Министерство образования Российской Федерации
Пермский Государственный технический университет
Кафедра информационных технологий и автоматизированных систем

Викентьева О. Л.

Конспект лекций по курсу «Алгоритмические языки и программирование»
(Основы языка C++, I семестр)

Пермь 2003

Введение

В первом семестре рассматриваются основные конструкции языка Си и базовая технология программирования (структурное программирование).

Структурное программирование – это технология создания программ, позволяющая путем соблюдения определенных правил уменьшить время разработки и количество ошибок, а также облегчить возможность модификации программы.

1.1. Алгоритм и программа

Алгоритм – точное предписание, определяющий вычислительный процесс, идущий от изменяемых начальных данных к конечному результату, т. е. это рецепт достижения какой-либо цели.

Совокупность средств и правил для представления алгоритма в виде пригодном для выполнения вычислительной машиной называется языком программирования, алгоритм, записанный на этом языке, называется программой.

Сначала всегда разрабатывается алгоритм действий, а потом он записывается на одном из языков программирования. Текст программы обрабатывается специальными служебными программами – трансляторами. Языки программирования – это искусственные языки. От естественных языков они отличаются ограниченным числом «слов» и очень строгими правилами записи команд (операторов). Совокупность этих требований образует синтаксис языка программирования, а смысл каждой конструкции – его семантику.

1.2.Свойства алгоритма

1. Массовость: алгоритм должен применяться не к одной задаче, а к целому классу подобных задач (алгоритм для решения квадратного уравнения должен решать не одно уравнение, а все квадратные уравнения).
2. Результативность: алгоритм должен приводить к получению результата за конкретное число шагов (при делении 1 на 3 получается периодическая дробь 0,3333(3), для достижения конечного результата надо оговорить точность получения этой дроби, например, до 4 знака после запятой).
3. Определенность (детерминированность) – каждое действие алгоритма должно быть понятно его исполнителю (инструкция к бытовому прибору на японском языке для человека не владеющего японским языком не является алгоритмом, т .к не обладает свойством детерминированности).
4. Дискретность – процесс должен быть описан с помощью неделимых операций, выполняемых на каждом шаге (т. е. шаги нельзя разделить на более мелкие шаги).

Алгоритмы можно представить в следующих формах:

1) словесное описание алгоритма.
2) графическое описание алгоритма.
3) с помощью алгоритмического языка программирования

1.2. Компиляторы и интерпретаторы

С помощью языка программирования создается текст, описывающий ранее составленный алгоритм. Чтобы получить работающую программу, надо этот текст перевести в последовательность команд процессора, что выполняется при помощи специальных программ, которые называются трансляторами. Трансляторы бывают двух видов: компиляторы и интерпретаторы. Компилятор транслирует текст исходного модуля в машинный код, который называется объектным модулем за один непрерывный процесс. При этом сначала он просматривает исходный текст программы в поисках синтаксических ошибок. Интерпретатор выполняет исходный модуль программы в режиме оператор за оператором, по

ходу работы, переводя каждый оператор на машинный язык.

1.3.Языки программирования

Разные типы процессоров имеют разный набор команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется языком программирования низкого уровня. Языком самого низкого уровня является язык ассемблера, который просто представляет каждую команду машинного кода в виде специальных символьных обозначений, которые называются мнемониками. С помощью языков низкого уровня создаются очень эффективные и компактные программы, т .к разработчик получает доступ ко всем возможностям процессора. Т .к. наборы инструкций для разных моделей процессоров тоже разные, то каждой модели процессора соответствует свой язык ассемблера, и написанная на нем программа может быть использована только в этой среде. Подобные языки применяют для написания небольших системных приложений, драйверов устройств и т. п..

Языки программирования высокого уровня не учитывают особенности конкретных компьютерных архитектур, поэтому создаваемые программы на уровне исходных текстов легко переносятся на другие платформы, если для них созданы соответствующие трансляторы. Разработка программ на языках высокого уровня гораздо проще, чем на машинных языках.

Языками высокого уровня являются:

1. Фортран – первый компилируемый язык, созданный в 50-е годы 20 века. В нем были реализован ряд важнейших понятий программирования. Для этого языка было создано огромное количество библиотек, начиная от статистических комплексов и заканчивая управлением спутниками, поэтому он продолжает использоваться во многих организациях.
2. Кобол – компилируемый язык для экономических расчетов и решения бизнес-задач, разработанный в начале 60-х годов. В Коболе были реализованы очень мощные средства работы с большими объемами данных, хранящихся на внешних носителях.
3. Паскаль – создан в конце 70-х годов швейцарским математиком Никлаусом Виртом специально для обучению программированию. Он позволяет выработать алгоритмическое мышление, строить короткую, хорошо читаемую программу, демонстрировать основные приемы алгоритмизации, он также хорошо подходит для реализации крупных проектов.
4. Бейсик – создавался в 60-х годах также для обучения программированию. Для него имеются и компиляторы и интерпретаторы, является одним из самых популярных языков программирования.
5. Си – был создан в 70- е годы первоначально не рассматривался как массовый язык программирования. Он планировался для замены ассемблера, чтобы иметь возможность создавать такие же эффективные и короткие программы, но не зависеть от конкретного процессора. Он во многом похож на Паскаль и имеет дополнительные возможности для работы с памятью. На нем написано много прикладных и системных программ, а также операционная система Unix.
6. Си++ - объектно-ориентированное расширение языка Си, созданное Бьярном Страуструпом в 1980г.
7. Java – язык, который был создан компанией Sun в начале 90-х годов на основе Си++. Он призван упростить разработку приложений на СИ++ путем исключения из него низкоуровневых возможностей. Главная особенность языка – это то, что он компилируется не в машинный код, а в платформно-независимый байт-код (каждая команда занимает один байт). Этот код может выполняться с помощью интерпретатора – виртуальной Java-машины (JVM).

2.Структура программы на Си++

Программа на языке Си имеет следующую структуру:

```
#директивы препроцессора
. . . . . .
#директивы препроцессора
функция а ( )
        операторы
функция в ( )
        операторы
void main ( )      //функция, с которой начинается выполнение программы
        операторы
                описания
                присваивания
                функция
                пустой оператор
                составной
                выбора
                циклов
                перехода
```

Директивы препроцессора - управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на СИ в виде текстового файла, проходит 3 этапа обработки:

1) препроцессорное преобразование текста ;
2) компиляция;
3) компоновка (редактирование связей или сборка).

После этих трех этапов формируется исполняемый код программы. Задача препроцессора - преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора. Директива начинается с #. Например,

1) #define - указывает правила замены в тексте.

#define ZERO 0.0

Означает , что каждое использование в программе имени ZERO будет заменяться на 0.0.

2) #include< имя заголовочного файла> - предназначена для включения в текст программы текста из каталога «Заголовочных файлов», поставляемых вместе со стандартными библиотеками. Каждая библиотечная функция Си имеет соответствующее описание в одном из заголовочных файлов. Список заголовочных файлов определен стандартом языка. Употребление директивы include не подключает соответствующую стандартную биб-

лиотеку, а только позволяют вставить в текст программы описания из указанного заголовочного файла. Подключение кодов библиотеки осуществляется на этапе компоновки, т. е. после компиляции. Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.

После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.

Программа представляет собой набор описаний и определений, и состоит из набора функций. Среди этих функций всегда должна быть функция с именем main. Без нее программа не может быть выполнена. Перед именем функции помещаются сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип void: void main ( ). Каждая функция, в том числе и main должна иметь набор параметров, он может быть пустым, тогда в скобках указывается (void).

За заголовком функции размещается тело функции. Тело функции - это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.

Определения - вводят объекты (объект - это именованная область памяти, частный случай объекта - переменная), необходимые для представления в программе обрабатываемых данных. Примером являются

```
int y = 10 ; //именованная константа
float x ;   //переменная
```

Описания - уведомляют компилятор о свойствах и именах объектов и функций, описанных в других частях программы.

Операторы - определяют действия программы на каждом шаге ее исполнения.

Пример программы на Си:

```
#include <stdio.h>  //препроцессорная директива
void main()         //функция
{                   //начало
printf("Hello! ");  //печать
}                   //конец
```

Контрольные вопросы

1. Из каких частей состоит программа на С++?
2. Чем определение отличается от объявления?
3. Перечислить этапы создания исполняемой программы на языке С++.
4. Что такое препроцессор?
5. Что такое директива препроцессора? Привести примеры директив препроцессора.
6. Составить программу, которая печатает текст «Моя первая программа на С++»

2. Базовые средства языка СИ++

2.1.Состав языка

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Алгоритмический язык также содержит такие элементы, только слова называют лексемами (элементарными конструкциями), словосочетания – выражениями, предложения – операторами. Лексемы образуются из символов, выражения из лексем и символов, операторы из символов выражений и лексем (Рис. 1.1)

Рис. 1.1. Состав алгоритмического языка

Таким образом, элементами алгоритмического языка являются:

1) *Алфавит языка СИ++,* который включает
- прописные и строчные латинские буквы и знак подчеркивания;
- арабские цифры от 0 до 9;
- специальные знаки “{},| []()+-/%*.\’:;&?<>=!#^
- пробельные символы (пробел, символ табуляции, символы перехода на новую строку).

2) Из символов формируются лексемы языка:
- *Идентификаторы* – имена объектов СИ-программ. В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, PROG1, prog1 и Prog1 – три различных идентификатора. Первым символом должна быть буква или знак подчеркивания (но не цифра). Пробелы в идентификаторах не допускаются.
- *Ключевые* (зарезервированные) слова – это слова, которые имеют специальное значение для компилятора. Их нельзя использовать в качестве идентификаторов.
- Знаки операций – это один или несколько символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в этой операции операндов.
- *Константы* – это неизменяемые величины. Существуют целые, вещественные, символьные и строковые константы. Компилятор выделяет константу в качестве лексемы (элементарной конструкции) и относит ее к одному из типов по ее внешнему виду.
- *Разделители* – скобки, точка, запятая пробельные символы.

2.1.1. Константы в Си++

Константа – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения.

Константы делятся на 5 групп:
- целые;
- вещественные (с плавающей точкой);
- перечислимые;
- символьные;
- строковые.

Компилятор выделяет лексему и относит ее к той или другой группе, а затем вну-

три группы к определенному типу по ее форме записи в тексте программы и по числовому значению.

Целые константы могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичная константа определяется как последовательность десятичных цифр, начинающаяся не с 0, если это число не 0 (примеры: 8, 0, 192345). Восьмеричная константа – это константа , которая всегда начинается с 0. За 0 следуют восьмеричные цифры (примеры: 016 – десятичное значение 14, 01). Шестнадцатеричные константы – последовательность шестнадцатеричных цифр, которым предшествуют символы 0x или 0X (примеры: 0xA, 0X00F).

В зависимости от значения целой константы компилятор по-разному представит ее в памяти компьютера (т. е. компилятор припишет константе соответствующий тип данных).

Вещественные константы имеют другую форму внутреннего представления в памяти компьютера. Компилятор распознает такие константы по их виду. Вещественные константы могут иметь две формы представления: с фиксированной точкой и с плавающей точкой. Вид константы с фиксированной точкой:[цифры].[цифры] (примеры: 5.7, . 0001, 41.).Вид константы с плавающей точкой: [цифры][.][цифры]E|e[+|-][цифры] (примеры:0.5e5, .11e-5, 5E3). В записи вещественных констант может опускаться либо целая, либо дробная части, либо десятичная точка, либо признак экспоненты с показателем степени.

Перечислимые константы вводятся с помощью ключевого слова enum. Это обычные целые константы, которым приписаны уникальны и удобные для использования обозначения. Примеры: enum { one=1, two=2, three=3,four=4};

enum {zero,one,two,three} – если в определении перечислимых константы опустить знаки = и числовые значения, то значения будут приписываться по умолчанию. При этом самый левый идентификатор получит значение 0, а каждый последующий будет увеличиваться на 1.

enum { ten=10, three=3, four, five, six};

enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} ;

Символьные константы – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, имеют тип char и занимают в памяти один байт, символьные константы, состоящие из двух символов, имеют тип int и занимают два байта. Последовательности, начинающиеся со знака \ , называются управляющими, они используются:

- Для представления символов, не имеющих графического отображения, например:
  \a – звуковой сигнал,
  \b – возврат на один шаг,
  \n – перевод строки,
  \t – горизонтальная табуляция.
- Для представления символов: \ , ’ , ? , ” (\\, \’ ,\? ,\”).
- Для представления символов с помощью шестнадцатеричных или восьмеричных кодов (\073, \0xF5).

Строковая константа – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы. Например: “\nНовая строка”,

“\n\”Алгоритмические языки программирования высокого уровня \”” .

2.2. Типы данных в Си++

Данные отображают в программе окружающий мир. Цель программы состоит в обработке данных. Данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

1) внутреннее представление данных в памяти компьютера;
2) множество значений, которые могут принимать величины этого типа;
3) операции и функции, которые можно применять к данным этого типа.

В зависимости от требований задания программист выбирает тип для объектов программы. Типы Си++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением. В Си++ определено 6 простых типов данных:

int (целый)
char (символьный)
wchar_t (расширенный символьный)
bool (логический)
float(вещественный)
double (вещественный с двойной точностью)

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов

short (короткий)
long (длинный)
signed (знаковый)
unsigned (беззнаковый)

2.2.1. Тип int

Значениями этого типа являются целые числа.

Размер типа int не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под него отводится 2 байта, для 32-разрядного – 4 байта.

Если перед int стоит спецификатор short, то под число отводится 2 байта, а если спецификатор long, то 4 байта. От количества отводимой под объект памяти зависит множество допустимых значений, которые может принимать объект:

short int - занимает 2 байта, следовательно, имеет диапазон –32768 ..+32767;

long int – занимает 4 байта, следовательно, имеет диапазон –2 147 483 648..+2 147 483 647

Тип int совпадает с типом short int на 16-разрядных ПК и с типом long int на 32-разрядных ПК.

Модификаторы signed и unsigned также влияют на множество допустимых значений, которые может принимать объект:

unsigned short int - занимает 2 байта, следовательно, имеет диапазон 0 ..65536;

unsigned long int – занимает 4 байта, следовательно, имеет диапазон 0..+4 294 967 295.

2.2.2. Тип char

Значениями этого типа являются элементы конечного упорядоченного множества символов. Каждому символу ставится в соответствие число, которое называется кодом символа. Под величину символьного типа отводится 1 байт. Тип char может использоваться со спецификаторами signed и unsigned. В данных типа signed char можно хранить значения в диапазоне от –128 до 127. При использовании типа unsigned char значения могут находиться в диапазоне от 0 до 255. Для кодировки используется код ASCII(American Standard Code foe International Interchange). Символы с кодами от 0 до 31 относятся к служебным и имеют самостоятельное значение только в операторах ввода-вывода.

Величины типа char также применяются для хранения чисел из указанных диапазонов.

2.2.3. Тип wchar_t

Предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например Unicode. Размер этого типа, как правило, соответствует типу short. Строковые константы такого типа записываются с префиксом L: L“String #1”.

2.2.4. Тип bool

Тип bool называется логическим. Его величины могут принимать значения true и false. Внутренняя форма представления false – 0, любое другое значение интерпретируется как true.

2.2.5. Типы с плавающей точкой.

Внутреннее представление вещественного числа состоит из 2 частей: мантиссы и порядка. В IBM-совместимых ПК величины типа float занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 – под мантиссу.

Величины типы double занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка его диапазон.

Если перед именем типа double стоит спецификатор long, то под величину отводится байтов.

2.2.6. Тип void

К основным типам также относится тип void Множество значений этого типа – пусто.

2.3. Переменные

Переменная в СИ++ - именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Перед использованием любая переменная должна быть описана. Примеры:

int a; float x;

Общий вид оператора описания:

[класс памяти][const]тип имя [инициализатор];

Класс памяти может принимать значения: auto, extern, static, register. Класс памяти определяет время жизни и область видимости переменной. Если класс памяти не указан явно, то компилятор определяет его исходя из контекста объявления. Время жизни может быть постоянным – в течение выполнения программы или временным – в течение блока. Область видимости – часть текста программы, из которой допустим обычный доступ к переменной. Обычно область видимости совпадает с областью действия. Кроме того случая, когда во внутреннем блоке существует переменная с таким же именем.

Const – показывает, что эту переменную нельзя изменять (именованная константа).

При описании можно присвоить переменной начальное значение (инициализация).

Классы памяти:

auto –автоматическая локальная переменная. Спецификатор auto может быть задан только при определении объектов блока, например, в теле функции. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют.

extern – глобальная переменная, она находится в другом месте программы (в другом файле или долее по тексту). Используется для создания переменных, которые доступны во всех файлах программы.

static – статическая переменная, она существует только в пределах того файла, где определена переменная.

register - аналогичны auto, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как auto.

Пример

```
int a; //глобальная переменная
void main(){
int b;//локальная переменная
extern int x;//переменная x определена в другом месте
static int c;//локальная статическая переменная
a=1;//присваивание глобальной переменной
int a;//локальная переменная a
```

```
a=2;//присваивание локальной переменной
::a=3;//присваивание глобальной переменной
}
int x=4;//определение и инициализация x
```

В примере переменная a определена вне всех блоков. Областью действия переменной a является вся программа, кроме тех строк, где используется локальная переменная a. Переменные b и c – локальные, область их видимости – блок. Время жизни различно: память под b выделяется при входе в блок (т. к. по умолчанию класс памяти auto), освобождается при выходе из него. Переменная c (static) существует, пока работает программа.

