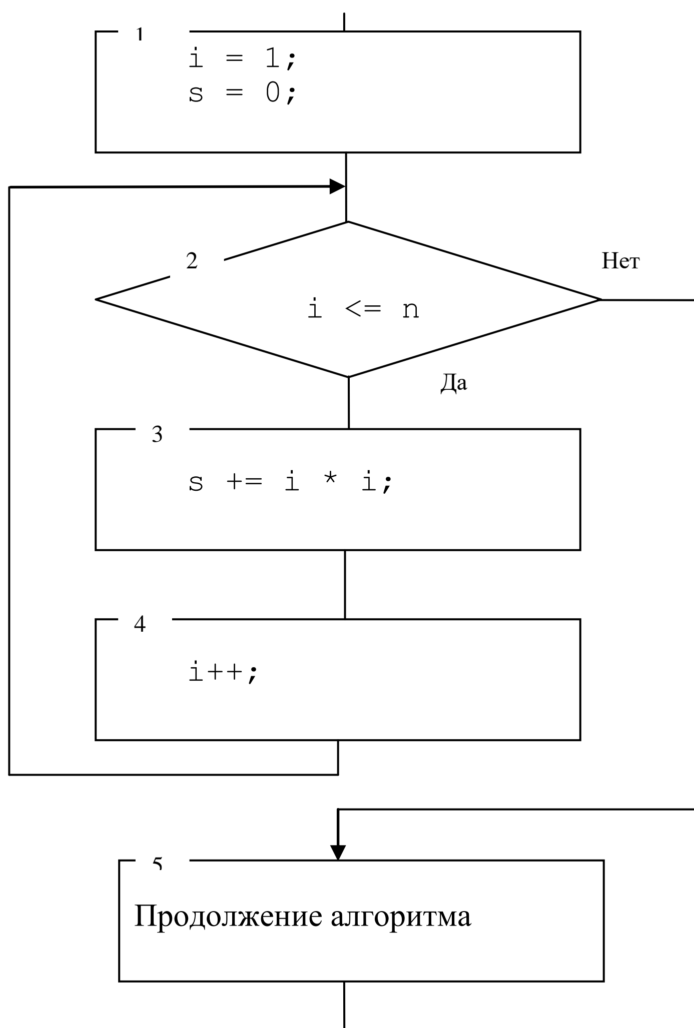
Для правильной работы цикла переменные i и s необходимо инициализировать следующим образом (символ 1):

```
i = 1;
s = 0;
```

В символе 4 для правильного продвижения цикла следует поместить инструкцию инкремента переменной i:

```
i++;
```

Для обеспечения n кратного выполнения цикла в символе 2 следует проверять отношение: `i <= n` .



27. Арифметические и итерационные циклы

Циклические алгоритмы можно разделить на две категории:

1. Арифметические циклы.
2. Итерационные циклы.

К арифметическим циклам относятся циклы, количество повторений которых заранее известно. Более точно, количество повторений таких циклов либо содержится в условии задачи, либо может быть вычислено до начала выполнения цикла.

Итерационные циклы – это циклы с неизвестным количеством повторений тела цикла. Более точно, количество повторений тела итерационного цикла становится известным только после завершения работы цикла.

Приведем примеры задач, решения которых требует организации соответственно арифметических и итерационных циклов.

27.1. *Задачи, приводящие к арифметическим циклам*

Задача 1. Табулировать функцию $y = x^2$ в n равноотстоящих точках, начиная от x = xn вплоть до x = xk.

Количество повторений цикла в этой задаче содержится в условии задачи. Оно должно быть равно значению величины.n. Дело в том, что при решении данной задачи следует n раз выполнить однотипные вычисления, для которых придется организовать цикл. Незнание того, как это выполнить не мешает интуитивно предполагать, что для решения потребуется именно арифметический цикл.

Задача 2. Табулировать функцию $y = x^2$ в равноотстоящих точках, начиная от x = xn вплоть до x = xk с шагом dx.

Эту задачу можно рассматривать как разновидность предыдущей задачи. Это обусловлено тем обстоятельством, что количество повторений цикла n может быть определено до начала работы цикла, если округлить значение, полученное в результате вычисления следующего выражения:

(xk – xn) / dx + 1.

27.2. *Задачи, приводящие к итерационным циклам*

Задача 1. Обработка последовательности произвольной длины.

Это значительный класс задач. Примером такой задачи может быть задача о вычислении суммы произвольного количества чисел, вводимых с клавиатуры. Для прекращения обработки могут использоваться следующие способы:

- Ввод так называемого стоп-кода.
- Ввод признака конца файла.

Важным элементом здесь является то обстоятельство, что решение о прекращении обработки (вычисление суммы) оперативно принимает пользователь программы во время ее выполнения.

Задача 2. Вычисление значения корня квадратного по формуле Герона.

Эта задача относится к обширному классу задач вычислительной математики, в котором используются приближенные вычисления.

28. Циклические управляющие инструкции

В языке Си имеются три циклические управляющие инструкции:

- **`for`**
- **`while`**
- **`do while`**

Остановимся вначале на общих вопросах организации циклов, управляемых рассматриваемыми инструкциями.

В общем случае каждый из таких циклов состоит из трех структурных частей:

- инструкций инициализации,
- собственно управляющей инструкции (**for**, **while** или **do while**),

- тела цикла – единственной инструкции, которая находится под управлением циклической инструкции.

Ограничение на количество управляемых инструкций является несущественным. Дело в том, что при необходимости размещения в теле цикла несколько инструкции, эти инструкции всегда могут быть объединены в составную инструкцию.

Инструкции инициализации, если они необходимы, должны всегда располагаться до циклической инструкции. Инструкция, составляющая тело цикла, при использовании циклических инструкций **for** и **while** должна располагаться непосредственно за циклической инструкцией, а при использовании циклической инструкции **do while** - должна находиться внутри циклической инструкции.

Каждая из циклических инструкций содержит управляющее условие, определяющее момент прекращения управления телом цикла. Прекращение управления при использовании каждой из трех циклических инструкций соответствует получению булевского значения false при оценке управляющего условия.

Циклы, управляемые инструкциями **for** и **while**, являются циклами с предусловием, а цикл, управляемый инструкцией **do while** – циклом с постусловием. В циклах с предусловием управляющее условие оценивается до выполнения тела цикла, а в циклах с постусловием – после выполнения тела цикла. Это означает, что при использовании первых двух разновидностей циклов возможна ситуация, когда тело цикла не выполнится ни разу. При использовании цикла, управляемого инструкцией **do while**, тело цикла должно выполниться хотя бы один раз.

Наиболее простую структуру имеют циклические инструкции **while** и **do while.** Их содержательная часть содержит управляющее условие, которое заключается в круглые скобки. Логика работы таких циклов чрезвычайно проста. Необходимо на каждом шаге цикла оценивать значение управляющего условия, а затем выполнять тело цикла или завершать его выполнение.

Циклическая инструкция for имеет более сложную структуру. Содержательная часть этой инструкции состоит из трёх выражений, заключенных в круглые скобки. Первое из этих выражений предназначено для записи элементов инициализации, второе выражение выполняет те функции, что и управляющее выражение в инструкциях **while** и **do while**. Последнее выражение предназначено для записи действий по “продвижению” цикла. Указанная особенность цикла **for** делает целесообразным использование этого вида циклов, когда все элементы управления циклом могут быть реализованы с помощью инструкции for. Отметим, что часто такая возможность имеется при организации арифметических циклов. В связи с этим цикл, управляемый инструкцией **for**, нашел наибольшее применение при программировании арифметических циклов. Это обусловило то положение, что работа цикла, управляемого инструкцией **for**, поясняется на примере программирования арифметического

цикла. Здесь же рассматриваются общие принципы организации арифметических циклов. Работа циклов, управляемых циклическими инструкциями **while** и **do while**, поясняется на примерах организации итерационных циклов.

29. Цикл, управляемый инструкцией for

Как отмечалось выше, в общем случае цикл, управляемый инструкцией **for,** может состоять из трех частей. К первой необязательной части цикла могут относиться элементы инициализации цикла. Вторую часть составляет сама инструкция **for**, а последнюю часть, часто называемую телом, – любая другая единственная инструкция языка Си. Говорят, что инструкция **for** управляет работой тела цикла. В случае необходимости управлять в цикле работой нескольких инструкций их следует объединять в составную инструкцию.

В общем случае цикл, построенный на основе инструкции **for,** имеет следующий формат:

```
[инструкции_инициализации]
for([выражение1];[выражение2];[выражение3])
     инструкция
продолжение_программы
```

Здесь *инструкции_инициализации* – необязательная группа инструкций, используемая для инициализации цикла, **for** – зарезервированное слово, *выражение1*, *выражение2* и *выражение3* – необязательные выражения, а *инструкция* – единственная инструкция языка Си.

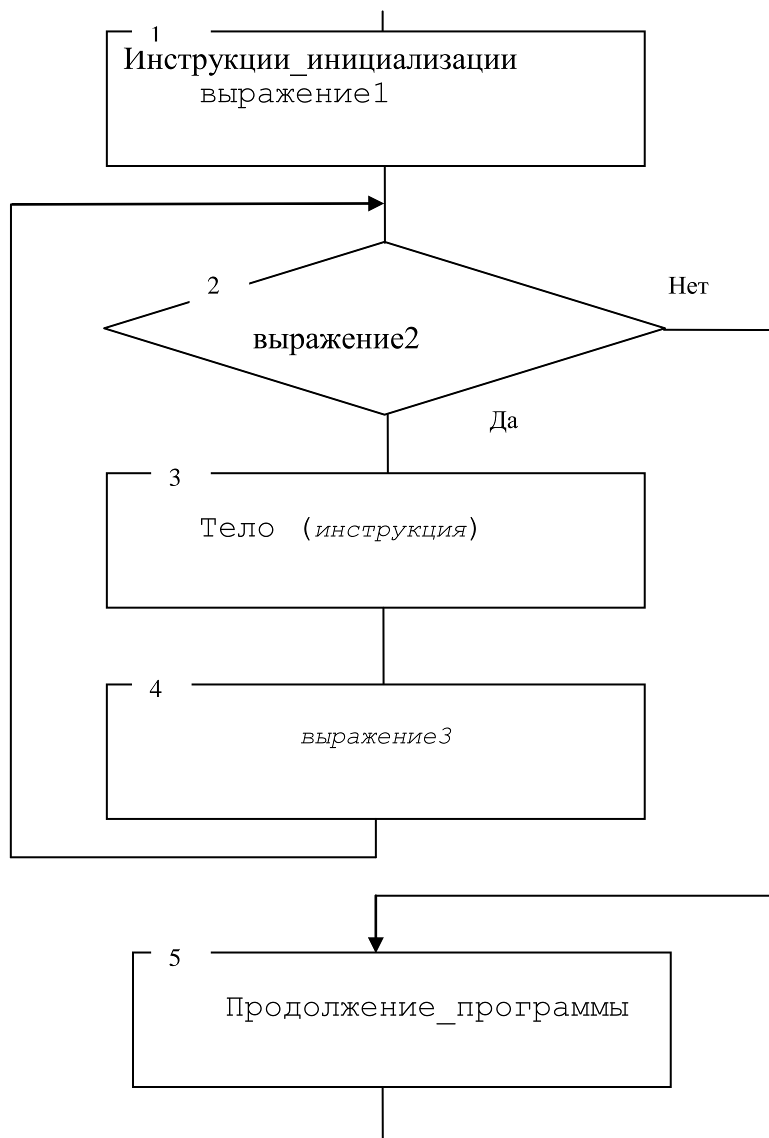
Выражения *выражение1*, *выражение2* и *выражение3*, входящие в состав инструкции **for**, имеют следующее функциональное назначение:

- *выражение1* – инициализация цикла,
- *выражение2* – проверка нахождения в цикле,
- *выражение3* – продвижение цикла.

Каждое из трех выражений является полным. Поэтому любой побочный эффект, который имеет место при вычислении каждого из этих выражений, должен завершиться до начала вычисления следующего выражения.

Использование внешних *инструкции_инициализации* может быть обусловлено соображениями повышения читабельности программного кода.

Работу рассматриваемого цикла удобно иллюстрировать следующей схемой алгоритма.



Следует отметить, что эта схема совпадает с функциональной схемой цикла, которая приводилась в пункте 1.26. Порядок работы цикла, управляемого инструкцией **for**, состоит в следующем:

- Выполняются *инструкции_инициализации* и вычисляется значение выражения *выражение1* (символ 1 схемы алгоритма).
- Оценивается значение выражения *выражение2* (символ 2). Если значение этого выражения равно true, то выполняется тело цикла (символ 3). В противном случае управление передается инструкции, непосредственно следующей за телом цикла (символ 5).
- После выполнения тела цикла вычисляется выражение *выражение3* (символ 4). После этого вновь вычисляется выражение *выражение2*.

Рекомендуется применять цикл, управляемый инструкцией **for** в простейших случаях, когда все элементы управления циклом можно сосредоточить внутри инструкции **for**.

Перейдем к рассмотрению примера организации цикла, управляемого циклической инструкцией **for**. Приведем решение типичной задачи,

требующей организации арифметического цикла. Вначале рассмотрим принципы организации арифметических циклов.

Принципы организации арифметических циклов.

Общим при организации арифметических циклов является использование специальной управляющей переменной целого типа. Такую переменную часто называют счетчиком. В качестве синонимов используются следующие термины: параметр цикла и индекс цикла.

В некоторых задачах в постановке задачи уже имеется переменная, которую можно использовать в качестве счетчика. Например, это имеет место в той задаче, которая будет использована для иллюстрации работы цикла, управляемого инструкцией **for**. В других задачах такую переменную необходимо вводить специально.

Очень важно придерживаться следующего положения: все значения, которые будет принимать управляющая переменная, должны определяться инструкцией **for**. Это означает, что эта инструкция должна определять для этой переменной:

- начальное значение,
- конечное значение,
- порядок изменения переменной в процессе выполнения цикла.

Заметим, что в инструкции **for** имеются средства, позволяющие полностью (в простейших случаях) контролировать использование управляющей переменной в цикле. Действительно, с помощью выражения *выражение1* можно задать начальное значение управляющей переменной, с помощью выражения *выражение2* можно задать условие, сравнивающее текущее значение управляющей переменной с ее последним значением, а с помощью выражения *выражение3* можно задать закон изменения управляющей переменной в процессе работы цикла.

Некоторые авторы книг по программированию считают нежелательным размещение в теле цикла, управляемого инструкцией **for**, дополнительных элементов управления работой цикла. С другой стороны не рекомендуется перегружать инструкцию **for** операциями, непосредственно не связанными с управлением счетчиком.

Перейдем теперь к рассмотрению конкретного примера.

Постановка задачи. Вычислить значение величины “y”, заданной следующим образом

$$y = \sum_{i=2}^{n} \frac{i}{i + x}$$

Решение. В настоящей задаче следует выполнить сложение n слагаемых, каждое из которых полностью определяется значением двух переменных: i (переменная суммирования) и x. В связи с тем обстоятельством, что количество слагаемых в вычисляемой сумме до начала работы цикла известно (оно равно n - 1), задача может быть решена путем организации арифметического цикла. Удобно ее решать с применение цикла, управляемого инструкцией **for**. В качестве управляющей переменной

(счетчика) удобно использовать переменную суммирования i. В этом цикле необходимо значения переменной i последовательно увеличивать от значения i == 2 до значения i == n. Инициализацию переменной суммирования (i) и сравнение текущего значения этой переменной с ее наибольшим значением (n) можно поручить инструкции **for**. Эта же инструкция может выполнять на каждом шаге цикла увеличение переменной суммирования на 1. Сложнее дело обстоит с реализацией процесса накопления суммы. В теле цикла необходимо вычислять очередное слагаемое по формуле i / (i + x) и добавлять его к уже накопленному значению суммы. Если y – имя переменной, выступающей в роли накопителя суммы, то для накопления суммы необходимо в теле цикла написать следующую инструкцию

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

Переменную – накопитель суммы “y” необходимо до входа в цикл обнулить. Это требуется для того, чтобы при первом выполнении тела цикла значение суммы было равно первому слагаемому.

Изложенные соображения приводят к следующей программе.

```
/* Файл summa.c */.
#include<stdio.h>
#include<conio.h>

int main(void)
{
   clrcsr();
   int i, n;
   double x, y;

   printf(“n=”)
   scanf(“%d”, &n);
   printf(“x=”);
   scanf(“%lf”, &x);

   y = 0;
   for(i = 2; i <= n; i++)
     y += i / (i + x);

   printf(“y=%0.4\n”, y);
   getch();
   return 0;
}
```

Замечание. Стандарт C99 разрешает определять переменные внутри инструкции **for**. Область действия такой переменной будет ограничена телом инструкции **for**. С целью продемонстрировать возможность свободного размещений определений переменных в теле функции и допустимость определения переменных внутри инструкции **for** приведем модифицированный вариант решения задачи.

```
/* Файл summa.c */.
#include<stdio.h>
#include<conio.h>
```

```
int main(void)
{
    clrcsr();
    int n;
    printf("n=")
    scanf("%d", &n);
    double x;
    printf("x=");
    scanf("%lf", &x);

    double y = 0;
    for(int i = 2; i <= n; i++)
      y += i / (i + x);

    printf("y=%0.4g\n", y);
    getch();
    return 0;
}
```

30. Цикл, управляемый инструкцией while

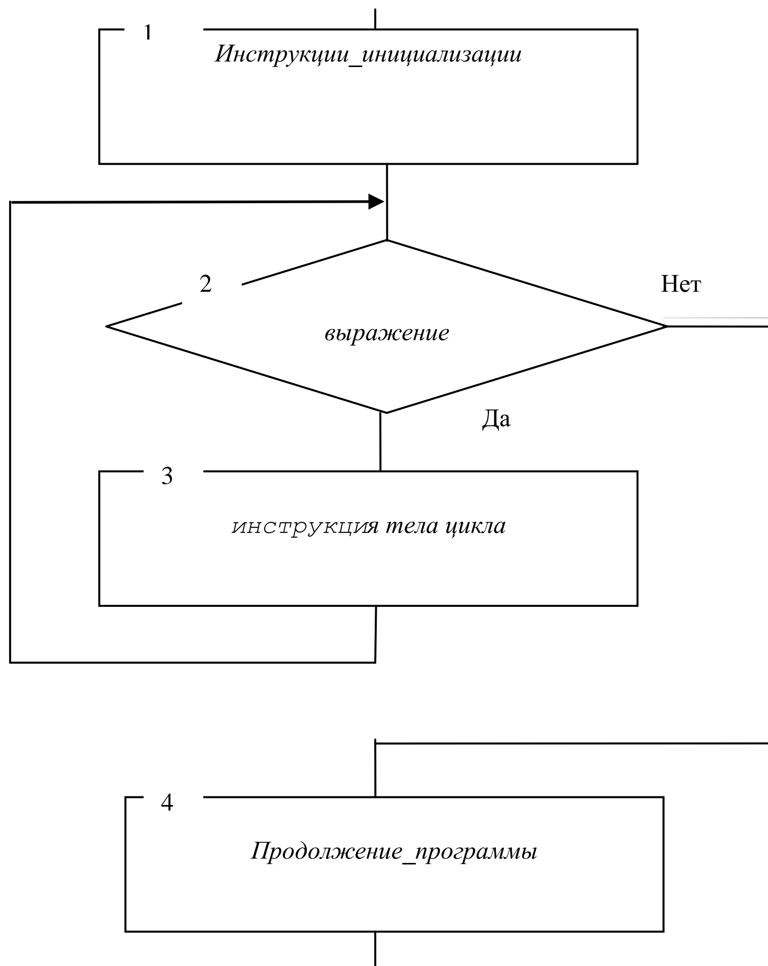
Вначале рассмотрим организацию цикла, управляемого этой инструкцией, а затем приведем ряд примеров.

