ОГЛАВЛЕНИЕ

**Введение....5**
**1Простейшие конструкции языка C++....7**
1.1Алфавит языка....7
1.2Структура программы....8
1.3Данные....10
1.3.1 Типы данных....10
1.3.2 Литералы ....12
1.3.3 Объявление переменных и поименованных констант....14
1.3.4* Перечисляемый тип....16
1.3.5* Объявление нового типа данных....16
1.4Операции ....17
1.5Выражения....21
1.6Элементарный ввод вывод....23
1.6.1Форматный ввод /вывод ....24
1.6.2 Ввод/вывод строк....26
1.6.3 Ввод/вывод символов....26
**2Управляющие операторы языка....29**
2.1Блок операторов и пустой оператор....29
2.2Оператор условной передачи управления....29
2.3Оператор выбора....33
2.4Операторы организации циклических процессов....35
2.4.1 Цикл с предусловием (Цикл-пока)....36
2.4.2 Цикл с постусловием (Цикл-до)....38
2.4.3 Оператор счетного цикла for....39
2.5Неструктурные операторы передачи управления....41
**3Сложные структуры данных. Адресная арифметика....44**
3.1Указатели и ссылки....44
3.1.1 Определение указателя. Типизированные и нетипизированные указатели и операции над ними....44
3.1.2 Понятие ссылки....47
3.1.3 Отличие ссылки от указателя....48
3.2Адресная арифметика....48
3.3Управление динамической памятью ....50
3.4Массивы....52
3.4.1 Одномерные массивы....53
3.4.2 Многомерные массивы....55
3.5Строки....59
3.5.1 Объявление и инициализация строк....59
3.5.2 Ввод и вывод строк....62
3.5.3 Функции, работающие со строками....64
3.6Структуры....68
3.7* Объединения....74
3.8Динамические структуры данных. Списки....75
3.8.1 Описание элементов списковых структур....76
3.8.2 Основные приемы работы....76
**4Функции. Модульное программирование....82**
4.1Функции C++....82
4.1.1 Классы памяти переменных....86
4.1.2 Параметры сложных структурных типов....88

4.1.3* Рекурсивные функции....96
4.1.4* Дополнительные возможности функций C++....98
4.2Модули C++....100
4.3* Средства создания универсальных подпрограмм....102
4.3.1 Параметры – многомерные массивы неопределенного размера....102
4.3.2 Параметры-функции....105
**5Файловая система....110**
5.1Механизм выполнения операций ввода/вывода. Типы файлов ....110
5.2Объявление, открытие и закрытие файлов ....110
5.3Работа с файловым указателем....111
5.4Текстовые файлы. Стандартные текстовые файлы....112
5.4.1 Ввод/вывод символов....113
5.4.2 Ввод/вывод строк....115
5.4.3 Форматный ввод/вывод ....117
5.5 Двоичные файлы....118
5.6Удаление и переименование файлов....122
**6 Препроцессор языка С....124**
6.1Команда #include....124
6.2Команды #define и #undef....124
6.3Команды условной компиляции....126
6.4Некоторые предопределенные макроопределения....128
**Литература....130**
**Приложение А Оптимизация кода программы....131**
**Приложение Б Некоторые опции компилятора и компоновщика....133**

МГТУ им. Н.Э. Баумана

Факультет «Информатика и Системы Управления»

Кафедра ИУ-6 «Компьютерные системы и сети»

Иванова Галина Сергеевна, Ничушкина Татьяна Николаевна,

Самарев Роман Станиславович

Средства процедурного программирования Microsoft Visual C++ 2008

Учебное пособие по дисциплинам

Алгоритмические языки и программирование,

Программирование, Системное программное обеспечение

МОСКВА

2010 год МГТУ им. Н.Э. Баумана

## АННОТАЦИЯ

Учебное пособие содержит описание средств процедурного программирования на Visual C++ в среде Microsoft Studio 2008. Подробно обсуждаются структура программы, типы данных, способы и особенности реализации вычислений, операторы организации ветвлений и циклов, а также адресная арифметика, основы работы с динамической памятью, особенности моделирования работы с массивами в C++. Большое внимание уделяется также организации подпрограмм и различным способам передачи параметров в них и организации файловой системы хранения данных на внешних носителях информации.

Пособие предназначено для студентов 1 курса кафедры «Компьютерные системы и сети» (ИУ6) и студентов, обучающихся по аналогичной программе на Аэрокосмическом факультете университета (АК5), которые изучают C++ в качестве второго языка программирования. Степень глубины проработки материала соответствует именно изложению второго языка. Однако пособие может быть полезно и студентам, изучающим C++ в качестве первого языка программирования. При первом знакомстве с материалом разделы, отмеченные звездочкой, целесообразно опустить.

# ВВЕДЕНИЕ

Язык Си был создан в 1972 году Денисом Ритчи (см. рисунок В.1). Языком системных программистов на тот момент был ассемблер, эффективность написания программ на котором весьма мала. Поэтому целью Дениса Ритчи было создание языка, специально предназначенного для написания системных программ.

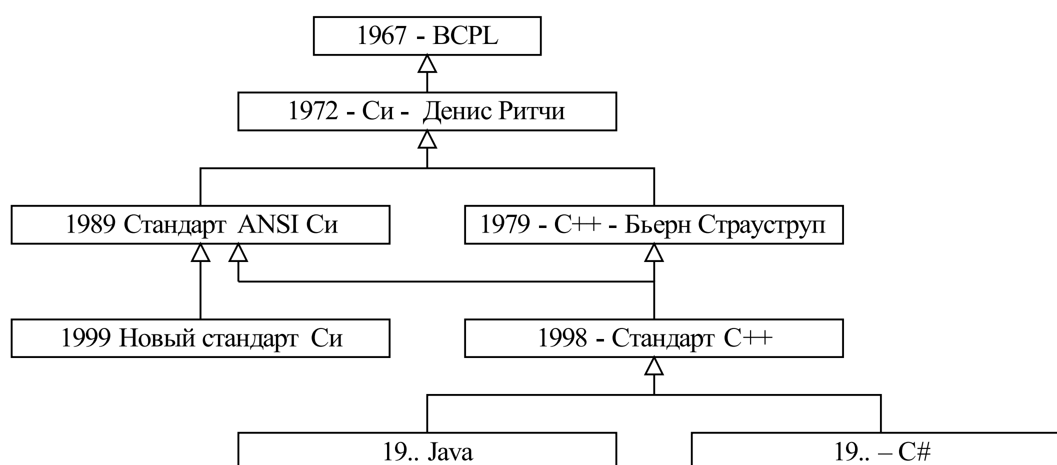



**Рисунок В.1** – Этапы развития языков семейства Си..С++

Язык в то время мало выделялся из множества других, однако в его пользу говорили компактный синтаксис, наличие библиотек часто используемых в системных программах подпрограмм и компиляторы для трех используемых на тот момент платформ: вычислительных машин семейства ЕС ЭВМ, АРМов (Автоматизированное Рабочее Место) и только появляющихся персональных ЭВМ на базе микропроцессора i8086.

Особое место среди других языков программирования Си занял после того, как Бьерн Страуструп создал на его основе язык С++, включающий средства объектной технологии программирования, а компания Microsoft приняла решение об придании ему статуса системного языка Windows.

В настоящее время Си и С++ – весьма популярные языки программирования, на базе которых построены более современные языки программирования, такие как С# и Java, предназначенные для создания сетевых программ.

*Основные достоинства* языка – компактный синтаксис, наличие большого количества специальных средств, упрощающих написание сложных системных программ, многоплатформенность.

*Основной недостаток* – «незащищенный» синтаксис, при котором в языке возможно существование близких по форме допустимых конструкций, что часто не позволяет иден-

тифицировать ошибку на этапе компиляции программы, а потому удлиняет и усложняет ее отладку.

Широко распространены следующие компиляторы C/C++:

- gcc – GNU C Compiler;
- Microsoft Visual C++;
- Intel C++ Compiler.

**Компилятор gcc (GNU C Compiler)** является свободно распространяемым программным продуктом и является де-факто стандартом для сборки C/C++ программ под операционными системами Linux и FreeBSD. Реализован для множества аппаратных платформ и различных операционных систем. Реализация для ОС Windows называется mingw. Используется для компиляции свободно распространяемых программных продуктов с открытыми исходными кодами.

**Коммерческий компилятор Microsoft Visual C++** для процессоров семейств x86, x86-64 и IA-64 наиболее распространенный компилятор для создания приложений для ОС Windows (включая различные её версии для различных платформ Win32, Win64, WinCE). Чаще всего используется совместно со средой разработки MS Visual Studio. В отличии от gcc, ориентируется не на соблюдение принятых стандартов C/C++ как таковых, а на внутренние спецификации Microsoft. Компилятор лучше оптимизирует код программ, чем mingw или gcc, однако не полностью с ним совместим.

**Коммерческий компилятор Intel C++ Compiler** для процессоров семейств x86, x86-64 и IA-64 позиционируется как оптимизирующий компилятор для приложений, критических к скорости работы или аппаратным ресурсам. Может использоваться в качестве замены Microsoft Visual C++ для ОС MS Windows, в том числе совместно с MS Visual Studio. Данный компилятор также существует для ОС Linux и Mac OS, однако распространен мало и используется только для создания коммерческих приложений. Не полностью совместим с gcc.

В настоящем пособии рассматривается компилятор Microsoft Visual C++ , входящий в состав Visual Studio 2008.

# 1 ПРОСТЕЙШИЕ КОНСТРУКЦИИ ЯЗЫКА С++

К простейшим конструкциям языка относятся операторы объявления данных и построения выражений. Кроме того в настоящей главе рассмотрены стандартные функции ввода вывода, используемые для ввода данных с клавиатуры и вывода результатов на экран дисплея.

## *1.1 Алфавит языка*

*Алфавит* языка С++ включает:

- строчные и прописные буквы латинского алфавита: A..Z и a..z;
- арабские цифры: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;
- шестнадцатеричные цифры: 0..9, a..f или A..F;
- специальные символы, например «+», «-», «*», «/», «=», «<», «&», «;» и т. д.;
- служебные слова.

Служебные слова зарезервированы в языке для специального применения, т.е. их нельзя использовать в качестве идентификаторов. Они применяются в конструкциях языка в качестве стандартных описателей или ключевых слов. Стандарт ANSI языка Си предусматривает следующий список служебных слов:

```
auto      default  extern  int       signed  typedef
break     do       float   long      sizeof  union
case      double   for     register  static  unsigned
char      else     goto    return    struct  void
continue  enum     if      short     switch  while
```

В разных реализациях есть дополнительные служебные слова, например, компиляторы фирмы Borland используют также: `asm cdecl far pascal const volatile`. Язык С++ добавляет еще несколько ключевых слов, например: `catch class friend inline new operator private`.

Из символов алфавита строятся конструкции – слова и предложения языка. Простейшей конструкцией является идентификатор.

*Идентификатор* – последовательность букв латинского алфавита, десятичных цифр и символов подчеркивания, начинающаяся не с цифры, например:

```
ABC   abc   Abc   ABc   AbC   _a  MY_Primer_1   Prim_123
```

Следует иметь в виду, что *прописные и строчные буквы в идентификаторах различаются*, т. е. идентификаторы `ABC abc Abc ABc AbC` с точки зрения компилятора языков Си и C++ различны.

Идентификаторы используют для обозначения имен переменных, констант, типов, подпрограмм и т. д.

На длину различаемой части идентификатора конкретные реализации накладывают ограничения. Так компилятор Visual C++ 2008 компании Microsoft различает 2048-х первых символов любого идентификатора.

### *1.2 Структура программы*

Программа, написанная на Си или C++ с использованием средств процедурного программирования, в общем случае включает:

- команды препроцессора;
- объявления переменных, констант и типов;
- прототипы функций;
- определения функций.

*Препроцессор* – это программа, которая обрабатывает исходный текст до компилятора. Посредством команд препроцессора, например, определяется перечень файлов, содержащих прототипы стандартных функций из библиотек, которые должны быть подключены при компиляции программы.

Выполнение программы всегда начинается с функции, имеющей имя `main()` – это основная функция программы, которая получает управление от операционной системы при запуске программы и возвращает управление операционной системе при завершении программы. Остальные функции прямо или косвенно получают управление от основной функции программы.

Описание каждой функции состоит из заголовка и тела функции:

**<Тип результата или void> <Имя функции> ([<Список параметров>])**

**{ [ < Объявление переменных и констант >]**

**<Операторы>**

**}**

*Примечание*. При описании конструкций языка далее будет использоваться специальная общепринятая нотация. Согласно этой нотации за исключением оговоренных случаев:

<…> – обозначает конструкцию языка, которая заменит свое описание;

[…] – обозначает необязательный элемент конструкции, который может отсутствовать.

Так в данном случае список параметров и объявление локальных переменных и констант в описании функции могут отсутствовать.

**Пример 1.1.** Программа определения наибольшего общего делителя двух чисел.

```
#include <locale.h>     // команды препроцессора подключают файлы прототипов
#include <stdio.h>      // функций ввода-вывода и подключения русских букв
#include <conio.h>
int a=18,b=24,c;        // объявление двух инициализированных и одной
                        // неинициализированной переменных
int nod(int a,int b);   // прототип функции nod()
int main()              // заголовок основной программы
{
    setlocale(0,"russian"); // установка кодов русских букв
    c=nod(a,b);             // вызов функции nod()
    printf("НОД=%d\n",c);   // вывод результата на экран
    puts("Нажмите любую клавишу для завершения..."); // сообщение
    _getch();               // обработка нажатия любой клавиши
    return 0;               // возврат нулевого кода завершения
}
int nod(int a,int b)    // заголовок функции nod()
{
    while (a!=b)            // цикл-пока a не равно b
        if (a>b) a=a-b;     // если a>b, то вычитаем из a b
        else b=b-a;         // иначе вычитаем из b a
    return a;               // возвращаем результат функции
}
```

Первое предложение программы – команда препроцессора `#include <locale.h>`. Она подключает к программе файл `locale.h`, содержащий прототип функции, которая обеспечивает возможность ввода/вывода русских букв. Команды препроцессора `#include <stdio.h>` и `#include <conio.h>` аналогично подключают файлы `stdio.h` и `conio.h`, содержащие прототипы используемых в программе функций

Оглавление

ввода/вывода `printf()` и `_getch()`. Поиск подключаемых файлов при этом выполняется в стандартных путях среды (см. раздел 6.1).

Затем объявлены внешние переменные целого типа `a`,`b` и `c`.

Далее следует прототип функции `nod()`. Прототипы используют, если *вызов функции предшествует ее описанию*, как в данном случае. Если бы функция `nod()` была описана первой, то прототип можно было бы в программу не включать.

Функция `nod()` определяет наименьший общий делитель двух целых чисел. Основная функция `main()` осуществляет вызов функции `nod()` и вывод результатов работы. Кроме этого она организует приостановку выполнения программы, пока пользователь не прочитает результат и не нажмет какую-либо клавишу.

## *1.3 Данные*

Любая программа существует для того, чтобы обрабатывать какие-либо данные. В качестве данных могут использоваться числа, а также текстовая информация, представленная в различных видах: вводимая с клавиатуры или из файла или наоборот выводимая на экран или в файл. При этом данные в программе могут быть неизменяемыми, т.е. *константами*, и изменяемыми, в которых сохраняются получаемые результаты.

### 1.3.1 Типы данных

*Тип данных* – это характеристика данных, которая определяет:

- *интерпретацию значений и множество операций*, которые могут выполняться над ними;
- *диапазон и шаг изменения хранимых значений*.

Стандартно определено некоторое количество скалярных (простых) типов данных, которые программист может использовать *без предварительного описания*. Во внутреннем представлении каждому типу сопоставлен собственный *формат записи* значений.

В Visual C++ 2008 принято различать целочисленные (интегральные) и вещественные типы данных. Значениями целочисленных типов данных могут быть положительные, отрицательные и нулевые целые числа. В таблице 1 приведены стандартные целочисленные типы данных.

**Таблица 1 -** Целочисленные типы данных Visual C++ 2008

| Имя типа | Подтипы | Размер, Байт | Интервал значений |
|---|---|---|---|
| **[int]** или **long** или **_int32** | `[signed] [int]`<br>`unsigned [int]`<br>`[signed] long`<br>`unsigned long` | 4 | $-2^{31}..\ 2^{31}-1$<br>$0..\ 2^{32}-1$ |
| **short** или **_int16** | `[signed] short`<br>`unsigned short` | 2 | -32768..32767<br>0..65535 |
| **char** или **_int8** | `[signed] char`<br>`unsigned char` | 1 | -128..127<br>0..255 |
| **long long** или **_int64** | `[signed] long long`<br>`unsigned long long` | 8 | $-2^{63}..\ 2^{63}-1$<br>$0..\ 2^{64}-1$ |
| **bool** | | 1 | false (0), true(1) |

Наиболее используемый тип `int` – целое число со знаком. Знак задается первым битом числа. При этом 0 – обозначает «плюс», а 1 – «минус». Квадратные скобки в таблице при указании типа `int` означают, что, если тип данных при объявлении в программе не указан, то им присваивается тип `int` по умолчанию.

Тип `long` в данной реализации совпадает по характеристикам с типом `int`. В прошлых версиях это было не так и может быть не так для компиляторов других фирм.

Типы `short` и `long long` применяются соответственно для небольших и очень больших чисел. Для хранения этих данных использованы аналогичные форматы, различающиеся длиной внутреннего представления.

Символы, как это принято в программировании, представляются своими кодами (номерами) по расширенной таблице ANSI. При этом тип `char` в Си и в C++ соответствует целому числу со знаком (!). Поэтому при работе с символами, коды которых превышают 127, используют беззнаковый подтип символьного типа `unsigned char`.

Логический тип в Си первоначально отсутствовал. При необходимости в качестве «истины» фигурировало любое целое число отличное от нуля. Соответственно ноль интерпретировался как «ложь». Однако позднее логический `bool` тип все-таки был добавлен в C++. Для этого типа данных, как это принято в других языках программирования: `true` = 1, `false` = 0. Для хранения значений типа `bool` используется 1 байт памяти.

Типы `_int32`, `_int16` и `_int64` были введены в Microsoft Visual C++ для простоты запоминания системы типов.

К интегральным типам формально относится и специальный тип, используемый для представления адресов. Этот тип будет описан позднее (см. раздел 3.1).

Описание стандартных вещественных типов данных приведено в таблице 2.

**Таблица 2** – Стандартные вещественные типы данных

| Тип | Размер, байтов | Значащих цифр | Минимальное положительное число | Максимальное положительное число |
|---|---|---|---|---|
| **float** | 4 | 6 | $1.175494351 \times 10^{-38}$ | $3.402823466 \times 10^{38}$ |
| **double (long double)** | 8 | 15 | $2.2250738585072014 \times 10^{-308}$ | $1.797693134862318 \times 10^{308}$ |

Кроме указанных выше типов данных стандартно задан еще один тип – *неопределенный*. Он обозначается служебным словом `void`. Этот тип используют в тех случаях, когда тип неизвестен или не может быть указан. Например, если функция не возвращает значений, являясь по сути процедурой, то тип ее возвращаемого значения указывают как `void`, т.е. «пустой, неопределенный».

### 1.3.2 Литералы

*Литерал* – это запись фиксированного числового, символьного или строкового значения в тексте программы. Литералы специально объявлять не надо, их используют при записи выражений или других конструкций языка. Компилятор, обнаружив литерал, относит его к соответствующему типу по форме записи и величине числового значения.