Если при определении начальное значение переменным не задается явным образом, то компилятор обнуляет глобальные и статические переменные. Автоматические переменные не инициализируются..

Имя переменной должно быть уникальным в своей области действия.

Описание переменной может быть выполнено или как объявление, или как определение. Объявление содержит информацию о классе памяти и типе переменной, определение вместе с этой информацией дает указание выделить память. В примере extern int x; - объявление, а остальные – определения.

2.4.Знаки операций в Си++

Знаки операций обеспечивают формирование выражений. Выражения состоят из операндов, знаков операций и скобок. Каждый операнд является, в свою очередь, выражением или частным случаем выражения – константой или переменной.

Унарные операции

&	получение адреса операнда
*	Обращение по адресу (разыменование)
-	унарный минус, меняет знак арифметического операнда
~	поразрядное инвертирование внутреннего двоичного кода целочисленного операнда (побитовое отрицание)
!	логическое отрицание (НЕ). В качестве логических значений используется 0 - ложь и не 0 - истина, отрицанием 0 будет 1, отрицанием любого ненулевого числа будет 0.

++	Увеличение на единицу:
	префиксная операция - увеличивает операнд до его использования,
	постфиксная операция увеличивает операнд после его использования.
	int m=1,n=2;
	int a=(m++)+n; // a=4,m=2,n=2
	int b=m+(++n);//a=3,m=1,n=3

- -	уменьшение на единицу: префиксная операция - уменьшает операнд до его использования, постфиксная операция уменьшает операнд после его использования.

sizeof	вычисление размера (в байтах) для объекта того типа, который имеет операнд
	имеет две формы
	sizeof выражение
	sizeof (тип)
	Примеры:
	sizeof(float)//4
	sizeof(1.0)//8, т. к. вещественные константы по умолчанию имеют тип double

Бинарные операции.

Аддитивные:

+	бинарный плюс (сложение арифметических операндов)
-	бинарный минус (вычитание арифметических операндов)

Мультипликативные:

*	умножение операндов арифметического типа
/	деление операндов арифметического типа (если операнды целочисленные, то выполняется целочисленное деление)
%	получение остатка от деления целочисленных операндов

Операции сдвига (определены только для целочисленных операндов).

Формат выражения с операцией сдвига:

операнд_левый операция_сдвига операнд_правый

<<	сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда, освободившиеся разряды обнуляются
>>	сдвиг вправо битового представления значения правого целочисленного операнда на количество разрядов, равное значению правого операнда, освободившиеся разряды обнуляются, если операнд беззнакового типа и заполняются знаковым разрядом, если – знакового

Поразрядные операции:

&	поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов (бит =1, если соответствующие биты обоих операндов=1)
|	поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов (бит =1, если соответствующий бит одного из операндов=1)
^	поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов(бит =1, если соответствующий бит только одного из операндов=1)

Операции сравнения: результатом являются true(не 0) или false(0)

<	меньше, чем
>	больше, чем
<=	меньше или равно
>=	больше или равно
==	Равно
!=	не равно

Логические бинарные операции:

&&	конъюнкция (И) целочисленных операндов или отношений, целочисленный результат ложь(0) или истина(не 0)
||	дизъюнкция (ИЛИ) целочисленных операндов или отношений, целочисленный результат ложь(0) или истина(не 0)

Операции присваивания

=, +=, -=, += и т.д.

Формат операции простого присваивания:

операнд1=операнд2

Леводопустимое значение (L-значение) – выражение, которое адресует некоторый участок памяти, т. е. в него можно занести значение. Это название произошло от операции присваивания, т. к. именно левая часть операции присваивания определяет, в какую область памяти будет занесен результат операции. Переменная – это частный случай леводопустимого выражения.

Условная операция.

В отличие от унарных и бинарных операций в ней используется три операнда.

Выражение1 ? Выражение2 : Выражение3;

Первым вычисляется значение выражения1. Если оно истинно, то вычисляется значение выражения2, которое становится результатом. Если при вычислении выражения1 получится 0, то в качестве результата берется значение выражения3.

Например:

x<0 ? -x : x ; //вычисляется абсолютное значение x.

Операция явного (преобразования) приведения типа.

Существует две формы: каноническая и функциональная:

1) (имя_типа) операнд
2) имя_типа (операнд)

(int)a //каноническая форма
int(a) //функциональная форма

2.5. Выражения

Из констант, переменных, разделителей и знаков операций можно конструировать выражения. Каждое выражение представляет собой правило вычисления нового значения.. Если выражение формирует целое или вещественное число, то оно называется арифметическим. Пара арифметических выражений, объединенная операцией сравнения, называется отношением. Если отношение имеет ненулевое значение, то оно – истинно, иначе – ложно.

Приоритеты операций в выражениях

Ранг	Операции
1	( ) [ ] -> .
2	! ~ - ++ -- & * (тип) sizeof тип( )
3	* / % (мультипликативные бинарные)
	+ - (аддитивные бинарные)
5	<< >> (поразрядного сдвига)
6	< > <= >= (отношения)
7	== != (отношения)
8	& (поразрядная конъюнкция «И»)
9	^ (поразрядное исключающее «ИЛИ»)
10	| (поразрядная дизъюнкция «ИЛИ»)
11	&& (конъюнкция «И»)
12	|| (дизъюнкция «ИЛИ»)
13	?: (условная операция)
14	= *= /= %= -= &= ^= |= <<= >>= (операция присваивания)
15	, (операция запятая)

Контрольные вопросы

1. Из каких элементов состоит естественный язык? Что является аналогами этих элементов в C++?
2. Что такое лексема? Привести примеры лексем в языке C++.
3. Что такое идентификатор? Правила записи идентификаторов.
4. Что такое константа? Как константа обрабатывается компилятором?
5. Какие типы констант существуют в C++. Привести примеры констант разных типов.
6. К какому типу относятся константы 192345, 0x56, 0xCB, 016, 0.7865, .0045, ‘c’, “x”, one, “one”, 5 , 5.?
7. Что такое тип данных?
8. Чем отличаются типы данных: float и double, char и wchar_t, int и short int?
9. Чем отличаются типы данных int и unsigned int?
10. Перечислить все типы данных, которые существуют в C++. Сколько места в памяти занимают данные каждого типа?
11. На что влияет количество памяти, выделяемое для данных определенного типа?

12. Что такое переменная? Чем объявление переменой отличается от ее определения? Привести примеры определений и объявлений.
13. Что такое класс памяти? Какие классы памяти существуют в C++? Привести примеры объявлений и определений переменных разных классов памяти.
14. Что такое выражение? Из чего состоит выражение?
15. Что такое операнд?
16. Какие операции можно применять к целочисленным данным? К вещественным данным? К символьным данным?
17. Что такое отношение?
18. В каком случае отношение считается ложным, а в каком – истинным?
19. Какие операции называются унарными? Привести примеры.
20. Какие операции называются бинарными? Привести примеры.
21. Что такое тернарная операция? Привести пример.
22. Какая разница между постфиксной и префиксной операцией инкремента (декремента)?
23. Какие операции присваивания существуют в C++?
24. Привести примеры выражений, содержащих операции присваивания, операции инкремента (декремента), аддитивные и мультипликативные операции. Пояснить, как они будут выполняться.
25. Что такое леводопустимое значение? Привести пример.
26. Чему будет равно значение выражений:

```
int z=x/y++; если int x=1, y=2;
int w=x%++y, если int x=1, y=2;
int a=++m+n++*sizeof(int); если int m=1, n=2;
float a=4*m/0.3*n; если float m=1.5; int n=5;
int ok=int(0.5*y)<short(x)++; если int x=10, y=3;
```

3. Ввод и вывод данных

В языке Си++ нет встроенных средств ввода и вывода – он осуществляется с помощью функций, типов и объектов, которые находятся в стандартных библиотеках. Существует два основных способа: функции унаследованные из Си и объекты Си++.

Для ввода/вывода данных в стиле Си используются функции, которые описываются в библиотечном файле stdio.h.

1)printf (форматная строка, список аргументов);

форматная строка - строка символов, заключенных в кавычки, которая показывает, как должны быть напечатаны аргументы. Например:

printf (“Значение числа Пи равно %f\n”, pi);

Форматная строка может содержать

1) символы печатаемые текстуально;
2) спецификации преобразования;
3) управляющие символы.

Каждому аргументу соответствует своя спецификация преобразования:

%d, %i - десятичное целое число;

%f - число с плавающей точкой;

%e,%E – число с плавающей точкой в экспоненциальной форме;

%u – десятичное число в беззнаковой форме;

%c - символ;

%s - строка.

В форматную строку также могут входить управляющие символы:

\n - управляющий символ новая строка;

\t – табуляция;

\a – звуковой сигнал и др.

Также в форматной строке могут использоваться модификаторы формата, которые управляют шириной поля, отводимого для размещения выводимого значения. Модификаторы – это числа, которые указывают минимальное количество позиций для вывода значения и количество позиций ля вывода дробной части числа:

%[-]m[.p]C, где

1. - задает выравнивание по левому краю,

m – минимальная ширина поля,

p – количество цифр после запятой для чисел с плавающей точкой и минимальное количество выводимых цифр для целых чисел (если цифр в числе меньше, чем значение p, то выводятся начальные нули),

C- спецификация формата вывода.

Пример

printf("\nСпецификации формата:\n%10.5d - целое,\n%10.5f - с плавающей точкой\
\n%10.5e – в экспоненциальной форме\n%10s - строка",10,10.0,10.0,"10");

Будет выведено:

Спецификации формата:

00010 – целое

10.00000 – с плавающей точкой

1.00000e+001 - в экспоненциальной форме

10 – строка.

2) scanf (форматная строка, список аргументов);

В качестве аргументов используются адреса переменных. Например:

scanf(“ %d%f ”, &x,&y);

При использовании библиотеки классов Си++, Используется библиотечный файл iostream.h, в котором определены стандартные потоки ввода данных от клавиатуры cin и вывода данных на экран дисплея cout, а также соответствующие операции

1) << - операция записи данных в поток;
2) >> - операция чтения данных из потока.

Например:

```
#include <iostream.h>;
. . . . . .
cout << “\nВведите количество элементов: ”;
cin >> n;
```

Контрольные вопросы

1. Что такое форматная строка? Что содержит форматная строка функции printf? Что содержит форматная строка функции scanf?
2. Что такое спецификация преобразования? Привести примеры спецификаций преобразования для различных типов данных.
3. Что будет выведено функцией
   printf("\nСреднее арифметическое последовательности чисел равно: %10.5f \nКоличество четных элементов последовательности равно%10.5d ",S/n,k);
4. Как записать вывод результатов из вопроса 3 с помощью операции << ?
5. Как выполнить ввод переменных x и y, где x типа long int, а y типа double с помощью функции scanf? С помощью операции >> ?

4. Основные операторы языка Си++»

4.1. Базовые конструкции структурного программирования

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.

Линейной называется конструкция, представляющая собой последовательное соединение двух или более операторов.

Ветвление – задает выполнение одного из двух операторов, в зависимости от выполнения какого либо условия.

Цикл – задает многократное выполнение оператора.

Следование	Ветвление	Цикл

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. Структурное программирование также называют программированием без goto, т. к. частое использование операторов перехода затрудняет понимание логики работы программы. Но иногда встречаются ситуации, в которых применение операторов перехода, наоборот, упрощает структуру программы.

Операторы управления работой программы называют управляющими конструкциями программы. К ним относят:

- составные операторы;
- операторы выбора;
- операторы циклов;
- операторы перехода.

4.2. Оператор «выражение»

Любое выражение, заканчивающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении этого выражения. Частным случаем выражения является пустой оператор ;.

Примеры:

```
i++;
a+=2;
x=a+b;
```

4.3. Составные операторы

К составным операторам относят собственно составные операторы и блоки. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается от составного оператора наличием определений в теле блока. Например:

```
{
n++;                    это составной оператор
summa+=n;
}

{
int n=0;
n++;                    это блок
summa+=n;
```

```
}
```

4.4. Операторы выбора

Операторы выбора - это условный оператор и переключатель.

1. Условный оператор имеет полную и сокращенную форму.

```
if (выражение-условие ) оператор;          //сокращенная форма
```

В качестве выражения-условия могут использоваться арифметическое выражение, отношение и логическое выражение. Если значение выражения-условия отлично от нуля (т. е. истинно), то выполняется оператор. Например:

```
if (x<y&&x<z)min=x;
if ( выражение-условие ) оператор1;                    //полная форма
else  оператор2;
```

Если значение выражения-условия отлично от нуля, то выполняется оператор1, при нулевом значении выражения-условия выполняется оператор2.Например:

```
if (d>=0)
{
x1=(-b-sqrt(d))/(2*a);
x2=(-b+sqrt(d))/(2*a);
cout<< “\nx1=”<<x1<<“x2=”<<x2;
}
else cout<<“\nРешения нет”;
```

2.Переключатель определяет множественный выбор.

```
switch (выражение)
{
case константа1 : оператор1 ;
case константа2 : оператор2 ;
. . . . . .
[default: операторы;]
}
```

При выполнении оператора switch, вычисляется выражение, записанное после switch, оно должно быть целочисленным. Полученное значение последовательно сравнивается с константами, которые записаны следом за case. При первом же совпадении выполняются операторы помеченные данной меткой. Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель. Если значение выражения, записанного после switch не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой default. Метка default может отсутствовать.

Пример:

```
#include <iostream.h>
void main()
{
        int i;
        cout<<"\nEnter the number";
        cin>>i;
        switch(i)
        {
        case 1:cout<<"\nthe number is one";
        case 2:cout<<"\n2*2="<<i*i;
        case 3: cout<<"\n3*3="<<i*i;break;
        case 4: cout<<"\n"<<i<<" is very beautiful!";
        default:cout<<"\nThe end of work";
        }
}
```

Результаты работы программы:

1. При вводе 1 будет выведено:

The number is one
2*2=1
3*3=1

2. При вводе 2 будет выведено:

2*2=4
3*3=4

3. При вводе 3 будет выведено:

3*3=9

4. При вводе 4 будет выведено:

4 is very beautiful!

5. При вводе всех остальных чисел будет выведено:

The end of work

4.5. Операторы циклов

Различают:

1) итерационные циклы;
2) арифметические циклы.

Группа действий, повторяющихся в цикле, называется его телом. Однократное выполнение цикла называется его шагом.

В итерационных циклах известно условие выполнения цикла.

1. Цикл с предусловием:

```
while (выражение-условие)
оператор;
```

В качестве <выражения-условия> чаще всего используется отношение или логическое выражение. Если оно истинно, т. е. не равно 0, то тело цикла выполняется до тех пор, пока выражение-условие не станет ложным.

Пример

```
while (a!=0)
{
cin>>a;
s+=a;
}
```

2. Цикл с постусловием:

```
do
оператор
while (выражение-условие);
```

Тело цикла выполняется до тех пор, пока выражение-условие истинно.

Пример:

```
do
{
cin>>a;
s+=a;
}
while(a!=0);
```

3. Цикл с параметром:

```
for ( выражение_1;выражение-условие;выражение_3)
оператор;
```

выражение_1 и выражение_3 могут состоять из нескольких выражений, разделенных запятыми. Выражение_1 - задает начальные условия для цикла (инициализация). Выражение-условие> определяет условие выполнения цикла, если оно не равно 0, цикл выполняется, а затем вычисляется значение выражения_3. Выражение_3 - задает изменение параметра цикла или других переменных (коррекция). Цикл продолжается до тех пор, пока выражение-условие не станет равно 0. Любое выражение может отсутствовать, но

разделяющие их « ; » должны быть обязательно.

Примеры использования цикла с параметром.

1) Уменьшение параметра:

```
for ( n=10; n>0; n--)
{ оператор};
```

2) Изменение шага корректировки:

```
for ( n=2; n>60; n+=13)
{ оператор };
```

3) Возможность проверять условие отличное от условия, которое налагается на число итераций:

```
for ( num=1;num*num*num<216; num++)
{ оператор };
```

4) Коррекция может осуществляться не только с помощью сложения или вычитания:

```
for ( d=100.0; d<150.0;d*=1.1)
{ <тело цикла>};
for (x=1;y<=75;y=5*(x++)+10)
{ оператор };
```

5) Можно использовать несколько инициализирующих или корректирующих выражений:

```
for (x=1, y=0; x<10;x++;y+=x);
```

4.6.Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

1) break - оператор прерывания цикла.

```
{
< операторы>
if (<выражение_условие>) break;
<операторы>
}
```

Т. е. оператор break целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла.

Пример:

```
// ищет сумму чисел вводимых с клавиатуры до тех пор, пока не будет введено 100 чисел или 0
for(s=0, i=1; i<100;i++)
{
cin>>x;
if( x==0) break; // если ввели 0, то суммирование заканчивается
s+=x;
}
```

2) continue - переход к следующей итерации цикла. Он используется, когда тело цикла содержит ветвления.

Пример:

```
//ищет количество и сумму положительных чисел
for( k=0,s=0,x=1;x!=0;)
{
cin>>x;
if (x<=0) continue;
k++;s+=x;
}
```

3) Оператор goto

Оператор goto имеет формат: goto метка;

В теле той же функции должна присутствовать конструкция: метка:оператор;

Метка – это обычный идентификатор, областью видимости которого является функция. Оператор goto передает управления оператору, стоящему после метки. Использование оператора goto оправдано, если необходимо выполнить переход из нескольких вложенных циклов или переключателей вниз по тексту программы или перейти в одно место функции после выполнения различных действий.