В общем случае цикл, управляемый инструкцией **`while`**, может состоять из трех частей. К первой части цикла относятся элементы его инициализации. Вторую часть составляет сама инструкция **`while`**, а последнюю часть, часто называемую телом, – любая инструкция языка Си. Говорят, что инструкция **`while`** управляет работой тела цикла. В случае, когда в теле цикла должны находиться несколько инструкций, их следует объединить в составную инструкцию. Таким образом, в общем случае цикл, построенный на основе инструкции **`while`**, имеет следующий формат:

```
Инструкции_инициализации
while(выражение)          /* Инструкция while      */
   инструкция          /* Единственная инструкция */
продолжение_программы
```

Заключенное в круглые скобки *выражение* относится к категории полных выражений (оно не является подвыражением другого выражения). Поэтому к моменту начала выполнения тела цикла все побочные эффекты при его вычислении будут завершены.

Работу цикла удобно пояснить фрагментом схемы алгоритма, приведенным ниже.



Тело цикла выполняется пока остается истинным `выражение`, проверяемое в инструкции **`while`**. Легко убедиться в том, что цикл, управляемый инструкцией **`while`**, относится к категории циклов с предусловием.

Приведем пример применения цикла, управляемого инструкцией **`while`**.

Постановка задачи. Последовательность положительных чисел произвольной длины вводится с клавиатуры. Вычислить сумму вводимых чисел.

Решение. Для решения этой задачи требуется организация цикла, в теле которого следует выполнять два действия:

- Вводить очередное число `x`,
- Увеличивать текущее значение суммы `summa` на величину введенного числа.

Для прекращения циклического процесса можно воспользоваться тем обстоятельством, что по условию задачи вводимые числа должны быть положительными. Можно предложить пользователю для окончания процесса вычислений ввести любое отрицательное число.

Ниже приводится программа, реализующая требуемые вычисления. В программе использованы следующие локальные переменные: `summa` - искомая сумма, x – очередное введенное число. В комментариях к программе определены функциональные компоненты цикла.

```
//Файл summa2.c
#include<stdio.h>

int main(void)
{
  double x, summa = 0;

     //          Инициализация цикла
    printf(“Введите положительное число (для завершения “
        “вычислений введите отрицательное число)”);
     scanf(“%lf”, &x);

    while(x > 0) //Инструкция while (Заголовок цикла)
    {             // начало тела цикла
      summa += x;
      printf(“Введите положительное число(для завершения“
           “вычислений введите отрицательное число)”);
      scanf(“%lf”, &x);
  }          // Конец тела цикла

  printf(“summa=%10.3g\n”, summa);
  return 0;
}
```

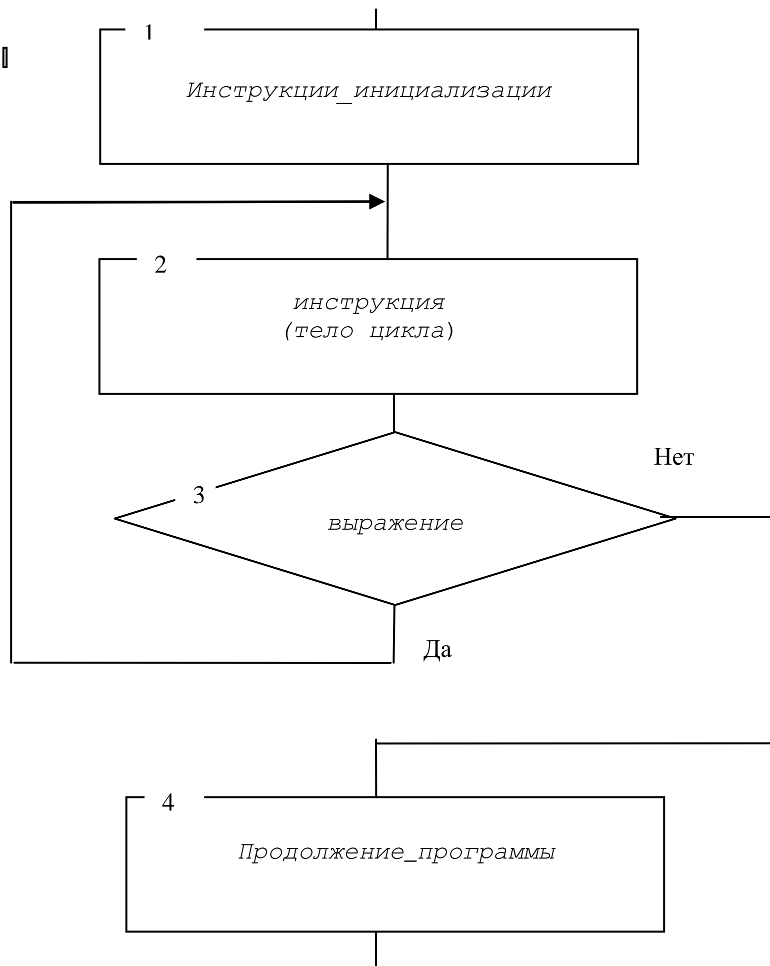
31. Цикл, управляемый инструкцией do … while

В отличие от ранее рассмотренных двух циклических управляющих инструкций цикл, управляемый инструкцией **do while**, относится к категории циклов с постусловием. Это обусловлено тем обстоятельством, что проверяемое условие расположено после тела цикла. В связи с этим тело такого цикла должно выполниться хотя бы один раз. Такой цикл целесообразно использовать в тех случаях, когда критерий продолжения (окончания) работы цикла формируется в теле цикла.

Цикл, управляемый инструкцией do while, имеет следующий формат:

```
Инструкции_инициализации
do                    /* Начало инструкции while             */
    инструкция        /* Единственная инструкция (тело цикла)*/
while(выражение);     /* Конец инструкции while              */
продолжение_программы
```

Работу цикла удобно пояснить схемой алгоритма, приведенной ниже.



Компонент `инструкция`, составляющая тело цикла (символ 2 схемы алгоритма), многократно выполняется в процессе работы цикла, пока логическая оценка компонента `выражение` (символ 3) дает значение, равное `true`.

Рассмотрим пример задачи, решение которой приводит к целесообразности применения цикла, управляемого инструкцией **`do while`**.

Постановка задачи.

Последовательность целых чисел a_i (i = 1,2, …) формируется в соответствии со следующим правилом:

$$a_i = \sum_{j=1}^{i} j^2$$

Требуется определить номер и значение первого элемента последовательности, который удовлетворяет условию $a_i > n$, где “n” – произвольное целое число.

Решение

С целью уяснения задачи получим вручную решение для конкретного значения величины “n”. Пусть значение величины `n == 25`. Получим следующую последовательность чисел: a_1 `== 1`, a_2 `== 5`, a_3 `== 14`, a_4 `== 30`. На этом ручные вычисления элементов последовательности чисел можно прекратить. Дело в том, что уже получен элемент

последовательности, удовлетворяющий заданному условию ($a_i > n$). Таким образом, искомый элемент последовательности имеет номер, равный 4 и значение, равное 30.

Следует отметить, что рассматриваемая задача всегда имеет решение, т.к. элементы a_i образуют возрастающую последовательность.

Решение данной задачи требует организации итерационного цикла, т.к. количество повторений цикла заранее неизвестно. Здесь удобно воспользоваться циклом, управляемым инструкцией **`do while`**. Дело в том, что разрабатываемый цикл должен выполниться хотя бы один раз.

Обозначим через `i` – номер очередного вычисляемого элемента последовательности, а через `current_item` – значение этого элемента. Для вычисления `current_item` следует использовать принцип накопления суммы. Желательно построить цикл таким образом, чтобы после окончания его работы значение переменной `i` совпадало с номером искомого элемента. Это можно обеспечить в том случае, когда продвижение цикла, обеспечиваемое инструкцией `i++;` будет расположено до рабочей части цикла, представленной инструкцией `current_item += i * i;`.

Программа, разработанная с учетом изложенных соображений, приведена ниже.

```
#include<stdio.h>
int main(void)
{
    int i, n;
    long current_item;
    printf("n=");
    scanf("%d", &n);

    i = 0;
    current_item = 0;
    do
    {
       i++;
       current_item += i * i;
    }while(current_item <= n);

    printf("Элемент последовательности с номером = %d"
            " и значением, равным %ld является первым"
            " элементом, удовлетворяющим заданному"
            " условию\n", i, current_item);
    getch();
    return 0;
}
```

32. Цикл с выходом

В этой разновидности цикла условие выхода проверяется внутри тела цикла, а не в его начале или конце. Такой вид цикла целесообразно

применять в том случае, когда проверка условия в начале или конце приводят к ситуации, которую иногда называют “полтора цикла”. Элементы такой ситуации имеют место в программе, приведенной в пункте 1.31. В этой программе имеются вызовы функций `printf()` и `scanf()` до входа в цикл, затем такие же вызовы этих функций находятся в конце тела цикла. Такое дублирование кода может привести к ошибкам при модификации текста программы в процессе ее эксплуатации. При модификации текста программист может забыть о необходимости согласованного изменения двух идентичных фрагментов кода, что и может оказаться источником ошибок.

Циклы с выходом реализованы только в Visual Basic. В языке Си такие циклы можно построить, используя циклические инструкции **for** и **while** и инструкцию **break**. Для этого достаточно организовать “бесконечный” цикл, В теле этого цикла расположить инструкцию **if**, которая совместно с инструкцией **break** позволила бы в требуемый момент завершить выполнение цикла.

Модифицируем программу, приведенной в пункте 1.31, заменив использованный в ней цикл циклом с выходом.

```
/* Файл summa2_2.c */
/* Модернизированный вариант решения задачи, приведенной в
   пункте 1.31. В новом варианте используется цикл с выходом
    */
   #include<stdio.h>
   int main(void)
   {
      double x, summa = 0;
      while(1)
      {
         printf("Введите положительное число (для завершения "
                   "вычислений введите отрицательное число)");
         scanf("%lf", &x);
         if(x <= 0)
             break;
         summa += x;
      }

      printf("summa=%10.3f\n", summa);
      return 0;
   }
```

33. Вложенные циклы

Вложенным циклом называют циклическую конструкцию, состоящую из нескольких циклов, в которой один из циклов в своем теле содержит один или несколько других циклов.

Цикл, в теле которого находится один или несколько других циклов называется внешним. Цикл, находящийся в теле другого цикла, называется внутренним.

Рассмотрим пример, приводящий к необходимости организации вложенных циклов.

Постановка задачи.

Вычислить значение суммы следующего вида

$$S = \sum_{k=1}^{n} k*(k+1)*(k+2)*\ldots*k^2$$

Решение

Для упрощения процесса разработки алгоритма, который необходимо организовать, перепишем исходное выражение для вычисления суммы в следующем виде:

$$s = \sum_{k=1}^{n} a_k, \text{ где}$$

$$a_k = \prod_{j=k}^{k*k} j$$

Из соотношений, приведенных выше, следует, что для вычисления искомой суммы s необходимо организовать два циклических вычислительных процесса. Первый из них предназначен для накопления суммы `s`, а второй – для вычисления очередного слагаемого a_k этой суммы.

Первый вычислительный процесс является основным. Второй вычислительный процесс должен определять значение очередного слагаемого суммы, которая накапливается с помощью первого вычислительного процесса. Очередное слагаемое должно вычисляется с помощью накопления произведения. Второй циклический процесс должен возобновлять свою работу на каждом шаге первого циклического процесса.

Из соображений, изложенных выше, следует, что для реализации рассматриваемых циклических процессов необходимо организовать вложенные циклы. Программа, предназначенная для решения рассматриваемой задачи, приведена ниже.

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

int main(void)
{
  int  k, j, n;
  long s, p;
  printf("n=");
  scanf("%d",&n);

  s = 0;                   /* Инициализация внешнего цикла */
  for(k = 1; k <= n; k++)  /* Заголовок внешнего цикла     */
  {                        /* Начало тела внешнего цикла   */
    p = 1;                 /* Инициализация внутреннего цикла*/
    for(j = k; j <= k * k; j++)/*Заголовок внутреннего цикла*/
```

```
      p *= j;                   /*Тело внутреннего цикла*/
    s += p;
  }                             /*Конец тела внешнего цикла*/

  printf("Сумма = %d\n", s)
  getch();
  return  0;
}
```

34. Понятие о варианте и инварианте цикла

При разработке циклов в последнее время все чаще используются понятия варианта и инварианта цикла. Основным из этих двух понятий является понятие инварианта цикла. Понятие варианта цикла используется реже: его применяют совместно с инвариантом цикла.

Вариант цикла — неотрицательное выражение, результат вычисления которого уменьшается на каждом шаге цикла.

Инвариант цикла — это логическое высказывание (утверждение), истинность которого должна быть обеспечена в следующих точках цикла:

- вначале работы цикла (после выполнения его инициализации),
- в конце каждого шага (перед очередным тестированием условия, проверяемого в цикле).

Выполнение этих двух условий должно гарантировать выполнение инварианта после окончания работы цикла.

Вариант и инвариант цикла принято оформлять в виде комментариев в тексте программы, предшествующем циклу.

Назначение рассматриваемых понятий является:

- Помочь программисту организовать цикл.
- Повысить читабельность программного кода.
- Помочь программисту убедиться в правильности работы цикла.

Пример 1. Вычисление суммы квадратов натуральных чисел.

Постановка задачи. Вычислить сумму квадратов первых `n` напуральных чисел.

Алгоритм решения этой задачи приведен в п. `1. 26` первой части пособия. Напишем вариант и инвариант цикла для программы, выполняющей необходимые вычисления.

Вариантом цикла в рассматриваемом примере может служить выражение `n - i + 1`, а инвариантом цикла — утверждение: просуммированы квадраты всех натуральных чисел из диапазона `[1, i - 1]`. На каждом шаге цикла его вариант цикла численно равен количеству чисел, квадраты которых следует суммировать, а инвариант содержит оценку диапазона чисел, для которых такое суммирование уже выполнено.

Напишем программу, снабдив ее комментариями, содержащими вариант и инвариант цикла. Кроме того, в программе помощью комментариев будут отмечены точки, в которых должна иметь место истинность инварианта цикла. При использовании для организации цикла инструкции **`for`** такие

точки должны находиться внутри заголовка цикла. Первая из точек должна находиться после выражения инициализации, а вторая после выражения, определяющего продвижение цикла.

```
#include<stdio.h>
int main(void)
{
   int i, n, s;
   printf("n=");
   scanf("%d", &n);

   /*  Вариант: n – i + 1 */
   /*
       Инвариант цикла: Просуммированы все числа из диапазона
       [1, i - 1]
   */

   s = 0;
   for(i = 1      /* Инвариант истинен */; i <= n;
            i++ /* Инвариант истинен */
       )
     s += i * i;  /* Инвариант ложен */

   /* Можно сделать вывод, что инвариант здесь истинен */
   printf("Сумма квадратов =%d", s);
   getch();
  return 0;
}
```

Проанализируем работу цикла, воспользовавшись понятиями варианта и инварианта. Для определенности будем полагать, что переменная "`n`" после выполнения операции ввода имеет значение, равное 3.

После инициализации цикла искомая сумма, представленная переменной `s`, обнуляется, а счетчик цикла (переменная `i`) оказывается равным 1. Выбранные инициализирующие значения переменных `i` и `s` обеспечивают истинность инварианта после выполнения выражения инициализации (`i = 1`) инструкции **`for`**. Действительно, для `i == 1` диапазон чисел (`[1, i - 1]`), квадраты которых просуммированы, оказывается пустым. То положение, что сумма `s` пустого диапазона равна нулю, обеспечивает истинность инварианта цикла в рассматриваемой точке цикла. Вариант цикла (`n – i + 1`) в этой точке оказывается равным 3.

На каждом шаге цикла его вариант уменьшается на 1 и на 1 увеличивается диапазон просуммированных квадратов чисел. После окончания работы цикла параметр цикла `i == n + 1`. С учетом этого вариант цикла в этой точке будет равен нулю, истинность инварианта цикла будет свидетельствовать о том, что все квадраты чисел из заданного по условию задачи диапазона [1, 3] просуммированы.

Предположим теперь, что при программировании цикла была допущена ошибка, состоящая в том, что в заголовке цикла (инструкции for) в качестве

логического выражения вместо правильного отношения i <= n было написано i < n. В этом случае после окончания работы цикла параметр цикла окажется равным n. С учетом этого вариант цикла в этой точке будет равен единице, истинность инварианта цикла будет свидетельствовать о том, что просуммированы квадраты чисел только из диапазона [1, 2]. Это позволяет сделать вывод о том, что квадрат одного числа остался не учтенным. Это позволит устранить допущенную ошибку в организации цикла.

Пример 2. Возведение в целочисленную степень.

Постановка задачи. Пусть требуется вычислить значение выражения a^n, не прибегая к использованию стандартной функции pow(). Будем считать основание степени a вещественным, а показатель n – неотрицательным целым числом.

Рассмотрим три различных варианта рещения рассматриваемой задачи. Каждый из вариантов решения будет иметь свой инвариант цикла. Дадим вначале общую характеристику предлагаемых решений.

В двух первых способах решение строится с использованием арифметического цикла, а в третьем способе для этих целей используется итерационный цикл.

Первый вариант решения. Наиболее очевидное решение может быть получено с учетом следующего соотношения

$$a^n = \prod_{i=1}^{n} a$$

В этом случае возведение в степень может быть заменено многократным умножением на основание степени. С этой целью введем переменную p, с помощью которой будем накапливать произведение. Ниже приводится фрагмент программы, выполняющей необходимые вычисления. После завершения цикла переменная p должна содержать искомый результат вычислений.

```
// Инвариант p == a^i
p = 1.0;
for(int i = 0 /* Инв */; i != n; i++ /* Инв */)
   p *= a;
```

Второй вариант решения. Для нахождения искомого результата воспользуемся следующей идеей. Заменим вычисление исходного выражения a^n многократным (циклическим) вычислением выражения p * a^k. В цикле необходимо уменьшать значение показателя k на 1 и увеличивать значение множителя p. Для обеспечения корректности такой замены достаточно при организации цикла потребовать выполнения следующего инварианта цикла: p * == a^n. Цикл должен закончить свою работу в момент, когда переменная k станет равной нулю. Тогда переменная p будет содержать искомый результат вычислений. Ниже приводится фрагмент программы, реализующий вычисление

```
// Инвариант p *     == a^n  , p – вычисленная часть степени
p = 1.0;
```

```
int k = n;
// Инв.
while(k != 0)
{
   p *= a;
   k--;
   // Инв.
}
```

Третий вариант решения.