В программе на C++ допускаются литералы пяти видов: целочисленные, вещественные, перечисляемые, символьные и строковые.

**Целочисленные литералы.** По форме записи целочисленные литералы могут быть десятичными, восьмеричными и шестнадцатеричными.

*Десятичный литерал* записывается как последовательность десятичных цифр, перед которой может стоять знак. При этом наличие незначащих нулей перед числом не допускается, например:

```
16, 56783, 0, -567, +7865.
```

*Восьмеричный литерал* записывается как последовательность цифр от 0 до 7, всегда начинающаяся с нуля, перед которой может также стоять знак, например:

```
016, 020, -0777.
```

*Шестнадцатеричный литерал* записывается как последовательность шестнадцатеричных цифр (цифры от 0 до 9 и буквы от a до f или от A до F), которая начинается сочетанием 0x и перед которой, как и в предыдущих случаях, может стоять знак:

`0x30, 0xF, 0xfa4, -0x56AD.`

В зависимости от значения целой константы компилятор представляет ее в памяти в соответствии с существующими стандартными *типами*. Для явного указания типа внутреннего представления программист может использовать суффиксы: «L», «l» – соответствует типу `long`, а «U», «u» – типу `unsigned long`, например:

`-64L, 067u.`

**Вещественные литералы** записывают в форматах с фиксированной или плавающей точкой. Форма записи «с фиксированной точкой» предполагает, что при записи числа десятичная дробь отделена от целой части числа точкой, например 0.0021.

Запись «с плавающей точкой» использует *экспоненциальную* форму. Число в экспоненциальной форме представляется в виде произведения мантиссы на степень основания системы счисления, например $2.1{\times}10^{-3}$. В программе на Си или C++ при этом используют строчную форму, в которой вместо $10^{-3}$ указывают e-3 или E-3. Тогда все число будет записано как 2.1e-3.

При этом также в явном виде может быть указан тип числа, определяющий его внутреннее представление: «F», «f» – `float`, «L», «l» – `long double`, например:

`66.   .045   .0   3.1459F   1.34e-12   -45E+6L   56.891.`

Без суффиксов F или L под вещественную константу отводится 8 байт (тип `long double`).

**Символьные литералы** – это отдельные символы или Esc-последовательности символов, заключенные в апострофы.

Символы используют, если для их ввода существуют клавиши на клавиатуре, например `'Z'`, `'*'`, `'$'` . При необходимости можно сразу записать несколько символов, например `'db1'`.

Esc-последовательностью называют последовательность символов, начинающуюся с символа «\». Такие последовательности используют для записи:

- кодов символов, отсутствующих на клавиатуре – в этом случае указывают «\» и коды ANSI в 8-ричном ('\ooo') или 16-ричном ('\xhh') виде, например '\012' – код символа в восьмеричном виде, `'\x07\x07'` – два символа, заданных кодами в шестнадцатеричном виде, '\0' – символ с кодом 0;

- служебных символов, таких как «′», «\», «?», «″» – косая черта перед ними говорит, что необходим код символа, а не подразумеваемые им операции, например `\′`, `\″`;
- кодов управляющих символов, например:

  `′\n′` – символ перехода на следующую строку,

  `′\t′` – символ горизонтальной табуляции,

  `′\a′` – символ звонка (тревога),

  `′\b′` – символ возврата на одну позицию и др.

**Строковые литералы** записывают как последовательность символов, заключенную в двойные апострофы (кавычки), например: `″Это пример строкового литерала″`.

В отличие от символьных строковые литералы во внутреннем представлении завершаются байтом, содержащим число «0». Это позволяет определять длину строкового литерала при выполнении операций с ним.

Среди символов строки также могут присутствовать Esc-последовательности, например: `″\nЭто строка, \nиначе -\″строковый литерал\″.″`

### 1.3.3 Объявление переменных и поименованных констант

Помимо литералов для работы с данными в программе используют поименованные константы или переменные.

*Поименованные константы* – неизменяемые данные, обращение к которым выполняется по имени. Такие константы должны быть специально объявлены в программе, и им должны быть назначены типы и заданы значения.

*Переменные* – поименованные данные, которые могут изменяться в процессе выполнения программы. Обращение к этим данным, как и к поименованным константам, осуществляется по имени. При объявлении переменной для нее необходимо указать тип значений, которые в эту переменную можно записать.

В каждый момент времени переменная хранит значение, записанное в нее ранее. Запись значения может выполняться сразу при объявлении переменной, тогда переменная называется *инициализированной*. Если при объявлении переменной начальное значение не задается, то значение переменной должно быть определено во время выполнения программы. Такая переменная называется *неинициализированной*. *Чтение неинициализированной переменной до записи в нее значения приводит к неправильному результату.*

Переменные и поименованные константы объявляют в конструкции следующего вида:

**[<Изменяемость>] [<Тип>] <Имя> [=<Значение>][, <Имя> [=<Значение>],…];**

где <Изменяемость> – описатель возможности изменения значений, этот описатель можно не указывать, тогда по умолчанию объявляется обычная переменная,

`const` – поименованная константа – неизменяемое значение,

`volatile` – независимо меняющаяся переменная, которая может обновляться в промежутках между явными обращениями к ней. Этот описатель отключает оптимизацию компилятора для указанной переменной, и чаще всего применяется при реализации программ с параллельным выполнением;

<Тип> – описатель типа данных: `int`, `char`, `float`, `double` и т.д., если описатель типа отсутствует, то по умолчанию берется `int`;

<Имя> – идентификатор переменной или константы;

<Значение> – начальное значение переменной или значение константы.

Примеры:

```
// объявление неинициализированных переменных
int f,c,d;          // три переменных для хранения целых чисел
float r;            // одна переменная для хранения вещественного числа
I,j;                // две переменные типа int (по умолчанию)
unsigned int max,min; // две беззнаковые целочисленные переменные
char c1,c2;           // две символьные переменные
unsigned char c5;     // переменная для хранения беззнакового кода символа
// объявление инициализированных переменных
double k=89.34; float eps=0.1e-4; char ch='G';
// поименованные константы
const long a=6;
const float pp=6.6e-34;
```

Переменные и поименованные константы могут быть объявлены в любом месте программы: вне всех функций, внутри функций, в любом месте функции, а в С++ – даже внутри управляющих операторов, например:

```
for(s=0,int i=1;i<n;i++) s+=i;//переменная i объявлена внутри оператора
```

Переменные, объявленные внутри оператора, существуют только во время его выполнения. Использование таких переменных часто приводят к ошибкам, а потому – не рекомендуется.

Основное условие размещения объявлений: объявление должно стоять до первого обращения к переменной или константе. Однако технология создания надежных программ рекомендует все необходимые объявления делать в начале соответствующей функции.

Кроме того, в объявлении может присутствовать описатель класса памяти, определяющий видимость переменной в подпрограммах и время хранения ее значения, например: `static`, `extern` и т. д. (см. раздел 4.1.1).

### 1.3.4 * Перечисляемый тип

Перечисляемый тип появился еще в Си, он позволяет определить собственный набор поименованных целочисленных значений для объявляемой переменной. Тип описывается следующим образом:

**`enum` [<Имя типа>]{<Имя>[=<Целое>][,<Имя>[=<Целое>]…]} <Список имен переменных>;**

где <Имя типа> – идентификатор перечисляемого типа, обязательно указывается, если объявление переменных предполагается выполнять в отдельной конструкции;

<Имя> – идентификатор целочисленного значения;

<Целое> – присваиваемое целое значение, может быть опущено, если значение, сопоставляемое идентификатору, при описании типа не указано, то берется предыдущее значение и к нему добавляется единица, для первого значения по умолчанию берется 0, например:

```
enum days (sun, mon, tue, fri=5, sat) day; // конструкции описывает
                 // тип days и объявляет переменную day, которая может принимать
                 // значения: sun=0, mon=1, tue=2, fri=5, set=6
```

С переменными перечисляемого типа можно выполнять необходимые операции, например: `Day=fri;` .

### 1.3.5 * Объявление нового типа данных

В С++ появилась конструкция, позволяющая определять новые типы переменных:

**`typedef` <Описание типа> <Имя объявляемого типа>;**

Например:

```
typedef unsigned int word; // переименовываем подтип беззнаковый целый
                           // в word
typedef enum {FALSE, TRUE} boolean; // объявляем новый тип boolean
                                    // и разрешаем переменным этого типа
                                    // принимать значения FALSE(0) и TRUE(1)
```

В программе можно объявлять переменные указанных типов, например:

```
word l,m;        boolean flag;
```

### *1.4 Операции*

Программирование вычислений предполагает, что над данными программы выполняются некоторые преобразования. Для этого в Си и С++ определены операции соответствующих типов: арифметические, логические, отношений, логические поразрядные, порядковые, сдвига, условная и присваивания. Типы данных, над которыми можно выполнять перечисленные операции, указаны в таблице 1.3.

**Таблица 1.3** – Область действия операций С++

| Тип операций | Типы данных | | | |
|---|---|---|---|---|
| | Интегральные | | | |
| | Целые | Символьный | Логический | Адресный |
| Арифметические | + | + | – | + |
| Логические | + | – | + | – |
| Отношений | + | + | + | + |
| Логические поразрядные | + | – | – | – |
| Порядковые | + | + | + | + |
| Сдвиги | + | – | – | – |
| Условная | + | + | + | + |
| Присваивания | + | + | + | + |

**Арифметические операции.** Арифметические операции делятся на:

- унарные «+» и «-», например: `-beta`, унарный плюс введен для симметрии с унарным минусом;
- бинарные: сложение «+», вычитание «-», умножение «*», деление «/», получение остатка целочисленного деления «%».

При делении двух целых чисел и при определении остатка от деления целых чисел операция выполняется с точностью до целых (!), т.е. 5/2 = 2, 5%2=1.

*Неявное преобразование типов.* Если бинарная операция выполняется над операндами различных типов, то перед операцией оба операнда преобразуются в тип с большей разрядной сеткой по следующим схемам:

```
char  ⇒ short  ⇒ int ⇒ long
float ⇒ double ⇒ long double
```

Целые числа при этом преобразуются в вещественные.

Например:

```
int a=5, b = 3; float c=9.3
…
a+b          // результат  8, тип int
a/b          // результат  1, тип int
a%b          // результат  2, тип int
a*b          // результат 15, тип int
c/b          // результат  3.1, тип float
(a+b)/(a-b*a) // результат  0, тип int
```

Для выполнения операций над некоторыми типами данных требуется *явное преобразование типов.* Такое преобразование может быть выполнено двумя способами: с использованием функции типа и применением канонической формы.

*Функциональное преобразование* записывается следующим образом:

**<Имя типа> (<Выражение>)**

Например:

```
int(3.14)=3          // будет отброшена дробная часть числа
float(2/3)=0.0       // сначала будет выполнено целочисленное деление
int('A')=65          // код символа
```

Однако функциональная запись не подходит для подтипов, когда имя типа состоит из нескольких описателей. В этом случае применяется *каноническая форма* преобразования:

**(Имя типа)<Выражение>**

Например:

```
(unsigned long)(x/3+2);  (long)25;  (unsigned char)123;
```

Если определить новый тип – тогда можно использовать и функциональное преобразование, например:

```
typedef unsigned long int uli;
uli(x/3-123);
```

**Логические операции.** К логическим операциям относятся:

`!` – логическое «не» – отрицание целочисленного операнда – унарная логическая операция, выполняется над целым числом, которое интерпретируются в логическое значе-

ние по правилу Си (0 – false, не 0 – true), результат – целочисленный 0 или не 0, интерпретируемые по тем же правилам;

`&&` – логическое «и» – конъюнкция целочисленных операндов, результат операции <Операнд1>&&<Операнд2> = «Истина» только тогда, когда оба операнда «Истина»;

|| – логическое «или» – дизъюнкция целочисленных операндов, результат операции <Операнд1>||<Операнд2> = «Истина», если хотя бы один операнд «Истина».

Чаще всего операндами логических операций являются результаты операций отношения. Например:

```
6>2&&3==3          // результат «Истина»
!(6>2&&3==3)       // результат «Ложь»
x!=0 && 20/x<5     // второе выражение вычисляется, если x!=0, иначе результат
                   // всегда «Ложь»
```

**Операции отношения.** Операции меньше «<», больше «>», равно «==», неравно «!=», меньше или равно «<=», больше или равно «>=» применяют к числам и символам, точнее кодам этих символов, в результате получают логическое значение «Истина» или «Ложь».

Например:

```
int a = 5;  int b = 3;  …
a > b  // результат «Истина»
a == b // результат «ложь»
```

**Логические поразрядные операции.** В C++ реализованы следующие логические поразрядные операции:

`~` – поразрядная инверсия битов («не») в битовых представлениях целых чисел,

`&` – поразрядная конъюнкция («и») битов в битовых представлениях целых чисел,

`|` – поразрядная дизъюнкция («или») битов в битовых представлениях целых чисел,

`^` – поразрядное исключающее «или» битов в битовых представлениях целых чисел.

Например:

```
6&5 // результат 4: 00000110 & 00000101 → 00000100
6|5 // результат 7: 00000110 | 00000101 → 00000111
6^5 // результат 3: 00000110 ^ 00000101 → 00000011
```

**Порядковые операции.** Существуют всего две порядковые операции, каждая из которых имеет две модификации:

++<Идентификатор>, <Идентификатор>++ (следующее);

- -<Идентификатор>, <Идентификатор> - - (предыдущее).

Местоположение знаков операций (до или после операнда) определяет, в какой момент осуществляется изменение операнда. Если знак стоит слева от операнда – то сначала значение изменяется, а потом принимает участие в вычислении. Если знак стоит справа от операнда – то сначала операнд принимает участие в вычислении, а затем меняется его значение.

Например:

а) `i++;`

б) `a=5;   i=3;   c=a*++i;` // результат операции c = 20, i=4

**Сдвиги.** В С++ используются две операции сдвига:

>> – сдвиг вправо битового представления целого числа на количество разрядов, задаваемое правым целочисленным операндом,

<< – сдвиг влево битового представления целого числа на количество разрядов, задаваемое левым целочисленным операндом.

Например:

`4<<2` // результат 16: 00000100 << 00010000

`5>>1` // результат 2: 00000101 >> 00000010

**Операции присваивания.** В С++ присваивание относится к операциям. При этом помимо простого присваивания «=», при котором в переменную левого операнда копируется результат выражения правой части, определены также операции присваивания «с накоплением»: «+=», «-=», «*=», «/=», «%=», «&=», «^=», «|=», «<<=», «>>=». Эти операции присваивают левому операнду результат выполнения заданной операции, выполненной над левым и правым операндами**.**

Например:

```
int k;
k=8;      // операция простого присваивания
k*=3;     // результат  24
k+=7;     // результат  31
```

**Условная операция** позволяет организовать выбор одного из двух выражений в зависимости от результата заданного логического выражения, т.е. реализовать простейший вариант ветвления. Это единственная операция в С++, которая выполняется над тремя операндами, т.е. является триарной. Операция имеет следующий формат записи:

**<Выражение 1>?<Выражение 2>:<Выражение 3>**

При выполнении первым вычисляется значение выражения 1. Если оно истинно, т.е. результат не равен 0, то вычисляется выражение 2, которое и становится результатом. Если

при вычислении выражения 1 получается 0, то вычисляется выражение 3, которое становится результатом.

Например:

```
x<0?-x:x;
printf("%3d%c%",a,i==n?' ':'\n');
```

### *1.5 Выражения*

Обычно последовательно выполняемые операции в программе компонуются в *выражения*. Различают простые и сложные выражения.

**Простые выражения** представляют собой строковую запись формул, которая может включать литералы, поименованные константы, переменные, обращения к стандартным функциям и знаки операций, такие как «+», «-» и т.д.

Порядок выполнения операций в выражении определяется приоритетами операций и их ассоциативностью (для последовательно записанных одинаковых операций). Для изменения порядка выполнения операций используются круглые скобки.

Грамматика C++ определяет 16 категорий приоритетов операций или рангов. Операции 1 ранга имеют высший приоритет, т.е. выполняются в первую очередь:

1. ( ) [ ] -> :: .
2. ! (не) + - (унарные) ++ -- &(адрес) *(указатель) `sizeof new delete`
3. .* ->*
4. * / %
5. + - (бинарные)
6. << >>
7. < <= > >=
8. = = ! =
9. &(поразрядное и)
10. ^(исключающее или)
11. | (поразрядное или)
12. &&
13. ||
14. ?:
15. = *= /= %= += -= &= ^= |= <<= >>=
16. ,

Например:

```
int  a=10, b=3; float ret; ret=a/b;  // результат: ret=3
c=1;  b=c++;       // результат: b=1, c=2
c=1;  sum=++c;     // результат: c=2 , sum=2
c=a<<4;            // эквивалентно c=a*16;
a+=b;              // эквивалентно a=a+b;
a=b=5;             // эквивалентно b=5; a=b;
a=(b=s/k)+n;       // эквивалентно b=s/k; a=b+n;
c=(a>b)?a:b;       // если a>b, то c=a, иначе c=b
```

**Сложные выражения** представляют собой последовательность простых, записанных через запятую «,»:

**<Выражение1>,<Выражение2>,...<Выражение n>**

По таблице приоритетов операций запятая имеет низший ранг, поэтому простые выражения, разделенные запятой, выполняются последовательно слева направо, а в качестве результата выражения берется тип и значение *самого правого выражения***.**

Например:

```
int m=5,z;
z=(m=m*5,m*3);   // результат:  m=25, z=75
int d,k;
k=(d=4,d*8);     // результат:  d=4, k=32 .
c=(a=5, b=a*a);  // эквивалентно a=5; b=a*a; c=b;
```

**Стандартные математические функции.** В выражениях можно применять стандартные математические функции из библиотеки `math`. При их использовании необходимо подключить файл прототипов функций `math.h`:

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

Наиболее часто используют следующие функции:

```
fabs(< Вещественное выражение>)   //  абсолютное значение вещественного числа
abs(<Целое выражение>)            //  абсолютное значение целого числа
sqrt(<Вещественное выражение>)    //  √x
exp(<Вещественное выражение>)     //  e^x
log(<Вещественное выражение>)     //  ln x
log10 (< Вещественное выражение >) //  log10 x
sin(<Вещественное выражение>)     //  sin x
cos(<Вещественное выражение>)     //  cos x
```

atan(<Вещественное выражение>) // arctg x

tan(< Вещественное выражение >) // tg x

acos (< Вещественное выражение >) // арккосинус

asin (< Вещественное выражение >) // арксинус

sinh(<Вещественное выражение>) // гиперболический синус

cosh(<Вещественное выражение>) // гиперболический косинус

Кроме этих функций еще достаточно часто используют функции, позволяющие получать последовательности случайных чисел. Их прототипы находятся в файле conio.h.

rand () – генерация случайного числа $0 \le x < 2^{15}-1$;

srand (<Целое число>) – инициализация генератора случайных чисел.

Например:

```
srand((unsigned)time(NULL));//инициализация датчика текущим временем
num=rand()/1000.0;  // генерация вещественного случайного числа
```

Для такой инициализации к проекту должен быть подключен файл time.h:

```
#include <time.h>
```

### *1.6 Элементарный ввод вывод*

Для ввода/вывода данных скалярных типов и строк обычно используют стандартные функции ввода/вывода, описанные в библиотеке stdio. Для того чтобы применять эти функции необходимо, чтобы программе был доступен файл stdio.h, содержащий прототипы функций из этой библиотеки. Для этого необходимо подключить этот файл с помощью директивы препроцессора include:

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

В библиотеке существуют три вида функций, организующих элементарный ввод с клавиатуры и вывод на экран:

- форматный ввод/вывод – для выполнения операций ввода/вывода над скалярными значениями, символами и строками;
- ввод/вывод строк;
- ввод/вывод символов.