Применение goto нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и только один выход.

Нельзя передавать управление внутрь операторов if, switch и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию, на операторы, которые стоят после инициализации. Пример:

```
int k;
goto m;
. . .
{
int a=3,b=4;
k=a+b;
m: int c=k+1;
. . .
}
```

В этом примере при переходе на метку m не будет выполняться инициализация переменных a , b и k.

4) Оператор return – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

return [выражение];

5. Примеры решения задач с использованием основных операторов Си++

«Начинающие программисты, особенно студенты, часто пишут программы так: получив задание, тут же садятся за компьютер и начинают кодировать те фрагменты алгоритма, которые им удается придумать сразу. Переменным дают первые попавшиеся имена типа x и y. Когда компьютер зависает, делается перерыв, после которого все написанное стирается, и все повторяется заново. Периодически высказываются сомнения в правильности работы компилятора, компьютера и операционной системы. Когда программа доходит до стадии выполнения, в нее вводятся произвольные значения, после чего экран становится объектом пристального удивленного изучения. «Работает» такая программа обычно только в бережных руках хозяина на одном наборе данных, а внесение в нее изменений может привести автора к потере веры в себя и ненависти к процессу программирования.

Ваша задача состоит в том, чтобы научиться подходить к программированию профессионально. В конце концов, профессионал отличается тем, что может достаточно точно оценить, сколько времени у него займет написание программы, которая будет работать в полном соответствии с поставленной задачей. Кроме «ума, вкуса и терпения», для этого требуется опыт, а также знание основных принципов, выработанных программистами в течение более, чем полувека развития этой дисциплины. Даже к написанию самых простых программ нужно подходить последовательно, соблюдая определенную дисциплину.» (Павловская Т. А., стр.109)

Решение задач по программированию предполагает ряд этапов:

1)Разработка математической модели. На этом этапе определяются исходные данные и результаты решения задачи, а также математические формулы, с помощью которых можно перейти от исходных данных к конечному результату.

2)Разработка алгоритма. Определяются действия, выполняя которые можно будет от исходных данных придти к требуемому результату.

3)Запись программы на некотором языке программирования. На этом этапе каждому шагу алгоритма ставится в соответствие конструкция выбранного алгоритмического языка.

4)Выполнение программы (исходный модуль ->компилятор ->объектный модуль -> компоновщик -> исполняемый модуль)

5)Тестирование и отладка программы. При выполнении программы могут возникнуть ошибки 3 типов:

a. синтаксические – исправляются на этапе компиляции;
b. ошибки исполнения программы (деление на 0, логарифм от отрицательного числа и т. п.) – исправляются при выполнении программы;
c. семантические (логические) ошибки – появляются из-за неправильно понятой задачи, неправильно составленного алгоритма.

Чтобы устранить эти ошибки программа должна быть выполнена на некотором наборе тестов. Цель процесса тестирования – определение наличия ошибки, нахождение места ошибки, ее причины и соответствующие изменения программы – исправление. Тест – это набор исходных данных, для которых заранее известен результат. Тест выявивший ошибку считается успешным. Отладка программы заканчивается, когда достаточное количество тестов выполнилось неуспешно, т. е. программа на них выдала правильные результаты.

Для определения достаточного количества тестов существует два подхода. При первом подходе программа рассматривается как «черный ящик», в который передают исходные данные и получают результаты. Устройство самого ящика неизвестно. При этом подходе, чтобы осуществить полное тестирование, надо проверить программу на всех входных данных, что практически невозможно. Поэтому вводят специальные критерии, которые должны показать, какое конечное множество тестов является достаточным для программы. При первом подходе чаще всего используются следующие критерии:

1)тестирование классов входных данных, т. е. набор тестов должен содержать по одному представителю каждого класса данных:

X	0	1	0	1	-1	1	-1
Y	0	1	1	0	1	-1	-1

2)тестирование классов выходных данных, набор тестов должен содержать данные достаточные для получения по одному представителю из каждого класса выходных данных.

При втором подходе программа рассматривается как «белый ящик», для которого полностью известно устройство. Полное тестирование при этом подходе заканчивается после проверки всех путей, ведущих от начала программы к ее концу. Однако и при таком подходе полное тестирование программы невозможно, т. к. путей в программе с циклами бесконечное множество. При таком подходе используются следующие критерии:

1)Тестирование команд. Набор тестов должен обеспечивать прохождение каждой команды не менее одного раза.

2)Тестирование ветвей. Набор тестов в совокупности должен обеспечивать прохождение каждой ветви не менее одного раза. Это самый распространенный критерий в практике программирования.

5.1. Программирование ветвлений

Задача №1. Определить, попадет ли точка с координатами (x, y) в заштрихованную область.

Исходные данные: x,y

Результат: да или нет

Математическая модель:

Ok=I || II || III || VI, где I, II, III, IV – условия попадания точки в заштрихованную область для каждого квадранта.

Квадрант I: Область формируется прямыми 0X и 0У, прямой, проходящей через точки (0,1)и(1,0) и прямой, проходящей через точки (0,3) и (2,0).

Необходимо определить уравнения прямых y=ax+b. Решаем две системы уравнений:

1) 1=a*0+b;
0=a*1+b;
2) 2=a*0+b;
0=a*3+b;

Из этих систем получаем следующие уравнения прямых:

y=-1x+1;
y=-2/3x+1;

Тогда условие попадания точки в I квадрант будет выглядеть следующим образом:

y>=-x+1&&y<=-2/3x+2&&y>=0&&x>=0.

Квадранты II и III: Область формируется прямыми 0X и 0У и двумя окружностями, описываемыми формулами $x^2+y^2=1$, $x^2+y^2=9$.

Тогда условие попадания точки во II и III квадранты будет выглядеть следующим образом:

$x^2+y^2>=1$&& $x^2+y^2<=9$&&&&x<=0.

Квадрант IV:

Область формируется двумя прямоугольниками. Точка может попадать либо в первый прямоугольник, либо во второй.

Условие попадания точки в IV квадрант будет выглядеть следующим образом:

(x>=0&&x<=1&&y<=-1&&y>=-3)|| (x>=1&&x<=3&&y<=0&&y>=-3) .

Программа:

```
#include <iostream.h>
#include <math.h>

void main()
{
        float x,y;
        cout<<"\nEnter x,y";
        cin>>x>>y;
        bool Ok=(y>=-x+1&&y<=2/3*x+2&&x>=0&&y>=0)||
        (pow(x,2)+pow(y,2)>=1&&pow(x,2)+pow(y,2)<=9&&x<=0)||
        (x>=0&&x<=1&&y<=-1&&y>=-3)||(x>=1&&x<=2&&y<=0&&y>=-3);
        cout<<"\n"<<Ok;
}
```

Тесты:

Квадрант	Исходные данные (X,Y)	Результат (Ok)
I	0.2,0.2	0
I	0.7,0.5	1
II	-0.5, 0.5	0
II	-2,0	1
III	-0.5,-0,5	0
III	-2,-1	1
IV	0,5,-0.5	0
IV	1.5, -1	1
Центр системы координат	0,0	0

5.2. Программирование арифметических циклов.

Для арифметического цикла заранее известно сколько раз выполняется тело цикла.

Задача №2

Дана последовательность целых чисел из n элементов. Найти среднее арифметическое этой последовательности.

```
#include <iostream.h>
#include <math.h>
void main()
{
        int a,n,i,k=0;
        double s=0;
        cout<<"\nEnter n";
        cin>>n;
                for(i=1;i<=n;i++)
                {
                        cout<<"\nEnter a";
                        cin>>a;
                        s+=a;k++;
                }
                s=s/k;
                cout<<"\nSr. arifm="<<s<<"\n";
}
```

Тесты

N	5

A	1,2,3,4,5, 3
S	3

Задача №3

S=1+2+3+4+. . . +N

```
#include <iostream.h>
#include <math.h>
void main()
{
        int n,i,s=0;
        cout<<"\nEnter n";
        cin>>n;
if(n<=0) {cout<<”\nN<=0”;return;}
                for(i=1;i<=n;i++)s+=i;
                cout<<"\nS="<<s<<"\n";
}
```

Тесты

n	S
n=-1	N<=0
n=0	N<=0
n=5	S=15

Задача №4

S=15-17+19-21+ . . ., всего n слагаемых.

```
#include <iostream.h>
#include <math.h>
void main()
{
        int n,i,s=0,a=15;
        cout<<"\nEnter n";
        cin>>n;
if(n<=0) {cout<<”\nN<=0”;return;}
for(i=1;i<=n;i++)
{
if(i%2==1)s+=a;
else s-=a;
a+=2;
}
cout<<"\nS="<<s<<"\n";
}
```

Тесты

n	S
n=-1	N<=0
n=0	N<=0
n=3	S=17

5.3. Итерационные циклы

Для итерационного цикла известно условие выполнения цикла.

Задача №5

Дана последовательность целых чисел, за которой следует 0. Найти минимальный элемент этой последовательности.

```
#include <iostream.h>
#include <math.h>
void main()
{
```

```
int a,min;
cout<<"\nEnter a";
cin>>a;
min=a;
while(a!=0)//for(;a!=0;)
{
cout<<"\nEnter a";
cin>>a;
if (a!=0&&a<min)min=a;
}
cout<<"\nmin="<<min<<"\n";
}
```

Тесты:

a	2	5 5	-3	-1 0	0
min	-1 0				

a	1 2	5 5	4	2 7	0
min	4				

a	-6	-4 3	-1 5	-1 0	0
min	-1 0				

Задача №6 : Найти сумму чисел Фибоначчи, меньших заданного числа Q.

```
#include<iostream.h>
void main()
{
        int a=1,b=1,s=2,Q,c;
        cout<<"\nEnter Q";
        cin>>Q;
        if(Q<=0)cout<<"Error in Q";
        else
        if(Q==1)cout<<"\nS=1";
        else
        {
c=a+b;
while(c<Q) //for(;c!=0;)
        {
                s+=c;
                a=b;
                b=c;
                c=a+b;
        }
        cout<<"\nS="<<s<<"\n";
}
}
```

Тесты:

Q	S
-1	Error in Q

0	Error in Q
1	1
2	2
1 0	20

Тесты:

Q	
-1	Error in Q
0	Error in Q
1	2
2	2 3
1 0	2 3 5 7 11

5.4. Вложенные циклы

Задача №7: Напечатать N простых чисел.

```
#include<iostream.h>
void main()
{
	int a=1,n,d;
	cout<<"\nEnter N";
	cin>>n;
	for(int i=0;i<n;)//внешний цикл
	{
	a++;d=1;
	do //внутренний цикл
	{
		d++;
	}
	while(a%d!=0);//конец внутреннего цикла
	if(a==d){
		cout<<a<<" ";
		i++;}

	}//конец внешнего цикла
}
```

6. Массивы

В языке Си/Си++ ,кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых. Стандарт языка определяет три способа получения производных типов:

- массив элементов заданного типа;
- указатель на объект заданного типа;
- функция, возвращающая значение заданного типа.

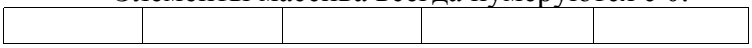
Массив – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти. Все элементы имеют одно имя - имя массива и отличаются индексами – порядковыми номерами в массиве. Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

6.1. Определение массива в Си/Си++

int a[100];//массив из 100 элементов целого типа

Операция sizeof(a) даст результат 400, т. е.100 элементов по 4 байта.

Элементы массива всегда нумеруются с 0.



0 1 2 …. 9

. 9

Чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс):

a[0] – индекс задается как константа,

a[55] – индекс задается как константа,

a[I] – индекс задается как переменная,

a[2*I] – индекс задается как выражение.

Элементы массива можно задавать при его определении:

int a[10]={1,2,3,4,5,6,7,8,9,10} ;

Операция sizeof(a) даст результат 40, т. е.10 элементов по 4 байта.

int a[10]={1,2,3,4,5};

Операция sizeof(a) даст результат 40, т. е.10 элементов по 4 байта. Если количество начальных значений меньше, чем объявленная длина массива, то начальные элементы массива получат только первые элементы.

int a[]={1,2,3,4,5};

Операция sizeof(a) даст результат 20, т. е.5 элементов по 4 байта. Длин массива вычисляется компилятором по количеству значений, перечисленных при инициализации.

6.2. Обработка одномерных массивов

При работе с массивами очень часто требуется одинаково обработать все элементы или часть элементов массива. Для этого организуется перебор массива.

Перебор элементов массива характеризуется:

- направлением перебора;
- количеством одновременно обрабатываемых элементов;
- характером изменения индексов.

По направлению перебора массивы обрабатывают :

- слева направо (от начала массива к его концу);
- справа налево (от конца массива к началу);
- от обоих концов к середине.

Индексы могут меняться

- линейно (с постоянным шагом);
- нелинейно (с переменным шагом).

6.2.1. Перебор массива по одному элементу

Элементы можно перебирать:

1) Слева направо с шагом 1, используя цикл с параметром

for(int I=0;I<n;I++){обработка a[I];}

2) Слева направо с шагом отличным от 1, используя цикл с параметром

for (int I=0;I<n;I+=step){обработка a[I];}

3) Справа налево с шагом 1, используя цикл с параметром

for(int I=n-1;I>=0;I--){обработка a[I];}

4) Справа налево с шагом отличным от 1, используя цикл с параметром

for (int I=n-1;I>=0;I-=step){обработка a[I];}

6.2.2 Формирование псевдодинамических массивов

При описании массива в программе надо обязательно указывать количество элементов массива для того, чтобы компилятор выделил под этот массив нужное количество памяти. Это не всегда бывает удобно, т. к. число элементов в массиве может меняться в зависимости от решаемой задачи. Динамические массивы реализуются с помощью указателей (см. далее).

Псевдодинамические массивы реализуются следующим образом:

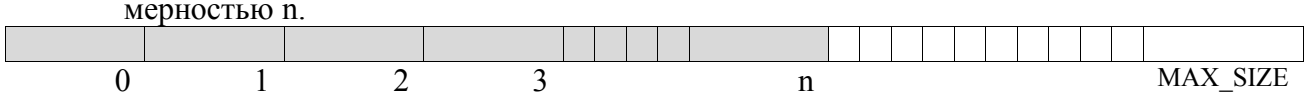
1) при определении массива выделяется достаточно большое количество памяти:

```
const int MAX_SIZE=100;//именованная константа
int mas[MAX_SIZE];
```

2) пользователь вводит реальное количество элементов массива меньшее N.

```
int n;
cout<<"\nEnter the size of array<"<<MAX_SIZE<<":";cin>>n;
```

3) дальнейшая работа с массивом ограничивается заданной пользователем размерностью n.



Т. о. используется только часть массива.

6.2.3. Использование датчика случайных чисел для формирования массива.

Датчик случайных чисел (ДСЧ) – это программа, которая формирует псевдослучайное число. Простейший ДСЧ работает следующим образом:

1) Берется большое число К и произвольное $x_0 \in [0,1]$.
2) Формируются числа x1=дробная_часть(x0*К); x2=дробная_часть(x1*К); и т. д.

В результате получается последовательность чисел x0, x1, x2,. . . беспорядочно разбросанных по отрезку от 0 до 1. Их можно считать случайными, а точнее псевдослучайными. Реальные ДСЧ реализуют более сложную функцию f(x).

В Си++ есть функция

int rand() – возвращает псевдослучайное число из диапазона 0..RAND_MAX=32767, описание функции находится в файле <stdlib.h>.

Пример формирования и печати массива с помощью ДСЧ:

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
}
```

В этой программе используется перебор массива по одному элементу слева направо с шагом 1.

Задача 1

Найти максимальный элемент массива.

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<”\nEnter the size of array:”;cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<” “;
}
int max=a[0];
for(I=1;I<n;I++)
if (a[I]>max)max=a[I];
cout<<”\nMax=”<<max”;
}
```

В этой программе также используется перебор массива по одному элементу слева направо с шагом 1.

Задача 2

Найти сумму элементов массива с четными индексами.

`#include<iostream.h>`
`#include<stdlib.h>`
`void main()`
`{`
`int a[100];`
`int n;`
`cout<<”\nEnter the size of array:”;cin>>n;`
`for(int I=0;I<n;I++)`
`{a[I]=rand()%100-50;`
`cout<<a[I]<<” “;`
`}`
`int Sum=0;`
`for(I=0;I<n;I+=2)`
`Sum+=a[I];//элементы с индексами 0, 2, 4… cout<<”\nSum=”<<Sum”;`
`}`
	Ввод массива

	`//Второй способ`
	`for(I=0;I<n;I++)`
	`if(I%2==0)Sum+=a[I]; ];//элементы с индексами 0, 2, 4…`
	`cout<<”\nSum=”<<Sum”;`

6.2.4. Перебор массива по два элемента

1) Элементы массива можно обрабатывать по два элемента, двигаясь с обеих сторон массива к его середине:

```
int I=0, J=N-1;
while( I<J)
    {обработка a[I] и a[J];I++;J--;}
```

2) Элементы массива можно обрабатывать по два элемента, двигаясь от начала к концу с

шагом 1(т. е. обрабатываются пары элементов a[1]и a[2], a[2]и a[3] и т. д.):
for (I=1;I<N;I++)
{обработка a[I] и a[I+1]}

3) Элементы массива можно обрабатывать по два элемента, двигаясь от начала к концу с шагом 2 (т. е. обрабатываются пары элементов a[1]и a[2], a[3]и a[4] и т. д.)
int I=1;
while (I<N-1)
{обработка a[I] и a[I+1];
I+=2;}

6.3. Классы задач по обработке массивов

1) К задачам 1 класса относятся задачи, в которых выполняется однотипная обработка всех или указанных элементов массива.
2) К задачам 2 класса относятся задачи, в которых изменяется порядок следования элементов массива.
3) К задачам 3 класса относятся задачи, в которых выполняется обработка нескольких массивов или подмассивов одного массива. Массивы могут обрабатываться по одной схеме – синхронная обработка или по разным схемам – асинхронная обработка массивов.
4) К задачам 4 класса относятся задачи, в которых требуется отыскать первый элемент массива, совпадающий с заданным значением – поисковые задачи в массиве.