```
// Инвариант p * b == a^n
p = 1.0;
int k = n;
double b = a;
while(k != 0)
{
   if(k % 2 == 1)
   {
      p *= b;
      k--;
   }else
   {
      b *= b;
      k /= 2;
   }
}
```

35. Объявления и определения (расширенное рассмотрение)

Объявления и определения уже обсуждались в первой части настоящего пособия в пункте 1.12. Рассмотрение этого вопроса ограничивалось наиболее простыми вариантами их применений, в которых объявлялись простые переменные встроенных типов. Во второй части пособия должны изучаться более сложные программные элементы (функции, указатели, массивы). В связи с этим представляется целесообразным вновь вернуться к рассмотрению этого вопроса.

В общем случае объявление программного элемента состоит из двух структурных частей:

- последовательности спецификаторов.
- списка описателей; элементы списка отделяются запятыми

Вначале объявления должна записываться последовательность спецификаторов, а затем список объявителей. Число объявителей в списке определяется числом, объявляемых имен. Объявляемое имя в качестве необязательного компонента может иметь список инициализаторов.

35.1. *Спецификаторы объявления*

Можно выделить следующие три категории спецификаторов:

- спецификаторы класса памяти.
- квалификаторы типа.
- спецификаторы типа.

35.1.1. Спецификаторы класса памяти.

Имеются следующие спецификаторы класса памяти:

- **`auto`**.
- **`extern`**
- **`regester`**
- **`static`**
- **`typedef`**

Класс памяти определяет время существования и вид связывания объявляемого имени. Ниже в таблице приводится назначение спецификаторов класса памяти приведено.

Спецификатор	Назначение
`auto`	Спецификатор допустим только в блоках и предназначен для указания на то, что переменная является локальной (автоматической). В программах обычно не используется в связи с тем, что по умолчанию локальные переменные имеют класс памяти **`auto`**.
`extern`	Этот спецификатор используется для указания на то, что определяемое имя имеет внешнее связывание.
`register`	Спецификатор применяется при работе с локальными переменными с той целью, чтобы рекомендовать компилятору хранить эту переменную в регистре. Современные компиляторы часто игнорируют эту спецификацию в связи с тем, что могут справиться с оптимизацией использования регистров лучше программиста.
`static`	Этот спецификатор может появляться в объявлениях функций и переменных. В объявлениях функций этот спецификатор указывает на внутренное связывание. В объявлениях переменных назначение спецификатора зависит от контекста, в котором встречается объявление. В том случае, когда объявление находится вне функций, то спецификатор указывает на внутреннее связывание. Для локальных переменных определяет размещение переменной в статической области оперативной памяти.
`typedef`	Определяет не имя функции или переменной, а имя типа. Имя типа располагается на месте объявителя.

35.1.2. Квалификаторы типа

Классификаторы **`const`** и **`volatile`** были введены в язык Си стандартом C89. Классификатор **`restrict`** был добавлен в язык стандартом C99.

Квалификатора **`const`** следует использовать в объявлениях тех объектов, значения которых должны оставаться неизменными. Имя константного объекта не должно встречаться в левой части оператора присваивания.

Квалификатор **`volatile`** предполагает, что значение объекта может внезапно измениться. На практике переменная, привязанная к некоторому аппаратному регистру, должна быть объявлена с использование этого

квалификатора. Наличие рассматриваемого квалификатора обязывает компилятор отказаться от выполнения оптимизации.

Квалификатор `restrict` используется только с указателями и будет рассмотрен в разделе 38.

35.1.3. Спецификаторы типа

Любой объявляемый объект должен иметь тип, определяемый одним или несколькими спецификаторами типа. Спецификаторы типа могут иметь следующий вид:

- последовательность спецификаторов встроенного (фундаментального) типа.
- имя перечисления.
- typedef-имя.

35.1.4. Использование спецификаторов

В объявлении спецификаторы могут быть записаны в любом порядке. Обычно программисты придерживаются одного из двух следующих форматов:

- спецификаторы класса памяти, квалификаторы типа, спецификаторы типа.
- спецификаторы класса памяти, спецификаторы типа, квалификаторы.

Первый формат является традиционным, однако второй формат в последнее время все чаще используется.

35.2. *Описатели*

Основным элементом каждого описателя является объявляемое имя. Этот элемент является обязательным. Кроме того, в объявитель может входить дополнительная информация, относящаяся к типу. В состав дополнительных компонентов могут входить классификаторы типа и некоторые операторы языка Си:

* - оператор разыменования (для объявления указателей см. пункты 17 и 38).

() – оператор функция (для объявления функций см. пункт 37).

[] – оператор индексирования (для объявления массивов см. пункт 39).

Оператор * в объявителе используется (по отношению к объявляемому имени) в качестве префикса, а операторы () и [] – в качестве суффикса. Можно провести некоторую аналогию между использованием операторов в объявителях и выражениях. Суффиксные операторы «сильнее» связаны с именем, чем префиксные.

Приведем примеры объявлений, В комментариях использованы следующие сокращения: ПС – послеовательность спецификаций, СО – список объявителей.

```
int n, m;
// ПС – спецификатор типа int, СО – два идентификатора перемен
// ных: n и m

double *px;
// ПС – спецификатор типа double, СО – идентификатор переменной
// px и префиксный оператор разыменования *
```

35.3. *Использование зарезервированного слова typedef для объявления синонимов типов*

Отметим, что typedef не позволяет объявлять новые типы. Его назначение состоит в том, чтобы назначать имя для уже существующего типа. В зависимости от вида существующего типа целесообразно различать три случая:

- встроенный (фундаментальный) тип.
- перечисление и структура.
- все прочие типы; их характерной особенностью является отсутствие у типа имени.

Для встроенных типов назначение существующему типу синонима может ставить своей целью упрощение редактирования текста при изменении требований к типу.

Для перечислений и структур применение typedef позволяет устранить из имени типа зарезервированное слово. Для перечислений таким именем является enum, а для структур – struct.

Для всех оставшихся случаев применение рассматриваемого способа позволяет упростить структуру новых объявлений.

Приведем ряд примеров.

Пример 1. Синоним для встроенного типа. Рассмотрим некоторый фрагмент кода, в котором будет объявлен синоним (integer) для встроенного типа int.

```
typedef int integer;
```

После объявления этого синонима у программиста появляется альтернатива, существо которой состоит в том, что наряду с обычным именем типа int, он имеет возможность пользоваться новым именем.

```
int   n;
integer m;
```

Бессистемное использование этих двух альтернативных возможностей пользы не приносит. Смысл в объявлении синонима в том случае, когда программист откажется от стандартного имени int, и будет использовать альтернативное integer.

```
integer n, m;
```

При этом появляется возможность его оперативного изменения. Например, если окажется, что тип int следовало бы заменить другим типом.

Например, типом double int. такая замена может быть осуществлена редактированием объявления имени integer.

```
typedef double int integer;
integer n, m;
```

Теперь переменные n и m имеют тип double int.

Пример 2. Синоним для типа, у которого нет своего имени. Обратимся к программному коду, приведенному ниже. В этом коде объявлен синоним ptr_dbl для существующего типа double*. Теперь программист имеет альтернативные возможности для объявления указателей на объекты типа double. Это и продемонстрировано в приведенном ниже коде.

```
typedef double* ptr_dbl;
double *pb1;
ptr_dbl pb2;
```

36. Функции

36.1. *Понятие об абстракции и инкапсуляции*

Известно, что сложность программирования обусловлена в первую очередь сложностью решаемых задач. Основным способом преодоления этих сложностей является применение декомпозиции. Декомпозиция сводится к тому, что исходная задача заменяется совокупностью подзадач, совместное решение которых приводит к искомому решению. Однако не любая декомпозиция заканчивается успешно. Типичной является следующая ситуация. Программные компоненты, реализующие решение отдельных подзадач работают корректно, а их объединение не работает. Классическим примером неудачной декомпозиции является попытка написать текст пьесы, распределив работу между отдельными исполнителями, следующим образом. Определяется состав действующих лиц. Затем каждому исполнителю указывается список действующих лиц пьесы и поручается написать часть текста пьесы, в которой они участвуют. Представляется, что результат будет самый плачевный.

Существует мнение, что качество декомпозиции может быть повышено в том случае, когда при ее выполнении применяется абстракция.

В науке и технике широко используется понятие абстракции. Под абстрагированием понимается процесс отделения основных характеристик объекта, явления или операции от их второстепенных характеристик. Применительно к программированию процесс абстрагирования можно трактовать следующим образом. При разработке любого программного компонента следует различать интерфейс и реализацию. Основные элементы должны составить интерфейс компонента, а второстепенные следует отнести к его реализации. Таким образом, основным результатом процесса абстрагирования является создание интерфейса, позволяющего клиенту разрабатываемого программного компонента воспользоваться возможностями, которые будут предоставляться компонентом.

Вторым понятием, которое наряду с абстракцией широко используется в современном программировании, является инкапсуляция. Назначение инкапсуляции состоит в том, чтобы сделать реализацию программного компонента недоступной для клиентского кода. Клиентский код должен получать доступ к возможностям программируемого компонента только через его интерфейс. Такой подход предоставляет определенные преимущества, как для разработчика, так и для клиента. Разработчику это дает возможность в дальнейшем изменять реализацию. Если при изменении реализации не произошло изменение интерфейса, то не возникает необходимости в изменении клиентского кода. С другой стороны инкапсуляция избавляет клиента от опасности испортить реализацию компонента.

В языке Си имеется развитый аппарат подпрограмм, которые называются функциями.

Прежде чем переходить к изучению функций, поставим ряд вопросом, на которые излагаемый материал позволял бы находить ответы. Следует иметь в виду, что на один и тот же вопрос можно получить, казалось бы, совершенно разные ответы. Причем каждый из ответов будет правильным. Это имеет место в тех случаях, когда рассматриваемая проблема оказывается сложной. Заметим, что это в полной мере относится к работе с функциями. Совокупность полученных ответов часто позволяет уяснить существо самой проблемы и возможных способ ее решения.

При использовании функций важно получить ответ на следующие вопросы:

1. Какова роль функций в программировании?
2. Какие разновидности функций имеются в языке Си?
3. Чем отличается понятие функции языка Си от понятия функции, принятого в математике?
4. В чем состоят преимущества и недостатки применения функций при программировании на языке Си?
5. Как оценить качество разработанной функции?
6. Как организованы функции языка Си?
7. Из каких структурных частей состоит функция языка Си?

На многие вопросы можно найти ответ, рассматривая функцию как процедурную абстракцию. Изучение функций языка Си начнем с установления связи между ними и процедурной абстракцией.

36.2. *Функция языка Си и процедурная абстракция*

Как отмечалось ранее, применение абстракции позволяет повысить качество декомпозиции. Язык Си позволяет реализовать один из видов абстракции, который принято называть процедурной абстракцией. Такой вид абстракции основан на применении подпрограмм. В процедурной абстракции используется способ, называемый абстрагированием через параметризацию.

Понятие параметров функции знакомо читателю по работе с библиотечными функциями.

36.3. *Понятие о функции*

Функцией называется *автономный* фрагмент программного кода, написанный на языке Си, имеющий собственное *имя*, предназначенный для решения некоторой задачи и *возвращающий значение* в точку вызова (последнее необязательное). Программной конструкцией для синтаксического оформления указанного фрагмента программного кода служит определение функции.

Прежде всего, отметим, что в языке Си предусмотрены две категории функций:

- Обычные функции.
- Встроенные функции или **inline** функции (стандарт C99).

Основное отличие между этими категориями функции состоит в том, что в выполняемом коде программы всегда присутствует только одна копия кода обычной функции. Программный код **inline** функции встраивается в каждую точку ее вызова.

Рассмотрим компоненты этого определения:

- *Имя*. Каждая функция должна иметь имя, которое позволяет обратиться к инструкциям, входящих в функцию. Такое обращение называется *вызовом функции*. Функцию можно вызывать из другой функции.
- *Автономность* – важнейшая характеристика функции. Язык Си обеспечивает синтаксическую независимость функции от другого программного кода.
- *Возвращаемое значение* позволяет вернуть результаты вычислений в точку вызова. Следует отметить, что возвращаемое значение является не единственным средством передачи в точку вызова результатов вычислений. Результаты вычислений можно вернуть, с помощью аппарата параметров функции, используя в качестве параметров указатели.

При работе с функциями языка Си следует различать три понятия:

- Определение функции.
- Объявление функции (прототип).
- Вызов функции.

Определение функции является ее реализацией. Написать функцию означает написать ее определение. Каждая функция в программе должна быть представлена только одним определением (это не относится к так называемым встроенным функциям или inline функциям).

В той точке программы, в которой функция должна выполнить требуемую от нее работу, к ней необходимо обратиться. Средством обращения к функции является вызов функции.

Появление прототипов функций связано с используемой в языке Си раздельной компиляцией отдельных модулей. Под модулем в языке Си понимается отдельный файл, имеющий расширение .c. Раздельная компиляция предполагает, что каждый модуль может разрабатываться, компилироваться и отлаживаться отдельно от других модулей. При этом вызов функции может оказаться в одном модуле, а ее определение – в другом модуле. Одно из преимуществ использования прототипа состоит в том, что его наличие позволяет компилятору выполнить корректную компиляцию вызова функции даже при условии недоступности ее определения. Программисту прототип позволяет написать корректный вызов функции. Перейдем к детальному рассмотрению программных компонентов, предназначенных для работы с функциями.

Следует учитывать, что определения функций в соответствии со стандартом языка Си не могут быть вложенными. Правда, некоторые расширения языка (например, вложенные функции реализованы среде rhide)

36.4. *Назначение функций*

Основное назначение подпрограмм и функций языка Си, в том числе состоит в следующем:

- Подпрограмма предназначена для целей структурирования программы. Подпрограмма является основным строительным блоком, который используется при разработке программ.
- Подпрограммы применяются для повторного использования кода.

Целесообразность в применении функции возникает каждый раз, когда ее применение позволяет улучшить структуру разрабатываемой программы. Дополнительным побуждением к разработке функции может служить то обстоятельство, что разрабатываемая функция будет вызываться из нескольких точек программного кода. В этом случае применение функции может привести к уменьшению общего программного кода. Кроме того, может оказаться, что задача, решаемая разрабатываемой функцией, является типичной для той прикладной области, в которой работает программист. В этом случае разработка функции будет способствовать повторному использованию кода.

Следует учитывать, что имеются и негативные последствия применения функций. Они связаны с временными накладными расходами, с которыми связано применение функций. Это положение характерно для обычных функций. Для inline функций временные накладные расходы отсутствуют. Особенно следует обращать внимание на этот фактор в том случае, когда функция вызывается в цикле.

36.5. *Определение функции*

В некоторых языках, например в языке Паскаль, различают две вида подпрограмм:

- Подпрограммы – процедуры.

- Подпрограммы – функции.

В языке Си формально имеется один вид подпрограмм, который и называется функцией. Однако функции языка Си существуют в двух разновидностях:

- Функции, возвращающие значение.
- Функции, не возвращающие значение.

Функции языка Си, возвращающие значение, являются аналогами функций языка Паскаль, а функции, не возвращающие значение - аналогами процедур языка Паскаль.

Прежде всего, следует различать старый и новый стиль организации функций. Старый стиль организации функций связывают с именами авторов классической книги по языку Си: Кернигана и Ритчи (стиль K&R). Новый стиль – это стиль, принятый в стандарте языка Си. В настоящее время при разработке новых программ старый стиль организации функций практически не используется. Учитывая это обстоятельство, основное внимание уделим новому стилю организации функций.

Определение функции состоит из двух структурных частей: заголовок и тело. Формально это можно представить следующим образом:

```
Заголовок_функции
{
 определения, объявления и инструкции
}
```

Заголовок функции определяет ее интерфейс с внешним миром (клиентским кодом), а ее тело - алгоритм, который она реализует. Все элементы, расположенные до открывающей фигурной скобки, составляют заголовок функции. Телом функции является блок.

Заголовок функции имеет следующую структуру:

```
тип имя([определения_формальных_параметров])
```

Первый элемент заголовка функции `тип` определяет, имеет ли определяемая функция возвращаемое значение. Функция не имеет возвращаемого значения в том случае, когда в качестве элемента *тип* используется зарезервированное слово **void**. Применение в качестве элемента *тип* любого другого спецификатора типа приводит к созданию функции, имеющей возвращаемое значение. Приведем примеры библиотечных функций, возвращающих и не возвращающих значение. Функция

Второй элемент заголовка функции `имя` является ее именем, которое используется для идентификации ее как функции.

Третий элемент заголовка `определения_формальных_параметров`**,** который не является обязательным, содержит список определений формальных параметров. Аппарат параметров (формальных и фактических) лежит в основе способа абстрагирования, который используется в процедурной абстракции. В отличие от определений переменных определения формальных параметров нельзя объединять в группы. Каждый формальный параметр должен быть определен отдельно.

Замечание. Стандарт С89 поддерживал так называемый принцип неявного **int**. В соответствии с этим принципом при отсутствии спецификатора типа по умолчанию предполагает использовании типа **int**. Стандарт С99 не рекомендует придерживаться этого принципа.

Формальные параметры имеют синтаксис переменных. Область видимости формальных параметров начинается от точки их определения и распространяется на все тело функции. Формальные параметры наряду с фактическими параметрами, используемыми в вызове функции, являются элементами связи по данным между функцией и точкой ее вызова (клиентским кодом). Формальные параметры могут рассматриваться как средство модификации алгоритма, предусмотренного функцией.

Тело функции пользователя стоится по тем же принципам, что и тело функции main(). В целом тело функции является блоком, который может содержать объявления переменных и выполняемые инструкции. Перейдем к рассмотрению принципа формирования функцией возвращаемого значения (для функций, такой результат возвращающих). Для этих целей тело функции должно содержать инструкцию **return**, имеющую следующий формат

```
return выражение;
```

Инструкция return, рассматриваемого вида, выполняет два действия:

1. Вычисляет значение своего компонента *выражение,* формирующего значение, возвращаемое функцией.
2. Прекращает вычисления, выполняемые в теле функции, и возвращает управление клиентскому коду.

Результатом выполнения функции, возвращающей значение, передается клиентскому коду в виде временной переменной, тип которой определяется элементом заголовка функции *тип*. Значение этой переменной возвращается в точку вызова функции и может использоваться в качестве операнда в том выражении, в котором находится вызов функции.

Заметим, что для прекращения вычислений в функции, не возвращаемых значения, используется инструкция, состоящая из одного зарезервированного слова **return**. Такая функция вообще может не содержать инструкции **return**. В этом случае вычисления в теле функции прекращаются в тот момент времени, когда управление достигает закрывающей фигурной скобки ее тела.

В качестве примера функции, возвращающей значение, приведем заголовок стандартной функции, предназначенной для возведения величины `x` в степень `y`:

```
double pow(double x, double y)
```

Заметим, что недопустимо написать заголовок функции `pow()` в следующем виде:

```
double pow(double x,y) /* Синтаксическая ошибка */
```

Функция `pow` принимает два аргумента, имеющих тип **`double`**, и возвращает в точку ее вызова значение типа **`double`**.

Приведем еще один пример заголовка функции:

```
void table(double xn, double xk, int n)
```

Учитывая принятие в заголовке имена, можно предположить, что функция, интерфейсом которой он (заголовок) является, будет использоваться для табулирования некоторой функции. Функция `table()` не возвращает значения, поэтому ее вызов не может использоваться в качестве операнда выражения, а должен использоваться как инструкция - выражение.

Телом любой функции, является блок. Блоком называется последовательность инструкций, заключенная в фигурные скобки. Тело функции может содержать вложенные блоки. В блоке могут встречаться инструкции двух видов: инструкции объявления и выполняемые инструкции. В стандарте C89 в блоке вначале следует располагать инструкции объявления, а затем выполняемые инструкции. В стандарте C99 это ограничение отсутствует, и инструкции объявления могут располагаться в любом месте блока. Используя эту возможность, можно располагать определение переменной в той точке программы, где она начинает использоваться. Это повышает читабельность программы.

Рассмотрим два примера определений функций пользователя. В первом из них функция имеет возвращаемое значение, а вторая не имеет возвращаемого значения.

Пример 1. Приведенная ниже функция (`max2()`) принимает два аргумента вещественного типа и возвращает значение набольшего из своих аргументов.

```
    double max2(double a, double b)
{
    double max;
    if(a > b)
       max = a;
    else
       max = b;
    return max;
}
```

В функции max2() результат вычислений фиксируется с помощью локальной переменной max. Приведем альтернативный вариант функции, решающей ту же задачу, что функция max2().

```
    double max2_2(double a, double b)
{
    if(a > b)
       return a;
    return b;
}
```

Сравним два варианта решения. Второй вариант (функция max2_2()) выглядит более компактной. Некоторые программиста возможно в качестве недостатка отметили бы наличие в теле этой функции двух инструкций **return**. Это можно рассматривать как нарушение принципа одного входа и одного выхода. Действительно лучше придерживаться принципа, в соответствии с которым функция имеет только одну точку выхода. Это

улучшает читабельность программного кода. Однако тело функции оказывается достаточно компактным, нарушение этого принципа не приводит к заметным негативным последствиям.

Пример 2. Табулирование функции $y = x^2$ в n равноотстоящих точках.