### 1.6.1 Форматный ввод /вывод

Ввод чисел, символов и строк с клавиатуры:

**int scanf(<Форматная строка>, <Список адресов переменных>);** // возвращает
//количество введенных значений или EOF(-1)

Вывод чисел, символов и строк на экран:

**int printf(<Форматная строка>, <Список выражений>);** // возвращает
// количество выведенных байтов

Форматная строка – это строка, которая помимо символов содержит управляющие спецификации вида:

%[-] [<Целое 1>] [.<Целое 2>] <Формат>

где «-» – выравнивание по левой границе,

<Целое 1> – ширина поля вывода;

<Целое 2> – количество цифр дробной части числа;

<Формат > – формат для ввода/вывода значения.

Основные форматы для ввода и вывода:

d – целое десятичное число;

u – целое десятичное число без знака;

o – целое число в восьмеричной системе счисления;

x – целое число в шестнадцатеричной системе счисления (% 4x – без гашения незначащих нулей);

f – вещественное число;

e – вещественное число в экспоненциальной форме;

c – символ;

p – ближний указатель (адрес);

s – символьная строка, вводит строку до первого пробела.

Кроме этого, форматная строка может содержать Esc-последовательности:

\n – переход на следующую строку;

\t – переход на следующую позицию табуляции;

\r – перевод каретки;

\f – перевод страницы;

\n hhh – вставка символа с кодом ANSI hhh (код задается в шестнадцатеричной системе счисления);

%% – печать знака %.

Примеры форматного ввода/вывода:

а) `i=26;`

`printf ("%-6d∪∪∪%%∪ %o∪ %x\n", i, i, i);`

Выведенная строка: `26∪∪∪∪∪∪∪%∪32∪1A ↵`

б) `scanf("%d  %d",&a,&b);`

Вводимые значения: 1) `24 28` 2) `24↵`

`28`

в) `scanf("%d,%d",&a,&b);`

Вводимые значения: `24,28`

г) `scanf("%s",name);`

Вводимые значения: `Иванов  Иван`

Результат ввода: `Иванов`

Последний пример демонстрирует, что по формату s строка вводится только до пробела. Чтобы ввести всю строку используют специальную функцию ввода строк или повторяют ввод в другие переменные из той же строки.

*Примечание*. Начиная с версии Visual C++ 2008 разработчики языка рекомендуют вместо стандартной функции `scanf` использовать функцию `scanf_s`, которая контролирует длину вводимой пользователем строки, позволяя исключить ошибку переполнения буфера. При вызове функции `scanf_s` максимально возможную длину вводимой строки указывают после соответствующего формата.

**Пример 1.2.** Пример работы функции `scanf_s()`.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int i,result;
   float fp;
   char  c,s[81];
   result = scanf_s( "%d %f %c %s", &i, &fp, &c, 1, s, 80 );
   printf( "The number of fields input is %d\n", result );
   printf( "The contents are: %d %f %c %s \n", i, fp, c, s);
   _getch();
}
```

Результат работы программы (полужирно выделены вводимые значения):

```
45 67.8
fghjk
The number of fields input is 4
The contents are: 45 67.800003 f ghjk
```

### 1.6.2 Ввод/вывод строк

Ввод строк с клавиатуры:

`char* gets`(<Строковая переменная>); // возвращает копию строки или NULL

Вывод строк на экран с переходом на следующую строку:

`int puts` (<Строковая константа или переменная>);

Примеры:

а) `puts("Это строка");`

Результат: `Это строка↵`

б) `gets(st);`

Вводимые значения: `Иванов Иван↵`

Результат: `Иванов Иван`

### 1.6.3 Ввод/вывод символов

Ввод символов с клавиатуры:

`int getchar();` // возвращает символ или EOF

Вывод символов на экран:

`int putchar`(<Символьная переменная или константа>);

Например:

```
ch=getchar();
putchar('t');
```

**Пример 1.3.** Программа определения корней квадратного уравнения $Ax^2+Bx+C=0$ при условии, что дискриминант неотрицателен.

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

// основная программа

```
int main(int argc, char* argv[])
{
    setlocale(0,"russian");
    float A,B,C,E,D,X1,X2;
    puts("Введите A,B,C:");
    scanf_s("%f  %f  %f",&A,&B,&C);
    printf("A=%5.2f  B=%5.2f   C=%5.2f\n",A,B,C);
    E=2*A;
    D=sqrt(B*B-4*A*C);
    X1=(-B+D)/E;
    X2=(-B-D)/E;
    printf("X1= %7.3f   X2=%7.3f\n",X1,X2);
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

В результате получим следующее (вводимая информация выделена полужирно):

```
Введите A,B,C:
1 -5 6
A= 1.00  B=-5.00   C= 6.00
X1=   3.000   X2=  2.000
Нажмите любую клавишу для завершения...
```

**Контрольные вопросы к главе 1**

1. Что такое тип данных и каково его назначение?

Ответ.

2. Дайте определение литерала. Приведите примеры литералов?

Ответ.

3. Что такое переменная и как она определяется в C++?

Ответ.

4. Какие операции над данными определены в C++?

Ответ.

5. Какие логические поразрядные операции реализованы в C++ и особенности их выполнения?

Ответ.

Оглавление

6. Какие операции присваивания Вы знаете? В чем особенности их выполнения?

Ответ.

7. Что такое выражение? Дайте определение простого выражения.

Ответ.

8. Что такое приоритет и ассоциативность операций в C++?

Ответ.

9. Какие стандартные функции можно использовать в C++ и какую библиотеку для этого нужно подключить?

Ответ.

10. Что такое форматная строка и как она используется при вводе и выводе данных?

Ответ.

# 2 УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ ЯЗЫКА

*Управляющими* называют операторы, способные изменять линейность процесса вычислений. К таким операторам относятся оператор условной передачи управления `if`, оператор выбора `switch`, операторы циклов `while`, `do`, `for` и операторы безусловной передачи управления `goto`, `break`, `continue`, `exit`.

## *2.1 Блок операторов и пустой оператор*

При объявлении управляющих операторов широко используют конструкции «блок операторов» и «пустой оператор».

*Блоком операторов* называют последовательность операторов, заключенную в фигурные скобки:

**{<Оператор>;… <Оператор>;}**

Например:

```
{
    f=a+b;
    a+=10;
}
```

Особенность блока операторов заключается в том, что при анализе синтаксиса он рассматривается как единый оператор. Это позволяет использовать его там, где по правилам записи конструкций возможен только один оператор, а по алгоритму программы должны выполняться несколько действий.

*Пустой оператор* включает только символ «;», перед которым нет никакого выражения или не завершенного разделителем оператора. Пустой оператор не предусматривает выполнения никаких действий. Он используется там, где синтаксис языка требует присутствия оператора, а по алгоритму программы никакие действия не должны выполняться.

## *2.2 Оператор условной передачи управления*

Оператором условной передачи управления называют конструкцию, позволяющую выбрать одну из возможных альтернатив вычислительного процесса в зависимости от результата условия. Он реализует алгоритмическую конструкцию Ветвление (см. рисунок 2.1).

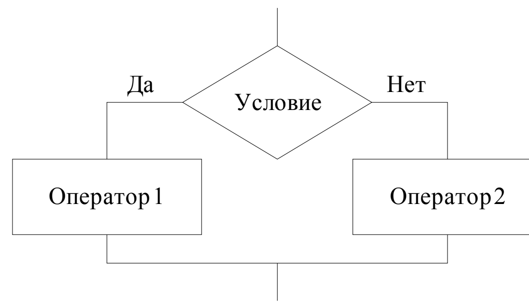



**Рисунок 2.1** – Конструкция Ветвление

Синтаксис оператора:

**`if` (<Выражение>) <Оператор;> [ else <Оператор;>]**

Квадратные скобки показывают, что описание ветви `else` – необязательно.

Выражение – любое выражение, записанное по правилам Си или С++. Если результат выражения не равен нулю, то выполняется оператор, следующий за выражением. Если результат выражения равен нулю, то выполняется оператор альтернативной ветви. При отсутствии описания альтернативной ветви управление передается следующему за `if` оператору.

В качестве оператора может записываться любой оператор С++, в том числе другой оператор условной передачи управления, пустой оператор и блок операторов.

Примеры:

а)
```
if (!b)puts("c - не определено");     // если b=0, то – ошибка,
 else {c=a/b; printf("c=%d\n", c);}   // иначе выводится значение c.
```

б)
```
if ((c=a+b)!=5) c+=b;
 else c=a;
```

в)
```
if ((ch=getchar())=='q')             //  если в ch введено q,
 puts ("Программа завершена."); //  то ...
 else puts ("Продолжаем работу..."); //  иначе ...
```

**Правило вложенности.** При записи двух вложенных операторов `if` с одной альтернативной ветвью возникает неоднозначность: не понятно, которому из двух операторов `if` принадлежит альтернативная ветвь (см. рисунок 2.2).

```
if (<Условие 1> )
 if <Условие 2> <Действие 1>
  else <Действие 2>
```

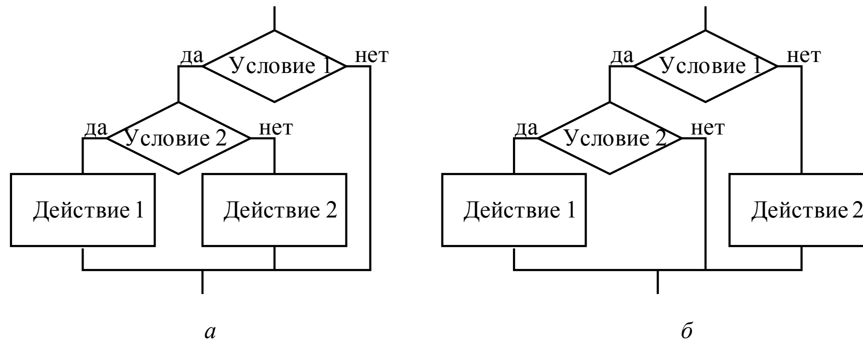



**Рисунок 2.2** – Варианты вложения конструкций `if`

Для разрешения коллизии используют правило вложения: *else всегда относится к ближайшему if*. Тогда приведенный выше фрагмент реализует вариант *а*. Для реализации варианта *б* используют блок операторов:

```
if <Условие 1>
   {
      if  <Условие 2> <Действие 1>
   }
else <Действие 2>
```

**Пример 2.1.** Написать программу решения системы уравнений:

$$\begin{cases} ax=b \\ x+cy=1 \end{cases}$$

Схема алгоритма решения системы уравнений приведена на рисунке 2.3.

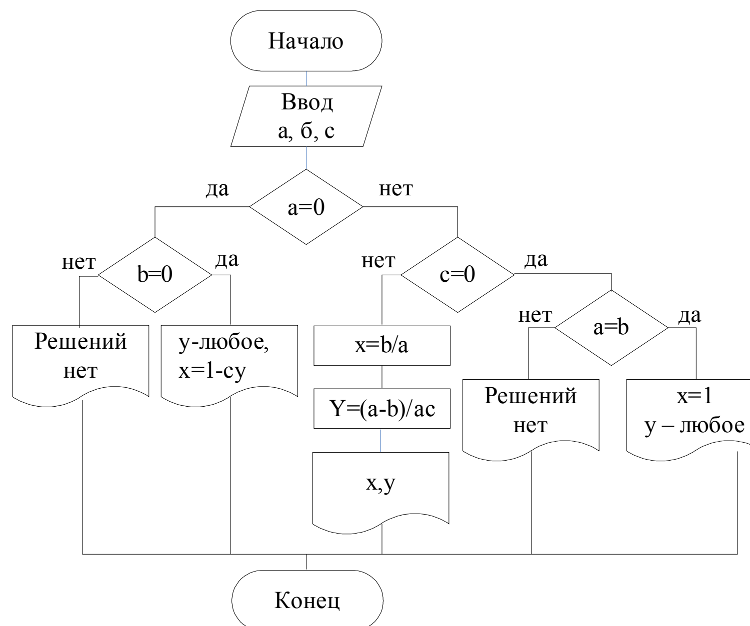



**Рисунок 2.3** – Схема алгоритма решения системы уравнений

Текст программы реализует приведенную схему алгоритма.

Оглавление

```
#include <locale>
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float y,x,a,b,c;
     puts("Введите a,b,c:");
     scanf("%f %f %f",&a,&b,&c);
     printf("a=%5.2f   b=%5.2f  c=%5.2f\n",a,b,c);
     if (a==0)
        if (b==0) puts("Решение не существует.");
        else printf("y - любое,  x=1-c*y");
     else
        if (c==0)
               if (a=b) puts("Решение не существует.");
               else puts("x=1, y - любое");
     else
     {
          x=b/a;     y=(a-b)/a/c;
          printf("x= %7.3f   y=%7.3f\n",x,y);
     }
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

Результат выполнения программы (полужирно выделены вводимые числа):

```
Введите a,b,c:
3 2 5
a= 3.00   b= 2.00  c= 5.00
x=   0.667   y=  0.067
Нажмите любую клавишу для завершения...
```

### *2.3 Оператор выбора*

Если количество альтернатив велико, а параметр выбора единственный, то можно использовать оператор выбора:

**switch (<Выражение>)**

**{**

**case <Элемент>: [<Операторы;>]**

**case <Элемент>: [<Операторы;>]**

**...**

**[default: <Операторы;>]**

**}**

где <Выражение> – параметр выбора альтернативы, результат выражения должен быть целого типа или его значение приводится к целому;

<Элемент> – целая константа, определяющая описанную за ней альтернативу.

<Операторы> – ноль, один или более операторов, выполняемые, если выполнилось равенство <Выражение>=<Элемент>.

После группы операторов данной альтернативы *выполняются операторы всех последующих альтернатив без проверки их параметров*, пока не встретится оператор `break`. Например:

```
switch (n_day)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5: puts("Go work!"); break;
    case 6: printf("%s","");
    case 7: puts("relax!");
}
```

При выполнении данного фрагмента, если переменная `n_day` содержит значение от 1 до 5, то выводится сообщение «`Go work!`», если `n_day=6`, то выводится «`Clean the yard and relax!`» и, наконец, если `n_day=7`, то выводится «`relax!`».

**Пример 2.2.** Разработать программу, вычисляющую значения нескольких функций. Функция выбирается пользователем из предлагаемого списка по соответствующему коду, вводимому в ответ на запрос пользователем:

Оглавление

```
Введите код:
1 - y=sin x
2 - y=cos x
3 - y=exp x
```

Схема алгоритма программы приведена на рисунке 2.4.

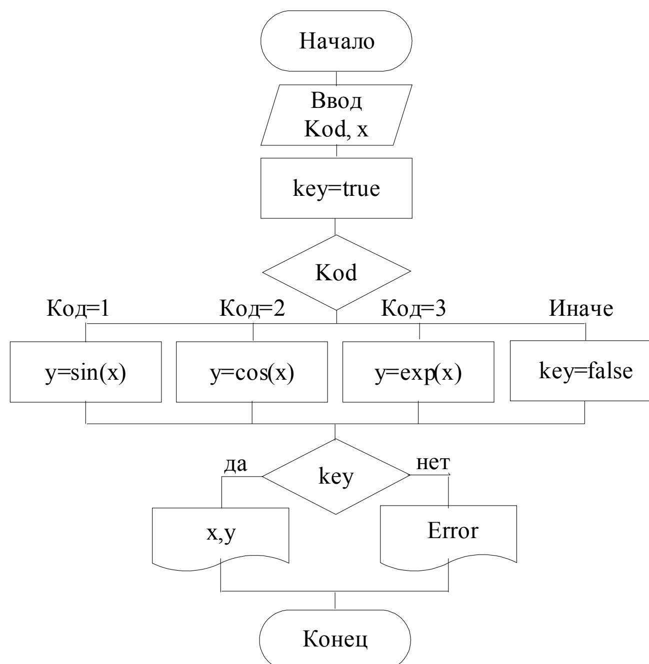



**Рисунок 2.4** – Схема алгоритма программы

Текст программы:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int kod,key;   float x,y;
     puts("Введите x:");      scanf_s("%f",&x);
     printf("x=%6.3f\n",x);
     puts("Введите код:");
     puts("1 - y=sin(x)");
     puts("2 - y=cos(x)");
     puts("3 - y=exp(x)");
     scanf_s("%d",&kod);
```

```
    key=1;
    switch(kod)
    {
        case 1: y=sin(x); break;
        case 2: y=cos(x); break;
        case 3: y=exp(x); break;
        default: key=0;
    }
    if (key) printf("x= %5.2f    y=%8.6f\n",x,y);
    else puts("Ошибка!");
    puts("Нажмите любую клавишу для завершения...");
   _getch();
    return 0;
}
```

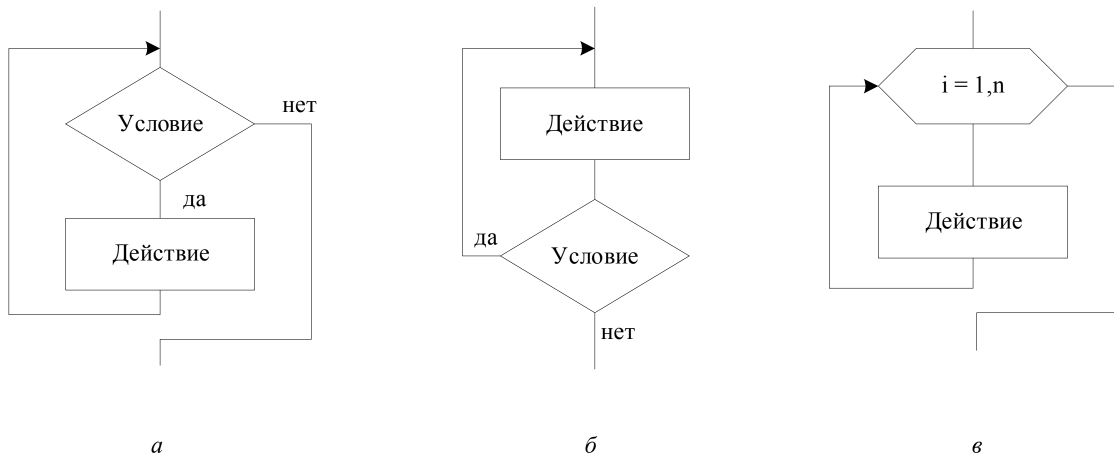
### *2.4 Операторы организации циклических процессов*

Операторы цикла задают многократное повторение некоторой последовательности действий. В программировании выделяют три разновидности циклов, отличающихся способом определения количества повторений: итерационные циклы, циклы с заданным числом повторений и поисковые циклы (рисунок 2.5).

**Рисунок 2.5 –** Виды циклических процессов и реализующие их операторы

*Итерационными* называются циклы, в которых количество повторений заранее неизвестно. Оно определяется некоторой целевой функцией. В частности, это может быть точность вычислений. Такие циклы, например, реализуются при решении задачи нахождения суммы бесконечного ряда, определения значения интеграла на отрезке, длины кривой и т.д. Для организации итерационного цикла в C++ существуют два оператора: реализую-

щие итерационные циклы: цикл с предусловием `while` (рисунок 2.6, *а*) и цикл с постусловием `do … while` (рисунок 2.6, *б*).