6.3.1. Задачи 1-ого класса

Решение таких задач сводится к установлению того, как обрабатывается каждый элемент массива или указанные элементы, затем подбирается подходящая схема перебора, в которую вставляются операторы обработки элементов массива. Примером такой задачи является нахождение максимального элемента массива или среднего арифметического массива.

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
int Sum=0;
for(I=0;I<n;I++)
Sum+=a[I];
Cout<<"Среднее арифметическое="<<Sum/n";
}
```

6.3.2. Задачи 2-ого класса

Обмен элементов внутри массива выполняется с использованием вспомогательной переменной:

```
int R=a[I];a[I]=a[J]; a[J]:=R; // обмен a[I] и a[J] элементов массива.
```

Пример1.
Перевернуть массив.

```
//формирование массива
for(int i=0,j=n-1;i<j;i++,j--)
{int r=a[i];
a[i]=a[j];
a[j]=r;}
//вывод массива
```

Пример 2.

Поменять местами пары элементов в массиве: 1и2, 3 и 4, 5 и 6 и т. д.

```
for(int i=0;i<n-1;i+=2)
{int r=a[i];
a[i]=a[i+1];
a[i+1]=r;}
```

Пример 3.

Циклически сдвинуть массив на к элементов влево (вправо).

```
int k,i,t,r;
cout<<"\nK=?";cin>>k;

for(t=0;t<k;t++)
{
        r=a[0];
        for(int i=0;i<n-1;i++)
                a[i]=a[i+1];
        a[n-1]=r;
}
```

6.3.3. Задачи 3-ого класса

При синхронной обработке массивов индексы при переборе массивов меняются одинаково.

Пример 1. Заданы два массива из n целых элементов. Получить массив c, где c[I]=a [I]+b[I].

```
For(int I=0;I<n;I++)c[I]=a[I]+b[I];
```

При асинхронной обработке массивов индекс каждого массива меняется по своей схеме.

Пример 2. В массиве целых чисел все отрицательные элементы перенести в начало массива.

```
int b[10];//вспомогательный массив
int i,j=0;
for(i=0;i<n;i++)
        if(a[i]<0){b[j]=a[i];j++;}//переписываем из a в b все отрицательные
элементы
for(i=0;i<n;i++)
        if(a[i]>=0){b[j]=a[i];j++;}// переписываем из a в b все положительные
элементы
for(i=0;i<n;i++) cout<<b[I]<<” “;
```

Пример3.

Удалить из массива все четные числа

```
int b[10];
        int i,j=0;
        for(i=0;i<n;i++)
                if(a[i]%2!=0){b[j]=a[i];j++;}

                for(i=0;i<j;i++) cout<<b[i]<<" ";
                        cout<<"\n";
```

6.3.4. Задачи 4-ого класса

В поисковых задачах требуется найти элемент, удовлетворяющий заданному условию. Для этого требуется организовать перебор массива и проверку условия. Но при этом существует две возможности выхода из цикла:

-нужный элемент найден ;

-элемент не найден, но просмотр массива закончен.

Пример1. Найти первое вхождение элемента К в массив целых чисел.

```
int k;
```

```
cout<<"\nK=?";cin>>k;
int ok=0;//признак найден элемент или нет
int i,nom;
for(i=0;i<n;i++)
        if(a[i]==k){ok=1;nom=i;break;}
if(ok==1)
                cout<<"\nnom="<<nom;
else
        cout<<"\nthere is no such element!";
```

6.4. Сортировка массивов

Сортировка – это процесс перегруппировки заданного множества объектов в некотором установленном порядке.

Сортировки массивов подразделяются по быстродействию. Существуют простые методы сортировок, которые требуют n*n сравнений, где n – количество элементов массива и быстрые сортировки, которые требуют n*ln(n) сравнений. Простые методы удобны для объяснения принципов сортировок, т. к. имеют простые и короткие алгоритмы. Усложненные методы требуют меньшего числа операций, но сами операции более сложные, поэтому для небольших массивов простые методы более эффективны.

Простые методы подразделяются на три основные категории:

-сортировка методом простого включения;

-сортировка методом простого выделения;

-сортировка методом простого обмена;

6.4.1. Сортировка методом простого включения (вставки)

Элементы массива делятся на уже готовую последовательность и исходную. При каждом шаге, начиная с I=2, из исходной последовательности извлекается I-ый элемент и вставляется на нужное место готовой последовательности, затем I увеличивается на 1 и т. д.

44	5	1	4	9	1
	5	2	2	4	8

готовая исходная

В процессе поиска нужного места осуществляются пересылки элементов больше выбранного на одну позицию вправо, т. е. выбранный элемент сравнивают с очередным элементом отсортированной части, начиная с J:=I-1. Если выбранный элемент больше a[I], то его включают в отсортированную часть, в противном случае a[J] сдвигают на одну позицию, а выбранный элемент сравнивают со следующим элементом отсортированной последовательности. Процесс поиска подходящего места заканчивается при двух различных условиях:

- если найден элемент a[J]>a[I];
- достигнут левый конец готовой последовательности.

```
int i,j,x;
for(i=1;i<n;i++)
{
x=a[i];//запомнили элемент, который будем вставлять
j=i-1;
while(x<a[j]&&j>=0)//поиск подходящего места
{
a[j+1]=a[j];//сдвиг вправо
        j--;
}
a[j+1]=x;//вставка элемента
}
```

6.4.2. Сортировка методом простого выбора

Выбирается минимальный элемент массива и меняется местами с первым элементом массива. Затем процесс повторяется с оставшимися элементами и т. д.

4	5	12	4	9	1
4	5		2	4	8

1 ми

н

```
int i,min,n_min,j;
for(i=0;i<n-1;i++)
{
        min=a[i];n_min=i;//поиск минимального
        for(j=i+1;j<n;j++)
                if(a[j]<min){min=a[j];n_min=j;}
                a[n_min]=a[i];//обмен
                a[i]=min;
}
```

6.4.3. Сортировка методом простого обмена

Сравниваются и меняются местами пары элементов, начиная с последнего. В результате самый маленький элемент массива оказывается самым левым элементом массива. Процесс повторяется с оставшимися элементами массива.

4	5	1	4	9	1
4	5	2	2	4	8

```
for(int i=1;i<n;i++)
for(int j=n-1;j>=i;j--)
if(a[j]<a[j-1])
{int r=a[j];a[j]=a[j-1];a[j-1]=r;}
}
```

6.5. Поиск в отсортированном массиве

В отсортированном массиве используется дихотомический (бинарный) поиск. При последовательном поиске требуется в среднем n/2 сравнений, где n – количество элементов в массиве. При дихотомическом поиске требуется не более m сравнений, если n- m-ая степень 2, если n не является степенью 2, то $n<k=2^m$.

Массив делится пополам S:=(L+R)/ 2+1 и определяется в какой части массива находится нужный элемент X. Т .к. массив упорядочен, то если a[S]<X, то искомый элемент находится в правой части массива, иначе - находится в левой части. Выбранную часть массива снова надо разделить пополам и т. д., до тех пор, пока границы отрезка L и R не станут равны.

1	3	8	10	11	15	19	21	23	37
0	1	2	3	4	5	6	7	8	9

L S R

S=(L+R)/2=4

```
int b;
cout<<"\nB=?";cin>>b;
int l=0,r=n-1,s;
do
{
        s=(l+r)/2;//средний элемент
```

```
        if(a[s]<b)l=s+1;//перенести леую границу
        else r=s;//перенести правую границу
}while(l!=r);
        if(a[l]==b)return l;
        else return -1;
```

7. Указатели

7.1. Понятии указателя

Указатели являются специальными объектами в программах на Си++. Указатели предназначены для хранения адресов памяти.

Пример: Когда компилятор обрабатывает оператор определения переменной, например, int i=10;, то в памяти выделяется участок памяти в соответствии с типом переменной (int=> 4байта) и записывает в этот участок указанное значение. Все обращения к этой переменной компилятор заменит на адрес области памяти, в которой хранится эта переменная.

a

Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом.

Указатели делятся на две категории: указатели на объекты и указатели на функции. Рассмотрим указатели на объекты, которые хранят адрес области памяти, содержащей данные определенного типа .

В простейшем случае объявление указателя имеет вид:

тип *имя;

Тип может быть любым, кроме ссылки.

Примеры:

```
int *i;
double *f, *ff;
char *c;
```

Размер указателя зависит от модели памяти. Можно определить указатель на указатель: int**a;

Указатель может быть константой или переменной, а также указывать на константу или переменную.

Примеры:

```
1. int i;          //целая переменная
const int ci=1; //целая константа
int *pi;          //указатель на целую переменную
const int *pci;//указатель на целую константу
```

Указатель можно сразу проинициализировать:

```
int *pi=&i;     //указатель на целую переменную
const int *pci=&ci;;//указатель на целую константу
   2. int*const cpi=&i;//указатель-константа на целую переменную
const int* const cpc=&ci;//указатель-константа на целую константу
```

Если модификатор const относится к указателю (т. е. находится между именем указателя и *), то он запрещает изменение указателя, а если он находится слева от типа (т. е. слева от *), то он запрещает изменение значения, на которое указывает указатель.

Для инициализации указателя существуют следующие способы:

1) Присваивание адреса существующего объекта:

1) с помощью операции получения адреса

```
int a=5;
int *p=&a; или int p(&a);
```

2) с помощью проинициализированного указателя

```
int *r=p;
```

3) адрес присваивается в явном виде

```
char*cp=(char*)0x B800 0000;
```

где 0x B800 0000 – шестнадцатеричная константа, (char*) – операция приведения

типа.

4) присваивание пустого значения:

```
int*N=NULL;
int *R=0;
```

7.2. Динамические переменные

Все переменные, объявленные в программе размещаются в одной непрерывной области памяти, которую называют сегментом данных (64К). Такие переменные не меняют своего размера в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из этой ситуации является использование динамической памяти. Динамическая память – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

Для создания динамических переменных используют операцию new, определенную в СИ++:

указатель = new имя_типа[инициализатор];

где инициализатор – выражение в круглых скобках.

Операция new позволяет выделить и сделать доступным участок динамической памяти, который соответствует заданному типу данных. Если задан инициализатор, то в этот участок будет занесено значение, указанное в инициализаторе.

```
int*x=new int(5);
```

Для удаления динамических переменных используется операция delete, определенная в СИ++:

delete указатель;

где указатель содержит адрес участка памяти, ранее выделенный с помощью операции new.

```
delete x;
```

В языке Си определены библиотечные функции для работы с динамической памятью, они находятся в библиотеке <stdlib.h>:

1) void*malloc(unsigned s) – возвращает указатель на начало области динамической памяти длиной s байт, при неудачном завершении возвращает NULL;
2) void*calloc(unsigned n, unsigned m) – возвращает указатель на начало области динамической для размещения n элементов длиной m байт каждый, при неудачном завершении возвращает NULL;
3) void*realloc(void *p,unsigned s) –изменяет размер блока ранее выделенной динамической до размера s байт, p – адрес начала изменяемого блока, при неудачном завершении возвращает NULL;
4) void *free(void *p) – освобождает ранее выделенный участок динамической памяти, p- адрес начала участка.

Пример:

```
int *u=(int*)malloc(sizeof(int)); // в функцию передается количество требуемой памяти в байтах, т. к. функция возвращает значение типа void*, то его необходимо преобразовать к типу указателя (int*).
free(u); //освобождение выделенной памяти
```

7.3. Операции с указателями

С указателями можно выполнять следующие операции:

1) разыменование (*);
2) присваивание;
3) арифметические операции (сложение с константой, вычитание, инкремент ++, декремент --);
4) сравнение;
5) приведение типов.

1) Операция разыменования предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, также используя операцию разыменования.

Примеры:

```
int a; //переменная типа int
int*pa=new int; //указатель и выделение памяти под динамическую переменную
*pa=10;//присвоили значение динамической переменной, на которую указывает указатель
a=*pa;//присвоили значение переменной a
```

Присваивать значение указателям-константам запрещено.

2) Приведение типов

На одну и ту же область памяти могут ссылаться указатели разного типа. Если применить к ним операцию разыменования, то получатся разные результаты.

```
int a=123;
int*pi=&a;
char*pc=(char*)&a;
float *pf=(float*)&a;
printf("\n%x\t%i",pi,*pi);
printf("\n%x\t%c",pc,*pc);
printf("\n%x\t%f",pf,*pf);
```

При выполнении этой программы получатся следующие результаты:

```
66fd9c  123
66fd9c  {
66fd9c  0.000000
```

Т. е. адрес у трех указателей один и тот же, но при разыменовании получаются разные значения в зависимости от типа указателя.

В примере при инициализации указателя была использована операция приведения типов. При использовании в выражении указателей разных типов, явное преобразование требуется для всех типов, кроме void*. Указатель может неявно преобразовываться в значения типа bool, при этом ненулевой указатель преобразуется в true, а нулевой в false.

3) Арифметические операции применимы только к указателям одного типа.

-Инкремент увеличивает значение указателя на величину sizeof(тип).

Например:

```
char *pc;
int *pi;
float *pf;
. . . . .
pc++;//значение увеличится на 1
pi++;//значение увеличится на 4
pf++;//значение увеличится на 4
```

- Декремент уменьшает значение указателя на величину sizeof(тип).

-Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

Например:

```
int a=123,b=456,c=789;
int*pi1=&a;
int *pi2=&b;
int*pi3=&c;
```

```
printf("\n%x",pi1-pi2);
printf("\n%x",pi1-pi3);
```

Результат

1

2

Суммирование двух указателей не допускается.

Можно суммировать указатель и константу:

```
pi3=pi3+2;
pi2=pi2+1;
printf("\n%x\t%d",pi1,*pi1);
printf("\n%x\t%d",pi2,*pi2);
printf("\n%x\t%d",pi3,*pi3);
```

Результат выполнения программы:

66fd9c 123

66fd9c 123

66fd9c 123

При записи выражений с указателями требуется обращать внимание на приоритеты операций.

8. Ссылки

8.1. Понятие ссылки

Ссылка – это синоним имени объекта, указанного при инициализации ссылки.

Формат объявления ссылки

тип & имя =имя_объекта;

Примеры:

```
int  x;// определение переменной
int&  sx=x;// определение ссылки на переменную x
const char& CR='\n';//определение ссылки на константу
```

8.1. Правила работы со ссылками:

1) Переменная ссылка должна явно инициализироваться при ее описании, если она не является параметром функции, не описана как extern или не ссылается на поле класса.
2) После инициализации ссылке не может быть присвоено другое значение.
3) Не существует указателей на ссылки, массивов ссылок и ссылок на ссылки.
4) Операция над ссылкой приводит к изменению величины на которую она ссылается

Ссылка не занимает дополнительного пространства в памяти, она является просто другим именем объекта.

Пример1:

```
#include <iostream.h>
void main()
{
int I=123;
int &si=I;
cout<<"\ni="<<I<<" si="<<si;
I=456;
cout<<"\ni="<<I<<" si="<<si;
I=0; cout<<"\ni="<<I<<" si="<<si;
}
```

Выведется

I=123 si=123

I=456 si=456

I=0 si=0

9. Указатели и массивы

9.1. Одномерные массивы и указатели

При определении массива ему выделяется память. После этого имя массива воспринимается как константный указатель того типа, к которому относятся элементы массива. Исключением является использовании операции sizeof (имя_массива) и операции &имя_массива.

Примеры:

```
int a[100];
int k=sizeof(a);// результатом будет 4*100=400 (байтов).
int n=sizeof(a)/sizeof(a[0]);//количество элементов массива
```

Результатом операции & является адрес нулевого элемента массива:

имя_массива==&имя_массива=&имя_массива[0]

Имя массива является указателем-константой, значением которой служит адрес первого элемента массива, следовательно, к нему применимы все правила адресной арифметики, связанной с указателями. Запись имя_массива[индекс] это выражение с двумя операндами: имя массива и индекс. Имя_массива - это указатель константа, а индекс определяет смещение от начала массива. Используя указатели, обращение по индексу можно записать следующим образом: *(имя_массива+индекс).

Пример:

```
for(int i=0;i<n;i++)//печать массива
cout<<*(a+i)<<" ";//к имени адресу массива добавляется константа i  и полученное // значение разыменовывается
```

Так как имя массива является константным указателем, то его невозможно изменить, следовательно, запись *(a++) будет ошибочной, а *(a+1) - нет.

Указатели можно использовать и при определении массивов:

```
int  a[100]={1,2,3,4,5,6,7,8,9,10};
int * na=a;//поставили указатель на уже определенный массив
int b=new int[100];//выделили в динамической памяти место под массив из 100 элементов
```

9.2. Многомерные массивы и указатели

Многомерный массив это массив, элементами которого служат массивы. Например, массив с описанием int a[4][5] – это массив из 4 указателей типа int*, которые содержат адреса одномерных массивов из 5 целых элементов (см. рис.).

int **a

Рис.

Инициализация многомерных массивов выполняется аналогично одномерным массивам. Примеры:

```
int a[3][4] = {{11,22,33,44},{55,66,77,88},{99,110,120,130}};//проинициализированы все //элементы массива
int b[3][4] = {{1},{2},{3}};//проинициализированы первые элементы каждой строки
int c[3][2]={1,2,3,4,5,6};//проинициализированы все элементы массива
```

Доступ к элементам многомерных массивов возможен и с помощью индексированных переменных и с помощью указателей:

a[1][1] – доступ с помощью индексированных переменных,

((a+1)+1) – доступ к этому же элементу с помощью указателей (см. рис.).

19.3. Динамические массивы

Операция new при использовании с массивами имеет следующий формат:

new тип_массива

Такая операция выделяет для размещения массива участок динамической памяти соответствующего размера, но не позволяет инициализировать элементы массива. Операция new возвращает указатель, значением которого служит адрес первого элемента массива. При выделении динамической памяти размеры массива должны быть полностью определены.

Примеры:

1. int *a=new int[100];//выделение динамической памяти размером 100*sizeof(int) байтов double *b=new double[10];// выделение динамической памяти размером 10*sizeof (double) байтов

2. long(*la)[4];//указатель на массив из 4 элементов типа long

la=new[2][4];//выделение динамической памяти размером 2*4*sizeof(long) байтов

3. int**matr=(int**)new int[5][10];//еще один способ выделения памяти под двумерный //массив

4. int **matr;
   matr=new int*[4];//выделяем память под массив указателей int* их n элементов
   for(int I=0;I<4;I++)matr[I]=new int[6];//выделяем память под строки массива

Указатель на динамический массив затем используется при освобождении памяти с помощью операции delete.

Примеры:

delete[] a;//освобождает память, выделенную под массив, если a адресует его начало

delete[]b;

delete[] la;

for(I=0;I<4;I++)delete [] matr[I];//удаляем строки

delete [] matr;//удаляем массив указателей

Пример

Удалить из матрицы строку с номером K

```
#include <iostream.h>
#include <string.h>

#include <iostream.h>
#include <stdlib.h>
void main()
{
        int n,m;//размерность матрицы
        int i,j;
        cout<<"\nEnter n";
        cin>>n;//строки
        cout<<"\nEnter m";
        cin>>m;//столбцы
        //выделение памяти
        int **matr=new int* [n];// массив указателей на строки
        for(i=0;i<n;i++)
                matr[i]=new int [m];//память под элементы матрицы
        //заполнение матрицы
        for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                matr[i][j]=rand()%10;//заполнение матрицы
        //печать сформированной матрицы
        for(i=0;i<n;i++)
        {
```

```
		for(j=0;j<m;j++)
			cout<<matr[i][j]<<" ";
		cout<<"\n";
	}
	//удаление строки с номером к
	int k;
	cout<<"\nEnter k";
	cin>>k;
	int**temp=new int*[n-1];//формирование новой матрицы
	for(i=0;i<n;i++)
		temp[i]=new int[m];
	//заполнение новой матрицы
	int t;
	for(i=0,t=0;i<n;i++)
		if(i!=k)
		{
			for(j=0;j<m;j++)
				temp[t][j]=matr[i][j];
			t++;
		}

		//удаление старой матрицы
	for(i=0;i<n;i++)
		delete matr[i];
	delete[]matr;
n--;
	//печать новой матрицы

	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
			cout<<temp[i][j]<<" ";
		cout<<"\n";
	}
}
```

10. Символьная информация и строки

Для символьных данных в Си++ введен тип char. Для представления символьной информации используются символы, символьные переменные и текстовые константы.

Примеры:

const char c='c'; //символ – занимает один байт, его значение не меняется

char a,b;//символьные переменные, занимают по одному байту, значения меняются

const char *s="Пример строки\n" ;//текстовая константа

Строка в Си++ - это массив символов, заканчивающийся нуль-символом – '\0' (нуль-терминатором). По положению нуль-терминатора определяется фактическая длина строки. Количество элементов в таком массиве на 1 больше, чем изображение строки.

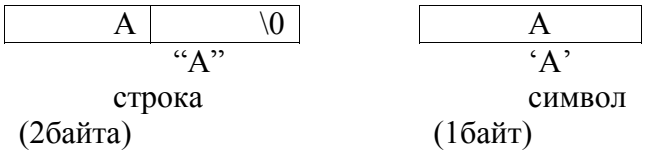


Рис.2. Представление строки и символа

Присвоить значение строке с помощью оператора присваивания нельзя. Поместить строку в массив можно либо при вводе, либо с помощью инициализации.

Пример

```
void main()
{
        char s1[10]="string1";
        int k=sizeof(s1);
        cout<<s1<<"\t"<<k<<endl;
        char s2[]="string2";
        k=sizeof(s2);
        cout<<s2<<"\t"<<k<<endl;
        char s3[]={'s','t','r','i','n','g','3'}
        k=sizeof(s3);
        cout<<s3<<"\t"<<k<<endl;
        char *s4="string4";//указатель на строку, ее нельзя изменить
        k=sizeof(s4);
        cout<<s4<<"\t"<<k<<endl;


}
```

Результаты:

string1 10 – выделено 10 байтов, в том числе под \0

string2 8 – выделено 8 байтов (7+1байт под \0)

string3 8 – выделено 8 байтов (7+1байт под \0)

string4 4 – размер указателя

Примеры:

char *s="String5"; - выделяется 8 байтов для строки

char*ss; - описан указатель

ss="String6";//память не выделяется , поэтому программа может закончиться аварийно.

char *sss=new char[10];//выделяем динамическую память

strcpy(sss,"String7");//копируем строку в память

Для ввода и вывода символьных данных в библиотеке языка СИ определены следующие функции:

int getchar(void) - осуществляет вод одного символа их входного потока, при этом

она возвращает один байт информации (символ) в виде значения типа int. Это сделано для распознавания ситуации, когда при чтении будет достигнут конец файла.

int putchar (int c) – помещает в стандартный выходной поток символ c.

char* gets(char*s) – считывает строку s из стандартного потока до появления символа ‘\n’, сам символ ‘\n’ в строку не заносится.

int puts(const char* s) записывает строку в стандартный поток, добавляя в конец строки символ ‘\n’, в случае удачного завершения возвращает значение больше или равное 0 и отрицательное значение (EOF=-1) в случае ошибки.

Примеры:

1. char s[20];
cin>>s;//ввод строки из стандартного потока
cout<<s;//вывод строки в стандартный поток
Результат работы программы:

При вводе строки “123 456 789”, чтение байтов осуществляется до первого пробела, т. е. в строку s занесется только первое слово строки “123/0”, следовательно, выведется: 123.

2. char s[20];
gets(s);//ввод строки из стандартного потока
puts(s);//вывод строки в стандартный поток
Результат работы программы:

При вводе строки “123 456 789”, чтение байтов осуществляется до символа ‘\n’, т. е. в s занесется строка”123 456 789\n\0”, при выводе строки функция puts возвращает еще один символ ‘\n’, следовательно, будет выведена строка “123 456 789\n\n”.

3. char s[20];
scanf(“%s”,s);//ввод строки из стандартного потока
printf(“%s”,s);//вывод строки в стандартный поток
Результат работы программы:

При вводе строки “123 456 789”, чтение байтов осуществляется до первого пробела, т. е. в строку s занесется только первое слово строки “123/0”, следовательно, выведется: 123. Т. к. s – имя массива, т. е. адрес его первого элемента, операция & в функции scanf не используется.

Для работы со строками существуют специальные библиотечные функции, которые содержатся в заголовочном файле **string.h**. Рассмотрим некоторые из этих функций:

Прототип функции	Краткое описание	Примечание
unsigned **strlen**(const char*s);	Вычисляет длину строки s.	
int **strcmp**(const char*s1, const char *s2);	Сравнивает строки s1 и s2.	Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный.
int **strcnmp**(const char*s1, const char *s2);	Сравнивает первые n символов строк s1 и s2.	Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный.
char***strcpy**(char*s1, const char*s2);	Копирует символы строки s1 в строку s2.	
char***strncpy**(char*s1, const char*s2, int n);	Копирует n символов строки s1 в строку s2.	Конец строки отбрасывается или дополняется пробелами.
char***strcat**(char*s1, const char*s2);	Приписывает строку s2 к строке s1	

char***strncat**(char*s1, const char*s2);	Приписывает первые n символов строки s2 к строке s1	
char***strdup**(const char*s);	Выделяет память и переносит в нее копию строки s	При выделении памяти используются функции

Пример1:

Дана строка символов, состоящая из слов, слова разделены между собой пробелами. Удалить из строки все слова, начинающиеся с цифры.

```
#include <stdio.h>
#include <string.h>
void main()
{
char s[250], //исходная строка
w[25], //слово
mas[10][25];//массив слов
puts("\nвведите строку");
gets(s);
int k=0,t=0,i,len,j;
len=strlen(s);
while(t<len)
{
for(j=0,i=t;s[i]!=' ';i++,j++)w[j]=s[i];//формируем слово до пробела
w[j]='/0';//формируем конец строки
strcpy(mas[k],w);//копируем слово в массив
k++;//увеличиваем счетчик слов
t=i+1;//переходим к следующему слову в исходной строке
}
strcpy(s,"");//очищаем исходную строку
for(t=0;t<k;t++)
if(mas[t][0]<'0'&&mas[t][0]>'9')//если первый символ не цифра
{
strcat(s,mas[t]);//копируем в строку слово
strcat(s," ");//копируем в строку пробел
}
puts(s);//выводим результат
}
```

Пример2:

Сформировать динамический массив строк. Удалить из него строку с заданным номером.

```
#include <iostream.h>
#include <string.h>
void main()
{
        int n;
        cout<<"\nN=?";cin>>n;
        char s[100];
        char**matr=new char*[n];
        for(int i=0;i<n;i++)
        {
                cout<<"\nS=?";
                cin>>s;
                matr[i]=new char[strlen(s)];
                strcpy(matr[i],s);
        }
```

```
	for(i=0;i<n;i++)
	{
		cout<<matr[i];
		cout<<"\n";
	}
	int k;
	cout<<"\nK=?";
	cin>>k;
	if(k>=n){cout<<"There is not such string\n";return;}
	char **temp=new char*[n-1];
	int j=0;

	for(i=0;i<n;i++)
		if(i!=k)
		{
			temp[j]=new char[strlen(matr[i])];
			strcpy(temp[j],matr[i]);
			j++;
		}

		n--;
	for(i=0;i<n;i++)
	{
		cout<<temp[i];
		cout<<"\n";
	}
}
```

11. Функции в Си++

С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части. В Си++ задача может быть разделена на более простые подзадачи с помощью функций. Разделение задачи на функции также позволяет избежать избыточности кода, т. к. функцию записывают один раз, а вызывают многократно. Программу, которая содержит функции, легче отлаживать.

Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

11. 1. Объявление и определение функций

Функция – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т. д.

Функция, во-первых, является одним из производных типов СИ++, а ,во-вторых, минимальным исполняемым модулем программы.

Любая функция должна быть объявлена и определена.

Объявление функции (прототип, заголовок) задает имя функции, тип возвращаемого значения и список передаваемых параметров.

Определение функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов.

тип имя_функции([список_формальных_параметров])
{ тело_функции}

Тело_функции – это блок или составной оператор. Внутри функции нельзя определить другую функцию.

В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь 2 формы:

1) return выражение;
2) return;

Первая форма используется для возврата результата, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Вторая форма используется, если функция не возвращает значения, т. е. имеет тип void. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед }.

Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.

Список формальных параметров – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имена можно не указывать.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. При вызове указываются: имя функции и фактические параметры. Фактические параметры заменяют формальные параметры при выполнении операторов тела функции. Фактические и формальные параметры должны совпадать по количеству и типу.

Объявление функции должно находиться в тексте раньше вызова функции, чтобы компилятор мог осуществить проверку правильности вызова. Если функция имеет тип не

void, то ее вызов может быть операндом выражения.

Пример:

Заданы координаты сторон треугольника. Если такой треугольник существует, то найти его площадь.

1. Математическая модель:
   1) l=sqrt(pow(x1-x2,2)+pow(y1-y2,2));//длина стороны треугольника
   2) p=(a+b+c)/2;
      s=sqrt(p*(p-a)*(p-b)*(p-c));//формула Герона
   3) проверка существования треугольника
      (a+b>c&&a+c>b&&c+b>a)
2. Алгоритм:
   1) Ввести координаты сторон треугольника (x1,y1),(x2,y2),(x3,y3);
   2) Вычислить длины сторон ab, bc, ca;
   3) Проверить существует ли треугольник с такими сторонами. Если да, то вычислить площадь и вывести результат.
   4) Если нет, то вывести сообщение.
   5) Если все координаты равны 0, то конец, иначе возврат на п.1.

```
#include <iostream.h>
#include <math.h>

double line(double x1,double y1,double x2,double y2)
{
//функция возвращает длину отрезка, заданного координатами x1,y1 и x2,y2
return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
double square(double a, double b, double c)
{
//функция возвращает площадь треугольника, заданного длинами сторон a,b,c
double s, p=(a+b+c)/2;
return s=sqrt(p*(p-a)*(p-b)*(p-c));//формула Герона
}
bool  triangle(double a, double b, double c)
{
//возвращает true, если треугольник существует
if(a+b>c&&a+c>b&&c+b>a) return true;
else return false;
}
void main()
{
double x1=1,y1,x2,y2,x3,y3;
double point1_2,point1_3,point2_3;
do
{
cout<<"\nEnter koordinats of triangle:";
cin>>x1>>y1>>x2>>y2>>x3>>y3;
point1_2=line(x1,y1,x2,y2);
point1_3=line(x1,y1,x3,y3);
point2_3=line(x2,y2,x3,y3);
if(triangle(point1_2,point1_3,point2_3)==true)
cout<<"S="<<square(point1_2,point2_3,point1_3)<<"\n";
else cout<<"\nTriagle doesnt exist";
}
```

```
while(!(x1==0&&y1==0&&x2==0&&y2==0&&x3==0&&y3==0));
}
```

11.2.Прототип функции

Для того, чтобы к функции можно было обратиться, в том же файле должно находиться определение или описание функции (прототип).

```
double line(double x1,double y1,double x2,double y2);
double square(double a, double b, double c);
bool  triangle(double a, double b, double c);
double line(double ,double ,double ,double);
double square(double , double , double );
bool  triangle(double , double , double );
```

Это прототипы функций, описанных выше.

При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей и храниться в откомпилированном виде в библиотеке объектных модулей. Это относится и к функциям из стандартных модулей. В этом случае определения библиотечных функций уже оттранслированные и оформленные в виде объектных модулей, находятся в библиотеке компилятора, а описания функций необходимо включать в программу дополнительно. Это делают с помощью препроцессорных команд include< имя файла>.

Имя_файла – определяет заголовочный файл, содержащий прототипы группы стандартных для данного компилятора функций. Например, почти во всех программах мы использовали команду #include <iostream.h> для описания объектов потокового ввода-вывода и соответствующие им операции.

При разработке своих программ, состоящих из большого количества функций, и , размещенных в разных модулях, прототипы функций и описания внешних объектов (констант, переменных, массивов) помещают в отдельный файл, который включают в начало каждого из модулей программы с помощью директивы include”имя_файла”.

11.3.Параметры функции

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

<u>При передаче по значению</u> выполняются следующие действия:

-вычисляются значения выражений, стоящие на месте фактических параметров;

-в стеке выделяется память под формальные параметры функции;

-каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

Пример:

```
double square(double a, double b, double c)
{
//функция возвращает площадь треугольника, заданного длинами сторон a,b,c
double s, p=(a+b+c)/2;
return s=sqrt(p*(p-a)*(p-b)*(p-c));//формула Герона
}
1) double s1=square(2.5,2,1);
2) double a=2.5,b=2,c=1;
double s2=square(a,b,c);
3) double x1=1,y1=1,x2=3,y2=2,x3=3,y3=1;
double s3=square(sqrt(pow(x1-x2,2)+pow(y1-y2,2)),//расстояние между 1и2
 sqrt(pow(x1-x3,2)+pow(y1-y3,2)), //расстояние между 1 и 3
sqrt(pow(x3-x2,2)+pow(y3-y2,2)));//расстояние между 2 и3
```

Стек

A	2.5
B	2
C	1
S	
P	

P и S – локальные переменные.

Т. о. в стек заносятся копии фактических параметров и операторы функции работают с этими копиями. Доступа к самим фактическим параметрам у функции нет, следовательно нет возможности их изменить.

При передаче по адресу в стек заносятся копии адресов параметров, следовательно, у функции появляется доступ к ячейке памяти, в которой находится фактический параметр и она может его изменить.

Пример.