```
    void table(double xn, double xk, int n)
  {
     int i;
     double x, dx;
     printf("%5s%10s%10s\n", "Number", "X", "Y");
     x = xn;
     dx = (xk - xn) / (n - 1);
     for(i = 1; i <= n; i++)
     {
        printf("%6d%10.3f%10.3f\n", i, x, x * x);
        x += dx;
     }
}
```

36.6. *Функции, возвращающие и невозвращающие значение*

В некоторых языках, например в языке Паскаль, различают две вида подпрограмм:

- Подпрограммы – процедуры.
- Подпрограммы – функции.

В языке Си имеется все лишь один вид подпрограмм – функция. Однако такая функция может играть роль, как подпрограммы – процедуры, так и подпрограммы – функции языка Паскаль. Дело в том, что функция языка Си может возвращать значение, а может его не возвращать. Функция языка Си, не имеющая возвращаемого значения, играет роль процедуры, а возвращающая значение – роль функции языка Паскаль. Указанием на отсутствие у функции языка Си возвращаемого значения является использование в ее заголовке в качестве спецификатора типа, возвращаемого значения, зарезервированного слова **void**. Примером такой функции является библиотечная функция clrscr(), прототип которой имеет следующий вид:

void clrscr(**void**);

Примером функции, возвращающей значение, является библиотечная функция sqrt(), прототип которой имеет следующий вид:

double sqrt(**double**);

Заметим, что на ранних этапах использования языка Си широко применялся принцип неявного int. Существо этого принципа сводится к следующему. В тех случаях, когда отсутствует указание на тип объекта или функции, то по умолчанию предполагается наличие спецификатора типа int. Использование этого принципа позволяло повысить компактность записи объявлений и определений. Например, заголовок функции main() можно было записать в следующем виде:

```
     /* Устаревший стиль*/
  main(void)
```

Стандарт С99 не поддерживает принцип неявного int. Поэтому в заголовке функции main() следует в явном виде указать тип возвращаемого значения:

```
    /* Современный стиль */
int   main(void)
```

Значение, возвращаемое функцией, формируется с помощью инструкции **return**, которая должна иметь следующий формат:

```
return [выражение];
```

Порядок выполнения этой инструкции состоит в следующем:

- Вычисляется значение элемента *выражение*.
- Завершаются вычисления в функции.
- Создается временный объект, тип которого определяется типом, указанным в спецификации типа заголовка функции.
- Во временный объект переписывается вычисленное значение элемента *выражение*. При этом могут выполняться предусмотренные языком автоматические преобразования типа.
- Управление передается в точку вызова.

36.7. *Вызов функции. Передача параметров по значению*

В языке Си средством обращения к функции является вызов функции. Вызов функции имеет следующий формат:

```
имя([список_фактических_параметров])
```

Вызов функции содержит три структурных элемента. Два из них являются обязательными, а один нет. Вызов функции начинается с обязательного элемента `имя`. Этот элемент определяет имя функции, к которой выполняется вызов. Вторым обязательным элементом вызова функции является оператор функция, в качестве которого в языке Си используется пара круглых скобок. Наличие этого оператора делает расположенный перед ними идентификатор именем функции. Последний элемент вызова `список_фактически_параметров` относится к категории необязательных элементов. Фактические параметры наряду с формальными параметрами, которые содержатся в заголовке определения функции, служат для организации обмена данными между функцией и вызвавшим ее программным кодом (между функцией и ее клиентом). Фактические и формальные параметры должны быть согласованы по количеству, порядку следования и типам. Фактические параметры имеют синтаксис выражений. Приведем примеры вызовов для функции max2(), определение которой было написано выше. Пусть объявлены три переменных типа double a, b и c. Тогда можно написать следующие вызовы рассматриваемой функции:

```
    max2(3.0, 4.0)
max2(a, b)
max(a * a + 5, b + 3)
max2(max2(a, b), c)
```

Таким образом, при работе с функцией max2() ее аргументами могут быть константы, переменные, арифметически выражения и даже вызовы самой функции max2().

Теперь перейдем к рассмотрению вопроса о том, где располагать вызов функции в клиентском коде. Возможны два варианта использования вызова функции в ее клиентском коде. В первом варианте вызов функции используется либо в качестве операнда выражения, либо в качестве фактического параметра функции. Такая возможность открывается в том случае, когда функция имеет возвращаемое значение. Примером такой функции является рассматриваемая функция max2(). В том случае, кода возвращаемым типом является тип void (это равносильно тому, что функция не имеет возвращаемого значения) вызов функции можно использовать только в составе инструкции – выражения. Примером такой функции может служить библиотечная функция clrscr():

```
    int main(void)
{
    clrscr();
    /*   …   */
}
```

Опишем порядок выполнения вызова функции.

1. Вычисления в точке вызова временно прерываются. Запоминается точка возврата и состояния некоторых регистров процессора.
2. Управление передается вызываемой функции. В стеке выделяется память для формальных параметрах и локальных переменных.
3. Вычисляются значения фактических параметров, и формальные параметры функции инициализируются вычисленными значениями, соответствующих им фактических параметров. Операцией инициализации ограничивается взаимосвязь между формальными и фактическими параметрами.
4. Начинает выполняться тело функции. Выполнение функции завешается в момент времени, когда встречается инструкция return. Если функция не имеет возвращаемого значения, то в ее теле может отсутствовать инструкция return. В этом случае вычисления заканчиваются в том момент, когда управление достигает закрывающей скобки тела функции.
5. Память, выделенная функции в стеке, освобождается. Восстанавливается исходное состояние процессора. Управление передается в точку вызова функции.

Отметим одну особенность передачи параметров по значению. Эта особенность состоит в том, что для возврата результата, полученного в функции, необходимо в качестве формального параметра указатели (см. пункт 1.36.8). Дело в том, что при использовании передачи по значению через параметры данные передаются только в функцию. Применение указателей в качестве параметров функции позволяет организовать дополнительный канал связи между вызывающей и вызываемой функциями.

Обратимся к примеру. Воспользуемся приведенным выше определением функции max2(). Напишем для этой функции тестовую программу.

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

double max2(double a, double b)
 {
     double max;
     if(a > b)
        max = a;
     else
        max = b;
     return max;
 }

int main(void)
{
   double a, b, c;
   printf("a=");
   scanf("%lf", &a);
   printf("b=");
   scanf("%lf", &b);
   printf("c=");
   scanf("%lf", &c);

   printf("max(a, b)=%10.3f\n", max2(a, b));
   printf("max(a * a + 5, b + 3)=%10.3f\n",
           max(a * a + 5, b + 3));
   printf("max(3.0, 5.0)=%10.3f\n", max2(3.0, 4.0));
   printf("max(a, b, c)=%10.3f\n", max2(max2(a, b), c));
   return 0;
}
```

36.8. *Внешние переменные и процедурная абстракция*

Существуют две категории переменных в программе, написанной на языке Си:

- локальные,
- внешние (глобальные).

Локальными переменными называются переменные, объявленные в теле функции. Внешние переменные – это переменные, объявленные вне функций. Память для внешних переменных выделяется в статической области основной памяти. Все переменные, память для которых выделяется в статической области принято называть статическими. Время статических переменных совпадает со временем выполнения программы. Локальные переменные – это переменные, относящиеся к реализации функции. Память для таких переменных по умолчанию выделяется в стеке. Принято переменные, память для которых выделяется в стеке, называть автоматическими. Локальную переменную можно разместить в статической

области памяти. Для этого в ее определении следует использовать зарезервированное слово static. Примером переменной, которую следует объявлять внутри функции, является счетчик цикла.[1]

Укажем на одно полезное нововведение стандарта C99, которое касается объявления локальных переменных. Дело заключается в том, что этот стандарт разрешает объявлять переменные внутри инструкции for. Причем область видимости этой переменной ограничивается телом цикла. Например:

```
/* Стандарт C89 */
int s =0, i;
for(i = 0; i < 10; i++)
   s += i * i;
printf("i=%d", i);
```

36.9. *Понятие о прототипе. Компиляция*

Понятие прототипа вначале появилось в языке C++. Затем оно было заимствовано языком Си, что нашло отражение в стандарте C89. Прототипом функции может служить ее заголовок, после которого поставлен символ точка с запятой. В прототипе необязательно указывать имена формальных параметров. Прототип для разработанной выше функции max2() может быть записан в одном из следующих видов:

```
double max2(double a, double b);
double max2(double, double);
```

Предпочтительнее указывать в прототипах имена параметров, наделяя их мнемоническим смыслом. Это повышает читабельность программы. Прототипы необходимы для корректной компиляции вызова функции. Следует отметить, что стандарт c89 не был последовательным в отношении использования прототипов. Он в отличие от языка C++ допускал компиляцию вызова функции при отсутствии прототипа. В этом случае компилятор создавал так называемое неявное объявление функции, которое имеет следующий формат:

```
int имя();
```

Пустые круглые скобки являются для компилятора указанием на то, что ему не предоставляется информация о формальных параметрах объявляемой функции.

В стандарте C99 неявные объявления больше не поддерживает.

```
/* Стандарт C99 */
int s =0;
for(int i = 0; i < 10; i++)
   s += i * i;
/* printf("i=%d", i); на переменную i ссылаться нельзя
                      она здесь невидна. */
```

[1] Этот материал следует согласовать с материалом, изложенным в перой части по переменным

36.10. *Старый стиль определения функции*

В соответствии со старым стилем функция состоит из трех составных частей:

- Заголовок.
- Определение формальных параметров.
- Тело функции.

Заголовок функции, написанной в соответствии со старым стилем, имеет следующий формат:

```
[тип] имя([список_формальных_параметров])
```

В отличие от нового стиля организации функций в старом стиле в заголовке функции в аргументных скобках не указывались типы формальных параметров. Кроме того, в старом стиле записи определений функции указывать тип возвращаемого значения указывать было не обязательно. При этом отсутствии спецификации типа по умолчанию предполагался тип int. Стандарт C99 отменил действие принципа умолчания в отношении типа int. Например, заголовок функции max2(), написанный в соответствии со старым стилем, имел бы следующий вид:

```
double max2(a, b)
```

Определения формальных параметров записывались вне заголовка функции и располагались после заголовка функции, но до ее тела.

Тело функции, записанной в соответствии со старым стилем, строится по тем же правилам, что и использовании нового стиля. Для иллюстрации организации функций в соответствии со старым стилем перепишем определение функции max2(), приведенное выше.

```
/* Определение функции max2(), записанное в соответствии со
   старым стилем*/
double max2(a, b)       // Заголовок функции
double a, b;         // Объявление формальных параметров
{                     // Начало тела функции
  double max;
  if(a > b)
      max = a;
  else
      max = b;
  return max;
}
```

37. Массивы в стиле языка C89

Массивом называется упорядоченная совокупность однотипных величин, объединенных одним именем. Упорядоченность хранящихся в массиве данных достигается путем использования системы индексов. Количество индексов определяет размерность массива. Различают одномерные массивы и многомерные массивы. При работе с элементами одномерных массивов используется один индекс, а при работе с многомерными массивами – два и более индексов.

Назначение массивов состоит в хранении некоторой совокупности данных для их последующей обработки.. Приведем примеры ситуаций, когда возникает необходимость в использовании массивов.

- Одна и та же совокупность однородных данных неоднократно используется в процессе обработки.
- Для отделения ввода данных от их обработки.
- Для сохранения однородной совокупности исходных данных с целью облегчения конечному пользователю анализа результатов вычислений.

Выше подчеркивалось, что целесообразность в использовании массива однородных данных. Это предполагает, что все элементы этой совокупности будут обрабатываться одинаково. Например, это характерно для обработки данных полученных в процессе табулирования некоторой математической функции. С другой стороны, объединять в массив элементы комплексного числа вряд ли целесообразно. Это обусловлено тем обстоятельством, что действительная и мнимая часть комплексного числа должна обрабатываться по-разному. В языке Си каждый массив располагается в непрерывной области памяти. Первый его элемент располагается по наименьшему адресу, а последний - по самому большему. В языке Си нумерация по любому из измерений массива начинается с нуля.

Следует отметить, что в языке Си имеется тесная связь между массивами и указателями. Наиболее ярким проявлением этой связи является то обстоятельство, что имя массива является константным указателем на его первый элемент.

При определении (объявлении) и при обращении к отдельным элементам массива используется бинарный оператор индексирования ([...]). Вначале будут рассмотрены, которые предусмотрены стандартом языка C89. Особенность этих массивов состоит в том, что размер памяти, которая должна быть для них выделена, должен быть известен на этапе компиляции. Затем в пункте 1.39. будут рассмотрены массивы с переменными размерами. Особенностью последнего вида массивов является то положение, что определение необходимего объем памяти может быть отложено до этапа выполнения программы.

37.1. *Определение и объявление массивов*

При работе с переменными, являющимися массивами, следует различать его объявление и определение. Здесь имеет место полная аналогия с обычными переменными. Определение массива позволяет компьютеру выделить память. Определение массива должно содержать информацию, достаточную для выделения памяти. Каждый массив должен иметь только одно определение. На количество объявлений массива ограничений нет. Отличительным признаком объявления является наличие в нем зарезервированного слова extern.

37.1.1. Определение и объявление одномерных массивов. Инициализация одномерных массивов при их определении. Доступ к элементам массива.

Определение одномерного массива отличается от определения простой переменной наличием после определяемого имени квадратных скобок, в которых записывается константное выражение целого типа, определяющее количество элементов массива. Определение может быть дополнено набором инциализаторов, который заключается в фигурные скобки.

Общий формат определения одномерного массива в стиле стандарта C89 имеет следующий:

<α> <β>[<γ>] = <μ>;

Здесь α- тип элементов, хранящихся в массиве; β - имя массива, [..] – оператор индексирования; γ – константное выражение, определяющее размер памяти, выделяемой для хранения элементов массива Конструкция «= <μ>» используется для инициализации массива; μ – список инициализаторов.

Обращает на себя внимание использование в объявлении и в определении массива оператора индексные скобки

Для обращения к отдельным элементам одномерного массива имеет сдующий формат:

<β>[<σ>]

Здесь β - имя массива, [] – оператор индексные скобки, σ-арифметическое целочисенное выражение, значение которого определяет индекс элемента.

Пример 1 Определение глобального и локального массива.

```
#define MSIZE1 20
int x[MSIZE];
int main(void)
{
   double z[5];
   z[0] = 10;

}
```

В этом примере имеются опредеделения двух одномерных массивов. Первый массив (x) является глобальным. Он предназначен для хранения 20 элементов типа. int. В связи с тем, что определение этого массива находится вне функций, его элементы в момент выделения памяти будут обнулены. Второй массив (z) является локальным. Он предназначен для хранения 5 элементов типа. double. Элементы этого массива будут содержать «мусор». В теле функции main() с помощью оператора присваивания элементу массива z с нулевым индексом присваивается значение 10.

Пример 2. Определение и объявление массивов.

Пусть в некоторой программе имеются два модуля. В первом модуле содержится функция main(), а во втором модуле с с именем module содержатся объявление и опрелеление массива x.

```
// Первый модуль
#inclde "module.h"
int main(void)
{
     // Работаем с массивом x
     x = 10;
}
// Второй модуль (module). Интерфейсный файл module.h
#define MSIZE 20
extern x[]; // Это объявление массива x

// Второй модуль. Файл реализации module.c
#include "module1.c"
int x[MSIZE];// Здесь выделяется память для массива x
```

В интерфейсном файле module.h содержится объявление массива x. а в файле реализации module.c находится определение массива x. Функция полчает доступ к этому массиву, что позволяет ей записать в элемент x[0] значение 10.

Пример 3. Инициализация массива во время определения

Рассмотрим программный код, приведенный ниже.

```
#define MSIZE 5
int main(void)
{
     int x1[MSIZE] = {1, 3, 5, 7, 9};
     int x2[MSIZE] = {1, 3, 5};
     int x3[MSIZE] = {1, 3, 5, 7, 9, 11};// Ошибка
     int x4[] =       {1, 3, 5, 7, 9};
     int x5[MSIZE] = {0};
}
```

В рассматриваемом коде объявлено пять массивов. Все массивы имеют одинаковый размер, определяемый константой MSIZE, равной 5. Здесть количество элементов в массиве совпадает с количеством инициализаторов. При объявлении массива x1 список инициализации содержит 5 констант. Элемент x1[0] этого массива инциализируется первой константой, равной 1, затем элемент x1[1] инициализируется константой 2 и т. д. Наконец, элемент x1[4] инициализируется последней константой из списка инициализации, равной 9. Количество элементов массива и количество инциализаторов может не совпадать. Если количество элементов массива превышает количество инициализаторов, то элементы массива, для которых «не хватило» инициализаторов обнуляются. Например, x2[3] == x2[4] == 0; Случай, когда инициализаторов превышает количество элементов массива, рассматривается как ошибочный. При компиляции определения массива x3 будет выдано сообщение об ошибке. Заметим, что при наличии списка инициализации размер требуемой памяти можно не указывать, что, и сделано при определении массива x4.

37.1.2. Определение и инициализация двумерных массивов

Общий формат определения одномерного массива в стиле стандарта C89 имеет следующий:

<α> <β>[<γ1>][<γ2>] = <μ>;

Здесь α- тип элементов, хранящихся в массиве; β - имя массива, [..] – оператор индексирования; γ1 – константное выражение, определяющее количество строк в двумерном массиве, а γ2 - константное выражение, определяющее количество столбцов в двумерном массиве. Конструкция «= <μ>» используется для инициализации массива; μ – список инициализаторов.

Приведем пример определения двумерного массива.

```
#define MROW 2
#define MCOL 3

int main(void)
{
    double m[MROW][MCOL] = {
                               {1, 3, 5},
                               {2, 4, 6}
                          };
}
```

В этом примере содержится определение двумерного массива m, в котором может храниться не более 2 строк и не более трех столбцов. Элементы массива инициализированы во время его определения. Первая строка массива содержит числа 1, 3 и 5, а вторая строка - 2, 4 и 6. Элементы массива имеют тип double.

37.2. *Операции с массивами*

Все операции с элементами массива выполняются поэлементно. Приведем два примера.

Пример 1. Копирование одномерных массивов.

Постановка задачи. Даны два массива одинаковых размеров x1 и x2. Скопировать часть n элементов массива x1 в массив x2.

Решение. Организуем арифметический цикл, рассчитанный на n повторений. Поэлементно будем копировать элементы массива x1 в массив x2. Программный код, представленный ниже, решает рассматриваемую задачу.

```
#define MSIZE 20
#include <stdio.h>
int main(void)
{
    int x1[MSIZE] = {1, 3, 5};
    int x2[MSIZE];
    int n;
    printf("n=");
    scanf("%d", &n);
    for(int i = 0; i < n; i++)
       x2[i] = x1[i];
```

```
    //..
}
```

В рассматриваемом коде «n» элементов массива x1 копируется в массив x2. Если ввести значение «n», равное MSIZE, то весь массив x1 , будет скопировани в массив x2. Заметим, что инстукция x1 = x2; в связи с тем, x1 – константный указатель.

Пример 2. Обработка двумерных массивов.

Постановка задачи. В двумерном массиве, хранятся данные типа double. Массив состоит из nr строк и nc столбцов. Требуется вычислить две суммы summa_row и summa_col. Здесь summa_row - сумма элементов в каждой строке, а summa_col – сумма элементов для каждого столбца.

Решение

При обращении к элементам двумерного массива следует использовать два индекса, причем каждый из индексов должен записываться в своих индексных скобках. Отметим, левый индекс определяет номер строки, в которй находится элемент массива, а правый – номер столбца. Например, если m – матрицы, то m[1][2] – элемент этой матрицы, находящийся в строки с индексом 1 и столбце с индексом 2.

Каждая из сумм summa_row и summa_col должна вычисляться отдельно с использованием вложенных циклов. Дело в том, что при вычислении суммы summa_row матрица должна просматриваться по строкам, а при вычислении summa_col матрица должна просматриваться по стобцам.

Алгоритм просмотра по строкам реализуется с помощью вложенных циклов. Причем:

- внешний цикл должен фиксировать индекс строки,
- внутренний цикл - изменять индекс столбца.