*а* *б* *в*

**Рисунок 2.6** – Операторы циклов в С++

*Счетными* называются циклы, в которых количество повторений заранее известно или легко определяется. Примерами подобных циклов являются вычисление степени числа, факториала числа, суммы определенного числа членов некоторой последовательности и т.д. В С++ для реализации счетного цикла используется оператор `for` (рисунок 2.6, *в*).

*Поисковыми* называют циклы, в которых осуществляется поиск некоторого единственного значения среди других. При этом сразу после нахождения этого значения необходимо выйти из цикла, а если значение не найдено по завершению проверки всех элементов – выдать соответствующее сообщение. Поскольку специальной конструкции для реализации поискового цикла в С++ не существует, его обычно реализуют как итерационный цикл с конъюнкцией условий: «Все проверено?» && «Нашли?».

### 2.4.1 Цикл с предусловием (Цикл-пока)

Оператор Цикл-пока имеет следующий формат:

**`while` (<Выражение>) <Оператор>;**

где <Выражение> – выражение возможно сложное, состоящее из нескольких простых, определяет условие выполнения цикла;

<Оператор> – любой оператор С++, в том числе блок операторов.

Цикл выполняется до тех пор, пока результат выражения отличен от нуля, т.е. «Истина». Если условие нарушено сразу при входе в цикл, то тело цикла не выполнится ни разу.

**Пример 2.3.** Вычислить сумму ряда при $x > 1$ с заданной точностью ε:

$$S = 1 - 1/x + 1/x^2 - 1/x^3 + \dots$$

Рекуррентная формула определения очередного n+1-го члена ряда: $R_{n+1} = - R_n / x$ .

На каждом шаге итерации сумма ряда определяется по формуле: $S = S + R_n$ .

Условие выхода из цикла: $| R_n | < \varepsilon$ .

Схема алгоритма приведена на рисунке 2.7.

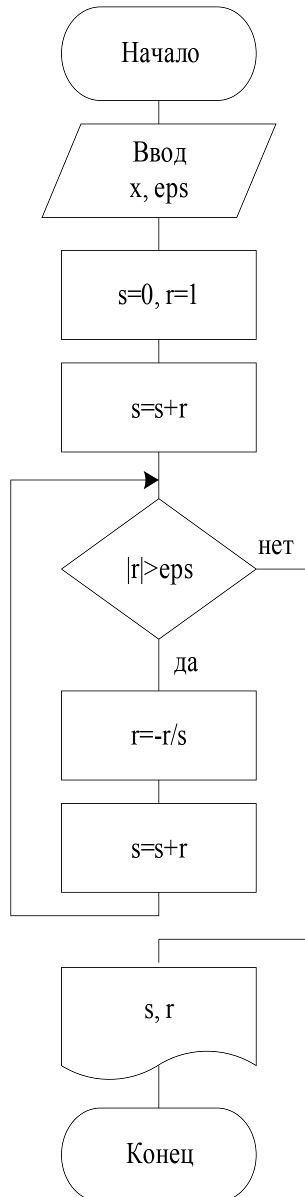



**Рисунок 2.7** – Схема алгоритма программы

Текст программы:

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

```
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float s, r,x,eps;
     puts("Введите x, eps:"); scanf_s("%f  %f", &x, &eps);
     s=0; r=1; s+=r;
     while (fabs(r)>eps)
     {
          r=-r/x;   s+=r;
     }
     printf("Результат= %10.7f  r=%10.8f\n", s,r);
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

При выполнении программа выдает следующий результат:

```
Введите x, eps:
3 0.0001
Результат=  0.7499873  r=-0.00005081
Нажмите любую клавишу для завершения...
```

### 2.4.2 Цикл с постусловием (Цикл-до)

Оператор цикла с постусловием отличается от оператора цикла с предусловием тем, что условие проверяется после выполнения оператора тела цикла. Цикл выполняется до тех пор, пока результат выражения «Истина», т.е. отличен от нуля. Поскольку проверка происходит после выполнения тела цикла, даже если условие сразу нарушено, тело цикла выполняется один раз. Формат оператора:

**do <Оператор > while (<Выражение>) ;** .

Цикл с постусловием легко реализовать через цикл с предусловием (см. рисунок 2.8).

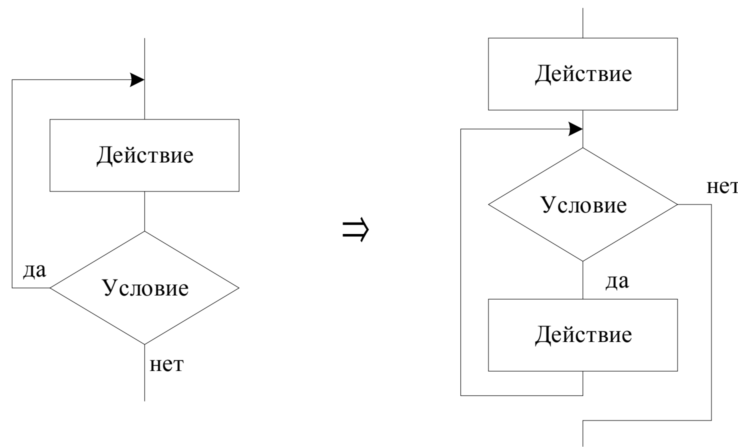



**Рисунок 2.8 –** Организация цикла с пост условием через цикл с предусловием

**Пример**. Разработать фрагмент программы, которая вводит только значения, входящие в заданный интервал.

```
do {
   printf("Введите значение от %d до %d:",low,high);
   scanf_s("%d",&a);
}
while (a<low || a>high);
```

### 2.4.3 Оператор счетного цикла for

Оператор реализует цикл, для которого известен диапазон и шаг параметра цикла:

**for ([<Выражение1>];[<Выражение2>];[<Выражение3>])[<Оператор>;]**

где <Выражение1> – инициализирующее выражение – выполняется один раз перед циклом и задает начальные значения переменным цикла. Может отсутствовать, но при этом точка с запятой остаются.

<Выражение2> – выражение условия – определяет предельное значение параметра цикла. Может отсутствовать, при этом точка с запятой остается.

<Выражение3> – выражение модификации – выполняется на каждой итерации после тела цикла, но до следующей проверки условия и обычно используется для изменения параметра(ов) цикла. Может отсутствовать;

<Оператор> – тело цикла – может быть любым оператором C++, блоком операторов или может отсутствовать.

Нетрудно заметить, что счетный цикл легко реализуется через цикл с предусловием (рисунок 2.9).

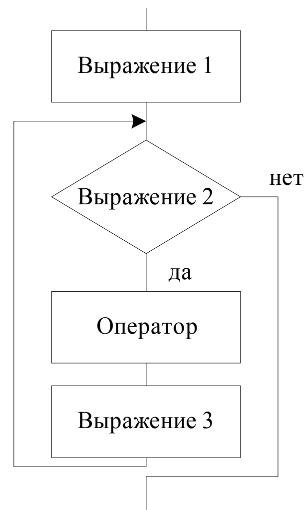



**Рисунок 2.9 –** Схема алгоритма оператора счетного цикла С++

Примеры:

а) `for(int i=0,float s=0;i<n;i++)s+=i;`

б) `for(int i=0,float s=0;i<n;i++,s+=i);` // отсутствует тело цикла

в) `int i=0;float s=0;`

`for(;i<n;s+=i++);`//отсутствует инициализирующее выражение и тело цикла

г) `for(;i<n;)s+=i++;`// отсутствуют инициализирующее и модифицирующие //выражения

д) `for(;;);` // бесконечный цикл, который ничего не делает.

**Пример 2.4.** Найти сумму N натуральных чисел.

На рисунке 2.10 приведена схема алгоритма программы.

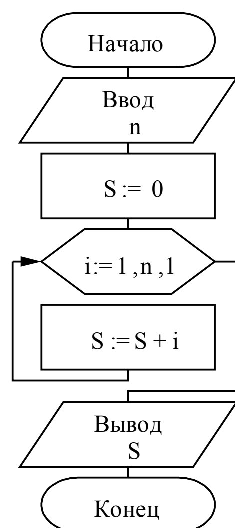



**Рисунок 2.10 –** Схема алгоритма программы

Текст программы:

Оглавление

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int i,n,s;
     puts("Введите количество членов последовательности:");
     scanf_s("%d",&n);
     for (i=1,s=0;i<=n;i++) s+=i;
     printf("Сумма=%5d  при n=%3d\n",s,n);
     puts("Нажмите любую клавишу для завершения...");
    _getch();
     return 0;
}
```

### *2.5 Неструктурные операторы передачи управления*

Помимо обычного набора операторов передачи управления C++ включает три оператора передачи управления, с точки зрения технологии структурного программирования нарушающие структурность программы. Использование этих операторов усложняет «чтение» программы и, возможно, свидетельствует о ее непродуманной структуре.

**Оператор goto.** Оператор безусловно передает управление в помеченную идентификатором точку:

**`goto` <Идентификатор>;**

Точка, в которую передается управление, отмечается меткой вида:

**<Идентификатор>:**

Например:

`again: ...` // метка

`goto again;` // оператор безусловной передачи управления

**Оператор break.** Оператор используется в операторах циклов и выбора: `for`, `while`, do...`while`, `switch` для выхода из конструкции до ее завершения. Он записывается в виде:

```
break;
```

**Пример 2.5**. Программа суммирования до 10 чисел вводимой последовательности. Завершение ввода и суммирования при получении отрицательного числа.

Оглавление

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main()
{
    setlocale(0,"russian");
    int s=0,i,k;
    puts("Введите последовательность до 10 чисел.");
    for(i=0;i<10;i++)
    {
        scanf_s("%d",&k);
        if (k<0) break; // по отрицательному числу - выход из цикла
        s+=k;
    }
    printf("Сумма чисел = %d\n",s);
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

**Оператор continue.** Оператор используется для пропуска нескольких операторов до конца тела цикла и возврата в цикл. Он записывается в виде:

```
continue;
```

**Пример 2.6.** Программа суммирования 10 вводимых целых положительных чисел. При вводе отрицательного числа выводится предупреждающее сообщение.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main()
{
    setlocale(0,"russian");
    int s=0,i=0,k;
    puts("Введите последовательность до 10 чисел.");
    while(i<10)
    {
        scanf_s("%d",&k);
```

```
        if (k<0)   // при вводе отрицательного числа
        {
            puts("Будьте внимательны.");
            continue;  // значение счетчика i и сумма не меняются
        }
        i++; s+=k;
    }
    printf("Сумма чисел = %d\n",s);
    puts("Нажмите любую клавишу для завершения...");
    _getch();   return 0;
}
```

**Контрольные вопросы к главе 2**

1. Какие операторы C++ называют управляющими?

Ответ.

2. Что такое блок операторов и его особенности?

Ответ.

3. В чем заключается правило вложенности при использовании оператора ветвления?

Ответ.

4. Каковы особенности выполнения оператора выбора switch?

Ответ.

5. Дайте определение итерационного цикла.

Ответ.

6. Какие циклические процессы называются счетными и почему?

Ответ.

7. В чем заключается особенность поискового цикла?

Ответ.

8. Назовите операторы реализации циклов в C++?

Ответ.

9. Чем оператор цикла с постусловием отличается от оператора цикла с предусловием?

Ответ.

10. Перечислите основные неструктурные операторы передачи управления. Их назначение и особенности применения.

Ответ.

# 3 СЛОЖНЫЕ СТРУКТУРЫ ДАННЫХ. АДРЕСНАЯ АРИФМЕТИКА

В C++ не существует развитой системы структур данных, как, например, в Паскале. Единственным действительно структурным типом данных являются структуры. Остальные сложные данные моделируются с использованием средств адресной арифметики, в основе которой лежит понятие адреса и операций над ним.

## *3.1 Указатели и ссылки*

Для хранения адресов в C++ используются специальные типы данных – указатели и ссылки. Существует несколько причин, по которым практически невозможно написать хорошую программу на языке C++ без использования указателей или ссылок, например:

а) использование указателей или ссылок позволяют функциям возвращать измененные значения в вызывающую программу;

б) с помощью указателей осуществляется доступ к динамически распределенной памяти;

в) применение указателей повышает эффективность функций обработки строк.

### 3.1.1 Определение указателя. Типизированные и нетипизированные указатели и операции над ними

*Указатель* – это переменная, в которой хранится адрес некоторого объекта программы: другой переменной, поименованной константы, подпрограммы и т.п. Формат объявления указателя выглядит так:

**[<И1>][<Тип данных>][<Тип указателя>] [И2]*<Имя>[=<Значение>];**

где <И1> – возможность изменения содержимого по адресу, записанному в указатель: если ничего не указано, то содержимое можно менять, если указано const, то нельзя;

<И2> – возможность изменения адреса, записанного в указатель: если ничего не указано, то адрес можно менять, если указано const, то нельзя;

<Тип данных> – тип данных, адрес которых хранит указатель, может быть любой тип, определенный в C++, в том числе void;

<Тип указателя> – определяется используемой моделью памяти, может быть `far` (дальний) или `near` (ближний). В настоящее время данный описатель не упо-

требляют, по умолчанию принимается `near`, что соответствует модели памяти большинства приложений Windows, которая называется `flat`.

Примеры объявления указателей:

1) `short a,*ptrs=&a;` // указатель ptrs инициализирован адресом переменной a;
2) `const short *ptrs;` // неизменяемое значение,
   // можно изменить адрес, например `ptrs=&b,`
   // но нельзя изменить содержимое `*ptrs=10;`
3) `short *const ptrs=&a;` // неизменяемый указатель,
   // можно менять содержимое `*ptrs=10,`
   // но нельзя изменить адрес `ptrs = &b.`

Различают указатели:

- типизированные – адресующие данные конкретного типа, например:
  ```
  int *b,*c ;          // указатели, хранящие адреса целых чисел
  long double *l;  // указатели, хранящие адреса вещественных чисел
  ```
- нетипизированные – не связанные с данными определенного типа, например:
  ```
  void * p;            // просто адрес области памяти
  ```

Типизированные указатели несут в себе сведения о размере памяти, адресуемой этим указателем. Нетипизированные – создаются как бы «на все случаи жизни». Они отличают-ся от типизированных указателей отсутствием сведений о размере соответствующего участка памяти, поэтому в них легко копировать адреса из указателей других типов.

В C++ определена адресная константа `NULL`. Эта константа означает адрес, который никуда не указывает или «нулевой адрес». Такой адрес используется как маркер «Пустой адрес», когда надо отметить, что адрес в указатель не записан и может быть присвоен ука-зателю любого типа, например:

```
int *pi=NULL;
void *b=NULL;
```

**Операции над указателями.** Над указателями, как над переменной любого типа, определены операции, которые можно над ними выполнять.

***1. Присваивание.*** Допускается присваивать указателю значение другого указателя того же типа или нулевого указателя, например:

```
int *p1,*p2;
float *p3,*p4;
void *p;...
```

// допустимые операции

```
 p1=p2; p4=p3; p1=NULL; p=NULL;    ...
```

// недопустимые операции

```
 p3=p2; p1=p3; …
```

Однако при необходимости выполнить запрещенную операцию присваивания можно использовать явное переопределение типа для приведения указателя одного типа к другому, например:

```
p3=(float*)p2;   p1=(int*)p3; .
```

**2. *Получение адреса (&).*** Результат операции – адрес некоторой области памяти, который можно присвоить указателю. Это можно сделать:

а) при помощи операции присваивания:

```
  int *pi,i=10;
    ...  pi=&i;
```

б) во время инициализации указателя при его объявлении:

```
  float b=5.7,*pf=&b;  .
```

***3. Доступ к данным по указателю (операция разыменования).*** Тип значений, получаемых при разыменовании указателя, совпадает с базовым типом данных указателя.

Например:

```
short c, a=5,*ptri=&a;
float d,p=2.4563;
void *b=&a;
…
c=*ptri;      // в результате c=5
*ptri=125;    // вместо 5 в память переменной a теперь записано число 125
```

*Нетипизированные указатели разыменовывать нельзя.* При необходимости разыменовать нетипизированный указатель требуется явно указать тип данных, например:

```
*b=6; ⇒ *(int*)b=6;
b=&p;
d=*b  => d=*(float *)b
```

***4. Операции отношения.*** Из шести операций отношений над указателями определены только две: проверка равенства (==) и неравенства (!=).

Например:

```
int sign = (p1 == p2);
if (p1!=NULL) {….}
```

**5.** ***Арифметические операции.*** В С++ над указателями разрешены следующие арифметические операции: сложение (+), вычитание (-), инкремент (++) и декремент (--).

Совокупность указанных операций и правила выполнения этих операций над указателями получили название *адресной арифметики* (см. раздел 3.2).

При необходимости указатель может содержать адрес другого указателя, который, в свою очередь, содержит адрес обычной переменной. Такая схема называется *косвенной адресацией*. Глубина косвенной адресации не ограничена, однако, почти всегда, можно ограничиться «указателем на указатель». Более громоздкие схемы могут привести к ошибкам, которые трудно обнаружить.

Переменная, представляющая собой указатель на указатель, объявляется следующим образом:

**<Имя типа> ** <Имя переменной>**

где <Имя типа> – любой допустимый тип С++, например:

```
float **ptr1;
int **ptr2;
```

При работе с косвенной адресацией указатели разыменовывают столько же раз, сколько раз выполнялась операция взятия адреса:

```
float a,*ptr1=&a,**ptr2;
puts("Input a");
scanf("%f",&a);
ptr2=&ptr1;
printf("%p    %3d\n",*ptr2,**ptr2);
```

### 3.1.2 Понятие ссылки

Кроме указателя в С++ для хранения адреса может использоваться ссылка. Ссылка определена как *альтернативное имя* уже существующего объекта. Основные достоинства ссылок проявляются при работе с возвращаемыми параметрами функций.

Правила описания ссылок:

**<Тип данных> &<Имя>[= <Выражение>]** или

**<Тип данных> &<Имя>[(<Выражение>)]**

Обе формы допустимы, например:

```
int L=127;
int &SL=L;        // первая форма
int &SL(L);       // вторая форма
```

В качестве выражения для инициализации ссылки обычно задают имя некоторого объекта, размещенного в памяти. Значением ссылки после инициализации становится *адрес* этого объекта, так в примере, приведенном выше, значением ссылки SL является адрес переменной L.

### 3.1.3 Отличие ссылки от указателя

Отличия ссылки от указателя следуют из их деклараций. Так указатель определяется как адрес и при работе с адресуемыми им данными необходимо использовать операцию разыменования. Ссылка объявляется как альтернативное имя, поэтому при работе с данными по ссылке разыменование не нужно:

```
int a,
     *ptri=&a, // указатель – адрес переменной a
     &b=a;     // ссылка – альтернативное имя переменной a
 …
a=3;  тоже самое, что и  *ptri=3;  тоже самое, что и  b=3;  (см. рисунок 3.1).
```

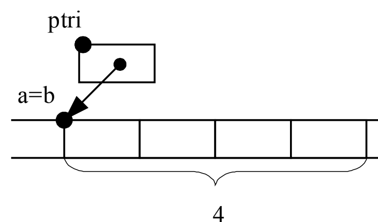



**Рисунок 3.1** – Переменная `a`, указатель `ptri` и ссылка `b`

## *3.2 Адресная арифметика*

Основное правило адресной арифметики определяется следующим образом: при увеличении или уменьшении адреса, хранящегося в указателе, на количество единиц n значение адреса изменяется на n, умноженное на размер элемента данных, единиц:

**<Указатель> + n ⇔ <Адрес> + n*sizeof(<Тип данных>)**

Например:

```
short a, *ptrs =&a;
```

1) `ptrs++;` // адрес в указателе меняется на длину элемента базового типа (см. рисунок 3.2);

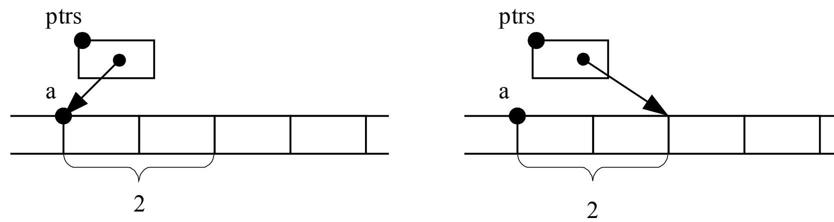



**Рисунок 3.2** – Увеличение указателя на единицу

2) `ptrs+=4;` // адрес в указателе меняется на 4 размера элемента базового типа (см. рисунок 3.3).

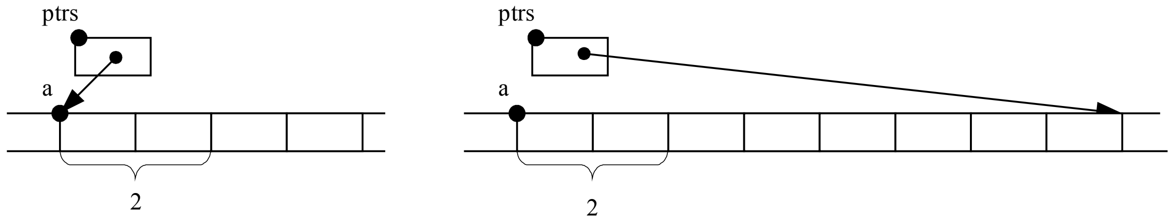



**Рисунок 3.3** – Увеличение указателя на четыре единицы

3) `*(ptrs+2)=2;` // *адрес в указателе не меняется* (см. рисунок 3.4)

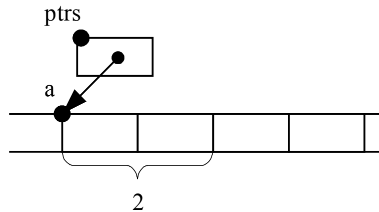



**Рисунок 3.4** – Обращение к заштрихованному полю без изменения указателя

**Особенности результатов выполнения операций адресной арифметики, связанные с реализацией языка C++.** В C++ принят обратный порядок размещения объектов в памяти. Это объясняется особенностями работы компилятора. При разборе текста, компилятор распознает и размещает в стек имена всех объектов, которые необходимо разместить в памяти. На этапе распределения памяти имена объектов выбираются из стека и им отводятся смежные участки памяти. А так как порядок записи в стек обратен порядку чтения, размещение объектов оказывается обратным, например:

```
int i1=10,i2=20,i3=30;
```

Первой будет в памяти размещена переменная `i3`, она будет иметь меньший адрес. Адрес переменной `i1` – самый большой, тогда, если указатель p определен как

```
int *p=&i2;
```

то `*(p+1)` ⇔ `i1`, а `*(p-1)` ⇔ `i3`.

### *3.3 Управление динамической памятью*

С++ включает набора средств, позволяющих организовать работу с динамически выделяемой памятью: унаследованный от Си и реализованный в С++. В программе не следует смешивать использование средств разных языков.

**Работа с динамической памятью с помощью средств Си.** Процедуры работы с динамической памятью, унаследованные от Си, размещены в библиотеке `alloc`, соответственно их прототипы находятся в файле `alloc.h`.

1. Процедура выделения памяти под одну переменную возвращает адрес начала области выделенной памяти или NULL, если память не выделена. Поскольку адрес не связан с данными конкретного типа, при записи значения в типизированный указатель необходимо явное преобразование типа.

Прототип процедуры:

```
void * malloc(size_t size);
```

где size – параметр, определяющий размер выделяемой области.

2. Процедура выделения памяти под несколько переменных возвращает адрес области выделенной памяти или NULL, если память не выделена. Память выделяется одним куском под размещение заданного количества элементов, поэтому для доступа к отдельным переменным можно использовать адресную арифметику.

Прототип процедуры:

```
void * calloc(size_t n, size_t size);
```

где size – параметр, определяющий размер выделяемой области;

n – количество областей.

Следует помнить, что ***любая память, динамически выделенная под размещение переменных, должна быть освобождена.***

3. Процедура освобождения памяти:

```
void free(void *block);
```

где block – параметр, определяющий адрес освобождаемого блока памяти.

Примеры:

а) выделение памяти под одно значение с проверкой результата выполнения операции:

```
int *a;
if ((a=(int *)malloc(sizeof(int)))==NULL)
{
     printf("Память для числа не выделена.");
```

```
    exit(1);
}
*a=-244;
*a+=10;
free(a);
```

б) выделение памяти под размещение нескольких значений:

```
int *list;
list = (int *) calloc(3,sizeof(int));
*list=-244;
*(list+1)=15;
*(list+2)=-45;
  …
free(list);
```

**Управление динамической памятью средствами С++.** Для работы с динамической памятью в С++ используют специальные операторы `new` и `delete`. Форматы записи этих операторов при работе с одним значением и несколькими значениями различны.

1. Операция выделения памяти под одно значение:

**<Типизированный указатель> =`new` <Тип>[(<Значение>)];**

где <Тип> – тип значения, под размещение которого выделяется память – определяет размер памяти, выделяемый данным оператором;

<Значение> – инициализирующее значение, которое, как показывают квадратные скобки, может быть опущено.

2. Операция освобождения памяти, выделенной под одно значение:

**`delete` <Типизированный указатель>;**

Примеры:

а) выделение памяти без проверки правильности завершения операции:

```
int *k;
k = new int;
*k = 85;
delete k;
```

б) с проверкой правильности выделения памяти:

```
int *a;
if ((a=new int(-244))==NULL)
{
    printf("Память для числа не выделена.");
```

```
        exit(1);    // завершение программы с признаком ошибки
    }
    delete a;
```

3. Операция выделения памяти под несколько значений:

**Указатель> = new <Тип>[<Количество>];**

где <Тип> – тип значения, под размещение которого выделяется память – определяет размер памяти, выделяемый для одного значения;

<Количество> – количество значений, которое необходимо разместить в памяти.

4. Операция освобождения памяти, выделенной под несколько значений. Квадратные скобки присутствуют в операторе и указывают, что освобождается несколько значений, память под которые отводилась одним оператором `new`:

**delete[] <Типизированный указатель>;**

Пример выделения памяти под переменные одним куском (см. рисунок 3.5):

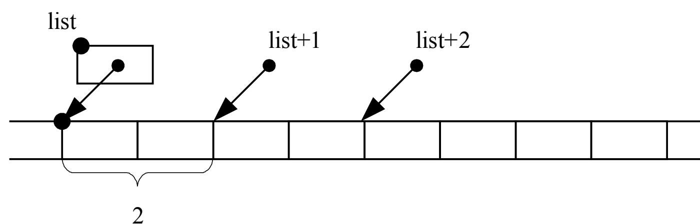



**Рисунок 3.5** – Использование адресной арифметики для обращения к переменным

```
short *list;
list = new int [3];
*list=-244;
*(list+1)=15;
*(list+2)=-45;
delete[] list;
```

### *3.4 Массивы*

Массив С++ – абстракция, используемая при работе с последовательно расположенными в памяти значениями одного типа. Доступ к этим значениям может осуществляться либо с применением возможностей адресной арифметики (как показано в предыдущем разделе), либо с указанием номера элемента (как обычно принято при работе с массивами).

При размещении в памяти младший адрес соответствует первому элементу массива, а старший – последнему. Однако *индексы элементов массива всегда начинаются с 0*.

Массивы могут быть одномерными, двумерными, трехмерными и многомерными. Для доступа к элементам многомерных массивов необходимо столько индексов, какова размерность массива. Двумерные и многомерные массивы расположены в памяти «построчно», т.е. правые индексы меняется быстрее, чем расположенные левее. Так элементы матрицы A(3,4), где 3 – количество строк, а 4 – количество столбцов, расположены в памяти в следующей последовательности: $a_{00}$, $a_{01}$,$a_{02}$,$a_{03}$, $a_{10}$, $a_{11}$,$a_{12}$,$a_{13}$, $a_{20}$, $a_{21}$,$a_{22}$,$a_{23}$.

Массив можно создать двумя способами:

- запросить память для размещения его элементов посредством оператора `new` (см. раздел 3.3);
- использовать конструкцию:

**<Тип элемента> <Имя>[<Размер1>] [<Размер2>] ...[= {<Список значений >}];**

где <Тип элемента> – скалярный или сложный (в том числе массив) тип последовательно располагаемых элементов;

<Имя> – имя массива – указатель, содержащий адрес первого элемента массива;

<Размер 1>, <Размер 2>, … <Размер n> – размерности массива по каждому из n измерений;

<Список значений> – перечень значений, инициализирующих выделяемое место в памяти.

Количество размерностей определяет мерность массива, если задан один размер, то массив – одномерный, если два, то – двумерный или матрица, если три, то – трехмерный, если больше, то – многомерный.

Массив в памяти не должен занимать более 2 Гб.

### 3.4.1 Одномерные массивы

По правилам Си и С++ одномерный массив можно объявить:

- статически – с использованием абстракции массив и указанием его размера, например:

```
int a[10]; // массив на 10 целых чисел, индекс меняется от 0 до 9
unsigned int koord[10]; // массив целых беззнаковых чисел
```

- динамически – объявив только указатель на будущий массив и выделив память под массив во время выполнения программы, например:

```
int *dinmas;      // объявление указателя на целое число
dinmas=new int [100]; // выделение памяти под массив на 100 элементов
```

```
...
delete [] dibmas;
```

**Инициализация одномерного массива при объявлении.** Массивы, объявляемые вне функций или с описателями `static` и `extern` (см. раздел 4.1.1), можно инициализировать. Инициализируемый массив можно объявлять без указания его размерности. Если массив объявлен с указанием количества элементов, то при инициализации должны быть заданы значения для всех элементов массива, например:

```
extern int a[5]={0,-36,78,3789,50};
```

Если массив объявлен без указания количества элементов, то количество элементов массива определяется количеством заданных значений, например:

```
extern long double c[]={7.89L,6.98L,0.5L,56.8L}; // 4 элемента
```

Если инициализируется массив, для которого объявлен только указатель, то память для размещения элементов массива выделять не надо, поскольку память выделяется для размещения констант, а затем адрес этой памяти заносится в указатель, например:

```
static short *m={2,3,5,8,12,0,56};        // массив на 7 элементов
```

**Доступ к элементам одномерного массива.** Доступ к элементам массива осуществляют по индексам. В качестве индекса можно указывать выражение с результатом целого (или символьного) типа. Если в качестве индекса указан целочисленный литерал или выражение над целочисленными литералами, то доступ называют *прямым*. Если в качестве индексов указано выражение, содержащее идентификаторы переменных, то доступ называют *косвенным*. Независимо от способа задания *индекс может меняться от 0 до величины на 1 меньшей размера*. Например, если объявлен массив `int a[5]`, то

а) прямой доступ:

```
a[0]=5;        // обращение к элементу с номером 0
```

б) косвенный доступ:

```
i=1; a[i+2]=5; // обращение к элементу с номером 3, который вычисляется
```

**Пример 3.1.** Программа определения максимального элемента массива и его номера:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float a[5],amax;  int i,imax;
     puts("Введите 5 значений:");
```

```
    for(i=0;i<5;i++) scanf("%f",&a[i]);
    amax=a[0];   imax=0;
    for(i=1;i<5;i++)
         if(a[i]>amax)
         {
              amax=a[i]; imax=i;
         }
    puts("Значения:");
    for(i=0;i<5;i++) printf("%7.2f ",a[i]);
    printf("\n");
    printf("Максимум = %7.2f  номер = %5d\n",amax, imax);
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Следует помнить, что по правилам Си и С++ независимо от способа объявления массива его имя – это имя переменной-указателя, содержащего адрес первого элемента массива. Поэтому для адресации элементов массива независимо от способа объявления можно использовать адресную арифметику. При этом следующие формы обращения эквивалентны:

```
(list+i)   ⇔ &(list[i])  // адреса элементов
*(list+i)  ⇔ list[i]     // значения элементов
```

### 3.4.2 Многомерные массивы

**Объявление многомерных массивов.** Так же, как и одномерные массивы, двух- и более мерные массивы можно объявить:

- статически, например:

```
int a[4][5]; // матрица элементов целого типа из 4 строк и 5 столбцов,
             // индексы меняются: первый от 0 до 3, второй  от 0 до 4
float b[10][20][2]; // трехмерный массив вещественных чисел из
                    // 10 строк,  20 столбцов и 2 слоев
```

- динамически, с помощью указателей, например:

```
short **matr; .
```

При статическом объявлении помять будет предоставлена одним куском по количеству определенных элементов. Элементы в памяти будут расположены построчно: элементы нулевой строки, элементы первой строки и т.д. (см. рисунок 3.6).

**Рисунок 3.6** – Размещение элементов статической матрицы в памяти

При динамическом описании обычно реализуют более удобные структуры. Например матрицу с указателями, хранящими адреса строк матрицы в явном виде – *массив динамических векторов* (см. рисунок 3.7):

```
float **D2;             // объявлен указатель на матрицу
D2=new float *[3];      // выделение памяти под массив указателей на строки
                        // матрицы
for(int i=0;i<3; i++)
       D2[i]=new float [4]; // выделение памяти под элементы строк
```

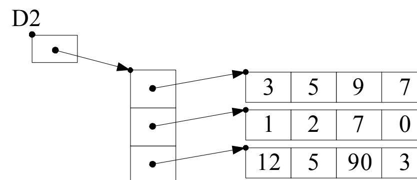



**Рисунок 3.7** – Массив динамических векторов

Обращение к элементам этой структуры может выполняться также как и к элементам матрицы, например:

```
D2[1][2]=3;
```

Точно так же, как и в одномерных массивах, для адресации элементов многомерного массива независимо от способа описания можно использовать адресную арифметику. Каждому уровню скобок при этом соответствует операция разыменования указателя, например:

```
int m[2][3][4];
```

`m` – «указатель указателя указателя», содержащий «адрес адреса адреса» первого элемента (см. рисунок 3.8).

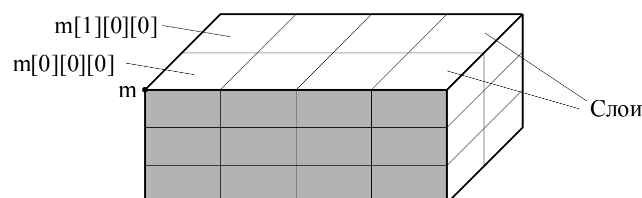



**Рисунок 3.8** – Трехмерная структура данных

Оглавление

Для обращения к элементу этого массива необходимо три операции разыменования:

```
*m   => m[0][?][?]
**m  => m[0][0][?]
```
**`***m => m[0][0][0]`**

Аналогично:

```
m[0][2][0] => *(*(*(m+0)+2)+0) => *(*(*m+2))
m[i][j][k] => *(*(*(m+i)+j)+k) =>*(*(*(i+m)+j)+k)
```

**Пример 3.2.** Написать программу, которая сортирует строки матрицы по возрастанию элементов с использованием указателей.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int **mas,*ptr;
int b,n,m,i,j,k;
int main()
{
     setlocale(0,"russian");
// Выделение памяти под матрицу
     printf("Введите n=");
     scanf_s("%d",&n);
     printf("Введите m=");
     scanf_s("%d",&m);
     mas=new  int *[n];        // выделение памяти под массив указателей
     for(i=0;i<n;i++)
          mas[i]=new int[m];   // выделение памяти под строки матрицы
//Заполнение матрицы данными
     for(i=0;i<n;i++)
     {
          printf("Введите %d элемента %d-й строки\n",m,i);
          for (j=0;j<m;j++)
               scanf_s("%d",&mas[i][j]);
     }
```

Оглавление

**// Вывод исходной матрицы на экран**

```
	puts("Введенная матрица:");
	for(i=0;i<n;i++)
	{
		for (j=0;j<m;j++)
			printf("%3d",mas[i][j]);
		printf("\n");
	 }
```

**// Сортировка строк матрицы - реализована через указатели**

```
	for(i=0;i<n;i++)
	{
		k=1;
		while(k!=0)
		{
			ptr=mas[i];
			for(k=0,j=0;j<m-1;ptr++,j++)
				if (*ptr>*(ptr+1))
				{
					b=*ptr;   *ptr=*(ptr+1);   *(ptr+1)=b;
					k++;
				}
		 }
	}
```

**// Вывод результата**

```
	puts("Сортированная матрица:");
	for(i=0;i<n;i++)
	{
		for (j=0;j<m;j++)
			printf("%3d",mas[i][j]);
		printf("\n");
	}
```

**// Удаление динамической матрицы**

```
	for(i=0;i<n;i++)         // удаление строк динамической матрицы
		delete[] mas[i];
	delete[] mas;          // удаление массива указжателей на строки
```

Оглавление

```
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

### *3.5 Строки*

В С++ символьная строка определяется как массив символов, который заканчивается *нуль-символом* – маркером конца строки «\0» (см. рисунок 3.9). Если маркер конца строки в символьном массиве отсутствует, то такой массив строкой не является. Маркер конца строки позволяет отслеживать реальную длину строки, которая может быть меньше размера массива.

| Э | т | о | | с | т | р | о | к | а | \0 |
|---|---|---|---|---|---|---|---|---|---|---|

Маркер конца строки

**Рисунок 3.9** – Внутреннее представление строки

Обращение к элементам строки осуществляется по индексу, так же как и к элементу массива. Соответственно и нумерация символов начинается с 0.

#### 3.5.1 Объявление и инициализация строк

Строку можно объявить теми же способами, что и одномерный массив символов. Единственно при объявлении надо учитывать, что любая строка должна заканчиваться маркером конца строки (значение 0), под который должен быть выделен один байт:

- объявление со статическим выделением памяти на этапе компиляции

`char` <Имя строки>[<Размер>] [= <Строковая константа>];

- объявление указателя на строку

`char` *<Имя указателя>[= < Строковая константа>];

Второй вариант предполагает, что либо строка инициализируется при объявлении, либо память под строку выделяют отдельно из области динамической памяти.

Примеры:

а) `char str[6];` // под строку выделено 6 байт, т.е. она может иметь длину до 5

б) `char *stroka = "Пример";` // под строку выделено 7 байт

в) `char *ptrstr;`

`ptrstr=new char[6];` // динамически выделили 6 байт

…

```
delete[] ptrstr;        // освободили память
```

Между способами а и б-в, так же, как и для массивов, существует существенное различие. В первом случае str – неизменяемый указатель, значение которого устанавливается один раз, когда под строку распределяется память (см. рисунок 3.10). К этому указателю нельзя применять адресную арифметику.

Во втором случае ptrstr и stroka – обычные указатели, которые можно изменять. Причем если указатель ptrstr утратит свое исходное значение, то станет невозможным корректное освобождение выделенной под строку памяти.

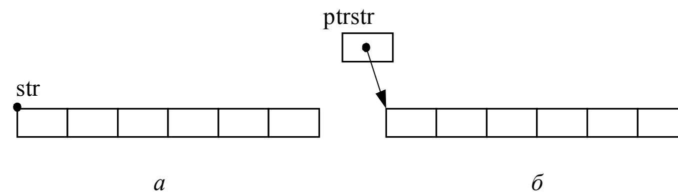



**Рисунок 3.10** – Различие между способами описания строки:

*а* – неизменяемый указатель; *б* – изменяемый указатель

Следует также иметь в виду, что, при различных способах объявления строк по-разному для этих строк выполняется оператор sizeof. Этот оператор используется для определения размера переменной в байтах и может быть применен для определения допустимого размера строки в случае, если память для строки выделена статически. Например:

а) правильное использование