```
void Change(int a,int b)//передача по значению
{int r=a;a=b;b=r;}

int x=1,y=5;
Change(x,y);
```

A	1	5
B	5	1
r		1

```
cout<<"x="<<x<<"y="<<y;
```

выведется: x=1y=5

```
void Change(int *a,int *b)//передача по адресу
{int r=*a;*a=*b;*b=r;}
int x=1,y=5;
Change(&x,&y);
```

A	&x	5
B	&y	1
r		1

```
cout<<"x="<<x<<"y="<<y;
```

выведется: x=5y=1

Для передачи по адресу также могут использоваться ссылки. При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

```
void Change(int &a,int &b)
{int r=a;a=b;b=r;}
int x=1,y=5;
Change(x,y);
```

A	&x	5
B	&y	1
r		1

```
cout<<"x="<<x<<"y="<<y;
```

выведется: x=5y=1

Использование ссылок вместо указателей улучшает читаемость программы, т. к. не надо применять операцию разыменовывания. Использование ссылок вместо передачи по значению также более эффективно, т .к. не требует копирования параметров. Если требуется запретить изменение параметра внутри функции, используется модификатор const. Рекомендуется ставить const перед всеми параметрами, изменение которых в функции не предусмотрено (по заголовку будет понятно, какие параметры в ней будут изменяться, а какие нет).

11.4. Локальные и глобальные переменные

Переменные, которые используются внутри данной функции, называются локальными. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т. к. память, выделенная такой переменной будет освобождаться.

```
int*f()
{
int a;
. . . .
return&a;// НЕВЕРНО
}
```

Глобальные переменные – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

Пример:

```
int a,b;//глобальные переменные
void change()
{
int r;//локальная переменная
r=a;a=b;b=r;
}
void main()
{
cin>>a,b;
change();
cout<<”a=”<<a<<”b=”<<b;
}
```

Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т. к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

11.5.Функции и массивы

11.5.1. Передача одномерных массивов как параметров функции

При использовании массива как параметра функции, в функцию передается указатель на его первый элемент, т. е. массив всегда передается по адресу. При этом теряется информация о количестве элементов в массиве, поэтому размерность массива следует передавать как отдельный параметр. Так как в функцию передается указатель на начало массива (передача по адресу), то массив может быть изменен за счет операторов тела функции.

Пример1:

Удалить из массива все четные элементы

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

int form(int a[100])
{
int n;
        cout<<"\nEnter n";
        cin>>n;

        for(int i=0;i<n;i++)
                a[i]=rand()%100;
        return n;
}
```

```
void print(int a[100],int n)
{
	for(int i=0;i<n;i++)
		cout<<a[i]<<" ";
	cout<<"\n";
}

void Dell(int a[100],int&n)
{
	int j=0,i,b[100];
	for(i=0;i<n;i++)
		if(a[i]%2!=0)
		{
			b[j]=a[i];j++;
		}
		n=j;
		for(i=0;i<n;i++)a[i]=b[i];

}

void main()
{
	int a[100];
	int n;
	n=form(a);
	print(a,n);
	Dell(a,n);
	print(a,n);
}
```

Пример 2

Удалить из массива все элементы, совпадающие с первым элементом, используя динамическое выделение памяти.

```
#include <iostream.h>
#include <stdlib.h>
int* form(int&n)
{

	cout<<"\nEnter n";
	cin>>n;
	int*a=new int[n];//указатель на динамическую область памяти
	for(int i=0;i<n;i++)
		a[i]=rand()%100;
	return a;
}
void print(int*a,int n)
{
	for(int i=0;i<n;i++)
		cout<<a[i]<<" ";
	cout<<"\n";
}

int*Dell(int *a,int&n)
{
	int k,j,i;
```

```
        for(k=0,i=0;i<n;i++)
                if(a[i]!=a[0])k++;
                int*b;
                b=new int [k];
                for(j=0,i=0;i<n;i++)
                if(a[i]!=a[0])
                {
                        b[j]=a[i];j++;
                }
                n=k;
                return b;
}

void main()
{
        int *a;
        int n;
        a=form(n);
        print(a,n);
        a=Dell(a,n);
        print(a,n);
}
```

11.5.2. Передача строк в качестве параметров функций

Строки при передаче в функции могут передаваться как одномерные массивы типа char или как указатели типа char*. В отличие от обычных массивов в функции не указывается длина строки, т. к. в конце строки есть признак конца строки /0.

Пример: Функция поиска заданного символа в строке

```
int  find(char *s,char c)
{
for (int I=0;I<strlen(s);I++)
if(s[I]==c) return I;
return –1
}
```

С помощью этой функции подсчитаем количество гласных букв в строке.

```
void main()
{
char s[255];
gets(s)
char gl=”aouiey”;
for(int I=0,k=0;I<strlen(gl);I++)
if(find(s,gl[I])>0)k++;
printf(“%d”,k);
}
```

11.5.3. Передача многомерных массивов в функцию

При передаче многомерных массивов в функцию все размерности должны передаваться в качестве параметров. По определению многомерные массивы в Си и СИ++ не существуют. Если мы описываем массив с несколькими индексами, например, массив int mas[3][4], то это означает, что мы описали одномерный массив mas, элементами которого являются указатели на одномерные массивы int[4].

Пример: Транспонирование квадратной матрицы

Если определить заголовок функции:

void transp(int a[][],int n){…..} – то получится, что мы хотим передать в функцию

массив с неизвестными размерами. По определению массив должен быть одномерным, и его элементы должны иметь одинаковую длину. При передаче массива ничего не сказано и о размере элементов, поэтому компилятор выдаст ошибку.

Самый простой вариант решения этой проблемы определить функцию следующим образом:

void transp(int a[][4],int n), тогда размер каждой строки будет 4, а размер массива указателей будет вычисляться.

```
#include<iostream.h>
const int N=4;//глобальная переменная
void transp(int a[][N],int n)
{
int r;
for(int I=0;I<n;I++)
for(int j=0;j<n;j++)
if(I<j)
{
r[a[I][j];a[I][j]=a[j][I];a[j][I]=r;
}
}
void main()
{
int mas[N][N];
for(int I=0;I<N;I++)
for(int j=0;j<Nlj++)
cin>>mas[I][j];
for(I=0;I<N;I++)
{
for(j=0;j<N;j++)
cout<<mas[I][j]
cout<<"\n";
}
transp(N,mas);
for(I=0;I<N;I++)
{
for(j=0;j<N;j++)
cout<<mas[I][j]
cout<<"\n";
}
}
```

12. Функции с начальными (умалчиваемыми) значениями параметров

В определении функции может содержаться начальное (умалчиваемое) значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Се параметры, описанные справа от такого параметра также должны быть умалчиваемыми.

Пример:

```
void print(char*name="Номер дома: ",int value=1)
{cout<<"\n"<<name<<value;}
```

Вызовы:

1. print();

Вывод: Номер дома: 1

2. print("Номер квартиры",15);

Вывод: Номер квартиры: 15

3. print(,15); - ошибка, т. к. параметры можно пускать только с конца

Поэтому функцию лучше переписать так:

void print(int value=1, char*name="Номер дома: ")

{cout<<"\n"<<name<<value;}

Вызовы:

1. print();

Вывод: Номер дома: 1

2. print(15);

Вывод: Номер дома: 15

3. print(6, "Размерность пространства");

Вывод: Размерность пространства: 6

13. Подставляемые (inline) функции

Некоторые функции в СИ++ можно определить с использованием служебного слова inline. Такая функция называется подставляемой или встраиваемой.

Например:

inline float Line(float x1,float y1,float x2=0, float y2=0)

{return sqrt(pow(x1-x2)+pow(y1-y2,2));}//функция возвращает расстояние от точки с координатами(x1,y1)(по умолчанию центр координат) до точки с координатами (x2,y2).

Обрабатывая каждый вызов подставляемой функции, компилятор пытается подставить в текст программы код операторов ее тела. Спецификатор inline определяет для функции так называемое внутреннее связывание, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода. При этом может увеличиваться размер программы, но исключаются затраты на передачу управления к вызываемой функции и возврата из нее. Подставляемые функции используют, если тело функции состоит из нескольких операторов.

Подставляемыми не могут быть:

-рекурсивные функции;

-функции, у которых вызов размещается до ее определения;

-функции, которые вызываются более одного раза в выражении;

-функции, содержащие циклы, переключатели и операторы переходов;

-функции, которые имеют слишком большой размер, чтобы сделать подстановку.

14. Функции с переменным числом параметров

В СИ++ допустимы функции, у которых при компиляции не фиксируется число параметров, и , кроме того может быть неизвестен тип этих параметров. Количество и тип параметров становится известным только в момент вызова, когда явно задан список фактических параметров. Каждая функция с переменным числом параметров должна иметь хотя бы один обязательный параметр. Определение функции с переменным числом параметров:

тип имя (явные параметры,. . .)

{тело функции }

После списка обязательных параметров ставится запятая, а затем многоточие, которое показывает, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции производить не нужно. Сложность заключается в определении начала и конца списка параметров, поэтому каждая функция с переменным числом параметров должна иметь механизм определения количества и типов параметров. Существует два подхода:

1) известно количество параметров, которое передается как обязательный параметр;
2) известен признак конца списка параметров;

Пример1

Найти среднее арифметическое последовательности чисел

//известен признак конца списка параметров

#include<iostream.h>

```
float sum(int k, . . .)
{
int *p=&k;//настроили указатель на параметр k
int s=0;
for(;k!=0;k--)
s+=*(++p);
return s/k;
}
void main()
{
cout<<"\n4+6="<<sum(2,4,6);//находит среднее арифметическое 4+6
cout<<"\n1+2++3+4="<<sum(4,1,2,3,4);//находит среднее арифметическое 1+2+3+4
}
```

Для доступа к списку параметров используется указатель *p типа int. Он устанавливается на начало списка параметров в памяти, а затем перемещается по адресам фактических параметров (++p).

Пример 2.

```
//известен признак конца списка параметров
#include<iostream.h>
int sum(int k, . . .)
{
int *p=&k;//настроили указатель на параметр k
int s=*p;//значение первого параметра присвоили s
for(int i=1;p!=0;i++)//пока нет конца списка
s+=*(++p);
return s/(i-1);
}
void main()
{
cout<<"\n4+6="<<sum(4,6,0);//находит среднее арифметическое 4+6
cout<<"\n1+2++3+4="<<sum(1,2,3,4,0);//находит среднее арифметическое 1+2+3+4
}
```

15. Перегрузка функций

Цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для обеспечения перегрузки необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем. Компилятор определяет какую функцию выбрать по типу фактических параметров.

Пример.

```
#include<iostream.h>
#include <string.h>
int max(int a,int b)
{
if(a>b)return a;
 else return b;
}
float max(float a,float b)
{
if(a>b)return a;
```

```
 else return b;
}
char*max(char*a,char*b)
{
        if(strcmp(a,b)>0) return a;
        else return b;
}
void main()
{
int a1,b1;
float a2, b2;
        char s1[20];
        char s2[20];
cout<<"\nfor int:\n";
cout<<"a=?";cin>>a1;
cout<<"b=?";cin>>b1;
cout<<"\nMAX="<<max(a1,b1)<<"\n";
cout<<"\nfor float:\n";
cout<<"a=?";cin>>a2;
cout<<"b=?";cin>>b2;
cout<<"\nMAX="<<max(a2,b2)<<"\n";
cout<<"\nfor char*:\n";
cout<<"a=?";cin>>s1;
cout<<"b=?";cin>>s2;
cout<<"\nMAX="<<max(s1,s2)<<"\n";
}
```

Правила описания перегруженных функций:

1) Перегруженные функции должны находиться в одной области видимости.
2) Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В разных вариантах перегруженных функций может быть разное количество умалчиваемых параметров.
3) Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или наличием ссылки.

Например, функции int&f1(int&,const int&){. . . } и int f1(int,int){. . . } – не являются перегруженными, т. к. компилятор не сможет узнать какая из функций вызывается: нет синтаксических отличий между вызовом функции, которая передает параметр по значению и функции, которая передает параметр по ссылке.

16. Шаблоны функций

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т. е. тип данных передается как параметр шаблона. Формат шаблона:

template <class имя_типа [,class имя_типа]>
заголовок_функции
{тело функции}

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: template<список параметров шаблона> и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

Пример 1.

```
//шаблон функции, которая находит абсолютное значение числа любого типа
template<class type>//type – имя параметризируемого типа
```

```
type abs(type x)
{
if(x<0)return -x;
else return x;
}
```

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как abs(-1.5), то компилятор сформирует определение функции double abs(double x){. . . }.

Пример 2.

```
//шаблон функции, которая меняет местами две переменных
template <class T>//T – имя параметризируемого типа
void change(T*x,T*y)
{T z=*x;*x=*y;*y=z;}
```

Вызов этой функции может быть :

```
long k=10,l=5;
change(&k,&l);
```

Компилятор сформирует определение:

```
void change(long*x,long*y){ long z=*x;*x=*y;*y=z;}
```

Пример 3.

```
#include<iostream.h>
template<class Data>
Data&rmax(int n,Data a[])
{
int im=0;
for(int i=0;i<n;i++)
if(a[im]<a[im])im=i;
return d[im];//возвращает ссылку на максимальный элемент в массиве
}
void main()
{int n=5;
int x[]={10,20,30,15};
cout<<”\nrmax(n,x)=”<<rmax(n,x)<<”\n”;
rmax(n,x)=0;
for(int i=0;i<n;i++)
cout<<x[i]<<” “;
cout<<”\n”;
float y[]={10.4,20.2,30.6,15.5};
cout<<”\nrmax(n,y)=”<<rmax(n,y)<<”\n”;
rmax(4,y)=0;
for(in i=0;i<n;i++)
cout<<y[i]<<” “;
cout<<”\n”;
}
```

Результаты:

```
rmax(n,x)=30
10 20 0 15
rmax(n,y)=30.6
10.4 20.2 0 15.5
```

Основные свойства параметров шаблона функций

1. Имена параметров должны быть уникальными во всем определении шаблона.

2. Список параметров шаблона не может быть пустым.
3. В списке параметров шаблона может быть несколько параметров, каждый из них начинается со слова class.

17. Указатель на функцию

Каждая функция характеризуется типом возвращаемого значения, именем и списком типов ее параметров. Если имя функции использовать без последующих скобок и параметров, то он будет выступать в качестве указателя на эту функцию, и его значением будет выступать адрес размещения функции в памяти. Это значение можно будет присвоить другому указателю. Тогда этот новый указатель можно будет использовать для вызова функции. Указатель на функцию определяется следующим образом:

тип_функции(*имя_указателя)(спецификация параметров)

Примеры:

```
1. int f1(char c){. . . . }//определение функции
int(*ptrf1)(char);//определение указателя на функцию f1
2. char*f2(int k,char c){. . . .}//определение функции
char*ptrf2(int,char);//определение указателя
```

В определении указателя количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.

Вызов функции с помощью указателя имеет вид:

(*имя_указателя)(список фактических параметров);

Пример.

```
#include <iostream.h>
void f1()
{cout<<"\nfunction f1";}
void f2()
{cout<<"\nfunction f2";}
void main()
{
void(*ptr)();//указатель на функцию
ptr=f2;//указателю присваивается адрес функции f2
(*ptr)();//вызов функции f2
ptr=f1;//указателю присваивается адрес функции f1
(*ptr)();//вызов функции f1с помощью указателя
}
```

При определении указатель на функцию может быть сразу проинициализирован.

```
void (*ptr)()=f1;
```

Указатели н функции могут быть объединены в массивы. Например, float(*ptrMas [4])(char) – описание массива, который содержит 4 указателя на функции. Каждая функция имеет параметр типа char и возвращает значение типа float. Обратиться к такой функции можно следующим образом:

```
float a=(*ptrMas[1])('f');//обращение ко второй функции
```

Пример.

```
#include <iostream.h>
#include <stdlib.h>
void f1()
{cout<<"\nThe end of work";exit(0);}
void f2()
{cout<<"\nThe work #1";}
void f3()
{cout<<"\nThe  work #2";}
void main()
{
void(*fptr[])()={f1,f2,f3};
int n;
while(1)//бесконечный цикл
{
cout<<"\n Enter the number";
cin>>n;
fptr[n]();//вызов функции с номером n
}
}
```

Указатели на функции удобно использовать в тех случаях, когда функцию надо передать в другую функцию как параметр.

Пример.

```
#include <iostream.h>
#include <math.h>
typedef float(*fptr)(float);//тип – указатель на функцию
float root(fptr f, float a, float b, float e)//решение уравнения методом половинного деления
//уравнение передается с помощью указателя на функцию
{float x;
do
{
x=(a+b)/2;
if ((*f)(a)*f(x)<0)b=x; else a=x;
}
while((*f)(x)>e&&fabs(a-b)>e);
return x;
}
float testf(float x)
{return x*x-1;}
void main()
{
float res=root(testf,0,2,0.0001);
cout<<"\nX="<<res;
}
```

18. Ссылки на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

тип_функции(&имя_ссылки)(параметры) инициализирующее_выражение;

Пример.

```
int f(float a,int b){. . . }//определение функции
int(&fref)(float,int)=f;//определение ссылки
```

Использование имени функции без параметров и скобок будет восприниматься как адрес функции. Ссылка на функцию является синонимом имени функции. Изменить значение ссылки на функцию нельзя, поэтому более широко используются указатели на

функции, а не ссылки.