В алгоритме просмотра по столбцам наоборот:

- внешний цикл должен фиксировать индекс столбца,
- внутренний цикл – изменять индекс строки.

```
#define MROW 5
#define MCOL 10

int main(void)
{
    double summa_row[MROW];
    int nr;
    printf("nr = ");
    scanf("%d", &nr);

    int nc;
    printf("nc = ");
    scanf("%d", &nc);

    double m[MROW][MCOL];
    // Ввод матрицы по строкам
    for(int r = 0; r < nr; r++)
    {
```

```
      for(c = 0; c < nc; c++)
      {
           printf("m[%d][%d]", c, r);
           scanf("%lf", &m[r][c]);
      }
   }
   // Вычисление суммы для каждой строки
   double summa_row[MROW];// массив для хранения суммы по
                           // строкам
   for(int r = 0; r < nr; r++)
   {
      double s = 0;
      for(c = 0; c < nc; c++)
      {
         s += m[r][c];
      }
      summa_row[r] = s;
   }
   // Вычисление суммы для каждого столбца
   double summa_col[MROW];// массив для хранения суммы по
                           // столбцам
   for(int c = 0; c < nc; c++)
   {
      double s = 0;
      for(r = 0; r < nr; r++)
      {
         s += m[r][c];
      }
      summa_col[c] = s;
   }
   // вывод результатов вычислений

}
   return 0;
```

37.3. *Размещение массивов в оперативной памяти*

Массивы занимают в памяти компьютера непрерывный участок памяти. Структура одномерного массива, как структура данных, хорошо соответствует одномерной природе структуры оперативной памяти компьютера. Иначе дело обстоит с двумерными массивами. Двумерность совместить с одномерностью оперативной памяти можно разными способами. В некоторых языках двумерные массивы хранятся по столбцам. Так двумерные массивы хранятся в языке Фортран. В языке Си двумерные массивы хранятся по строкам. Вначале первая строка, затем вторая и так далее.

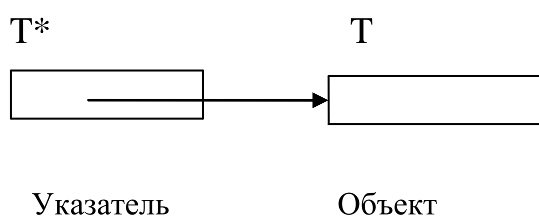
38. Указатели

Указатель – это программный элемент (переменная или выражение), значением которого является адрес переменной (указатель на объект) или функции (указатель на функцию). Вначале рассмотрим указатели на объект.

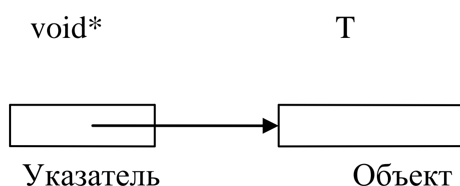
Целесообразно различать типизированные и нетипизированные указатели. Значением типизированного указателя может быть адрес объекта только определенного типа. Значением нетипизированного указателя может быть объект любого типа.

При работе с указателями следует использовать модель, состоящую из двух компонентов: собственно указателя и объекта, на который он установлен. Обозначим через T – тип объекта, на который должен быть установлен указатель. Тогда модель указателя может быть изображена в следующих двух вариантах (см. рис.).

А) Для типизированных указателей



Б) Для нетипизированных указателей



Из приведенной модели следует, что при работе с указателями недостаточно написать определение указателя (тем самым, выделив для него память), но необходимо обеспечить его инициализацию, состоящую в том, что в указатель записывается адрес того объекта, на который он должен быть установлен.

Для дальнейшего необходимо рассмотреть два оператора: разыменования и взятия адреса.

38.1. *Операторы разыменования и взятия адреса*

Это два основных оператора при работе с указателями. В результате выполнения оператора разыменования возвращается тот объект, на который установлен указатель. Основным оператором разыменования является префиксный унарный оператор “*”. Пусть в программе объявлен типизированный указатель p. Тогда, образовав выражение *p, получим доступ к тому объекту, адрес которого содержится в указателе p. Выражение *p относится к категории lvalue. Такое выражение может использоваться как

в левой, так и в правой части оператора присваивания. Если T - тип объекта, на который может быть установлен указателя p, относится к категории арифметических типов, то допустима инструкция присваивания следующего вида:

*p = 10;

Оператор взятия адреса является унарным префиксным оператором, позволяющим получить адрес своего операнда. Использование оператора взятия адреса приводит к образованию указателя-выражения. Приведем пример. Пусть в программе объявлены переменная n типа int и типизированный указатель ptr, значением которого могут быть адреса объектов типа int. Тогда с помощью следующей инструкции можно инициализировать указатель ptr:

ptr = &n;

Приведенная выше инструкция присваивания записывает в указатель-переменную ptr значение указателя-выражения &n.

38.2. *Области применения указателей*

Можно выделить следующие основные области применения указателей:

1. Использование указателей в качестве параметров функций для возврата результатов вычислений.
2. Работа с объектами, размещенными в динамической памяти.
3. Замена манипуляций со сложными объектами данных манипуляцией с адресами этих объектов.

38.3. *Нулевой указатель*

Нулевой указатель – это такое значение указателя, которое не может быть адресом переменной или функции. Наиболее часто используемыми представлениями нулевого указателя как константы являются 0, 0L и NULL. Считается хорошим стилем программирования всем указателям, которые не работают с реальными объектами (переменными) или функциями, было присвоено значение NULL

38.4. *Определение указателя*

В определении указателя используется оператор разыменования “*”. Упрощенный формат определения указателя имеет следующий вид:

```
type     * name;
```

Здесь *type* – спецификация типа, определяющая тип объекта, адрес которого будет храниться в указателе; *name* – имя (идентификатор) указателя. Заметим, что иногда тип объект, на который установлен указатель, называют базовым типом. Как уже отмечалось, в записи определений используется синтаксис выражений. Продемонстрируем это положение на примере. Пусть имеется следующее определение указателя:

int * ptr;

Здесь *type* – это ***int***, а ptr – *name*. Приведенное выше определение можно “прочитать”, используя синтаксис выражений, следующим образом.

Применение оператора разыменования к имени ptr позволяет получить объект типа **int**. Отсюда следует тот факт, что ptr является именем указателя

Во время определения указатель может быть инициализирован. В качестве инициализаторов указателя при его объявлении могут использоваться:

- Нулевой указатель.
- Указатель-переменная
- Указатель-выражение.

Приведем примеры инициализации указателей во время их определения.

```
/* Примеры инициализации указателей при их определении */
int* p = NULL; /* Инициализация указателя p нулевым указателем*/

int* p2 = p;   /* Инициализация указателя p2 значением
                  существующего указателя p*/

int n = 5;
int* p3 = &n;  /* Инициализация указателя значением
                  указателя-выражения &n */
/*    . . . . . . . . . . . . . . . . . . . . . . . . . . . . */
```

38.5. *Недействительный указатель*

Недействительный указатель (invalid pointer) – это ненулевой указатель, содержимое которого не является адресом переменной или функции. Любое использование недействительного указателя (разыменование такого указателя, сравнение его со значением NULL, передача его в качестве параметра в функцию) в стандарте языка Си приводит к неопределенному поведению программы. Рассмотрим возможные причины появления недействительных указателей.

Имеется ряд причин появления недействительных указателей:

- Отсутствие инициализации указателей,
- Ошибки, допущенные при преобразовании целочисленных значений к указательному типу.
- Освобождение динамической памяти.
- Ошибки, допущенные при использовании адресной арифметики.

Здесь ограничимся рассмотрением только первой из перечисленных выше причин. Одна из наиболее распространенных причин появление недействительных указателей связана с отсутствием инициализации указателей. Ниже приводится типичный пример программного кода, в котором используется неинициализированный указатель.

```
/*     Ошибка в программном коде*/
    double x;
    double* pn;
    x = modf(10.2, pn); // Использование неинициализированного
```

```
                                // указателя
/* ............................*/
```

38.6. *Операции с указателями*

В языке Си указатели наделены большими возможностями. Об этом свидетельствует размер перечня тех операций, которые можно выполнять с указателями:

- Разыменование,
- Взятие адреса,
- Присваивание,
- Индексирование указателей
- Арифметические действия с указателями,
- Сравнение указателей.

38.6.1. Дополнительные сведения о разыменовании указателей

Как отмечалось выше, разыменование – операция позволяющая получить доступ к объекту, на который установлен указатель. Однако эта операция допустима не ко всем указателям. Остановимся на тех категориях указателей, к которым нельзя применять разыменование:

- Недействительный указатель.
- Нетипизированный указатель.
- Нулевой указатель

Приведем примеры недопустимых разыменований.

```
/* Примеры недопустимых разыменований указателей */

int* p = NULL;
*p = 10;        /* Недопустимо разыменовывать нулевой указатель*/

int n = 5;
void* p3 = &n;
*p3 = 10;       /* Недопустимо разыменовывать нетипизированный
                   указатель p3*/

char* p4;
*p4 = ’A’;      /* Недопустимо разыменовывать
                   неинициализированный указатель p4 */
/*    . . . . . . . . . . . . . . . . . . . . . . . . . . . */
```

Заметим, что к категории операторов разыменования относятся еще два оператора:

- Оператор [], который используется при работе с массивами.
- Оператор ->, который используется при работе со структурами.

38.6.2. Присваивание указателей

Одному указателю можно присвоить значение другого указателя. Однако некоторые присваивания, которые допускает язык Си, могут привести к некорректной работе программы. Остановимся на этом вопросе подробнее. Рассмотрим ряд частных случаев. Начнем с присваивания типизированных указателей. Корректность такой операции не вызывает сомнений только в том случае, когда совпадают базовые типы обоих указателей. В противном случае такую операцию следует считать недопустимой. Приведем пример программного кода, в котором будут встречаться оба вида присваиваний.

```
/*   Корректные и некорректные присваивания типизированных
              указателей                                    */
int n = 10;
double x = 20.3;
int *p1, p2;
double*  p3;

p1     = &n;    /* допустимое присваивание    */
p2 = p1;        /* допустимое присваивание    */
p3 = p2;        /* недопустимое присваивание, т. к. не
                   совпадают базовые типы указателей    */
/* конец программного кода примера            */
```

Отметим, что, несмотря на семантическую недопустимость присваивания p3 = p1 в приведенном выше примере компилятор языка Си ограничивается выводом предупреждения (Компилятор языка C++ в этой ситуации выводит сообщение об ошибке). Работа языка Си с указателями свидетельствует о его пониженной типизации. Это обстоятельство следует рассматривать как крайне опасным последствиям. Одна из рекомендаций по преодолению этих неприятностей заключается в предварительной компиляции программного кода компилятором языка C++.

Одному указателю можно присвоить значение другого указателя. Однако некоторые присваивания, которые допускает язык Си, могут привести к некорректной работе программы. Остановимся на этом вопросе подробнее. Рассмотрим ряд частных случаев. Начнем с присваивания типизированных указателей. Корректность такой операции не вызывает сомнений только в том случае, когда совпадают базовые типы обоих указателей. В противном случае такую операцию следует считать недопустимой. Приведем пример программного кода, в котором будут встречаться оба вида присваиваний.

```
/*   Корректные и некорректные присваивания типизированных
              указателей                                    */
int n = 10;
double x = 20.3;
int *p1, p2;
double*  p3;

p1     = &n;    /* допустимое присваивание    */
p2 = p1;        /* допустимое присваивание    */
```

```
p3 = p2;        /* недопустимое присваивание, т. к. не совпадают
                   базовые типы указателей    */
/* конец программного кода примера            */
```

Отметим, что, несмотря на недопустимость присваивания p3 = p1 в приведенном выше примере компилятор языка Си ограничивается выводом предупреждения (Компилятор языка С++ в этой ситуации выводит сообщение об ошибке).

38.6.3. Операция взятия адреса для указателя. Указатели на указатели

Пусть имеется следующий фрагмент программы

```
//..
int n = 5;
int *p = &n;
int **pp = &p;
```

Переменная pp является указателем на указатель или двухуровневым указателем. Эта переменная получена применением оператора взятия адреса & к указателю p. Указатели на указателиприменяются на практике. Например, в качестве параметров функций (см. прототип функции strtok).

38.6.4. Указатели и операция индексирования

Рассмотрим следующий фрагмент программного кода:

```
#define MSIZE 5
#include <stdio.h>
int main(void)
{
     int x[MSIZE] = {1, 3, 5, 7, 9};
     int *p = &x[1];
     printf("p[1] = %d", p[1]);
     // ..
}
```

В рассматриваемом фрагменте программы объявлен указатель p, который инициализирован адресом второго элемента массива x. Затем функции printf() к указателю p применен оператор индексные скобки []. В результате выполнения этого фрагмента программы на экране получим p[1] = 5.

38.6.5. Арифметические операции с указателями

Арифметические операции применяются к укзателям, которые работают с массивами. К числу допустимых арифметических операций к таким указателям относятся:

- ++ (инкремент),
- --(декремент),
- сложение с целым числом,
- вычитание целого числа.
- вычисление разности двух указателей.

Рассмотрим слеющий программный код:

```
#define MSIZE 5
#include <stdio.h>
int main(void)
{
     int x[MSIZE] = {1, 3, 5, 7, 9};
     int *p1 = &x[1];
     int *p2 = &x[3];
     // ..
}
```

В этом программном коде объявлены указатели p1 и p2, инциализированныt адресами элементов массива x. Допустимы следующие выражения с этими указателями:

- p1++ и ++p1,
- p1-- и p1--,
- p1 + 2 и p2 – 2
- p2 – p1.

В результате вычисления всех выражений, исключая последнее выражение (p2 – p1), получается адрес. Результатом вычисления разности p2 – p1 будет целое число. Рассмотрим выражение p1++. При вычислении этого выражения следует учитывать возвращаемое значение и побочный эффект. Возращаемым значения выражения p++ является исходное значение указателя p. Побочный эффект при вычислении выражения p++ состоит в увеличении операнда на единицу. Однако адрес, хранящийся в указателе p, увеличивается таким образом, чтобы указатель был установлен на следующий элемент массива. При работе в 32 разрядной среды адрес, хранящийся в указатель, увеличиться на 4 байта. Таким образом, арифметические операции, выполняемые с указателями, оказываются масштабированными. В качесте масштаба используется объем памяти, которую занимает один элемент массива, с которым работает указатель.

38.6.6. Сравнение указателей

Указатели можно сравнивать. Сравнение имеет смысл только для указателей на один и тот же массив. При соблюдении этого условия операции ==, !=, <, >, <=, >= будут выполняться корректно.

38.7. *Указатели и динамическая память*

Как отмечалось выше, одной из основных областей применения указателей – работа с динамической памятью. Использование динамической памяти позволяет повысить эффективность использования оперативной памяти компьютера. При этом возрастают трудности программирования. В первую очередь это связано с тем, что программист должен следить за выделением и освобождением память самостоятельно.

В стандарте языка Си определены четыре функции, предназначенные для выделения и освобождения динамической памяти. К ним относятся следующие функции:

- malloc().
- calloc(),
- realloc(),
- free().

Первые три функции служат для выделения динамической памяти. Последняя функция предназначена для освобождения памяти. На начальном этапе знакомство с вопросами, связанными с работой с динамической памятью, достаточно ограничиться функциями malloc() и free().

Функция malloc

```
#include<stdlib.h>
void* malloc(size_t size);
```

Эта функция предпринимает попытку динамически выделить непрерывный блок памяти, размер которого равен size_t байт. В случае успеха функция возвращает указатель на начало выделенной памяти. В случае неуспеха функция вернет указатель NULL. Следует учитывать, что после завершения работы с памятью, выделенной функцией malloc(), ее следует освободить вызовом функции free().

В приведенном ниже примере делается попытка динамически выделить память для строкового буфера размером в MAXSIZE байт. Если попытка закончится неудачей, то будет вызвана функция пользователя error().

```
#include<stdlib.h>
#define MAXSIZE 129
int main(void)
{
     char* buf;
     if(NULL==(buf = malloc(MAXSIZE)))
         error();
     /*   ..................*/
}
```

Функция calloc

Объявление этой функции имеет следующий вид

```
#include<stdlib.h>
void* calloc(size_t nmemb, size_t size);
```

Делает попытку выделить память для массива из nmemb объектов, размер каждого из которых равен size байт. В случае успеха функция calloc() возвращает указатель на начало выделенной памяти. В случае неуспеха возвращается указатель NULL. Выделенная память заполняется двоичными нулями.

```
#include<stdlib.h>
#define MAXSIZE 129
int main(void)
{
```

```
        double* arr;
        int     n;
        if(NULL==(arr = calloc(MAXSIZE)))
           error();
        /*    ..................*/
    }
```

Функция realloc

Объявление этой функции имеет следующий вид

```
#include<stdlib.h>
void* realloc(void* ptr, size_t size);
```

Основное назначение функции `realloc()` состоит в изменении размера выделенной динамической памяти. Такое использование рассматриваемой функции имеет место при условии, что ее первый параметр (`ptr`) не равен `NULL`. В том же случае, когда значение параметра ptr равно `NULL`, функция `realloc()` ведет себя подобно функции `malloc()`.

Выполнение рассматриваемой функции в стандартах С89 и С99 несколько различаются, хотя конечный результат один и тот. Вначале рассмотрим работу этой функции в соответствии со стандартом С89. При использовании с параметром ptr, отличным от нуля, функция `realloc()`, делает попытку изменить размер объекта, на который установлен указатель ptr, до размера, определяемого значением параметра `size`. Если содержимое нового и старого объекта имеет общую часть, то содержимое общей части остается неизменным. Если размер нового объекта превышает размер старого объекта, то содержимое новой области памяти неопределенно. В соответствии со стандартом С89 значение, возвращаемое функцией, может совпадать, а может и не совпадать со значением параметра `ptr`. В соответствии со стандартом С99 функция `realloc()` должна освобождать старый блок динамической памяти и выделять новый блок. Поэтому значение, возвращаемое функцией `realloc()` в соответствии с этим стандартом всегда отличается от значения параметра `ptr`.

Приведем пример применения функции `realloc()`. Пусть в некоторой программе необходимо иметь массив, состоящий из “`n`” ‘элементов типа `double`. Предположим далее, что позже в зависимости от выполнения некоторых условий может потребоваться удвоение выделенной памяти для хранения элементов динамического массива. Ниже приведем фрагменты программного кода, которые решают задачу по выделению динамической памяти для рассматриваемой задачи.

```
#include<stdlib.h>
int main(void)
{
  double* p = NULL;
  int n;
  /*                                        */
  p = malloc(sizeof(double) * n);
  if(!p)
  {
```

```
    puts("Ошибка при выделении памяти");
    exit(1);
  }
  /*                                        */
  if(/*  Проверка условия необходимости удвоения памяти */)
  {
       p = realloc(p, siztof(double) *2 * n);
       if(!p)
       {
          puts("Ошибка при выделении памяти");
          exit(1);
       }

  }
  /*                                                     */
  return 0;
}
```

Функция free

Объявление этой функции имеет следующий вид

```
#include<stdlib.h>
void free(void* ptr);
```

Рассматриваемая функция предназначена для освобождения памяти, на которую установлен указатель ptr, Эта память должна быть предварительно выделена одной из следующих функций: calloc(), malloc() и realloc(). Если значение параметра ptr равно NULL, то функция free() никаких действий не выполняет. В противном случае, если указатель ptr не установлен на память, выделенную одной из функций, предназначенных для выделения динамической памяти, поведение функции free() не определенно. Значение указателя ptr после успешного освобождения памяти считается неопределенным, такие указатели относятся к категории недействительных указателей. Рекомендуется после освобождения памяти указателю ptr присваивать значение NULL.