```
char str1[5]; …
size_t a = sizeof(str1);  // a=5, допустимо 4 символа
```

б) неправильное использование

```
char * str1 = new [5]; …
size_t a = sizeof(str1);  // a=4 или 8! Поскольку получаем
                          // размер указателя, а не массива
```

**Инициализация строк.** Аналогично одномерному массиву строки, объявляемые вне функций или описанные как внешние extern или статические static (см. раздел 4.1.1), можно инициализировать, например:

а) `extern char str1[5] = {’A’,’B’,’C’,’D’,’\0’};` // система выделит // 5 байт и в них разместятся символы, включая «\0»

б) `static char str1[12] = {’A’,’B’,’C’,’D’,’\0’};` // система выделит // 12 байт и в них разместятся 5 символов, включая «\0», // содержимое остальных символов не определено

в) `extern char str1[] = "ABCD";` // система выделит 4 байта для размещения

// символов и 1 байт под символ «\0» !

г) `static char *str2 = "ABCD";` // система выделит 4 байта для размещения

// символов и 1 байт под символ «\0» !

Независимо от способа описания строки во всех случаях имя строки является указателем и содержит адрес строки.

**Строковые литералы.** При выполнении различных операций со строками, в строковых функциях, при инициализации строк часто используют *строковые литералы*, которые представляют собой последовательность символов, заключенных в кавычки, например:

`"Это строка"` .

При компиляции строковые литералы размещаются в статической памяти, а в использующий эти литералы оператор записываются их адреса, например:

char * str1= `"Это строка";` // указатель str1 будет содержать адрес строки

**Массивы строк.** Существует два варианта создания массивов строк:

- массив строк указанной максимальной длины (матрица символов):

`char` <Имя>[<Количество строк>][<Мах длина строки>] [= <Значение>];

где <Количество строк> – определяет, сколько строк можно записать в массив,

< Мах длина строки> – определяет максимальный размер сохраняемых строк,

<Значение> – определяет список инициализирующих строковых литералов;

- массив указателей на строки, память под которые отводится отдельно:

`char` * <Имя>[<Количество строк>] [= <Значение>];

Примеры:

а) `char ms[4][7]={"весна","осень","зима","лето"};` // объявлен

// «прямоугольный» символьный массив (см. рисунок 3.11, *а*)

б) `char *mn[4]={"весна","осень","зима","лето"};` // объявлен и

// инициализирован массив указателей на символьные строки разной длины

// (см. рисунок 3.11, *б*)

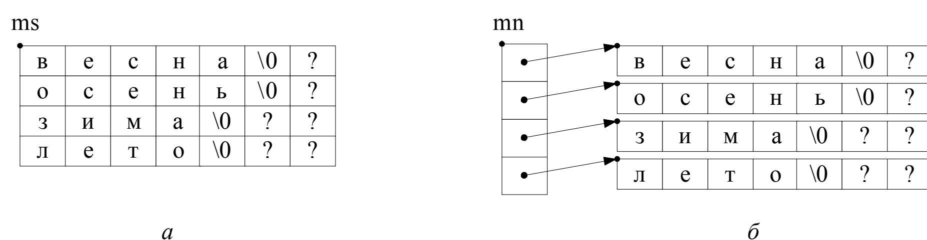



**Рисунок 3.11** – Символьная матрица и массив указателей на строки

### 3.5.2 Ввод и вывод строк

Как уже говорилось ранее под строку, в которую осуществляется ввод символов, память обязательно должна быть выделена. Выделение памяти под строку выполняется либо на этапе компиляции – статически, либо на этапе выполнения – динамически, например:

```
char st1[10];          // память выделена статически, 10 байтов
char * st2;            // память не выделена, просто объявлен указатель
st2=new char [10];     // динамически выделена память 10 байтов
```

**Ввод строк.** Для ввода строк можно использовать две функции – `gets` и `scanf`.

*Функция gets().* Функция считывает в указанную в параметре память символы. При этом маркер новой строки '\n', который создается при нажатии на клавиатуре клавиши Enter, заменяется маркером конца строки '\0'. Если при считывании строки происходит сбой или вводится пустая строка, то функция возвращает нулевой адрес `NULL`, иначе она возвращает адрес памяти, в которую введена строка, т.е. адрес, записанный в параметре-указателе.

Прототип функции:

```
char * gets(const * char s);
```

где s – указатель, содержащий адрес памяти, которая предназначена для хранения строки.

Функцию можно вызывать:

- как процедуру, тогда возвращаемое значение не учитывается, например:
  ```
  char name[10];
  gets(name);
  ```
- как функцию, что позволяет проанализировать результат ввода, например:
  ```
  char *ptrs=gets(name);
  if (ptrs!=NULL) …
  ```

Соответственно функция `gets` может использоваться в конструкциях, подобных `while (gets(name)!=NULL){…}` или `if (gets(name)==NULL)…`, для организации циклов и проверок.

В последней версии Visual C++ во избежание ошибки переполнения буфера при вводе строки, вызванной тем, что пользователь может ввести строку длиннее, чем размер буфера, вместо gets() рекомендуется использовать функцию

```
char *gets_s( char *s, size_t sizeInCharacters);
```

где `sizeInCharacters` – допустимый размер вводимой строки.

*Функция scanf().* Функция вводит строки по формату `%s` до первого пустого символа (пробела, знака табуляции, конца строки). Если в формате указывается размер, например %10s, то функция читает не более указанного количества символов строки. Функция `scanf()` возвращает количество считанных полей или константу EOF(-1), если прочитан конец файла на устройстве ввода (комбинация CTRL+Z при вводе с клавиатуры).

Функцию принято вызывать как процедуру:

```
char name[10];
scanf("%s",name);
```

Кроме того, функция может использоваться в конструкциях, подобных

```
while (scanf("%s",name)!=EOF){…} или
if (scanf("%s",name)==NULL)
```

для организации циклов и проверок.

Как уже указывалось ранее, вместо данной функции в последней версии Visual C++ рекомендуется использовать функцию `scanf_s()`, которой после указателя на строки или массивы передаётся их допустимый размер.

**Вывод строк.** Вывод строк выполняется с помощью функций `puts()` и `printf()`.

*Функция puts().* Функция выводит строку по адресу, указанному в качестве аргумента. В качестве аргумента можно указать строковую константу. При компиляции вместо этой константы в функцию будет подставлен адрес, по которому эта константа размещена компилятором. Вывод происходит до символа конца строки «\0», поэтому он обязательно должен присутствовать. Каждая выводимая по `puts()` строка начинается с новой строки. В качестве результата возвращает количество выведенных символов.

Прототип функции:

```
int puts(char *s);
```

Функцию можно вызывать:

- как процедуру:

```
char name[10]; int count;
puts(name);
```

- как функцию:

```
count=puts("Example function PUTS ");
printf("vivod  %4d symbols\n",count);
```

*Функция printf().* Использует в качестве аргумента указатель на строку. Вывод строк осуществляют по формату %s, что позволяет выводить строки только до пробела, поэтому используются для вывода строк без пробельных разделителей. Функция менее удобна, чем

puts, но более гибка, так как автоматического перехода на новую строку не выполняет и, следовательно, позволяет объединить при выводе в одной строке экрана строки из нескольких переменных. Для перехода на новую строку необходимо указать в форматной строке символ `'\n'`, например:

```
char name="Студент",MSG="GOOD";
printf("%s %s\n",name,MSG);
```

### 3.5.3 Функции, работающие со строками

Для обработки строк в С++ предусмотрено большое количество функций, размещенных в библиотеках `string`, `stdlib`. Соответственно для работы с этими функциями необходимо подключить файлы: `string.h` и `stdlib.h`. Ниже приведены прототипы наиболее часто используемых функций. В зависимости от формата часть из них может использоваться как функции или как процедуры.

От Си С++ унаследовал то, что символьные параметры функций передаются как целые числа. Физически при этом используется только младший байт числа. Однако в качестве аргументов этих функций можно использовать и символы, поскольку при вызове они автоматически преобразуются в целые числа.

1) `size_t strlen(char *s);` // функция возвращает длину строки параметра

*Примечание*. Тип `size_t` применяют для указания размера (длины). В зависимости от установки 32-х или 64-х разрядной модели памяти число типа size_t – это целое число со знаком размером 4 или 8 байт соответственно.

2) `char *strcat(char *dest,const char *src);` // функция объединяет
// строки (добавляет вторую строку к первой) и возвращает дубликат адреса
// первой строки, содержащей объединение строк

3) `int strcmp(const char *s1,const char *s2);` // функция сравнения
// строк, возвращает разницу кодов в ANSI первого различного символа
// строк `s1` и `s2`, если строки равны, то возвращает 0

4) `char *strcpy(char *dest,const char *src);` // функция копирует
// вторую строку в первую и возвращает дубликат адреса первой строки

5) `char *strncpy(char *dest, const char *src, size_t maxlen);`
// функция копирует `maxlen` символов из `src` в `dest` и возвращает
// дубликат адреса строки-результата `dest`

6) `char *strchr(const char *s, int c);` // функция определяет адрес

// первого вхождения символа **c** в строку **s**, иначе возвращает `NULL`

7) `char *strstr(const char *s1, const char *s2);` // функция

// возвращает адрес первого вхождения строки `s2` в строку `s1`, если

// вхождение отсутствует, то возвращает `NULL`

8) `char *strtok_s(char *str1,const char *str2, char **ptrptr);`

// возвращает указатель на следующее слово (лексему) в строке str1,

// при этом символы, образующие строку str2, являются разделителями,

// определяющими лексему, если лексемы не обнаружены – возвращается

// `NULL`. Параметр `ptrptr` используется для хранения позиции в строке.

// *Использование ранее применяемой функции `strtok()` при параллельном*

// *программировании может приводить к ошибкам*

9) `int atoi(const char *s);` // функция возвращает целое число,

// в символьном виде записанное в строке **s**

10) `double atof(const char *s);` //функция возвращает вещественное число

// двойной точности, в символьном виде записанное в строке **s**

11) `char *itoa(int value,char *s,int radix);` // функция

// осуществляет преобразование целого числа `value` в строку **s**

// в соответствии с заданной системой счисления `radix` (от 2 до 36).

12) `char *_gcvt(double value, int digits, char *buffer );`

// функция преобразует число `value` в строку `buffer` с учетом параметра

// `digits`, который определяет количество значащих цифр

13) `char *_ecvt(double value,int count,int *dec,int *sign);`

// функция преобразует число `value` в строку `buffer` с учетом параметров

// `count` – количество цифр,

14) `char *_fcvt(double value,int count,int *dec,int *sign);`

// функция преобразует число `value` в строку результата

// `count` – количество цифр, `dec`, `sign` – позиции точки и знака

15) `int sscanf(` const char *buffer, const char *format [, argument ] ... );

// функция для разбора элементов строки buffer по указанному формату

16) `int sprintf(` char *buffer, const char *format [,argument] ... );

// функции для формирования строки из элементов по указанному формату

Для предотвращения возможной ошибки переполнения буфера в MS Visual C++ 2008 добавлены функции `strcat_s`, `strcpy_s`, `strncpy_s`, в которых указывается допустимая длина строки, принимающей результат.

**Пример 3.3.** Разработать программу, которая выделяет слова их исходной строки с использованием функции `strtok_s`().

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main( void )
{
    // исходная строка
    char string[]="A string\tof ,,tokens\nand some more tokens";
    // строка разделителей
    char seps[] = " ,\t\n", *token, *context;
    setlocale(0,"russian");
    token = strtok_s( string,seps,&context);
    while(token!=NULL)
   {
        printf("%s ",token);
        token=strtok_s(NULL,seps,&context);
   }
    puts("\nНажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Результаты:

```
A string of tokens and some more tokens
Нажмите любую клавишу для завершения…
```

**Пример 3.4.** Демонстрация преобразования результатов для вывода.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
```

```
{
     setlocale(0,"russian");
     int      decimal,sign;   // позиция точки и знака
     char     *buffer;
     int      precision = 10;    // точность
     double   source = 3.1415926535;
     buffer =_ecvt( source, precision, &decimal, &sign );
     printf ("source: %12.10f\nbuffer: '%s'\n
                     decimal: %d\nsign: %d\n",
                     source, buffer, decimal, sign );
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

Результат работы программы:

```
source: 3,1415926535
buffer: '3141592654'
decimal: 1
sign: 0
Нажмите любую клавишу для завершения...
```

**Пример 3.5.** Составить программу, осуществляющую ввод строк, содержащих имя, фамилию, отчество и год рождения, записанные через пробел. Каждая введенная строка должна преобразовываться в строку, содержащую фамилию, инициалы и возраст в 2010 году. Например:

Petrov Petr Petrovich 1987 => Petrov P.P. 23

```
#include <string.h>
#include <stdlib.h>
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     char st[40],strez[40],strab[40],*ptr1,*ptr2,*ptr3;
     int old;
```

```
 while ( puts("Введите строку или Ctrl-z:"),
              gets_s(st, sizeof(st)-1)!=NULL)
 {
      strcpy(strez,st);            // копирование st в strez
      ptr1=strchr(strez,' ');      // поиск первого пробела
      *(ptr1+2)='.';               // вставка точки после инициала
      ptr2=strchr(st,' ');         // поиск первого пробела
      ptr2=strchr(ptr2+1,' ');     // поиск второго пробела
      strncpy(ptr1+3,ptr2+1,1);    // копирование второго инициала
      strncpy(ptr1+4,". \0",3);    //вставка ".",пробела и конца строки
      ptr3=strchr(ptr2+1,' ');     // поиск третьего пробела
      old=2010-atoi(ptr3+1);       // определение возраста
      strcat(strez,itoa(old,strab,10));// добавление возраста
      puts(strez);
 }
 puts("Нажмите любую клавишу для завершения...");
 _getch();
 return 0;
}
```

Результаты выполнения программы (вводимые данные выделены полужирным):

```
Введите строку или Ctrl-z:
Ivanov Ivan Ivanovich 1958
Ivanov I.I. 52
Введите строку или Ctrl-z:
Petrov Petr Petrovich 1987
Petrov P.P. 23
Введите строку или Ctrl-z:
^Z
Нажмите любую клавишу для завершения...
```

### *3.6 Структуры*

Иногда, при составлении программ необходимо объединить в единое целое разнородную, но логически связанную информацию. Например, нам необходимо хранить данные библиотечной карточки, содержащей следующую информацию о книге: фамилии и

инициалы авторов; название книги; место издания; издательство; год издания; количество страниц. Объединить такую разнородную информацию можно связав ее в структуру.

*Структура* – это набор разнотипных переменных, объединенных общим именем. Объявление структуры создает ее шаблон, который можно использовать при создании переменных типа этой структуры.

Переменные, входящие в структуру, называются ее *членами* (элементами или полями). Как правило, все члены структуры логически связаны между собой.

Объявление структур может выполнить двумя способами: так как это было предложено делать в Си и с использованием оператора создания нового типа `typedef`.

**Объявление структуры по правилам Си:**

```
struct [<Имя структуры>] {<Описание полей>} [<Список переменных>];
```

Такое описание структуры позволяет организовать два варианта реализации:

а) описание структуры выполняется отдельно от объявления списка переменных, например:

```
struct student
{
    char name[22],family[22];
    int old;
}; // описание структуры
struct student stud1,stud2,*ptrstud;  // объявление переменных
```

б) описание структуры выполняется совместно с объявлением переменных, в этом случае имя структуре можно не присваивать, например:

```
struct
{
     char name[22], family[22];
     int old;
} stud1, stud2, *ptrstud;
```

**2. Описание структуры средствами С++:**

```
typedef struct {<Описание полей>} <Имя типа структуры>;
```

Фактически посредством `typedef` мы объявляем новый тип данных, например:

```
typedef struct
{
     char name[22],family[22];
     int old;
} student;
```

Оглавление

В данном случае `student` – это имя нового типа данных. Соответственно возможно объявление переменных этого типа, например:

```
student stud1,stud[10],*ptrstud; // объявляются: переменная, массив и
                                  // указатель типа student
```

В последнем случае память под размещение структуры надо запрашивать специально, например:

```
ptrstud=new student;
```

Обращение к элементам структуры выполняется с указанием имени переменной и имени поля:

<Имя переменной>.<Имя поля>

<Имя массива>[<Индекс>].<Имя поля>

(*<Имя указателя>).<Имя поля>

Примеры:

```
stud1.name
stud[i].name
(*ptrstud).name
```

Если переменная – указатель, то удобнее использовать сокращенную форму записи обращения к элементу структуры

<Имя указателя> -> <Имя поля>, например:

```
ptrstud->name
```

Статические и внешние структуры (см. раздел 4.1.1) при объявлении можно инициализировать. При этом значения полей указываются после символа равенства в фигурных скобках через запятую, например:

```
static student stud1={"Петр","Петров",18},
               *ptrstud={"Иван","Иванов",19};
```

Чаще всего структуры используются как элементы массивов. Чтобы объявить массив структур, необходимо определить тип структуры и объявить массив переменных этого типа. Как во всех массивах, нумерация элементов массива структур начинается с 0, а переменная, содержит адрес первого элемента массива.

**Пример 3.6.** Написать программу формирования массива данных об игрушках, содержащих их название, количество и стоимость, и определения товара с наибольшей стоимости (см. рисунок 3.12).

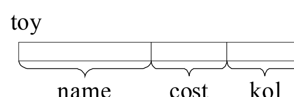



**Рисунок 3.12** – Структура toy

Оглавление

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef struct
{
     char name[15]; float cost; int kol;
}toy;
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     toy mas[10];
     int i,n=-1,k;
     char st[15];
     float sumcost=0,maxcost=0,num;
     st[0]='\0';
     while( (puts("Введите название или end"), scanf("\n%s",st),
               strcmp(st,"end")!=0)&&(n<9) )
     {
          n++;
          strcpy(mas[n].name,st);
          printf("Введите стоимость и количество:");
          scanf("%f %d",&mas[n].cost,&mas[n].kol);
     }
     puts("=======    Список товаров   ======");
     puts(" N   Название Стоимость Количество");
     puts("==================================");
     for(i=0;i<=n;i++)
     {
        printf("%3d   %10s",i+1,mas[i].name);
        printf(" %9.2f  %5d\n",mas[i].cost,mas[i].kol);
        sumcost=mas[i].cost*mas[i].kol;   // определение суммы
        if (sumcost>maxcost)
        {
         maxcost=sumcost;      // определение максимальной стоимости товара
```

```
            strcpy(st,mas[i].name);   // сохранение названия товара
          }
     }
     printf("Товар %10s имеет максимальную стоимость= %8.3f\n",
                                                         st,maxcost);
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

Результат работы программы (вводимые данные выделены полужирно):