Пример.

```
#include <iostream.h>
void f(char c)
{cout<<"\n"<<c;}
void main()
{
void (*pf)(char);//указатель на функцию
void(&rf)(char);//ссылка на функцию
f('A');//вызов по имени
pf=f;//указатель ставится на функцию
(*pf)('B');//вызов с помощью указателя
rf('C');//вызов по ссылке
}
```

19. Типы данных, определяемые пользователем

19.1.Переименование типов

Типу можно задавать имя с помощью ключевого слова typedef:

typedef тип имя_типа [размерность];

Примеры:

```
typedef unsigned int UNIT;
typedef char Msg[100];
```

Такое имя можно затем использовать также как и стандартное имя типа:

```
UNIT a,b,c;//переменные типа unsigned int
Msg str[10];// массив из 10 строк по 100 символов
```

19.2.Перечисления

Если надо определить несколько именованных констант таким образом, чтобы все они имели разные значения, можно воспользоваться перечисляемым типом:

enum [имя_типа] {список констант};

Константы должны быть целочисленными и могут инициализироваться обычным образом. Если инициализатор отсутствует, то первая константа обнуляется, а остальным присваиваются значение на единицу большее, чем предыдущее.

Пример:

```
Enum Err{ErrRead, ErrWrite, ErrConvert);
Err error;
. . . .
switch(error)
{
case ErrRead: …..
case ErrWrite: …..
case ErrConvert: …..
}
```

19.3.Структуры

Структура – это объединенное в единое целое множество поименованных элементов данных. Элементы структуры (поля) могут быть различного типа, они все должны иметь различные имена.

Форматы определения структурного типа следующие:

1. struct имя_типа //способ 1

```
{
тип 1 элемент1;
тип2 элемент2;
. . .
};
```

Пример:

```
struct Date//определение структуры
{
int day;
int month;
int year;
};
Date birthday;//переменная типа Date
```

2) struct //способ 2

```
{
тип 1 элемент1;
тип2 элемент2;
. . .
} список идентификаторов;
```

Пример:

```
struct
{
int min;
int sec;
int msec;
}time_beg,time_end;
```

В первом случае описание структур определяет новый тип, имя которого можно использовать наряду со стандартными типами.

Во втором случае описание структуры служит определением переменных.

3) Структурный тип можно также задать с помощью ключевого слова typedef:

```
Typedef struct //способ 3
{
floar re;
float im;
}Complex;
Complex a[100];//массив из 100 комплексных чисел.
```

19.3.1. Инициализация структур.

Для инициализации структур значения ее полей перечисляют в фигурных скобках.

Примеры:

```
1. struct Student
{
 char name[20];
int kurs;
float rating;
};
Student s={”Иванов”,1,3.5};
2. struct
{
char name[20];
char title[30];
float rate;
}employee={“Петров", “директор”,10000};
```

Работа со структурами

19.3.2. Присваивание структур

Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование.

```
Student ss=s;
```

19.3.3. Доступ к элементам структур

Доступ к элементам структур обеспечивается с помощью уточненных имен:

Имя_структуры.имя_элемента

employee.name – указатель на строку «Петров»;

employee.rate – переменная целого типа со значением 10000

Пример:

```
#include <iostream.h>
void main()
{
struct Student
{
char name[30];
char group[10];
float rating;
};
Student mas[35];
```

```
//ввод значений массива
for(int i=0;i<35;i++)
{
cout<<”\nEnter name:”;cin>>mas[i].name;
cout<<”\nEnter group:”;cin>>mas[i].group;
cout<<”\nEnter rating:”;cin>>mas[i].rating;
}
cout<<”Raitng <3:”;
for( i=0;i<35;i++)
if(mas[i].name<3)
cout<<”\n”<<mas[i].name;
}
```

19.4.Указатели на структуры

Указатели на структуры определяются также как и указатели на другие типы.

```
Student*ps;
```

Можно ввести указатель для типа struct, не имеющего имени (способ 2):

```
Struct
 {
char *name;
int age;
} *person;//указатель на структуру
```

При определении указатель на структуру может быть сразу же проинициализирован.

```
Student *ps=&mas[0];
```

Указатель на структуру обеспечивает доступ к ее элементам 2 способами:

1.(*указатель).имя_элемента

2. указатель->имя_элемента

```
cin>>(*ps).name;
cin>>ps->title;
```

20. Битовые поля

Битовые поля – это особый вид полей структуры. При описании битового поля указывается его длина в битах (целая положительная константа).

Пример:

```
struct {
int a:10;
int b:14}xx,*pxx;
. . . .
xx.a=1;
pxx=&xx;
pxx->b=8;
```

Битовые поля могут быть любого целого типа. Они используются для плотной упаковки данных. Например, с их помощью удобно реализовать флажки типа «да» / «нет».

Особенностью битовых полей является то, что нельзя получить их адрес. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

21. Объединения

Объединение (union)- это частный случай структуры. Все поля объединения располагаются по одному и тому же адресу. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в такой переменной может храниться только одно значение. Объединения применяют для экономии памяти, если известно, что более одного поля не потребуется. Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа.

Пример

```
union{
```

```
char s[10];
int x;
}u1;
```

Рис.3. Расположение объединения в памяти

И s, и x располагаются на одном участке памяти. Размер такого объединения будет равен 10 байтам.

```
Пример1:
//использование объединений
enum paytype{CARD,CHECK};//тип оплаты
struct{
paytype ptype;//поле, которое определяет с каким полем объединения будет
// выполняться работа
union{
        char card[25];
        long check;
        };
}info;
switch (info.ptype)
{
case CARD:cout<<"\nОплата по карте:"<<info.card;break;
case CHECK:cout<<"\nОплата чеком:"<<info.check;break;
}
```

22. Динамические структуры данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

-линейные списки;
-стеки;
-очереди;
-бинарные деревья;

Они отличаются способом связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.

Наиболее простой динамической структурой является линейный однонаправленный список, элементами которого служат объекты структурного типа (рис.4).

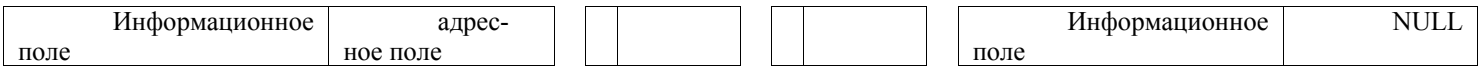


Рис.4. Линейный однонаправленный список

22.1. Линейный однонаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

```
struct  имя_типа
{
информационное поле;
```

адресное поле;
};

Информационное поле – это поле любого, ранее объявленного или стандартного, типа;
адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Информационных полей может быть несколько.

Примеры.

```
1. struct Node
{
int key;//информационное поле
Node*next;//адресное поле
};
2. struct point
{
char*name;//информационное поле
int age;//информационное поле
point*next;//адресное поле
};
```

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом (пример 1), либо строкой (пример 2).

Над списками можно выполнять следующие операции:

-начальное формирование списка (создание первого элемента);
-добавление элемента в конец списка;
-добавление элемента в начало списка;
-удаление элемента с заданным номером;
-чтение элемента с заданным ключом;
-вставка элемента в заданное место списка (до или после элемента с заданным ключом);
-упорядочивание списка по ключу

и др.

Пример1. Создание и печать однонаправленного списка

```
#include <iostream.h>
#include<string.h>
//описание структуры
struct point
{char *name;//информационное поле
int age;//информационное поле
point*next;//адресное поле
};

point* make_point()
//создание одного элемента
{
        point*p=new(point);//выделить память
        char s[20];
        cout<<"\nEnter the name:";
        cin>>s;
p->name=new char[strlen(s)+1];//выделить память под динамическую строку символов
        strcpy(p->name,s);//записать информацию в строку символов
        cout<<"\nEnter the age";
        cin>>p->age;
        p->next=0;//сформировать адресное поле
return p;
```

```
}
void print_point(point*p)
//печать информационных полей одного элемента списка
{
	cout<<"\nNAME:"<<p->name;
	cout<<"\nAGE:"<<p->age;
	cout<<"\n--------------------------------\n";
}

point* make_list(int n)
//формирование списка из n элементов
{
	point* beg=make_point();//сформировать первый элемент
	point*r;
	for(int i=1;i<n;i++)
	{
		r=make_point();//сформировать следующий элемент
		//добавление в начало списка
		r->next=beg;//сформировать адресное поле
		beg=r;//изменить адрес первого элемента списка
	}
	return beg;//вернуть адрес начала списка
}
int print_list(point*beg)
//печать списка, на который указывает указатель beg
{
	point*p=beg;//p присвоить адрес первого элемента списка
int k=0;//счетчик количества напечатанных элементов
	while(p)//пока нет конца списка
	{
		print_point(p);//печать элемента, на который указывает элемент p
		p=p->next;//переход к следующему элементу
	k++;
	}
return k;//количество элементов в списке
}

void main()
{
	int n;
	cout<<"\nEnter the size of list";
		cin>>n;
	point*beg=make_list(n);//формирование списка
if(!print_list(beg)) cout<<"\nThe list is empty";}//печать списка
```

Пример 2. Удаление из однонаправленного списка элемента с номером k (рис 5.).

Рис. 5. Удаление элемента с номером k из однонаправленного списка

```
point*del_point(point*beg,int k)
//удаление элемента с номером к
{
        point*p=beg,//поставить вспомогательную переменную на начало списка
*r;//вспомогательная переменная для удаления
        int i=0;//счетчик элементов в списке
        if(k==0)
        {//удалить первый элемент
                beg=p->next;
                delete[]p->name;//удалить динамическое поле name
                delete[]p;//удалить элемент из списка
                return beg;//вернуть адрес первого элемента списка
        }
        while(p)//пока нет конца списка
        {
                if(i==k-1)//дошли до элемента с номером k-1, чтобы поменять его
поле next
                {//удалить элемент
                        r=p->next;//поставить r на удаляемый элемент
                        if(r)//если p не последний элемент
                        {
                        p->next=r->next;//исключить r из списка
                        delete[]r->name;//удалить динамическое поле name
                        delete[]r;//удалить элемент из списка
                        }
                else p->next=0;//если p -последний элемент, то в поле next при-
своить NULL
                }
                p=p->next;//переход к следующему элементу списка
                i++;//увеличить счетчик элементов
        }
return beg;//вернуть адрес первого элемента}
```

22.2. Работа с двунаправленным списком

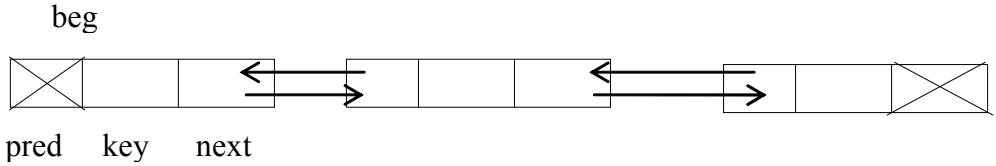


Рис. 6 Двунаправленный список

Пример 3.

1. Создать двунаправленный список, выполнить удаление элемента с заданным номером, добавление элемента с заданным номером, печать полученных списков.

```
#include <iostream.h>
struct point//описание структуры
```

```
{
	int key;//ключевое поле
	point* pred,*next;//адресные поля
};
point*make_list()
{
	int n;
	cout<<"n-?";cin>>n;
	point *p,*r,*beg;
	p=new (point);//создать первый элемент
	beg=p;//запомнить адрес в переменную beg, в которой хранится начало
списка
	cout<<"key-?";cin>>p->key;//заполнить ключевое поле
	p->pred=0;p->next=0;//запомнить адресные поля
	for(int i=1;i<n;i++)//добавить элементы в конец списка
	{
		r=new(point);//новый элемент
		cout<<"key-?";cin>>r->key;//адресное поле
		p->next=r;//связать начало списка с r
		r->pred=p;//связать r с началом списка
		r->next=0;//обнулить последнее адресное поле
		p=r;//передвинуть p на последний элемент списка
	}
	return beg;//вернуть первый элемент списка
}
void print_list(point *beg)
{
	if (beg==0)//если список пустой
	{
	cout<<"The list is empty\n";
	 return;
 }
	point*p=beg;
	while(p)//пока не конец списка
	{
		cout<<p->key<<"\t";
		p=p->next;//перейти на следующий
	}
	cout<<"\n";
}
point* del_point(point*beg, int k)
{
	point *p=beg;
	if(k==0)//удалить первый элемент
	{
		beg=beg->next;//переставить начало списка на следующий элемент
		if(beg==0)return 0;//если в списке только один элемент
		beg->pred=0;//обнулить адрес предыдущего элемента
		delete p;//удалить первый
		return beg;//вернуть начало списка
	}
//если удаляется элемент из середины списка
	for(int i=0;i<k-1&&p!=0;i++,p=p->next);//пройти по списку либо до элемента
с предыдущим номером, либо до конца списка
```

```
		if(p==0||p->next==0)return beg;//если в списке нет элемента с номером k
		point*r=p->next;//встать на удаляемый элемент
		p->next=r->next;//изменить ссылку
		delete r;//удалить r
		r=p->next;//встать на следующий
		if(r!=0)r->pred=p;//если r существует, то связать элементы
		return beg;//вернуть начало списка
	}
	point* add_point(point *beg,int k)
	{
		point *p;
		p=new(point);//создать новый элемент и заполнить ключевое поле
		cout<<"key-?";cin>>p->key;
		if(k==0)//если добавляется первый элемент
		{
			p->next=beg;//добавить перед beg
			p->pred=0;//обнулить адрес предыдущего
			beg->pred=p;//связать список с добавленным элементом
			beg=p;//запомнить первый элемент в beg
			return beg;//вернуть начало списка
		}
			point*r=beg;//встать на начало списка
		for(int i=0;i<k-1&&r->next!=0;i++,r=r->next);//пройти по списку либо до конца списка, либо до элемента с номером k-1
		p->next=r->next;//связать p с концом списка
		if(r->next!=0)r->next->pred=p;//если элемент не последний, то связать конец списка с p
		p->pred=r;//связать p и r
		r->next=p;
		return beg;//вернуть начало списка
	}


	void main()
	{
		point*beg;
	int i,k;
	do
	{
		cout<<"1.Make list\n";
		cout<<"2.Print list\n";
		cout<<"3.Add point\n";
		cout<<"4.Del point\n";
		cout<<"5.Exit\n";
	cin>>i;
		switch(i)
		{
		case 1:
			{beg=make_list();break;}
		case 2:
			{print_list(beg);break;}
		case 3:
			{
				cout<<"\nk-?";cin>>k;
```

```
                        beg=add_point(beg,k);
                        break;
                }
        case 4:
                {
                        cout<<"\nk-?";cin>>k;
                        beg=del_point(beg,k);
                        break;
                }
        }
}
while(i!=5);
}
```

23. Ввод-вывод в С

Файл – это именованная область внешней памяти. Файл имеет следующие характерные особенности:

1. имеет имя на диске, что дает возможность программам работать с несколькими файлами;
2. длина файла ограничивается только емкостью диска.

Особенностью С является отсутствие в этом языке структурированных файлов. Все файлы рассматриваются как не структурированная последовательность байтов. При таком подходе понятие файла распространяется и на различные устройства. Одни и те же функции используются как для обмена данными с файлами, так и для обмена с устройствами.

Библиотека С поддерживает три уровня ввода-вывода:

- потоковый ввод-вывод;
- ввод-вывод нижнего уровня;
- ввод-вывод для консоли портов (зависит от конкретной ОС).

Рассмотрим потоковый ввод-вывод.

23.1. Потоковый ввод-вывод

На уровне потокового ввода-вывода обмен данными производится побайтно, т. е. за одно обращение к устройству (файлу) производится считывание или запись фиксированной порции данных (512 или 1024 байта). При вводе с диска или при считывании из файла данные помещаются в буфер ОС, а затем побайтно или порциями передаются программе пользователя. При выводе в файл данные также накапливаются в буфере, а при заполнении буфера записываются в виде единого блока на диск. Буферы ОС реализуются в виде участков основной памяти. Т .о. поток – это файл вместе с предоставленными средствами буферизации. Функции библиотеки С, поддерживающие обмен данными на уровне потока позволяют обрабатывать данные различных размеров и форматов. При работе с потоком можно:

1. Открывать и закрывать потоки (при этом указатели на поток связываются с конкретными файлами);
2. Вводить и выводить строки, символы, форматированные данные, порции данных произвольной длины;
3. Управлять буферизацией потока и размером буфера;
4. Получать и устанавливать указатель текущей позиции в файле.

Прототипы функций ввода-вывода находятся в заголовочном файле <stdio.h>, который также содержит определения констант, типов и структур, необходимых для обмена с потоком.

23.2. Открытие и закрытие потока

Прежде, чем начать работать с потоком, его надо инициировать, т. е. открыть. При этом поток связывается со структурой предопределенного типа FILE, определение которой находится в файле <stdio.h>. В структуре находится указатель на буфер, указатель на текущую позицию и т. п. При открытии потока возвращается указатель на поток, т. е. на объект типа FILE. Указатель на поток должен быть объявлен следующим образом:

```
#include <stdio.h>
. . . . . .
FILE*f;//указатель на поток
```

Указатель на поток приобретает значение в результате выполнения функции открытия потока:

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

где const char*filename – строка, которая содержит имя файла, связанного с потоком,

const char*mode – строка режимов открытия файла.

Например:

```
f=fopen(“t.txt”,”r”);
```

где t.txt – имя файла, r – режим открытия файла.