Отметим, что функция free() не имеет средств, предназначенных для передачи информации о возникновении ошибки при освобождении памяти.
Рассмотрим пример применения функции free().

```
#include<stdlib.h>
#define MAXSIZE 129
int main(void)
{
     char* buf;
     if(NULL == (buf = malloc(MAXSIZE)))
         error();
     /*   .................. */
     /*   Строка buf теперь не нужна */
     free(buf);
     buf = NULL;
     /*   .................. */
}
```

38.8. *Указатели – параметры функций. Имитация передачи по ссылке*

В некоторых языках программирования, например в языке Паскаль, имеются два способа передачи параметров в функцию: передача по значению и по ссылке. В языке Си существует только один способ – передача параметров по значению. Применение указателей в языке Си открывает возможность имитировать (моделировать) передачу по ссылке. Это достигается путем установления с помощью указателя - параметра связи с памятью, выделенной в точке вызова функции. Это позволяет использовать указатели – параметры функций того, чтобы вернуть через параметры результаты вычислений, полученные в функции. Рассмотрим два примера.

Пример 1 Обмен значений двух переменных.

Постановка задачи. Даны две переменные a и b типа double. Необходимо выполнить обмен значений этих переменных. Решение необходимо оформить в виде функции.

Решение.

Прежде всего, следует убедиться в том, что попытка решить эту задачу, используя функцию, имеющую следующий интерфейс

```
void swap(double a, double b);
```

закончится неудачей. Вернуть обмен значений параметров a и b, выполненный в теле функции swap() не удается.

```
void swap(double *p1, double *p2)
{
    double temp   = *p1;
              *p1 = *p2;
              *p2 = temp;
}

// Клиентский код
int main(void)
{
      double a = 2, b = 3;
      swap(&a, &b);
}
```

Следует убедиться в том, что при использовании рассматриваемого варианта функции swap() обмен значений переменных a и b имеет место. Этого удалось достичь за счет того, что адреса этих переменных во время вызова рассматриваемой функции оказались записанными в парметры функции. В результате в теле функции swap() с помощью разыменования указателей осуществляется работа с памятью, выделенной в точке вызова. Обратимся, например, следующей инструкции , которая находится в теле функции swap()

```
*p1 = *p2;
```

Нетрудно убедиться в том, что эта инстукция работает с переменными a и b, которые находятся в клиентском коде, выполняя копирование переменной b в переменную a.

Пример 2 Обмен значений двух указателей.

Постановка задачи. Даны два указателя p1 и p1 типа double*. Решение необходимо оформиты в виде функции.

Решение.

```
void swap_pointers(double **p1, double **p2)
{
    double* pt    = *p1;
              *p1 = *p2;
              *p2 = *pt;
}

// Клиентский код
int main(void)
{
      double a = 2;
      double *pa = &a;
      double b = 3;
      double *pb = &b;

      swap_pointers(&pa, &pb);
      // ..
}
```

38.9. *Указатели на функцию*

Обычно параметры функций используются для организации интерфейса по данным между контекстом точки вызова и контекстом функции. Применение указателей на функции позволяет ввести в рассмотрение новый вид параметров. Параметры функций, относящиеся к этому новому виду, позволяют в разрабатываемую функцию передавать другие функции. В результате возникает возможность повысить степень универсальности разрабатываемой функции.

38.9.1. Определение понятия указатель на функцию

Указатель на функцию – это либо выражение, значением которого является адрес функции, либо переменная, в которой хранится адрес функции. Адресом функции чаще всего является адрес начала ее программного кода. Однако в некоторых компиляторах адресом функции может быть адрес объекта, который содержит адрес начала программного кода функции.

38.9.2. Классификация указателей на функцию

В языке Си существуют две разновидности указателей на функцию:

- указатель – выражение,

* указатель переменная.

38.9.3. Указатель – выражение

Как было отмечено выше, указателем выражение является выражение, значением которого является адрес функции. Приведем пример.

Пусть имеется прототип функции для вычисления квадрата числа типа double. Тогда выражение &sqr - это выражение – указатель. Здесь & - унарный оператор взятия адреса, который и предписывает компилятору вычислить адрес йункции.

38.9.4. Указатель – переменная

Указатель - переменная это переменная, предназначенная для хранения адреса функции. Рассмотрим формат определения такой переменной.

```
<β> (* <α>) (<γ>)
```

* * - оператор разыменования,

α – идентификатор, объявленного указателя,

* β - тип значения, возвращаемого функцией, на которую может быть установлен указатель.
* γ – список типов параметров функции, на которую может быть установлен указатель.

Приведем пример объявления указателя – переменной.

```
double (*pf)(double);
```

Это объявление переменной. Имя переменной pf. значением этой переменной может служить адрес любой функции, которая принимает один параметр типа double и которая возращает значения типа double.

Учитывая изложенное выше, можно сказать, что определение указателя на функцию задает множество функций, с которыми он может использоваться. Указатель – переменная pf, объявленная выше, может работать, например, со следующими функциями.

```
double sqr(double x)
{
   return x * x;
}

double cube(double x)
{
   return x * x * x;
}
```

Вернемся к объявлению указателя – переменной pf. Рассмотрим вопрос о назначении используемых в нем скобок. Здесь их две пары.

```
double (*pf)(double);
```

Вторая пара скобок – это оператор функция. Возникает вопрос: «Какую роль играет первая пара скобок?». С целью разобраться в этом вопросе на

время удалим эту пару скобок. Тогда рассматриваемое выражение примет следующий вид.

```
double *pf(double);
```

Полученное выражение является объявлением функции, которая принимает один параметр типа double и возращает указатель типа double*. Для того чтобы убедиться в этом вспомним, что в синтаксис объявлений в языке Си заложен синтаксис выражений. Для определения назначения имени pf в рассматриваемом выражении следует рассмотреть подвыражение *pf(double). Здесь к имени pf применяются два оператора: * и () . Трактовка имени pf зависит от приоритетов этих операторов. Если обратиться к таблице приоритетов операторов, то можно убедиться в том, что оператор функция () имеет более высокий приоритет по сравнению с оператором разыменования *. Отсюда следует, что оператор () сильнее связывает имя pf по сравнению с оператором *. Итак, в новой редакции pf является именем функции, а не именем указателя.

Вернемся теперь к исходному объявленияю

```
double (*pf)(double);
```

Первые скобки в этом выражении слияют на порядок группировoвки операторов и операндов. Теперь имя pf становится именем указателя.

38.9.5. Инициализация указателя на функцию

Указатель – переменная может быть инициализирована во время ее объявления либо нулевым указателем, либо адресом функции того типа, на который может быть установлен указатель.

Пусть имеются две функции, имеющие следующие прототипы.

```
double sqr(double x);
double cube(double x);
```

Объявим три указателя на функцию:

```
double (*pf1)(double) = &sqr;
double (*pf2)(double)= cube;
double (*pf3)(double)= 0;
```

Синтаксис, с которым объявлены первые две переменные (pf1 и pf2), несколько отличаются. Синтаксис, с которым объявлена первая переменная (pf1) не вызывает сомнений в корректности. Справа от знака присваивания находится выражение, значением которого является адрес функции. Могут возникнуть сомнения в корректности второго объявления, где объявлена переменная pf2. В этом объявлении в качестве инициализатора используется имя функции, а не ее адрес. Однако этот код нормально компилируется. Дело в том, что языке Си всегда в том коде, где ожидается указатель на функцию, а встречается имя функции, выполняется автоматическое преобразование имени функции в указатель.

Замечание. Операция инициализации выполняется и при использовании казателя на функцию в качестве формального параметра функции.

38.9.6. Операции с указателями на функцию

С указателями на функцию можно выполнять следующие операции.

1. Вызов функции, на которую установлен указатель.
2. Присваивание однотипных указателей.
3. Указатель можно использовать в качестве формального параметра функции.
4. Функция может возвращать значение, имеющее тип указателя на функцию.

Вызов функции через указатезатель на функцию.

Вначале рассмотрим вызов функции с помощью указателя на функции. Существуют две формы вызова через указатель на функцию.

Первая форма.

```
(*<α>)(<β>)
```

Вторая форма

```
<α>(<β>)
```

Здесь α имя указателя на функцию, а β – список фактических параметров вызываемой функции. Первая форма строго следует синтакису языка Си, а во второй форме выполняется автоматическое разыменование указателя на функцию. В качестве примера рассмотрим следующий программный код.

```
double (*pf)(double) = &sqr;
printf("%0.4g\n", (*pf)(3)));
printf("%0.4g\n",  pf(4));
```

В результате выполнения первого вызова функции printf() на экране появится число 9, а при выполнении второго вызова функции printf() на экране появится число 16.

Присваивание указателей на функцию.

Любому указателю на функцию может быть присвоено значение нулевого указателя. Например.

```
double (*pf)(double);
pf = 0;
```

Можно присваивать указателю на функцию значение другого однотипного указателя на функцию. Например.

```
double (*pf1)(double) = sqr;
double (*pf2)(double) = cube;
double (*pf3) (double, double);
pf1 = pf2; //Нормально.Указатели pf1 и pf2 имеют один и тот же
           //тип
pf1 = pf3; //Ошибка. Указатели pf1 и pf3 имеют разные типы.
```

Для иллюстрации использования указателей на функцию в качесте параметров функций рассмотрим два примера

38.9.7. Табулирование произвольной функции одного переменного

Постановка задачи

Написать функцию пользователя, предназначенную для табулирования произвольной математической функции одного переменного. Табулирование

следует выполнять в “n” равноотстоящих точках, начиная от x = xнач вполоть до x = xкон.

Решение

Программа, предназначенная для демонстрации решения поставленной задачи, состоит из трех файлов:

- Файл `bibl.c`, который содержит определение функции `table()`, выполняющей табулирование, и определения табулируемых функций: `sqr()` и `cube()` .
- Файл `bibl.h`, который содержит объявления используемых функций.
- Файл `main.c`, который содержит тестовую программу.

```
/*Файл bibl.h                              */
double sqr(double x);
double cube(double x);
/* Объявления других табулируемых функций */

void table(double (*pf)(double x), double xn, double xk,
                int n);


/*Файл bibl.c                              */
#include <stdio.h>
/*
   Вычисление квадрата аргумента. Первая из табулируемых
   функций.
*/
double sqr(double x)
{
   return x * x;
}

/*
   Вычисление третьей степени аргумента. Вторая из
табулируемых
   функций
*/
double cube(double x)
{
   return x * x * x;
}

/* Определения других табулируемых функций*/

/*
   Функция,выполняющая табулирование. Через первый параметр
   этой функции передается табулируемая функция
*/
```

```
void table(double (*pf)(double x), double xn, double xk,
                int n)
   {
      double x, dx;
      int i;
      printf("%5s%10s%10s\n", "НОМЕР", "АРГУМЕНТ",
"ФУНКЦИЯ");
      x = xn;
      dx = (xk - xn) / (n - 1);
      for(i = 1; i <= n; i++)
      {
         printf("%5d%10.2f%10.2\n", i, x, f(x));
         x += dx;
      }
   }


/*Файл main.c                              */
#include"bibl.h"
int main(void)
{
   double xn, xn:
   int n;
   printf("Количество расчетных точек=")
   scanf("%d", &n);
   printf("Начальное значение аргумента=");
   scanf("%lf", &xn);
   printf("Конечное значение аргумента=");
   scanf("%lf", &xк);

   /* Табулирование функции y = x2 */
   table(sqr, xn, xk, n);

   /* Табулирование функции y = x3 */
   table(cube, xn, xk, n);

   return 0;
}
```

38.9.8. Использование стандартной функции qsort

Функция qsort() предназначена для сортировки элементов массивов в порядке возрастания. Название qsort связано с первоначальной реализацией этой функции, в которой использовался алгоритм быстрой сортировки. В текущем стандарте не оговаривается, какой алгоритм сортировки должен использоваться. Рассмотрим вначале прототип этой функции.

```
#include <stdlib.h>
void qsort(void* base, size_t nmemb, size_t size,
           int (* compar)(const void* p1, const void*p2));
```

Здесь:

- base – указатель на первый элемент массива, содержащего сортируемые элементы,
- nmemb – количество сортируемых элементов,
- size – размер памяти, занимаемый одним элементом,
- compar – указатель на функцию, выполняющую сравнение сортируемых элементов.

Остановимся отдельно на функции, которая должна выполнять сравнение. Эта функция должна получать два параметра, которые указывают на объекты, подлежащие сравнению. Эта функция должна возвращать отрицательное число, если первый элемент меньше второго, положительное число, если первое элемент больше второго и, наконец, возвращает нулевое число – в случае равентсва сравниваемых элементов.

Отметим универсальность функции qsort(). С помощью этой функции можно сортировать массивы с произвольным типом элементов массива. Универсальность функции обеспечивается использованием нетипизированных указателей.

Рассмотрим пример использования функции qsort().

Постановка задачи. Сортировать в порядке возрастания элементы массива ar. Массив состоит из n элементов типа double.

Решение.

Напишем функцию пользователя compar (имя функции может быть и другим). Интерфейс разрабатываемой функции задается последним (четвертым) параметром функции qsort() и имеет следующий вид.

```
int compare(const void* p1,const void* p1);
```

Для упрощения алгоритма введем две вспомогательные переменные x1 и x2 типа double. Эти переменные должны представлять в разрабатываемой функции сравниваемые элементы массива. Для нахождения значений этих переменных необходимо нетипизированные указатели p1 и p2 преобразавать к указателям на тип сравниваемых элеменов (double*), а затем выполнить их разыменование. В результате можно получить следующее определение функции.

Определение функции сравнения

```
int compare(const void* p1, const void* p2)
{
    double x1 = *(double*)p1;
    double x2 = *(double*)p2;
    if(x1 < x2)
       return -1;
    if(x1 > x2)
       return 1;
    return 0;
}
```

Клиентский код.

```
int compare(const void* p1, const void* p2);
int main(void)
{
```

```
    double ar[] = {25, 4, 8, -1};
    int n = sizeof(ar) / sizeof(*ar);
    qsort(ar, n, sizeof(*ar), compar);
    for(int i = 0; i < n; i++)
       printf("%0.4g", ar[i]);
    return 0;
}
```

Результат выполнения программы:
-1
4
8
25

39. Указатели и массивы

Между указателями и массивами в языке Си имеется тесная связь. Эта связь проявлется в сдудующем:

- имя массива является константным укзателем.
- массив, передаваемый в качестве аргумента в вызове функции, «вырождается» в указатель.
- к любому указателю синтаксически применима операция индексирования (семантически это оправдано только для указателей, работающих с массивами).

40. Строки

На практике часто возникают задачи, связанные с обработкой текста. Текстом называется упорядоченная последовательность строк. Это положение приводит к необходимости рассмотрения вопроса о преставлении строк и способах их обработки.

Строкой в программировании называется последовательность символов переменной длины. Для работы с отдельными символами в языке Си используется тип char. А как обстоит дело со строками в языке Си? Строкового типа в языке Си нет. Концепция работы со строками в языке Си базируется на двух положениях:

1. Строки в языке Си хранятся в массивах, элементы которых имеют тип char.
2. Строки в символьном массиве хранятся в виде последовательности символов, дополненной нуль-символом.

Нуль-символ – символ, код которого равен нулю (‘\0’).

40.1. *Строковый литерал*

Строковый литерал – это последовательность символов, заключенная в кавычки. Например: “”Hello.

Какой тип имеет строковый литерал? Строковый литерал “Hello” имеет тип char[6]. В общем случае строковый литера, состоящий из “n” символов

имеет тип char[n + 1]. Наличие слагаемого 1 учитывает необходимость хранить нуль-символ.

Таким образом, символьный литерал – это разновидность символьного массива. Возможна т другая трактовка символьного литерала, в соответствии с которой строковый литерал является указателем. Например, char * p = “Hello”; В этом примере указатель p инициализируется адресом строкового литерала.

Часто текст строкового литерала не помещается на одной строке экрана. Возможны два способа продолжения ее на следующей строке. Рассмотрим вначале первый способ. Он состоит в использовании в конце текущей строки символа обратная косая черта. Например

```
printf(“Эта длинная строка \
введена на нескольких\
строчках, которые заканчиваются \
обратной чертой”);
```

Отступы на строках продолжения не следует делать, а если они сделаны, то пробелы, их представляющие будут внесены в строку.

Второй способ продолжить строку состоит в следующем: необходимо в конце текущей строки поставить символ “ (т. е.) закрыть строку, а на следующей строке экрана открыть новую строку. Например:

```
printf(“Начало строки”
       “продолжение строки”);
```

40.2. *Символические строковые константы*

Символическая строковая константа – это константа, имеющая имя. Возможны два способа объявления символических констант:

- использование директивы препроцессора define
- использование ключевого слова const.

Примеры.

```
#define MESSAGE “Ошибка ввода”
const char message2[] = “Ошибка ввода”ж
```

40.3. *Строковые переменные*

Строковая переменная – это идентификатор массива, элементы которого имеют тип char. Этот массив должен содержать текст строки, в конце которой должен находиться нуль-символ. Рассмотрим следующий фрагмент программного кода.

```
#define MLEN 128
char str1[MLEN];
int main(void)
{
   char  str2[MLEN];
   // …
```

```
}
```

Здесь str1 является пустой строкой. Дело в том, что str1 является глобальной переменной, а все глобальные переменные инициализируются нулями. С другой стороны str2 – локальная переменная. Такая переменная содержит “мусор”, среди которого может нуль-символ отсутствовать.

40.4. *Инициализация строковых переменных*

Возможны два способа инициализации строковой переменной:

- строковым литералом.
- символьным массивом

Ограничимся рассмотрением наиболее широко применяемого способа инициализации с помощью строкового литерала. Пример.

```
char str3[10] = “Hello”;
```

В результате выполнения этой строки программного кода будет выделено 10 байт памяти. В первых шести байт этой памяти будет записан текст константы, а затем нуль-символ.

Отметим, что при наличии инициализатора требуемый объем памяти можно не указывать. Пример.

```
char str4[10] = “Hello”;
```

В этом случае будет выделено шесть байт памяти.

Следует отметить, что инициализировать строковую переменную можно только в момент ее определения. Ниже следующий код содержит синтаксическую ошибку.

```
char str4[10];
str4 = “Hello”;  // Синтаксическая ошибка
```

40.5. *Операции со строковыми переменными*

Со строковыми переменными можно выполнять только поэлементные операции. Все остальные операции:

- присваивание (копирование),
- сравнение,
- поиск подстрок и др.

следует выполнять либо с помощью библиотечных функций, либо разрабатывать собственную (пользовательскую) функцию.

Для работы с библиотечными функциями, предназначенными для работы со строками, следует подключать заголовочный файл string.h.

40.6. *Ввод строк*

Имеется ряд библиотечных функций, помощью которых можно выполнять ввод строк:

- scanf(),
- gets(),
- fgets().

Начнем рассмотрение с функции scanf().

40.6.1. Функция scanf()

Особенность функции scanf() состоит в том, что с ее помощью нельзя вводить строки, содержащие пробелы. Эта функция может использоваться только для ввода отдельных слов. Приведем пример.

```
#define MLEN 129
#include<stdio.h>

int main(void)
{
   char st[MLEN];
   prinf(“Enter a string: ”);
   scanf(“%s”, st);
   printf(“%s”, st);
   /*   */
   return 0;
}
```

Следует учитывать, что переменная st является указателем и при ее передаче в функцию scanf() оператор взятия адреса & не нужен. Протокол работы с программой, приведенной выше, будет иметь следующий вид:

```
Enter a string: Hello, world<Enter>
Hello,
```

Выше с помощью подчеркивания выделен ввод пользователя. Из протокола видно, что введенным оказалось только первое слово (Hello,).

Другой особенностью функции scanf() является возможность ограничения количества вводимых символов. Для этого в спецификации преобразования функции следует использовать дополнительный параметр, как это показано на примере, приводимом ниже:

```
#define MLEN 129
#include<stdio.h>

int main(void)
{
   char st[MLEN];
   prinf(“Enter a string: ”);
   scanf(“%5s”, buffer);
   printf(“%s”, buffer);
   /*   */
   return 0;
}
```

Протокол работы с программой, приведенной выше, будет иметь следующий вид:

```
Enter a string: 1234567890<Enter>
12345
```

В этом примере спецификация преобразования содержит дополнительный параметр в виде числового литерала (5). В рассматриваемом случае воспринимается не более 5 вводимых символов.