```
Введите название или end
Kukla
Введите стоимость и количество: 256 23
Введите название или end
Mishka
Введите стоимость и количество: 400 20
Введите название или end
end
=======    Список товаров   ======
 N   Название Стоимость Количество
==================================
  1      Kukla   256,00     23
  2     Mishka   400,00     20
Товар     Mishka имеет максимальную стоимость 8000,000
Нажмите любую клавишу для завершения...
```

Поля структуры могут быть не только скалярными, но и структурными. Так членами структуры могут быть массивы данных скалярных типов, массивы массивов. Кроме того, элементами структуры могут быть другие структуры, получившие названия *вложенных*. Глубина вложения компилятором не ограничивается. Обращение к таким членам тоже осуществляется с помощью точечной нотации, после чего идет обращение к элементу по правилам обращения, определяемому типом элемента.

**Пример 3.7.** Написать программу определения среднего балла каждого студента и группы в целом после сдачи трех экзаменов.

Структуры данных:

1) test – данные об экзамене: название экзамена и отметка (см. рисунок 3.13, *а*);
2) student – фамилия студента и массив результатов 3 экзаменов (см. рисунок 3.13,*б*).

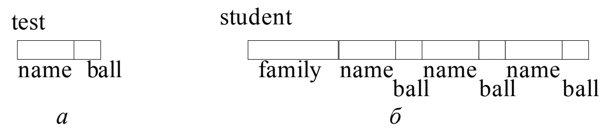



**Рисунок 3.13** – Структуры test (*а*) и student (*б*)

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
typedef struct    // данные об экзамене
{
     char name[10];  int ball;
} test;
typedef struct   // данные о студенте
{
     char family[22];
     test results[3];
}student;
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     student stud[10]; // массив данных о студентах
     int i,n=0; float avarstud,avarage=0;
     while (puts("Введите фамилию, предметы и оценки или end"),
        scanf("\n%s",stud[n].family),
        strcmp(stud[n].family,"end")!=0)
     {
          for (avarstud=0,i=0; i<3; i++)
          {
               scanf("\n%s %d",stud[n].results[i].name,
                                     &stud[n].results[i].ball);
               avarstud+=stud[n].results[i].ball;
          }
```

```
        printf("Среднее:%s=%5.2f\n",stud[n].family,avarstud/3);
        avarage+=avarstud;
        n++;
    }
    printf("Средняя оценка в группе=%5.2f\n",avarage/n/3);
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

### *3.7 * Объединения*

Объединение – это средство описания области памяти, используемой для хранения переменных разных типов. Объединения позволяют хранить в разное время различные данные или интерпретировать один и тот же набор битов по-разному.

Объявление объединения напоминает объявление структуры:

**union <Имя объединения> {<Список элементов объединения>}**

**[<Список переменных [и значений]>];**

Например:

```
union mem          // наложение полей (см. рисунок 3.14)
{
    double d;
    long l;
    int k[2];
};
```

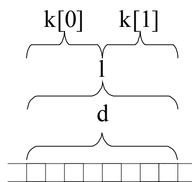



**Рисунок 3.14** – Наложение описаний переменных в объединении

Это объявление фактически объявляет новый тип – объединение под названием `mem`. После объявления можно объявлять переменные указанного типа, массив таких переменных и указатели на переменные данного типа. Чтобы объявить переменную, нужно либо указать ее имя в конце объявления типа, либо применить отдельный оператор объявления, например:

```
union mem comp,vfsun[10],*ptrun;
```

В языке C++ не запрещается указывать перед этим объявлением ключевое слово `union`, но это излишне, так как имя объединения полностью определяет его тип, например: Например:

```
mem comp,vfsun[10],*ptrun;
```

При объявлении переменной типа объединения компилятор автоматически выделяет память, достаточную для хранения наибольшего члена объединения.

Для доступа к членам объединения используются те же синтаксические конструкции, что и для доступа к членам структуры.

Объединения можно использовать для специального преобразования типов, поскольку к хранящимся в объединении данным можно обращаться разными способами. Например, объединение можно использовать для манипуляции байтами, образующими значение вещественного числа, изменять его точность или выполнять необычное округление. Кроме того, объединения, например, могут использоваться для выделения отдельных полей из одного длинного.

### *3.8 Динамические структуры данных. Списки*

*Динамическими* принято называть структуры данных, которые создаются в процессе выполнения программы. Понятие «создаются» можно интерпретировать по-разному:

- создаются – в смысле «получают память и начинают реально существовать»;
- создаются – в смысле «организуются, строятся из некоторых элементов».

К первому типу относятся динамические массивы, строки и структуры, память под которые выделяется во время выполнения программы. К структурам второго типа относятся списки.

*Список* – структура, при организации которой использованы указатели, содержащие адреса следующих элементов. Элемент списка состоит из двух частей: информационной и адресной. *Информационная* часть содержит поля данных. *Адресная* – включает от одного до n указателей, содержащих адреса других элементов. Количество связей, между соседними элементами списка определяет его связность: односвязные, двусвязные, n-связные.

По структуре списки бывают линейными, древовидными и сетевыми. На линейных списках обычно реализуют разные дисциплины обслуживания:

- очередь – дисциплина обработки элементов данных, в которой добавление элементов выполняется в конец, а удаление – из начала;

- стек – дисциплина обработки элементов данных, в которой добавление и удаление элементов осуществляется с одной стороны;
- дек – дисциплина обработки элементов данных, в которой добавление и удаление элементов может выполняться с двух сторон.

### 3.8.1 Описание элементов списковых структур

Для описания элемента списковой структуры используют структуры, одно или несколько полей которых – указатели на саму эту структуру. Ниже приведены примеры описания элементов односвязного и двусвязного списков.

Элемент односвязного списка:

```
struct element // тип элемента
{
    char name[16];      // информационное поле 1
    char telefon[7];    // информационное поле 2
    element *p;         // адрес следующего элемента
};
```

Элемент двусвязного списка:

```
struct element          // тип элемента
{
    char name[16];      // информационное поле 1
    char telefon[7];    // информационное поле 2
    element *prev;      // адресное поле «предыдущий»
    element *next;      // адресное поле «следующий»
};
```

### 3.8.2 Основные приемы работы

Создание списковой структуры предполагает:

- описание элемента списка;
- объявление указателей для работы со списком;
- создание пустого списка;
- добавление элементов к списковой структуре и удаление их из нее в процессе работы.

Рассмотрим эти операции на конкретном примере.

1. Описание элемента списка:

```
struct element   // тип на элемента
{
      int num;        // целое число
      element *p;   // указатель на следующий элемент
};
```

2. Описание переменной – указателя списка и нескольких переменных-указателей в статической памяти:

```
element * first,     // адрес первого элемента
          *n,*f,*q;  // вспомогательные указатели
```

3. Исходное состояние – «список пуст»:

```
first=NULL;
```

**Добавление элементов в список.** Возможно несколько вариантов добавления элементов к списку:

- добавление элемента к пустому списку;
- перед первым, например, при построении списка по типу стека;
- после последнего, например, при построении списка по типу очереди;
- после/перед заданным элементом, например при построении сортированного списка.

1 Добавление элемента к пустому списку:

```
first=new element;    // запросили память под элемент
first->num=5;         // занесли данные в информационное поле
first->p=NULL;        // записали признак конца списка NULL
```

2 Добавление элемента перед первым (по типу стека):

```
q=new element;        // запросили память под элемент
q->num=4;             // занесли данные в информационное поле
q->p=first;           // записали в новый элемент адрес первого
first=q;              // записали в качестве первого адрес нового элемента
```

3 Добавление элемента после первого (по типу очереди):

```
q=new element;        // запросили память под элемент
q->num=4;             // занесли данные в информационное поле
q->p=NULL;            // записали признак конца списка NULL
first->p=q;           // записали признак конца списка NULL
```

**Удаление элемента из списка.** При удалении элемента из списка также возможны варианты:

- удаление первого элемента;
- удаление элемента с адресом q;
- удаление последнего элемента.

1. Удаление первого элемента

```
q=first;              // скопировали адрес первого элемента
firs=first->p;        // запомнили адрес нового первого элемента
delete q;             // удалили бывший первый элемент
```

2. Удаление элемента с адресом q

```
f=first;              // скопировали адрес первого элемента
while(f->p!=q) f=f->p;  // нашли предыдущий элемент
q=q->p;               // перешли к следующему элементу
delete f->p;          // удалили элемент с адресом q
f->p=q;               // сменили адрес в предыдущем элементе
```

3. Удаление последнего элемента

```
f=q=first;            // скопировали адрес первого элемента
while(q->p!=NULL)     // нашли последний и предыдущий элементы
{
   f=q;
   q=q->p;
}
f->p=NULL;            // объявили предыдущий элемент последним
delete q;             // удалили последний элемент
```

**Пример 3.8**. Написать программу, которая формирует список деталей, содержащий наименование и диаметр детали. Удалить из списка все детали с диаметром, меньшим 1.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
struct zap  // тип элемента
{
    char det[10];  float diam;   zap *p;
};
```

```
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     zap a,*r,*q,*f;
     r=new zap;
     r->p=NULL;
     puts("Вводите названия деталей и их диаметр:");
     scanf("%s %f\n",r->det,&r->diam);
     while( (scanf_s("\n%s",a.det,sizeof(a.det)-1 )),
               strcmp(a.det,"end")!=0 )
     {
          scanf("%f",&a.diam);
          q=r;
          r=new zap;
          strcpy(r->det,a.det);
          r->diam=a.diam;
          r->p=q;
     }
     // удаление записей
     q=r;
     do
     {
          if (q->diam<1)
          {
               if( q==r)
               {
                    r=r->p;   delete(q);   q=r;
               }
               else
               {
                    q=q->p;   delete(f->p);  f->p=q;
               }
          }
          else
          {
               f=q; q=q->p;
```

```
            }
        } while(q!=NULL);
        q=r;
        puts("Результаты:");
        if(q==NULL) puts("Данные отсутствуют.");
        else
            do
            {
                printf("%s %5.1f\n",q->det,q->diam);
                q=q->p;
            } while (q!=NULL);
        puts("Нажмите любую клавишу для завершения...");
        _getch();
        return 0;
}
```

### Контрольные вопросы к главе 3

1. Дайте определение указателя. В чем его физический смысл?

Ответ.

2. Что такое ссылка и какое ее основное достоинство?

Ответ.

3. Поясните отличие ссылки от указателя.

Ответ.

4. Что такое адресная арифметика и основные правила ее выполнения?

Ответ.

5. Назовите основные операции по выделению и освобождению динамической памяти.

Ответ.

6. Что такое массивы и для чего их используют? Как можно создать массивы?

Ответ.

7. Как в C++ определить символьную строку? Какие существуют способы определения строк и чем они отличаются?

Ответ.

8. Какие стандартные функции определены для обработки строк и какую библиотеку для этого нужно подключить?

Ответ.

9. Что такое структура в С++ и для каких целей она используется?

Ответ.

10. Что такое динамические структуры данных?

Ответ.

11. Приведите пример описания элемента односвязного списка.

Ответ.

# 4 ФУНКЦИИ. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Для улучшения эффективности программ в языках высокого уровня были разработаны средства модульного программирования, предусматривающие использование подпрограмм и модулей.

*Подпрограмма* – это относительно самостоятельный фрагмент программы, соответствующим образом оформленный и снабженный именем.

В зависимости от способов описания и вызова различают подпрограммы двух видов: процедуры и функции.

Процедуры предназначены для выполнения некоторых действий, например, печати строки, а функция – позволяют получить задаваемую скалярную величину, которую возвращает в качестве результата.

Си и С++ декларированы как языки *функционального* программирования. Поэтому в Си и С++ нет понятия процедуры, как средства языка. Однако существует возможность создания функций, которые не возвращают значения и реализуют действия, свойственные процедурам.

## *4.1 Функции С++*

Каждая программа на С++ обязательно должна включать единственную главную функцию с именем `main`. Кроме нее в программу может входить произвольное количество функций, выполнение которых прямо или косвенно инициируется функцией `main`.

Функции можно описывать в любом месте программы и даже в другом файле, но только не внутри другой функции.

Различают объявление и описание функций. *Описание функции* состоит из заголовка, при необходимости используемого в качестве прототипа, и собственно тела функции:

**<Тип результата или void> <Имя функции> ([<Список параметров>])**

**{ [ < Объявление переменных и констант >]**

**<Операторы>**

**}**

*Объявление функций* предполагает включение в начало программы *прототипов* (заголовков) всех используемых функций, тогда описания функций может быть приведены в любом порядке. Если прототипы функций в начале программы не помещать, то все функции должны быть описаны до своего вызова.

Функция может получать данные двумя способами:

а) неявно – с использованием видимых в других функциях переменных;

б) явно – через параметры.

Неявная передача:

- приводит к большому количеству ошибок, вызванных возможной «порчей» глобальных данных подпрограммами;
- жестко связывает подпрограмму и данные, существенно снижая степень универсальности подпрограмм.

При отсутствии параметров список можно заменить служебным словом void или опустить, например:

```
int print(void); int print();
```

**Формальные и фактические параметры.** *Формальными* называют параметры, определенные в заголовке описания функции. При перечислении для каждого формального параметра помимо имени задают тип, например:

```
float max(float a,char b){…}
```

*Фактическими* называют параметры, задаваемые при вызове функции. В математике такие параметры называют аргументами. В качестве аргументов можно использовать литералы, поименованные константы, переменные и выражения.

Совокупность формальных параметров определяет *сигнатуру функции.* Сигнатура функции зависит от количества параметров, их типа и порядка размещения в спецификации формальных параметров.

При задании аргументов следует помнить, что формальные и фактические параметры, относящиеся к одной функции, должны совпадать: по количеству параметров, по их типу и по порядку следования, например:

```
int k,l,n=6; float d=567.5,m=90.45
void fun(int a,float c,float b){….}   // описание функции fun
fun(n,d,m);         // правильный вызов, аргументы – переменные  или константы
fun(4,8.7,0.1);     // правильный вызов, аргументы – литералы
fun(n%4-3,8.7,0.1); // правильный вызов, один из аргументов – выражение
fun(4,8.7);         //  ошибка в количестве параметров
fun(4.67, 5,7);     // ошибка в типах параметров
fun(3,m,d);//не обнаруживаемая компилятором ошибка в порядке следования
           // аргументов
```

**Оператор возврата значения функции.** Если описывается подпрограмма-функция, которая возвращает в вызывающую программу формируемое значение, то в теле функции обязательно наличие оператора возврата `return`, передающего это значение, например:

```
int max(int a,int b)
{
     if (a>b) return a
     else     return b
}
```

Вызов функции может выглядеть, например, так: `k=max(i,j);` .

В соответствии со стандартом, если функция возвращающая значение, не использует оператора `return`, то в вызывающую программу возвращается случайное значение (называемое *мусором*).

При необходимости функция может содержать несколько операторов возврата `return`, как в примере выше. Встретив любой из них, функция возвращает значение и управление в вызывающую программу.

Если функция не объявлена со спецификацией void, то ее можно использовать в качестве операнда в любом выражении, например:

`K=max(a,b)/max(c,e);` .

**Пример 4.1.** Разработать подпрограмму, определяющую максимальное из двух вещественных чисел.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
float max(float a,float b);        // прототип функции
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     printf("Результат = %f\n",max(4.6,5.8));
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

```
float max(float a,float b) // описание функции
{
     return (a>b)?a:b;
}
```

Если описывается подпрограмма-процедура, то она, либо не должна возвращать результатов, либо должна возвращать их через параметры (см. далее). Оператор возврата значений в подпрограммах-процедурах не используется.

**Возврат результатов через параметры функции (процедуры).** Существует два способа передачи аргументов в подпрограммы через параметры. Первый из них известен как «передача параметров по значению». В этом случае в подпрограмму передаются *копии фактических параметров*, и никакие изменения этих копий не возвращаются в вызывающую программу.

Второй способ называют «передачей параметров по ссылке». При использовании этого метода в подпрограмму передаются *адреса фактических параметров*, соответственно, все изменения этих параметров в подпрограмме происходят с переменными основной программы. *По умолчанию в языках Си и С++ применяется передача по значению.*

Если подпрограмма меняет передаваемое значение, и это измененное значение надо вернуть в вызывающую программу, то используют один из двух способов:

а) в качестве параметра *применяют указатель* и при вызове передают адрес;

б) в качестве параметра *применяют ссылку* и при вызове просто указывают имя параметра.

В обоих вариантах подпрограмма получает *копию адреса*, по которому надо записать возвращаемое значение. Например:

а) возврат значения с использованием параметров-указателей:

описание функции: `void prog(int a,int *b){*b=a;}`

вызов функции: `prog(c,&d);`

б) возврат значения с использованием параметров-ссылок:

описание функции: `void prog(int a, int &b){b=a;}`

вызов функции: `prog(c,d);`

Если наоборот надо *запретить изменение параметра*, переданного адресом, то его объявляют с описателем const, например:

```
int prog2(const int *a) {…}
```

### 4.1.1 Классы памяти переменных

В С++ переменные могут быть объявлены как вне, так и внутри функций. При этом каждой переменной присваивается класс памяти.

*Класс памяти* определяет:

- часть памяти программы, в которой размещается переменная (сегмент данных или сегмент стека);
- область действия, т.е. доступность переменной из функций;
- время жизни переменной, т.е. как долго она хранится в памяти.

Различают пять классов памяти:

**1. Внешние переменные (**описатель **extern).** Размещение – глобальная память программы (сегмент данных), область действия – все файлы все функции программы, в которой она определена, время жизни – с момента вызова программы и до возврата управления операционной системы.

По умолчанию, если переменная описана вне функции, то она – внешняя, т.е. внешние переменные вне функций, можно объявлять без описателя `extern`, например:

```
int a;                // внешняя переменная (по умолчанию)
int main()
{   extern int a;…    }   // внешняя переменная
abc()
{   extern int a;…    }   // внешняя переменная
```

В примере все три объявления относятся к одной и той же переменной, которая объявлена трижды.

**2. Автоматические переменные (**описатель **auto).** Размещение – локальная память (сегмент стека программы), область действия – внутри функции или оператора, где она определена, время жизни – с момента вызова функции и до момента возврата управления.

Если функция, содержащая автоматические переменные, вызывает другую функцию, то вызванная функция имеет доступ к автоматическим переменным вызвавшей ее функции, но не наоборот.

Исключением является случай, когда автоматическая переменная вызываемой подпрограммы имеет то же имя, что и внешняя переменная или автоматическая переменная вызывающей подпрограммы. Если такое происходит, то автоматическая переменная вызываемой подпрограммы «перекрывает» доступ к другим переменным с тем же именем. Так если к функциям предыдущего примера добавить функцию

```
bcd()
```

```
{   int a;…   }
```
// автоматическая переменная, «перекрывающая» внешнюю

то внешняя переменная a становится в подпрограмме bcd() е недоступной: все обращения из этой подпрограммы будут выполняться к внутренней автоматической переменной a.

По умолчанию все переменные, описанные внутри функции, – автоматические, т.е. описатель auto для них можно не применять, например:

```
int main()
{    int a;…    }          // автоматическая переменная (по умолчанию)
abc()
{    auto int a;…    }    // автоматическая переменная
```

Автоматические переменные функции `main` обладают особыми свойствами: время их жизни совпадает со временем работы всей программы и они доступны по имени из всех ее функций, что по свойствам делает эти переменные похожими на внешние.

**3. Статические переменные (**описатель **`static`).** Размещение – глобальная память подпрограммы (сегмент данных), область действия – внутри функции, в которой она определена, время жизни – все время работы программы. В отличие от автоматической статическая переменная не исчезает, когда функция завершает работу, по сравнению с внешней – недоступна из всех функций программы, кроме той, в которой была определена.

```
abc()
{
   int a=1;             // автоматическая переменная
   static int b=1;      // статическая переменная
   …
   a++;     // каждый раз инициализируется заново
   b++;     // инициализируется один раз при первом обращении и
   …        // увеличивается с каждым вызовом функции
}
```

**4. Внешние статические переменные** (описатель **`extern static`**). Размещение – глобальная память модуля (файла), область действия – внутри всех функций модуля (файла) программы, где она определена, время жизни – с момента вызова программы и до возврата управления операционной системе.