Файл связанный с потоком можно открыть в одном из 6 режимов

Режим	Описание режима открытия файла
r	Файл открывается для чтения, если файл не существует , то выдается ошибка при исполнении программы.
w	Файл открывается для записи, если файл не существует, то он будет создан, если файл уже существует, то вся информация из него стирается.
a	Файл открывается для добавления, если фай не существует, то он будет создан, если существует, то информация из него не стирается, можно выполнять запись в конец файла
r+	Файл открывается для чтения и записи, изменить размер файла нельзя, если файл не существует , то выдается ошибка при исполнении программы.
w+	Файл открывается для чтения и записи, если файл не существует, то он будет создан, если файл уже существует, то вся информация из него стирается.
a+	Файл открывается для чтения и записи, если фай не существует, то он будет создан, если существует, то информация из него не стирается, можно выполнять запись в конец файла

Поток можно открывать в текстовом (t) или двоичном режиме(b). В текстовом режиме поток рассматривается как совокупность строк, в конце каждой строки находится управляющий символ ‘\n’. В двоичном режиме поток рассматривается как набор двоичной информации. Текстовый режим устанавливается по умолчанию.

В файле stdio.h определена константа EOF, которая сообщает об окончании файла (отрицательное целое число).

При открытии потока могут возникать следующие ошибки:

- файл, связанный с потоком не найден (при чтении из файла);
- диск заполнен (при записи);
- диск защищен от записи (при записи) и т. п.

В этих случаях указатель на поток приобретет значение NULL (0). Указатель на поток, отличный от аварийного не равен 0.

Для вывода об ошибке при открытии потока используется стандартная библиотечная функция из файла <stdio.h>

void perror (const char*s);

Эта функция выводит строку символов, не которую указывает указатель s, за этой строкой размещается двоеточие пробел и сообщение об ошибке. Текст сообщения выбирается на основании номера ошибки. Номер ошибки заносится в переменную int errno (определена в заголовочном файле errno.h).

После того как файл открыт, в него можно записывать информацию или считывать информацию, в зависимости от режима.

Открытые файлы после окончания работы рекомендуется закрыть явно. Для этого используется функция:

int fclose(FILE*f);

Изменить режим работы с файлом можно только после закрытия файла.

Пример:

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
FILE *f;
char filename[20];
cout<<”\nEnter the name of file:”; cin>>filename;
if(f=fopen(filename,”rb”)==0)//открываем для чтения в бинарном режиме и проверяем
```

```
// возникает ли ошибка при открытии файла
{
perror(strcat“error in file :”,filename);//strcat складывает две строки
exit(0);//выход из программы
}
. . . . .
fclose(f);
}
```

Для текстового файла:

```
if(f=fopen(filename,”rt”)==0)//открываем для чтения и проверяем возникает ли
ошибка при //открытии файла
if(f=fopen(filename,”r”)==0)//открываем для чтения и проверяем возникает ли ошиб-
ка при //открытии файла
```

23.3. Стандартные файлы и функции для работы с ними

Когда программа начинает выполняться, автоматически открываются несколько потоков, из которых основными являются:

- стандартный поток ввода (stdin);
- стандартный поток вывода (stdout);
- стандартный поток вывода об ошибках (stderr).

По умолчанию stdin ставится в соответствие клавиатура, а потокам stdout и stderr - монитор. Для ввода-вывода с помощью стандартных потоков используются функции:

- getchar()/putchar() – ввод-вывод отдельного символа;
- gets()/puts() – ввод-вывод строки;
- scanf()/printf() – форматированный ввод/вывод.

Функции рассматривались, когда мы рассматривали строковые и символьные данные. Теперь мы можем связать их со стандартными потоками: ввод осуществляется из стандартного потока stdin вывод осуществляется в стандартный поток stdout. Аналогично работе со стандартными потоками выполняется ввод-вывод в потоки, связанные с файлами.

23.4. Символьный ввод-вывод

Для символьного ввода-вывода используются функции:

- int fgetc(FILE*fp), где fp – указатель на поток, из которого выполняется считывание. Функция возвращает очередной символ в форме int из потока fp. Если символ не может быть прочитан, то возвращается значение EOF.

- int fputc(int c, FILE*fp), где fp – указатель на поток, в который выполняется запись, c – переменная типа int, в которой содержится записываемый в поток символ. Функция возвращает записанный в поток fp символ в форме int . Если символ не может быть записан, то возвращается значение EOF.

Пример:

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
FILE *f;
char c;
char *filename=”f.txt”;
if((f=fopen(filename,”r”)==0)
{
perror(filename);exit(0);
}
while(c=fgetc(f)!=EOF)
```

```
putchar(c);//вывод с на стандартное устройство вывода
fclose(f);
}
```

23.5. Строковый ввод-вывод

Для построчного ввода-вывода используются следующие функции:

1) char* fgets(char* s,int n,FILE* f), где

char*s – адрес, по которому размещаются считанные байты,

int n – количество считанных байтов,

FILE* f – указатель на файл, из которого производится считывание.

Прием байтов заканчивается после передачи n-1 байтов или при получении управляющего символа ‘\n’. Управляющий символ тоже передается в принимающую строку. Строка в любом случае заканчивается ‘\0’. При успешном завершении считывания функция возвращает указатель на прочитанную строку, при неуспешном – 0.

2) int puts(char* s, FILE* f), где

char*s – адрес, из которого берутся записываемые в файл байты,

FILE* f – указатель на файл, в который производится запись.

Символ конца строки (‘\0’) в файл не записывается. Функция возвращает EOF, если при записи в файл произошла ошибка, при успешной записи возвращает неотрицательное число.

Пример:

```
//копирование файла in в файл out
int MAXLINE=255;//максимальная длина строки
FILE *in,//исходный файл
*out;//принимающий файл
char* buf[MAXLINE];//строка, с помощью которой выполняется копирование
in=fopen(“f1.txt”,”r”);//открыть исходный файл для чтения
out=fopen(“f2.txt”,”w”);//открыть принимающий файл для записи
while(fgets(buf,MAXLINE,in)!=0)//прочитать байты из файла in в строку buf
fputs(buf,out);//записать байты из строки buf в файл out
fclose(in);fclose(out);//закрыть оба файла
```

23.6. Блоковый ввод-вывод

Для блокового ввода-вывода используются функции:

1) int fread(void*ptr,int size, int n, FILE*f), где

void*ptr – указатель на область памяти, в которой размещаются считанные из файла данные,

int size – размер одного считываемого элемента,

int n – количество считываемых элементов,

FILE*f – указатель на файл, из которого производится считывание.

В случае успешного считывания функция возвращает количество считанных элементов, иначе – EOF.

2) int fwrite(void*ptr,int size, int n, FILE*f), где

void*ptr – указатель на область памяти, в которой размещаются считанные из файла данные,

int size – размер одного записываемого элемента,

int n – количество записываемых элементов,

FILE*f – указатель на файл, в который производится запись.

В случае успешной записи функция возвращает количество записанных элементов, иначе – EOF.

Пример:

```
struct Employee
{
char name[30];
char title[30];
```

```
float rate;
};
void main()
{
Employee e;
FILE *f;
if((f=fopen(“f.dat”,”wb”))==NULL)
{
cout<<”\nCannot open file for writing”;
exit(1);
}
int n;
//запись в файл
printf(“\nN-?”);
scanf(“%d”,&n);
for(int i=0;i<n;i++)
{
//формируем структуру e
printf(“\nname:”);scanf(“%s”,&e.name);
printf(“\ntitle:”);scanf(“%s”,&e.title);
printf(“\nrate:”);scanf(“%s”,&e.rate);
//записываем e в файл
fwrite(&e,sizeof(Employee),1,f);
}
fclose(f);
//чтение из файла
if((f=fopen(“f.dat”,”rb”))==NULL)
{
cout<<”\nCannot open file for reading”;
exit(2);
}
while(fread(&e,sizeof(Employee)1,f)
{
printf(“%s % s%f”, e.name, e.title, e.rate)
}
fclose(f);
}
```

23.7. Форматированный ввод-вывод

В некоторых случаях информацию удобно записывать в файл без преобразования, т. е. в символьном виде пригодном для непосредственного отображения на экран. Для этого можно использовать функции форматированного ввода-вывода:

1) int fprintf(FILE *f, const char*fmt,. . .) , где
FILE*f – указатель на файл, в который производится запись,
const char*fmt – форматная строка,
. . . – список переменных, которые записываются в файл.
Функция возвращает число записанных символов.

2) 1) int fscanf(FILE *f, const char*fmt, par1,par2, . . .) , где
FILE*f – указатель на файл, из которого производится чтение,
const char*fmt – форматная строка,
par1,par2,. . . – список переменных, в которые заносится информация из файла.
Функция возвращает число переменных, которым присвоено значение.

Пример:

```
void main()
{
```

```
FILE *f;
int n;
if((f=fopen(“int.dat”,”w”))==0)
{
perror(“int.dat”);
exit(0);
}
for(n=1;n<11;n++)
fprinf(f,”\n%d   %d”,n,n*n);
fclose(f);
if((f=fopen(“int.dat”,”r”))==0)
{
perror(“int.dat”);
exit(1);
}
int nn;
while(fscanf(f, ”%d%d”,&n,&nn))
printf(“\n%d %d”,n,nn);
fclose(f);
}
```

23.8. Прямой доступ к файлам

Рассмотренные ранее средства обмена с файлами позволяют записывать и считывать данные только последовательно. Операции чтения/записи всегда производятся, начиная с текущей позиции в потоке. Начальная позиция устанавливается при открытии потока и может соответствовать начальному или конечному байту потока в зависимости от режима открытия файла. При открытии потока в режимах “r” и “w” указатель текущей позиции устанавливается на начальный байт потока, при открытии в режиме “a” - за последним байтом в конец файла. При выполнении каждой операции указатель перемещается на новую текущую позицию в соответствии с числом записанных/прочитанных байтов.

Средства прямого доступа дают возможность перемещать указатель текущей позиции в потоке на нужный байт. Для этого используется функция

int fseek(FILE *f, long off, int org), где

FILE *f - – указатель на файл,

long off – позиция смещения

int org – начало отсчета.

Смещение задается выражение или переменной и может быть отрицательным, т. е. возможно перемещение как в прямом, так и в обратном направлениях. Начало отсчета задается одной из определенных в файле <stdio.h> констант:

SEEK_SET ==0 – начало файла;

SEEK_CUR==1 – текущая позиция;

SEEK_END ==2 – конец файла.

Функция возвращает 0, если перемещение в потоке выполнено успешно, иначе возвращает ненулевое значение.

Примеры:

```
fseek(f,0L,SEEK_SET); //перемещение к началу потока из текущей позиции
fseek(f,0L,SEEK_END); //перемещение к концу потока из текущей позиции
fseek(f,-(long)sizeof(a),SEEK_SET); //перемещение назад на длину переменной a.
```

Кроме этой функции, для прямого доступа к файлу используются:

```
long tell(FILE *f);//получает значение указателя текущей позиции в потоке;
void rewind(FILE *f);//установить значение указателя на начало потока.
```

23.9. Удаление и добавление элементов в файле

Пример 1:

```
void del(char *filename)
{
//удаление записи с номером x
FILE *f, *temp;
f=fopen(filename,”rb”);//открыть исходный файл для чтения
temp=fopen(“temp”,”wb”)//открыть вспомогательный файл для записи
student a;
          for(long i=0;.fread(&a,sizeof(student),1,f);i++)
                    if(i!=x)
                    {
                   fwrite(&a,sizeof(student)1,temp);
                    }
                    else
                    {
                              cout<<a<<" - is deleting...";
                    }
                    fclose(f); fclose(temp);
                    remove(filename);
    rename(“temp”, filename);
}
```

Пример 2:

```
void add(char *filename)
{
//добавление в файл
          student a;
          int n;
          f=fopen(filename,”ab”)открыть файл для добавления
          cout<<"\nHow many records would you add to file?";
          cin>>n;
          for(int i=0;i<n;i++)
          {
                    прочитать объект
             fwrite(&a,sizeof(student),1,f);//записать в файл
          }
fclose(f);//закрыть файл

}
```

24. Вопросы к экзамену.

1. Алгоритм и его свойства. Способы записи алгоритма. Программа. Языки программирования. Примеры алгоритмов и программ.
2. Структура программы на языке С++. Примеры. Этапы создания исполняемой программы.
3. Состав языка С++. Константы и переменные С++.
4. Типы данных в С++.
5. Выражения. Знаки операций.
6. Основные операторы С++ (присваивание, составные, выбора, циклов, перехода). Синтаксис, семантика, примеры
7. Этапы решения задачи. Виды ошибок. Тестирование.
8. Массивы (определение, инициализация, способы перебора).
9. Сортировка массивов (простой обмен, простое включение, простой выбор).
10. Поиск в одномерных массивах (дихотомический и линейный).
11. Указатели. Операции с указателями. Примеры
12. Динамические переменные. Операции new и delete. Примеры.
13. Ссылки. Примеры.
14. Одномерные массивы и указатели. Примеры.
15. Многомерные массивы и указатели. Примеры.
16. Динамические массивы. Примеры.
17. Символьная информация и строки. Функции для работы со строками (библиотечный файл string.h).
18. Функции ввод-вывода (scanf(), printf(), puts(), gets(), putchar(),getchar()).
19. Функции в С++. Формальные и фактические параметры. Передача параметров по адресу и по значению. Локальные и глобальные переменные. Примеры.
20. Прототип функции. Библиотечные файлы. Директива препроцессора #include.
21. Передача одномерных массивов в функции. Примеры.
22. Передача многомерных массивов в функции. Примеры.
23. Передача строк в функции. Примеры.
24. Функции с умалчиваемыми параметрами. Примеры.
25. Подставляемые функции. Примеры.
26. Функции с переменным числом параметров. Примеры.
27. Перегрузка функции. Шаблоны функций. Примеры.
28. Указатели на функции. Примеры.
29. Ссылки на функции. Примеры.
30. Типы данных, определяемые пользователем (переименование типов, перечисление, структуры, объединения). Примеры.
31. Структуры. Определение, инициализация, присваивание структур, доступ к элементам структур, указатели на структуры, битовые поля структур.
32. Динамические структуры данных (однонаправленные и двунаправленные списки).
33. Создание списка, печать, удаление, добавление элементов (на примере однонаправленных и двунаправленных списков).
34. Потоковый ввод-вывод в С++. Открытие и закрытие потока. Стандартные потоки ввода-вывода.
35. Символьный, строковый, блоковый и форматированный ввод-вывод.
36. Прямой доступ к файлам.
37. Создание бинарных и текстовых файлов, удаление, добавление, корректировка элементов, печать файлов.

25. Примеры задач для подготовки к экзамену

1. Определить, попадет ли точка с координатами (x, y) в указанную область.

2. Дана последовательность целых чисел из n элементов. Найти:
    - - среднее арифметическое;
    - - (максимальное значение;
    - - количество отрицательных элементов;
    - -номер минимального элемента;
    - -количество четных чисел;
    - - минимальный из четных элементов этой последовательности.

3. Дана последовательность целых чисел, за которой следует 0. Найти:
    - - среднее арифметическое;
    - - (максимальное значение;
    - - количество отрицательных элементов;
    - -номер минимального элемента;
    - -количество четных чисел;
    - - минимальный из четных элементов этой последовательности.
4. Найти сумму чисел Фибоначчи, меньших заданного числа Q.
5. Напечатать N простых чисел.
6. Дан массив целых чисел. Найти:
    - - среднее арифметическое;
    - - (максимальное значение;
    - - количество отрицательных элементов;
    - -номер минимального элемента;
    - -количество четных чисел;
    - - минимальный из четных элементов этого массива.
7. Дан массив целых чисел. Перевернуть массив.
8. Дан массив целых чисел. Поменять местами пары элементов в массиве: 1и2, 3 и 4, 5 и 6 и т. д.
9. Циклически сдвинуть массив на К элементов влево (вправо).
10. Найти первое вхождение элемента К в массив целых чисел.
11. Удалить из динамической матрицы строку с номером К.
12. Дана строка символов, состоящая из слов, слова разделены между собой пробелами. Удалить из строки все слова, начинающиеся с цифры.
13. Сформировать динамический массив строк. Удалить из него строку с заданным номером.
14. Заданы координаты сторон треугольника. Если такой треугольник существует, то найти его площадь. Решить задачу с использованием функций.
15. Дан массив int a[100]. Удалить из массива все четные элементы.
16. Дан массив int *a.Удалить из массива все элементы, совпадающие с первым элементом, используя динамическое выделение памяти.
17. Найти количество цифр в строке символов, используя функции.
18. Удалить из однонаправленного (двунаправленного) списка элемент с заданным номером (ключом).
19. Добавить в однонаправленный (двунаправленный) список элемент с задан-

ным номером.

20. Удалить из бинарного файла, в котором записаны целые числа все четные элементы.
21. Добавить в бинарный файл, в который записаны элементы типа

struct Student
{char name[20];int age;};
К элементов после элемента с заданной фамилией.

22. Удалить из текстового файла все четные строки.
23. Добавить порядковый номер в каждую строку текстового файла.
24. struct Date
{char Month[15];int Day;}
В файле содержатся даты типа Date. Заменить все даты, у которых поле Month равно “Май”,”Июнь” или ”Июль” на даты, у которых поле Day не меняется а поле Month меняется на “Август”.
25. В текстовом файле заменить все строки, начинающиеся с буквы ‘f’ на строки, начинающиеся с буквы ‘a’.