Отметим, что при организации ввода строк программист обязан предусмотреть выделение памяти для вводимой строки. Ниже приведен пример, в котором отсутствует выделение памяти для вводимой строки.

```
#include<stdio.h>

int main(void)
{
   char *s;
   prinf("Enter a string: ");
   scanf("%5s", s);  /* Ошибка*/
   printf("%s", s);
   /*   */
   return 0;
}
```

Ошибка состоит в том, что используется неинициализированный указатель s.

40.6.2. Опасная функция gets()

Прототип этой функции имеет следующий вид:

```
#include<stdio.h>
char* gets(char* str);
```

Рассматриваемая функция читает символы, вводимые с клавиатуры в символьный массив, на который установлен указатель `str`. Это чтение выполняется до тех пор, пока не встретится либо символ "новая строка" (\n), либо конец файла. После записи последнего прочитанного символа в массив `str` добавляется нуль – символ. Если встречается символ "новая строка", то он отбрасывается. Если выполнение функции завершено успешно, то она возвращает указатель `str`. Функция `gets()` вернет значение `NULL` в том случае, когда при достижении конца файла ни один из символов не оказался прочитанным, причем содержимое массива, на который установлен указатель `str`, останется без изменения. Значение `NULL` является возвращаемым значением и в том случае, когда будет обнаружена ошибка чтения, но в этом случае содержимое массива считается не определенным. Приведем пример применения функции `gets()`. В этом примере вначале запрашивается имя пользователя, а затем компьютер выводит приветствие.

```
/* Greeting.c */
#include<stdio.h>
#include<string.h>
#define MAX 81

int main(void)
{
   char name[MAX];
   char out[MAX] = "Привет, Вам ";
   puts("Введите Ваше имя");
   gets(name);
```

```
    printf("Привет, Вам %s", name);
    puts(out);
    return 0;
}
```

Протокол работы с программой greeting имеет следующий вид

```
Введите Ваше имя
Иван
Привет, Вам Иван
```

Существенным недостатком рассматриваемой функции является возможность переполнения массива, на который установлен указатель str. В связи с этим функцию gets() не рекомендуют использовать в коммерческих приложениях. Для этих целей можно воспользоваться функцией fgets().

40.6.3. Использование функции fgets()

В связи с тем, что функция gets() пользуется плохой репутацией, в качестве альтернативы этой функции предлагается использовать функцию файлового ввода-вывода fgets().Для обсуждения возможности использования этой функции для консольного ввода приведем прототип функции fgets():

```
#include<stdio.h>
char*  fgets(char* str, int n, FILE* stream);
```

Рассматриваемая функция имеет два дополнительных параметра, которые отсутствуют у функции gets(). Первый из дополнительных параметров (**int** n) служит для ограничения количества символов, которые могут быть прочитаны в массив str из буфера клавиатуры. Второй дополнительный параметр (FILE stream) при использовании функции fgets() определяет файл, с которым должна работать эта функция. Для консольного ввода достаточно в ее вызове в качестве параметра stream взять имя стандартного потока, предназначенного для работы с клавиатурой (stdin).

Функция fgets() в форме, предназначенной для ввода с клавиатуры, позволяет записать в массив, на который указывает указатель str, не более n – 1 символа. Ввод прекращается, как только встретится символ новой строки (который записывается в массив) или символ конца файла. За последним введенным символом добавляется нуль-символ. В случае успешного завершения функция вернет указатель строку str. Если прочитан конец файла, а ни один символ не был введен, то содержимое массива оказывается неизменным, а функция вернет значение NULL. Если во время ввода имела место ошибка, то функция вернет значение NULL, а содержимое массива str оказывается неопределенным. Приведем пример.

```
#include<stdio.h>
#include<string.h>
#define MAXSIZE 81

int main(void)
{
    char buf[MAXSIZE];
    char* s = NULL;
```

```
    fgets(buf, sizeof(buf), stdin);
    s = strchr(buf, '\n');/* Ищем символ '\n' в прочитанной
                             строке */
    if(s != NULL)
      *s = '\0';          /* Запись символа '\0'вместо
                             символа '\n' */
    return 0;
}
```

С целью приблизить работу функции fgets() к работе функции gets(), которую она призвана заменить, в рассматриваемом примере добавлен программный код, удаляющий из массива, используемого для ввода строки (buf), символ новой строки (\n). Для этой цели используется функция strchr() и инструкция if.

40.7. *Вывод строк*

Для ввода строк с клавиатуры можно воспользоваться следующими библиотечными функциями:

- `printf()`,
- `puts()`,
- `fputs()`.

Начнем рассмотрение с функции printf().

40.7.1. Функция printf()

Строки могут выводиться с помощью библиотечной функции printf(). Для вывода строкового литерала достаточно поместить его текст в форматной строке вызова рассматриваемой функции. Пусть необходимо вывести стандартное приветствие. Это можно выполнить с помощью следующего вызова функции printf():

```
printf("Hello, world");
```

В более сложных случаях следует при выводе строк использовать спецификацию типа s. В качестве примера приведем вывод заголовка таблицы:

```
printf("%5s%10s%10s", "НОМЕР", "АРГУМЕНТ", "ФУНКЦИЯ");
```

40.7.2. Функция puts()

Объявление функции puts() имеет следующий вид

```
#include<stdio.h>
int puts(const char* str);
```

Функция puts() выводит на экран дисплея строку, на которую установлен указатель str. Нуль – символ этой строки преобразуется в символ новой строки, который выводится на экран. Последнее приводит к тому, что после вывода строки str курсор перейдет на начало новой строки экрана.

При успешном выполнении функция puts() возвращает неотрицательное число, а в случае сбоя – значение EOF.

Рекомендуется использовать функцию puts() вместо printf() в тех случаях, когда необходимо вывести отдельное сообщение для пользователя, которое не сопровождается выводом и вводом данных.

Приведем пример применения функции puts().

```
/* Демонстрация вывода строк с применением функции puts()*/
/*                      Файл puts.c                        */
#include<stdio.h>
int main(void)
{
    char*  msg1   = "Не";
    char*  msg2   = "применяйте";
    char*  msg3   = "функцию";
    char*  msg4   = "gets()";
    char*  msg5   = "в коммерческих";
    char*  msg6   = "приложениях!!!";
    puts(msg1);
    puts(msg2);
    puts(msg3);
    puts(msg4);
    puts(msg5);
    puts(msg6);
    return 0;
  }
Результат выполнения программы будет иметь следующий вид:
Не
применяйте
функцию
gets()
в коммерческих
приложениях!!!
```

40.7.3. Функция fputs()

Функция fputs() предназначена для работы с файлами. Она не имеет особых преимуществ перед функцией puts(), которая используется для консольного вывода строк. Ее целесообразно применять совместно с функцией ввода строк fgets(). Функция fgets() имеет очевидные преимущества перед функцией gets(), специально предназначенной для консольного ввода строк. Это и делает оправданным рассмотрение этой функции.

Прототип рассматриваемой функции имеет следующий вид:

```
#include<stdio.h>
int fputs(const char*s, FILE* stream);
```

Рассматриваемая функция имеет один дополнительный параметр, которого нет у функции puts().Этот дополнительный параметр (FILE stream) при использовании функции fgets() определяет файл, с которым должна работать эта функция. Для консольного ввода достаточно в ее вызове в качестве параметра stream взять имя стандартного потока, предназначенного для работы с клавиатурой (stdout).

Эта функция (в том случае, когда она вызвана со вторым параметром, равным stdout), выводит на экран дисплея строку символов, на которую установлен указатель s. При этом нуль-символ, которым заканчивается выводимая строка отбрасывается. В отличие от puts() функция fputs() не дополняет выводимую строку символом “новая строка”. Если во время вывода строки имеет место ошибка, то функция fputs() вернет значение EOF. При успешном завершении рассматриваемой функции возвращается неотрицательное число.

40.8. *Библиотечные функции для обработки строк*

40.8.1. Функция strlen()

Эта функция предназначена для определения длины строки. Ее прототип имеет следующий вид:

```
#include<string.h>
size_t strlen(const char* str);
```

Тип size_t является разновидностью целочисленного типа. Функция strlen() возвращает длину строки, на которую установлен указатель str, причем строка должна заканчиваться “нуль – символом”. “Нуль – символ” во время определения длины строки не учитывается. Пример применения функции strlen().

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

int main(void)
{
    clrscr();
    char str[81];
    printf(“Введите строку:”);
    gets(str);
    printf(“Длина введенной строки =%d”, strlen(str));
    getch();
    return 0;
}

Протокол работы с программой имеет следующий вид.
Введите строку:Это строка

Длина введенной строки=10
```

40.8.2. Функции strcpy() и strncpy()

Прежде всего, следует отметить, что в языке Си отсутствуют встроенные средства для копирования строк. Применять для этой цели оператор присваивания нельзя. При компиляции приведенного ниже кода будет выдано сообщение об ошибке.

```
/*       Программный код, содержащий ошибку    */
```

```
char str1[30] = "Hello";
char str2[30];
str2 = str1;   /* Недопустимый код, т.к. str2 – константный
  указатель   */
```

Для копирования строк в языке Си следует использовать библиотечные функции strcpy() и strncpy(). Обращает на себя похожесть имен этих функций. В соответствии с принципом образования имен, принятым в библиотеке string это означает, что функция strncpy() имеет дополнительный параметр n.

Объявления функций strcpy() и strncpy() имеют следующий вид:

```
#include<string.h>
char* strcpy(char* out_str, const char* in_str);
char* strncpy(char* out_str, const char* in_str, size_t n);
```

Обе функции (strcpy() и strncpy) копируют содержимое строки in_str в строку out_str. Параметр in_str должен указывать на строку, которая заканчивается нуль - символом. До вызова рассматриваемых функций необходимо выделить память для хранения новой строки. Функции strcpy() и strncpy() эту память не выделяют. Обе функции возвращают значение указателя out_str. Обе функции заканчивают копирование в том случае, когда в строке str_in встречается нуль – символ. Функция strncpy() выполняет копирование более осторожным образом. Это связано с наличием у этой функции третьего параметра (параметр n), который ограничивает количество копируемых символов. Количество символов, которые могут быть скопированы функцией strncpy() не может быть больше n. Заметим, что в предельном случае, когда в скопированных n символах строки in_str не встретился нуль \ символ, то выходная строка не будет заканчиваться нуль – символом. Если массивы in_str и out_str перекрываются поведение функции strcpy() не определено.

В следующем фрагменте кода строка Hello копируется в строку str.

```
char str[81];
strcpy(str, "Hello");
```

Типичной ошибкой при работе с функцией strcpy() является передача ей неправильного указателя на строку str_in. Например, некорректным оказывается следующий фрагмент кода:

```
void foo()
{
   char str1[25] = "Hello";
   char* str2;
   strcpy(str1, str2);
   /* другой код         */
}
```

Ошибка в приведенном выше коде состоит в отсутствии инициализации указателя str2.

40.8.3. Функции strcat() и strncat()

Для целей объединения строк можно использовать две функции: strcat() и strncat(). Вторая из этих функций (strncat()) в отличие от первой

ограничивает количество символов, добавляемых в память, в которой происходит объединение строк.

Объявления рассматриваемых функций имеют следующий вид:

```
#include<string.h>
char* strcat(char* out_str, const char* in_str);
char* strncat(char* out_str, const char* in_str,
              size_t n);
```

Функция strcat() присоединяет содержимое строки in_str к строке out_str. Параметр in_str должен указывать на строку, которая заканчивается нуль - символом. Конечный "нуль – символ", первоначально завершающий строку out_str, перезаписывается первым символом строки in_str. Функция strcat() возвращает значение указателя out_str. Если массивы in_str и out_str перекрываются поведение функции strcat() не определено.

В следующей программе две строки читаются с клавиатуры, затем объединенная строка выводится на экран дисплея.

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

int main(void)
{
   char buf1[64], buf2[64];
   gets(buf1);
   gets(buf2);
   strcat(buf1, buf2);
   printf(buf1);
   return 0;
}
```

Перейдем к рассмотрению функции strncat(). Эта функция присоединяет не более n символов строки in_str в конец строки out_str. Если "нуль – символ" строки in_str достигнут раньше, чем будут прочитаны n символов строки in_str, то этот символ копируется, и процесс присоединения заканчивается. Если среди скопированных n символов не встретился нуль – символ, то он добавляется в строку in_str и процесс присоединения на этом заканчивается. В этом случае в выходную строку записывается n + 1 символ. Если в вызове функции n или отрицательно, то функция не изменяет выходную строку out_str.

40.8.4. Функция strcmp()

Объявление этой функции имеет следующий вид

```
#include<string.h>
int strcmp(const char* in_str1, const char* in_str2);
```

Функция strcmp() выполняет так называемое лексографическое сравнение строк. Функция возвращает нулевое значение, если строки совпадают. Функция возвращает положительное значение, если строка in_str1 > in_str2. Наконец, функция возвращает отрицательное значение, если in_str1 < in_str2. Строка in_str1 считается больше строки in_str2, если первый

несовпадающий ее символ имеет код, превышающий код соответствующего ему символа строки in_str2.

Ниже приводится пример применения функции strcmp().

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

int main(void)
{
   char name[32] = "Tom";
   /* Выводится положительное число */
   printf("%d\n", strcmp(name, "Alic");   /*

   /* Выводится отрицательное число */
   printf("%d\n", strcmp(name, "Victor");

   /*       Выводится нуль          */
   printf("%d\n", strcmp(name, "Tom");
   return 0;
}
```

40.9. *Массивы строк*

Ограничимся рассмотрением массива строк, построенного на основе двумерного символьного массива с фиксированными размерами. Правила работы с массивом строк рассмотрим на простом примере.

Постановка задачи. Необходимо ввести с клавиатуры “n” строк(n <= 20), длина которых не превышает 80 символов. Вводимые строки записать в строковый массив. После завершения ввода выполнить вывод строк, сохраненных в массиве. Ввод и вывод организовать с помощью функций пользователя.

Решение.

```
// File main.c
#include <stdio.h>
#include "Strings.h"
// File main.c
#include <stdio.h>
#include "Strings.h"
int main(void)
{
    char strs[MSTRS][MLEN + 1];
    int n;
    printf("n=");
    scanf("%d%*c", &n);

    input_arr_strs(strs, n);
    printf("\n\n");
    output_arr_strs(strs, n);
    return 0;
}

// File Strings.h
```

```
#ifndef STRINGS_H_INCLUDED
#define STRINGS_H_INCLUDED
#define MSTRS  20
#define MLEN   80
void input_arr_strs(char ar[][MLEN + 1], int n);
void output_arr_strs(const char ar[][MLEN + 1], int n);

#endif // STRINGS_H_INCLUDED

// File Strings.c
#include "Strings.h"
#include <stdio.h>
void input_arr_strs(char ar[][MLEN + 1], int n)
{
    for(int i = 0; i < n; i++)
    {
        printf("#%d->", i + 1);
        gets(ar[i]);
    }
}

void output_arr_strs(const char ar[][MLEN + 1], int n)
{
    printf("%5s\t%-50s\n", "I", "STS");
    for(int i = 0; i < n; i++)
    {
        printf("%5d\t%-50s\n", i + 1, ar[i]);

    }
}
```

```
E:\ksv\2012_2013\IP\InOutArrStrs\bin\Debug\InOutArrStrs.exe
n=3
#1->11     vvv    bbb
#2->2      xxx    ddd
#3->3      sss


    I   STS
    1   11     vvv    bbb
    2   2      xxx    ddd
    3   3      sss

Process returned 0 (0x0)   execution time : 44.406 s
Press any key to continue.
```

41. Структуры

Структура – это вторая разновидность агрегатных типов данных языка Си. Структура – совокупность величин, объединенных одним именем. Компоненты структуры называют:

- элементами,
- полями,
- членами.

От уже рассмотренных выше массивов структура отличается в следующих отношениях:

- отсутствует упорядоченность между компонентами,
- в отличие от массивов компоненты структуры могут иметь разный тип.
- другой способ обращения к компонентам; к компонентам структуры следует обращаться по имени их полей (а не индексу как это имеет место в случае массивов).

Обратим внимание на неоднозначность термина структура. В языке Си под этим термином наряду со структурой как типом данных понимают переменную, имеющую тип структуры.

41.1. *Объявление структур*

Общий формат, позволяющий наряду с типом объявлять и переменные, имеет следующий вид:

```
struct[<α>]
{
    <β>
}[γ];
```

Здесь:

- struct – ключевое слово языка Си,
- α - необязательный элемент, называемый тегом,
- β - список объявлений полей,
- γ - список переменных, имеющих тип объявляемой структуры.

Элементы объявления α и γ могут отсутствовать. Список γ подчиняется синтаксису обычных переменных. Переменные, включенные в этот список, могут использоваться только в составе структурной переменной. Определение такого формата выполняют две функции:

- объявляют структуру как тип данных.
- определяет переменные, имеющие этот тип.

Рассмотрим ряд примеров.

Пример 1. Совместное определение типа и переменных

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
```

```
} employee, worker;
```

Тегом определяющей здесь структуры является здесь идентификатор person_salary. Структура имеет два поля. Первым полем является строка fio, предназначенная для хранения фамилии, имени и отчества. Второе поле имеет тип double и предназначено для хранения сведений о заработке. Здесь же определены две переменные, имеющие структурный тип:

- employee (служащий),
- worker (рабочий).

Здесь необходимо следует иметь в виду следующее. В программе, написанной на языке C++, тег является именем типа. В языке Си в качестве имени типа следует использовать конструкцию struct <α (тег)> .

Пример 2. Раздельное определение типа и переменных

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};
//…
struct person_salary employee, worker;
```

Пример 3. Компонент тег можно опускать

```
#define MLEN 81
struct
{
   char    fio[MLEN];
   double  salary;
} employee, worker;
```

В этом примере определен так называемый анонимный тип. У этого типа нет тега. Поэтому на него нельзя ссылаться в другом программном коде. Польза от этого кода в определенных здесь двух переменных employee и worker.

Пример 4. Определение типа, переменной и указателя на структуру.

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};
//…
struct person_salary employee;
struct person_salary* p = &employee;
```

Здесь дополнительно объявлен и инициализирован указатель на структуру типа struct_salary.

41.2. *Обращение к полям структуры*

Для обращения к отдельным полям структуры в языке Си служат два оператора:

- .
- ->

Оператор . используется в том случае, когда известна переменная, имеющая тип структуры. Оператор -> при работе со структурой через указатель. Оба оператора относится к категории бинарных операторов. Эти операторы имеют следующий формат:

- <α1> . <β>.
- <α2> -> <β>.

Здесь α1 – переменная, имеющая тип структуры, α2 – указатель, установленный на объект типа структуры, а β - поле структуры.

Важно отметить, что результат вычисления рассматриваемых выражений имеет все свойства своего правого операнда. К ним относятся значение и тип результата. Рассмотрим следующий пример. Пусть имеется следующий программный код.

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};
//…
struct person_salary employee;
struct person_salary* p = &employee;
```

Тогда можно написать следующий программный код, содержащий обращения к полям объекта employee.

```
strcpy(employee.fio, "Petrov Ivan Ivanovich");
puts(p->fio);
p->salary = 10000;
printf("%10.3f\n", employee.salary);
```

41.3. *Инициализация структур*

Структура может быть инициализирована во время своего определения. Ограничимся примером.

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};

int main(void)
{
    struct person_salary new_person = {
                                          "Ivanov I.I",
```

```
                                        25000
                                     };
     // …
}
```

41.4. *Операции над структурами*

Со структурой как со структурной переменной можно выполнять следующие операции:

- присваивание,
- передавать в функцию по значению,
- передавать в функцию с использованием указателя,
- структуру можно использовать как значение, возвращаемое функцией.

Остальные операции со структурами выполняются по компонентно.

Пример 1. Присваивание структур.

Постановка задачи. Имеются две структуры person1 и person2. Требуется скопировать поля структуры person1 в структуру person2.

Решение

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};

int main(void)
{
    struct person_salary person1 = {
                                         "Ivanov I.I",
                                          25000
                                     };

     struct person_salary person2 = {
                                          "Petrov P.P",
                                          17000

                                     };
      person2 = peson1;
      // …
}
```

После выполнения этого фрагмента программного кода все поля структуры person1 оказались скопированными в структуру person2.

Пример 2. Возврат функцией значения типа структуры.

Постановка задачи. Заданы значения полей. Требуется сформировать структуру (построить структурную переменную) структуру.

Решение

```
#define MLEN 81
struct person_salary
```

```
{
   char    fio[MLEN];
   double  salary;
};

// Прототип функции, которая формирует структуру
struct person_salary form_struct(const char* fio, double
                                 salary);

int main(void)
{
   struct person_salary = form_struct("Petrov P.P.", 20000);
   //…

}

struct person_salary form_struct(const char* fio, double
                                 salary)
{
     struct person_salary temp;
     strcpy(temp.fio, fio);
     temp.salary = salary;
     return temp;
}
```

Пример 3. Использование указателей для передачи структур в качестве аргументов функций.

Постановка задачи. Выполнить обмен значений двух структур.

Решение

```
#define MLEN 81
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};


void swap_struct(struct person_salary* p1,
                 struct person_salary* p2);

int main(void)
{
    struct person_salary person1, person2;
    //…
    swap_struct(&person1, &person2);
    //…
}

void swap_struct(struct person_salary* p1,
                 struct person_salary* p2)
{
     struct person_salary temp = *p1;
```

```
    *p1                       = *p2;
    *p2                       = temp;
}
```

41.5. *Массивы структур*

Массивы структур широко используются на практике.

41.5.1. Объявление массива структур

При определении массива структур используются обычные правила определения массивов:

<α> <β>[<γ>];

Здесь α – имя типа элементов, β - имя массива, γ – размер массива.

Приведем пример.

```
#define MLEN  81
#define MSIZE 20
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};

struct person_salary ar[MSIZE];

// …
}
```

Здесь объявлен массив структур. В массиве можно хранить до 20 структур, имеющих тип struct person_salary.

41.5.2. Идентификация элементов массива структур

При обращении к полям элемента структур могут возникнуть синтаксические сложности. Для их преодоления следует учитывать, что вначале надо получить доступ к элементу массива. Для этого необходимо использовать оператор []. Затем, используя оператор “.”, получить доступ к полю структуры. Дальнейшие действия зависят от типа поля.

Пример.

Пусть имеется массив, объявленный следующим образом.

```
#define MLEN  81
#define MSIZE 20
struct person_salary
{
   char    fio[MLEN];
   double  salary;
};
struct person_salary arк[MSIZE];
```

Рассмотрим выражения, приведенные в таблице, приведенной ниже.

Выражение	Пояснение
arr[0].salary	Обращение к полю salary 1-ой

	структуры в массиве arr
arr[1].fio	Обращение к полю fio 2-ой структуры в массиве arr
arr[1].fio[2]	Обращение к 3-му символу поля fio 2-ой структуры в массиве arr

42. Работа с внешними устройствами

Язык Си не определяет операции ввода – вывода. Такие операции выполняются с помощью библиотечных функций. Библиотечные функции, предназначенные для этих целей можно разделить на две категории:

- потоковые функции,
- низкоуровневые функции.

Потоковые функции нашли более широкое применение на практике по сравнению с низкоуровневыми. Этому способствовало то обстоятельство, что потоковые функции были включены в стандартную библиотеку языка Си. Применение таких функций не ухудшают переносимость программы.

Низкоуровневые библиотечные функции учитывают особенности той среды, в которой будет выполняться программа. Это позволяет повысить их эффективность по сравнению с потоковыми функциями. Недостаток – ухудшению переносимости программы. Ограничимся рассмотрением потоковых функций.

42.1. *Понятие потока*

Поток – это своего рода обобщенное устройство ввода-вывода. Такое понятие введено в рассмотрение для того, чтобы при программировании операций ввода – вывода можно было не учитывать особенности:

- конкретного типа внешнего устройства,
- среды, в которой выполняется программа.

Все потоки ведут себя похожим образом. Работа их не зависит от типа физического устройства.

Различают потоки двух видов:

- текстовые,
- двоичные.

42.1.1. Текстовый поток

Текстовый поток – это последовательность символов, организованная в строки. Каждая строка такого потока должна заканчиваться символом новая строка ‘\n’. В конце последней строки такой символ не является обязательным.

Рассмотрим работу потока в режиме чтения. Организация строки, поступающей на вход потока, может быть любой. Она (организация) определяется внешним устройством. На выходе текстового потока строка гарантировано заканчивается символом новая строка.

При работе потока в режиме вывода картина обратная. На входе потока строка гарантировано заканчивается символом новая строка. Организация строки на выходе потока может быть любой.

В зависимости от среды, в которой выполняется программа, при использовании текстового потока могут иметь место некоторые преобразования. Например, в среде Windows символ новая строка '\n' преобразуется в последовательность символов CR LF.

Таким образом, при использовании текстового потока может и не быть однозначного соответствия между символами, который читаются (записываются) и теми символами, хранятся на внешнем устройстве.

Кроме того, при использовании текстового потока в режиме чтения особым способом обрабатывается символ конца файла. В таблице кодов ASCII этот символ имеет код 0x1A. Эта особенность состоит в том, что вся часть содержимого файла, расположенная справа от этого символа (от конца файла) не читается.

42.1.2. Двоичный поток

Двоичный поток – это последовательность байтов, которая взаимно однозначно соответствует байтам на внешнем устройстве. При этом никаких преобразований символов не происходит, и нет символов, играющих особую роль. Количество символов, которые записываются (читаются), и тех символов, которые находятся на внешнем устройстве одинаково.

42.2. *Файлы*

В языке Си файлом может быть все что угодно, начиная от диского файла и кончая терминалом или принтером. Поток связывается с определенным файлом при выполнении операции открытия. Как только файл будет открыт, можно осуществлять обмен данными между ним и программой.

Связь между файлом и программой разрывается в результате выполнения операции закрытия. При закрытии файла, открытого для вывода, содержимое (если оно есть) связанного с ним потока записывается на внешнее устройство. Этот процесс, который называется дозаписью потока гарантирует, что никакие данные не останутся в буфере.

Если работа программы завершается нормально, то все файлы автоматически закрываются. В случае аварийного завершения работы программы файлы автоматически не закрываются.

42.2.1. Указатель файла

У каждого потока, связанного с файлом, имеется структура, содержащая информацию о файле. Эта структура имеет тип FILE. Этот тип объявлен в заголовочном файле stdio.h. Для каждого файла, с которым она будет работать, должен существовать так называемый указатель файла. Указатель файла – указатель на структуру типа FILE.

Указатель файла определяет конкретный файл и используется связанным с ним потоком при выполнении операций ввода – вывода.

42.2.2. Функция fopen()

Объявление этой функции имеет следующий вид.

```
#include<stdio.h>
FILE* fopen(const char* filename, const char* mode);
```

Эта функция открывает поток и связывает с ним определенный файл. Первый параметр (filename) рассматриваемой функции определяет открываемый файл. Второй параметр функции (mode) задает режим, в котором должен быть открыт файл.

В случае успеха функция вернет указатель типа FILE* на созданную структуру данных. Этот указатель может использоваться файловыми функциями для выполнения операций ввода – вывода. В случае возникновения ошибки функция fopen() вернет нулевой указатель.

В следующем фрагменте программного кода функция используется для открытия файла с именем file1.txt в режиме записи. Предполагается, что файл находится в текущем каталоге (иначе необходимо указывать путь к каталогу). Во фрагменте используется обычный прием, который состоит в проверке значения, возвращаемого функцией fopen().

```
FILE* fp;
if((fp = fopen("file1.txt", "w")) == NULL)
{
   printf("Не могу открыть файл file1.txt");
   exit(1);
}
```

Возможные режимы открытия потоков представлены в приводимой ниже таблице

Режим	Пояснение
r	Открыть текстовый поток и связать его с существующим файлом
w	Открыть текстовый поток и связать его с вновь созданным файлом
a	Открыть текстовый файл в режиме дозаписи. Если файл еще не существует, то он будет создан.
rb, wb и ab	Отличаются соответственно от режимов r, w и a только тем, что открывают двоичный поток.
r+, w+ и a+	Отличаются соответственно от режимов r, w и a только тем, что поток открывается в режиме чтения / записи.
r+b, w+b и a+b	Отличаются соответственно от режимов rb, wb и ab только тем, что поток открываются в режиме чтения / записи.

42.2.3. Функция fclose()

Объявление этой функции имеет следующий вид.

```
#include<stdio.h>
int fclose(FILE* stream);
```

Функция fclose() закрывает поток stream. Перед закрытием потока все связанные с ним буферы сбрасываются. При успешном выполнении функция fclose() возвращает 0, а в случае ошибки – EOF.

42.2.4. Функция feof()

Объявление рассматриваемой функции имеет следующий вид:

```
#include<stdio.h>
int feof(FILE* stream);
```

Функция feof() проверяет, достигнут ли конец файла при выполнении очередной операции чтения. Параметр stream является указателем на поток, с которым работает файл.

Если указатель текущей позиции файла установлен на конец файла, то функция feof() вернет ненулевое значение, в противном случае возвращается нуль.

42.2.5. Стандартные потоки

В начале выполнения программы автоматически открываются три стандартных потока.

Поток	Файловый указатель	Внешнее устройство
Ввода	stdin	клавиатура
Вывода	stdout	Экран дисплея
Ошибок	stderr	Экран дисплея

42.2.6. Классификация функций потокового ввода – вывода

Все функции потокового ввода – вывода можно разделить на четыре категории:

- форматированный ввод – вывод,
- посимвольный ввод – вывод,
- построковый ввод – вывод,
- блоковый ввод – вывод.

42.2.7. Форматированный ввод – вывод

При форматированном вводе - выводе используются следующие функции:

- fprintf() – для записи в файл,
- fscanf() - для чтения из файла.

Функция fprintf() объявлена следующим образом:

```
#include<stdio.h>
int fpintf(FILE* stream, const char* format[, аргумент, …] );
```

Рассматриваемая функция является обобщенным вариантом функции printf(). Здесь появился дополнительный параметр stream, которого нет в функции printf. Параметр stream является указателем на поток, который должен быть предварительно открыт функцией fopen().

Функция fprintf() возвращает число выведенных символов или отрицательное число, если произошла ошибка вывода.

Функция fscanf() объявлена следующим образом:

```
#include<stdio.h>
int fscanf(FILE* stream, const char* format[, указатель, …]
);
```

Эта функция в свою очередь является обобщенным вариантом функции scanf(). Здесь также появился дополнительный параметр stream – указатель на поток.

В результате своего выполнения функция fscanf() возвращает количество просмотренных, преобразованных и сохраненных входных полей; отсканированные, но несохраненные поля не учитываются. При достижении конца файла функция fscanf() возвращает значение EOF. Если ни одного поля не сохранено, то функция вернет нуль.

42.2.8. Построковый ввод – вывод

Для организации построкового ввода – вывода могут использоваться следующие функции:

- fputs(),
- fgets().

Функция fputs() объявлена следующим образом:

```
#include<stdio.h>
int fputs(const char* s, FILE* stream);
```

Функция fputs() выводит строку, на которую установлен указатель s в поток, на который установлен указатель stream. Символ '\0', завершающий выводимую строку в поток не записывается. Если в процессе записи имела место ошибка, то рассматриваемая функция вернет EOF, в случае успеха возвращаемым значением функции fputs() является неотрицательное числа.

Функция fgets() объявлена следующим образом:

```
#include<stdio.h>
char*  fputs(char* s, int n, FILE* stream);
```

Функция fgets() читает не более n – 1 символов из потока на, который установлен указатель stream, в символьный массив, на который установлен указатель s. Существуют два способа завершения процесса чтения символов:

- прочитан символ новая строка ('\n') ,
- прочитан n – 1 символ.

В первом случае прочитанный символ новая строка записывается в символьный массив, а затем строка завершается нуль – символом. Во втором случае символ новая строка оказывается непрочтенным и незаписанным в символьный массив. Однако и в этом случае в символьный массив будет записан нуль – символ.

После успешного выполнения функция fgets() возвращает указатель на символьный массив, в который производилась запись. Если имело место достижение конца файла при условии, что ни один символ не был прочитан, возвращаемым значением функции fscanf() является NULL, а содержимое массива, на который установлен указатель s, оказывается неименным.

42.2.9. Блоковый ввод – вывод

Блоковый ввод – вывод обеспечивается функциями fread() и fwrite().

Объявление функции fwrite() имеет следующий вид:

```
#include<stdio.h>
size_t  fwrite(const void* ptr, size_t n, size_t nmemb, FILE* stream);
```

Функция fwrite() помещает до nmemb элементов (блоков), каждый из которых размером size, из массива адресуемого указателем ptr, в поток, на который установлен указатель stream.

Функция fwrite() возвращает число успешно записанных элементов (блоков). Это значение может оказаться меньше, чем nmemb, если имеет место ошибка.

```
#include<stdio.h>
size_t  fread(void* ptr, size_t n, size_t nmemb, FILE* stream);
```

Функция fwrite() читает до nmemb элементов (блоков), каждый из которых размером size, в массив адресуемого указателем ptr, из потока, на который установлен указатель stream.

Функция fread() возвращает число успешно прочитанных элементов (блоков). Это значение может оказаться меньше, чем nmemb, если имеет место ошибка или обнаружен конец файла.

42.2.10. Примеры решенных задач

Пример 1 Табулирование функции с записью результатов на диск.

Постановка задачи. Табулировать функцию y = x^2 в n равноостоящих точках, начиная от значения x = xn вплоть до x = xk. Результаты табулирования записать в текстовый файл.

Решение.

Один из возможных вариантов решения рассматриваемой задачи имеет следующий вид.

```
/*                    Файл table.c.                    */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

void table_file(double xn, double xk, int n, FILE* out);
int main(void)
{
     int n;
     double x1, x2;
     FILE* out_file;

     clrscr();
     printf("Количество расчетных точек = ");
     scanf("%d", &n);
     printf("Начальное значение аргумента=");
     scanf("%lf", &x1);
     printf("Конечное значение аргумента=");
```

```
        scanf("%lf", &x2);

        if((out_file = fopen("out_file.txt", "w") == NULL))
        {
           printf("Не могу открыть файл out_file.txt\n");
           exit(1);
        }

        table_file(x1, x2, n, out_file);
        fclose(out_file);
        printf("Файл сформирован\n");
        getch();

        return 0;
   }

   void table_file(double xn, double xk, int n, FILE* out)
   {
       double x, dx;
       int    i    ;
       fprintf(out,   "%5s%15s%15s\n",   "Номер",   "Аргумент",
"Функция");

       dx = (xk - xn) /(n - 1);
       x  = xn;
       for(i = 1; i <= n; i++)
       {
         fprintf(out, "%5d%15.2f%15.2f\n", i, x, x * x);
         x += dx;
       }
   }
```

Пример 2. Запись содержимого двух массивов в текстовый файл

Постановка задачи. Имеются два массива, содержащие числа типа double. Записать содержимое этих массивов в текстовый файл. Вывод организовать в виде таблицы, снабженной заголовком. Таблица должна содержать три колонки. В первой колонке должен выводиться порядковый номер элемента, в двух других элементы массивов. В заголовке следует указать имена выводимых массивов.

Решение

Оформим решение данной задачи в виде функции. Эта функция должна иметь шесть параметров. Первый из них должен служить для передачи в функцию файлового указателя на файл, в который будет выполняться вывод. Два параметра необходимы для передачи массивов, значения которых следует выводить в файл. Затем еще два параметра необходимы для передачи в функцию имен выводимых массивов. Наконец, последний параметр должен служить для передачи в функцию количества выводимых элементов массива.

```
#include<stdio.h>
 void save_arrays(FILE* out, double* arr1, double* arr2,
                  char* name1, char* name2, int num)
```

```
{
    int i;
    fprinf(out, "%6s%15s%15s\n", "НОМЕР", name1, name2);
    for(i = 0; i < n; i++)
    {
      fprinf(out, "%6d%15.2f%15.2f\n", i, arr1[i],
arr2[i]);
    }
}
```

Пример 3. Чтение из таблицы, хранящейся в файле, двух числовых массивов

Постановка задачи. В текстовом файле имеется таблица, в которой хранится содержимое двух массивов. Таблица содержит три колонки и снабжена заголовком, который находится в первой строке файла. В первой колонке представлен порядковый номер элемента, в двух других - элементы массивов.

Оформим решение данной задачи в виде функции. Эта функция должна иметь три параметра. Первый из них должен служить для передачи в функцию файлового указателя на файл, из которого будет выполняться чтение. Два других параметра необходимы для передачи массивов, в которые будет выполняться чтение данных. Количество прочитанных элементов массивов функция должна вернуть в качестве возвращаемого значения. Для чтения файла будет использоваться функция fgets из стандартной библиотеки языка С. Основу алгоритма функции должен составлять итерационный цикл чтения файла. До входа в цикл необходимо прочитать заголовок таблицы, а в его теле последовательно читать строки таблицы. Строка, полученная в результате чтения заголовка таблицы, в программе не используется.

```
#include<stdio.h>
/*       Чтение массивов                           */
int load_arrays(FILE* in, double* arr1, double* arr2)
{
   int count = 0;
   char buf[255];
   /* Чтение заголовка таблицы */
   fgets(buf, sizeof(buf), in);
   while(fgets(buf, sizeof(buf), in) != NULL)
   {
      sscanf(buf, "%*d%lf%lf", arr1 + couПримерnt, arr2 +
count);
      count++;
   }
   return count;
}
```

Пример 4. Обработка числовой матрицы, хранящейся на диске

Постановка задачи. Числовая матрица находится в текстовом файле. Первая строка файла содержит данные о размере матрицы. Вычислить сумму элементов в каждой строке матрицы.

Решение

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

```
#include<stdlib.h>
void summa_row_in_text_file(FILE* in, FILE* out)
{
   int nrow;
   int ncol;
   int row;
   int col;
   double s, x;
   fscanf(in, "%d%d", &row, &col);
   for(row = 0; row < nrow; row++)
   {
      s = 0;
     for(col = 0; col < ncol; col++)
     {
          fscanf(in, "%lf", &x);
          s += x;
     }
     fprintf(out, "%10.3f\n", s);
   }
 }
```

42.2.11. Прямой доступ к файлу

При использовании прямого доступа к файлу программист имеет возможность оперативно выйти на нужный участок файла, а затем выполнить требуемую операцию. Переход на заданный участок файла осуществляется с помощью функций, выполняющих позиционирование указателя файловой позиции. К таким функциям относятся следующие функции:

- fseek(),
- ftell(),
- fgetpos(),
- fsetpos().

Прямой доступ обычно реализуется для двоичных файлов с использованием функций fread() и fwrite().

42.2.11.1. Функция fseek()

```
#include <stdio.h>
int fseek(FILE* stream, long int offset, int whreefrom);
```

Функция fseek() обеспечивает прямой (произвольный) доступ к открытому потоку. Параметры: stream – файловый указатель, offset – смещение в символах от некоторой точки отсчета и wherefrom – параметр, определяющий точку отсчета. В языке Си определены три символические константы (SEEK_SET, SEEK_CUR и SEEK_END), с помощью которых можно задать точку отсчета wherefrom (см. таблицу, приведенную ниже).

Константа	Точка отсчета
SEEK_SET	Начало файла
SEEK_END	Конец файла
SEEK_CUR	Текущая позиция в файле