Например:

*файл Mod1.cpp:*

```
int a;                  // внешняя переменная (по умолчанию)
extern static int b;    // внешняя статическая переменная
```

*файл Mod2.cpp:*

```
int abc()
{ h=a; …}
```

Переменная a доступна в обоих файлах, переменная b – только в первом.

**5. Регистровые переменные** (описатель **register).** Регистровые переменные аналогичны автоматическим, но по возможности должен их размещать в регистровой памяти процессора, например:

```
register int a;
```

Такое размещение ускоряет операции над этими переменными. Однако свободных регистров обычно очень мало (1-2), и возможность их использования сильно зависит от остального текста программы. Если регистры заняты, то переменная размещается аналогично переменной auto.

По возможности следует использовать автоматические переменные. Это снижает зависимость функций и существенно уменьшает количество ошибок в программах.

### 4.1.2 Параметры сложных структурных типов

**Передача массивов в подпрограммы.** В силу специфики организации массивов в Си и С++ массивы передаются в подпрограммы как указатели на первый элемент. *Размерность массива по первому индексу при этом не контролируется*. Это вызвано тем, что С++ проверяет соответствие фактических и формальных параметров функции только по порядку и по типу, причем понятие типа при этом не включает конкретной размерности массива.

Следовательно, при передаче одномерного массива в качестве параметра достаточно указать, что этот параметр – массив или вообще описать указатель на элемент, а при работе с многомерными массивами можно не указывать размерность по первому индексу.

Тогда, допустимы следующие варианты описания параметров-массивов:

а) `int x[5]` ⇔ `int x[]` ⇔ `int *x` // размерность проверяться не будет

б) `int y[4][8]` ⇔ `int y[][8]` // будет проверяться размерность массива
// только по второму размеру

При этом если в функцию передается массив, значения которого не должны меняться внутри функции, его следует описать как const, например:

```
int k23(const int *a);
```

**Пример 4.2.** Написать функцию формирования в отдельном массиве сумм строк матрицы.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
const int stringNum=5;
void summa(const float x[][3], float y[],int n);
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float a[stringNum][3]={{3.1,2.4,1.6},
                    {6.1,9.3,1.3},
                    {2.4,1.9,1.4},
                    {-9.0,4.5,1.9},
                    {7.2,8.0,1.8}},
     float y[stringNum];
     summa(a,y, stringNum);
     for (int i=0;i< stringNum;i++) printf("%6.1f",y[i]);
     puts("\nНажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
void summa(const float x[][3], float y[],int n)
{
    int i,j;
    for(i=0;i<n;i++)
        for(y[i]=0,j=0;j<3;j++) y[i]=x[i][j];
}
```

**Пример 4.3.** Написать подпрограмму удаления из матрицы l-ой строки и k-го столбца. Исходная матрица должна формироваться тестирующей программой с помощью датчика случайных чисел.

```
#include <locale.h>
```

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <time.h>
#include  <stdlib.h>
void delsts(int a[][10],int & n,int & m,int l,int k)
{
	int i,j;
	for(i=l;i<n-1;i++)                // удаление строки
		for(j=0;j<m;j++) a[i][j]=a [i+1][j];
	for(j=k;j<m-1;j++)                // удаление столбца
		for(i=0;i<n;i++) a[i][j]=a[i][j+1];
	n--;  m--;
}
int main(int argc, char* argv[])
{
	setlocale(0,"russian");
	int matr[10][10],n,m,l,k,i,j;
	puts("Введите n,m<=10");
	scanf("%d %d",&n,&m);
	puts("Исходная матрица:");
	srand( (unsigned)time( NULL )); // установка датчика случайных чисел
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		{
			matr[i][j]=rand()/1000;   // вызов датчика случайных чисел
			printf("%4d",matr[i][j]);
		}
		printf("\n");
	}
	printf("Введите l< %5d   k<%5d  для удаления\n",n,m);
	scanf("%d %d",&l,&k);
	delsts(matr,n,m,l,k);      // вызов функции удаления l строки и  k столбца
	puts("Полученная матрица:");
```

```
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++) printf("%4d",matr[i][j]);
		printf("\n");
	}
	puts("Нажмите любую клавишу для завершения...");
	_getch();
	return 0;
}
```

**Параметры – строки.** При разработке функций, работающих со строками, обычно используют прием, который применяется в стандартных функциях обработки строк: обеспечение возможности вызова функций как процедур и как функций. Это достигается тем, что адрес результирующей строки дублируется и возвращается еще и как результат функции.

Рассмотрим несколько примеров.

**Пример 4.4.** Написать подпрограмму удаления «лишних» пробелов. Лишними при этом считать многократные пробелы между словами и пробелы перед началом и после завершения предложения. Функция получает в качестве входного параметра константную строку, преобразует ее и возвращает в качестве результата преобразованную строку и указатель на строку-результат.

Описание функции:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
char * strdel(const char * tstring,char * trez)
{
	char *ptr;
	strcpy(trez,tstring);
	while( (ptr=strstr(trez,"  "))!=NULL ) strcpy(ptr,ptr+1);
	return trez;
}
```

```
int main(int argc, char* argv[])
{
	setlocale(0,"russian");
	char st[40],st2[40],*ptr2;
	puts("Введите строку: слова и пробелы:");
	gets(st);
	puts("Исходная строка:");
	puts(st);
	strdel(st,st2);           // вызов подпрограммы как процедры
	puts("Полученная строка 1");
	puts(st2);
	printf("Полученная строка 2:\n");
	ptr2=new char [40];
	puts(strdel(st,ptr2)); // вызов подпрограммы как функции
	puts("Нажмите любую клавишу для завершения...");
	_getch();
	return 0;
}
```

**Пример 4.5.** Написать подпрограмму нахождения максимального слова строки. Адрес этого слова возвращается также как результат функции.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
char * maxworld(const char * s,char* slmax);
int main(int argc, char* argv[])
{
	setlocale(0,"russian");
	char st[80],maxsl[10];
	puts("Введите строку: слова и пробелы:");
	gets(st);
	printf("В строке слово ");
	// вызов подпрограммы как функции
	printf("""%s"" - с мах длиной.\n",maxworld(st,maxsl));
	maxworld(st,maxsl); // вызов подпрограммы как процедуры
```

```
	printf("В строке слово ");
	printf("""%s"" - с мах длиной.\n", maxsl);
	puts("Нажмите любую клавишу для завершения...");
	_getch();
	return 0;
}
char * maxworld(const char * s,char* slmax)
{
	char slovo[10];
	unsigned int i,j,dls,maxl;
	dls=0;slmax[0]='\0';maxl=0;j=0;
	for(i=0;i<=strlen(s);i++)
	{
		if ((s[i]==' ')||(s[i]=='\0'))
		{
			slovo[j]='\0';
			if (dls>maxl)
			{
				maxl=dls;
				strcpy(slmax,slovo);
			}
			slovo[0]='\0';   j=0;  dls=0;
		}
		else
		{
			dls++;
			slovo[j++]=s[i];
		}
	}
	return slmax;
}
```

**3. Параметры-структуры.** В отличие от массивов и строк переменная типа «структура» не является указателем, поэтому если структура передается в качестве параметра, то для нее действуют те же правила, как и для скалярных значений. Так для передачи в подпрограмму параметров типа «структура», значения которых необходимо вернуть в вызыва-

ющую программу, необходимо использовать ссылки или указатели. В случае, когда изменение полей структуры не требуется, рекомендуется для исключения лишнего копирования всей структуры передавать её по ссылке со спецификатором const.

**Пример 4.6.** Дан массив целых чисел на 10 элементов. Данные о массиве объединены в структуру `massive`, содержащую 3 поля: массив, его текущий размер и сумму элементов. Написать подпрограмму, получающую структуру `massive` в качестве параметра, вычисляющую сумму элементов массива и возвращающую эту структуру, как результат с вычисленной суммой элементов.

Реализовать передачу изменяемой структуры в подпрограмму можно с использованием указателя или ссылки. Результат будет одинаков, а вот синтаксис описания и вызова подпрограммы будут отличаться.

*Вариант 1*. Использование указателя на структуру в качестве параметра функции:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
struct mas
{
     int n, a[10], sum;
};
int summa(struct mas *x);
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int i;  struct mas massive;
     puts("Введите количество элементов:");
     scanf("%d",&massive.n);
     puts("Введите элементы:");
     for (i=0;i<massive.n;i++) scanf("%d",&massive.a[i]);
     printf("Сумма элементов=%4d.\n",summa(&massive));
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

```
int summa(struct mas *x) // передается указатель на структуру
{
     int i,s=0;
     for (i=0;i<x->n;i++)    s+=x->a[i]; // суммирование элементов
     x->sum=s;                // запись суммы элементов в поле структуры
     return s;
}
```

*Вариант 2.* Использование указателя на структуру в качестве параметра функции.

При использовании ссылки на структуру изменится текст функции и ее вызов.

Текст функции:

```
int summa(struct mas &x) // передается ссылка на структуру
{
     int i,s=0;
     for (i=0;i<x.n;i++)    s+=x.a[i]; // суммирование элементов
     x.sum=s;                // запись суммы элементов в поле структуры
     return s;
}
```

Вызов: `printf("Сумма элементов=%4d.\n",summa(massive));`

Однако следует понимать, что при использовании массива структур имя массива является указателем, который хранит адрес структуры.

**Пример 4.7.** Программа определяет сумму элементов каждой из трех структур, записывает эту сумму в соответствующее поле структуры и определяет общую сумму.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
struct mas
{
     int n, a[10], sum;
};
int summa(struct mas x[]);
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int i,k;
     struct mas massive[3];
```

```
    for (k=0;k<3;k++)
    {
        puts("Введите количество элементов:");
        scanf("%d",&massive[k].n);
        puts("Введите элементы:");
        for (i=0;i<massive[k].n;i++)
            scanf("%d",&massive[k].a[i]);
    }
    printf("Сумма элементов=%4d.\n",summa(massive));
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
int summa(struct mas x[])   // в функцию передается указатель на массив
{
    int i,k,s,s1=0;
    for (k=0;k<3;k++,x++)      /* x++ - осуществляет переадресацию на
                                  следующий элемент структуры */
    {
        for (s=0,i=0;i<x->n;i++)
        s+=x->a[i];
        x->sum=s;
        s1+=s;
    }
   return s1;
}
```

#### 4.1.3 * Рекурсивные функции

Рекурсивная подпрограмма подразумевает организацию вычислений, при которой процедура или функция обращается к самой себе.

Рассмотрим несколько примеров.

**Пример 4.8.** Вычисление наибольшего общего делителя. Линейная рекурсия

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

```
#include <conio.h>
int nod(int a,int b)
{
     if(a==b) return a;    // базисное утвержление
     else
     {
          if (a>b) return nod(a-b,b);    // рекурсивное утверждение
          else  return nod(a,b-a);       // рекурсивное утверждение
     }
}
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int a,b;
     puts("Введите два целых числа:");
     scanf("%d %d",&a,&b);
     printf("\nНаибольший общий делитель %5d  %5d = %5d\n",
                                   a,b,nod(a,b)); // вызов функции
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

**Пример 4.9.** Вычисление n-го числа Фибоначчи. Древовидная рекурсия

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int fib(int n)
{
     if((n==1)||(n==2)) return 1; // базисное утверждение
     else
     {
          return fib(n-1) + fib(n-2); //  рекурсивное утверждение
     }
}
```

Оглавление

```
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int n;
     puts("Введите номер элемента Фибоначчи:");
     scanf("%d",&n);
     printf("\n %7d Номер =",n);
     printf("%10d\n",fib(n));   // Вызов рекурсивной функции
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

### 4.1.4 * Дополнительные возможности функций С++

При создании С++ разработчики несколько расширили возможности по организации функций по сравнению с теми, которые использовались в Си. Так появились подставляемые функции, параметрическая перегрузка функций и функции с параметрами, задаваемыми по умолчанию.

**1. Подставляемые функции.** *Вставляемыми* или *подставляемыми* функциями в С++ называются функции, код которых вставляется в то место программы, где они вызываются. Такие функции используют для сокращения времени вызова подпрограммы, так как в этом случае не выполняется процедура вызова функции и обратной передачи управления. Естественно подставляемые функции не должны быть большими, а то используемый прием потеряет смысл.

Для обозначения подставляемых функций используется служебное слово `inline`, например функция, возвращающая абсолютное значение параметра:

```
inline int abs(int a) {return a>0?a:-a;)}
```

Традиционно на использование подставляемых функций накладываются ограничения. Такая функция:

- не должна содержать циклов;
- не может быть рекурсивной или виртуальной;
- не должна вызываться более одного раза в выражении;
- не должна вызываться до своего определения (а не до объявления, как обычная функция).

Если вставка не возможна, то служебное слово inline игнорируется, функция компилируется независимо, а при вызове используется стандартный механизм подключения.

В процессе оптимизации компилятор Microsoft Visual C++ самостоятельно принимает решение о вставке или вызове объявленных inline функций, поэтому он никаких предупреждающих сообщений не выдает.

**2. Переопределяемые функции.** В C++ функции различаются по списку, количеству и типам параметров. Поэтому было разрешено определять несколько вариантов одной и той же функции с одинаковыми именами, но с разными списками параметров. При вызове, компилятор по списку аргументов определяет нужный аспект функции и вызывает требуемую ее реализацию.

```
int lenght(int x,int y){return sqrt(x*x+y*y);}
int lenght(int x,int y,int z){return sqrt(x*x+y*y+z*z);}
```

**3. Параметры функции, принимаемые по умолчанию.** C++ предоставляет возможность при определении функции присваивать значения некоторым параметрам. Эти значения параметры будут принимать по умолчанию, если при вызове функции соответствующие параметры не будут указаны. Однако такие параметры должны располагаться в конце списка параметров, поскольку пропускать параметры при вызове функции не разрешается.

Значение по умолчанию задается с помощью синтаксической конструкции, очень похожей на инициализацию переменной, например:

```
void InitWindow(int xSize=80,int ySize=25,int barColor=BLUE,
                                   int FrameColor=CYAN){...}
```

Теперь функцию можно вызвать, пропуская часть параметров, например:

```
InitWindow(); // все параметры берутся по умолчанию
InitWindow(20,10); // меняются размеры окна, остальные – по умолчанию
```

Но, если нужно изменить, например, цвет, оставив остальные параметры неизменными, то все предыдущие параметры следует повторить.

```
InitWindow(80,25,GREEN); // меняем цвет рамки окна, остальные –
                         // по умолчанию
InitWindow(80,25,BLUE,GREEN); // меняем цвет фона окна, остальные –
                              // по умолчанию
```

### *4.2 Модули C++*

Среда Visual C++ позволяет создавать и отлаживать программы, использующие не только стандартные, но и пользовательские библиотеки подпрограмм – модули. Модуль C++ обычно включает два файла: заголовочный файл с расширением «.h» и файл реализации с расширением «.cpp».

*Заголовочный файл* играет роль интерфейсной секции модуля. В него помещают объявление экспортируемых ресурсов модуля:

- прототипы (заголовки) процедур и функций,
- объявление переменных, типов и констант.

Заголовочный файл подключают командой `#include "<Имя модуля>.h"`, записываемой в файле реализации программы или другого модуля, если они используют ресурсы описываемого модуля.

*Файл реализации* представляет собой секцию реализации модуля. Он должен содержать команды подключения используемых модулей, описания экспортируемых процедур и функций, а также объявление внутренних ресурсов модуля. Файл реализации подключается командой `#include "<Имя модуля>.cpp"`.

При создании первый (главный) файл проекта уже содержит заготовку основной функции программы – функции `main()`. Для создания файлов модуля и добавления их к проекту необходимо вновь вызвать многошаговый Мастер заготовок. Это делается с использованием команды меню File/New. Выполнение этой команды при открытом проекте вызовет открытие окна Мастера заготовок на вкладке Files, на которой необходимо выбрать тип файла, добавляемого к проекту.

**Пример 4.10.** Разработать модуль для нахождения наибольшего общего делителя для двух целых чисел.

*Файл Mod.h:*

```
#ifndef modh_20100810
#define modh_20100810
int nod(int a,int b);
#endif
```

Дополнительные команды препроцессора в файле mod.h позволяют исключить повторную компиляцию текста программы при многократном подключении заголовочного файла (см. раздел 6.3).

Имя переменной препроцессора `modh_20100810` – уникальный идентификатор, который точно не встречается в других библиотеках. Для получения этого имени в примере для `mod.h` использована следующая схема:

<ИМЯФАЙЛА><РАСШИРЕНИЕ>_<ГОД><МЕСЯЦ><ДЕНЬ (создания)>.

*Файл Mod.cpp:*

```
#include "Mod.h"
int nod(int a,int b)
{
     while (a!=b)
          if (a>b) a=a-b; else b=b-a;
     return a;
}
```

*Файл main.cpp:*

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include "mod.h"
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int a=18,b=24,c;
     c=nod(a,b);
     printf("Наибольший общий делитель=%d\n",c);
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

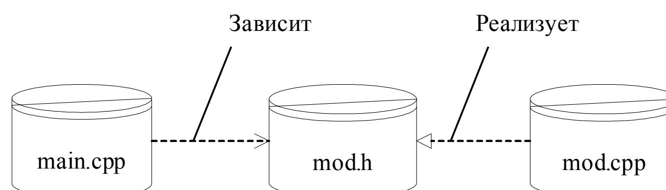



**Рисунок 4.1** – Диаграмма взаимодействия модуля и программы

### *4.3 * Средства создания универсальных подпрограмм*

При проектировании сложной программы, разработчик стремиться создать как можно более универсальные подпрограммы, чтобы сократить их количество и общий объем программы. Для этого C++ представляет определенные возможности. Ниже рассмотрены некоторые из них.

#### 4.3.1 Параметры – многомерные массивы неопределенного размера

Как было показано в разделе в C++ осуществляется контроль размеров массива, начиная со второй размерности, что ограничивает применение подпрограмм, делая их не универсальными, зависящими от размеров массивов.

В этом случае рекомендуется использовать вспомогательные массивы указателей на одномерные массивы, которые в свою очередь могут быть массивами указателей. По каждой размерности массив является одномерным, и по правилам C++ его размерность может быть опущена в спецификации формальных параметров. Такой подход позволяет в теле функции обрабатывать многомерные массивы с изменяющимися размерами.

**Пример 4.10.** Разработать подпрограмму, независящую от размерности матрицы, которая заменяет в матрице все отрицательные элементы нулевыми.

Дополним матрицу вектором указателей на ее строки (см. рисунок 4.2):

```
float B[3][4]={1.2,-4.9 ,5.0,-8.1,
               -3,6.1,-8.5,9.6,
               3.3,-6.7,-1.2,7.8};
float *ptr[]={&B[0],&B[1],&B[2]};
```

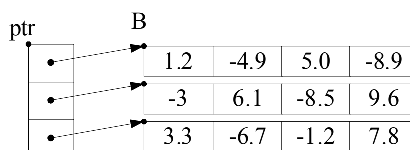



**Рисунок 4.2** – Структура матрицы с дополнительным массивом

Адреса начала строк, получаемые посредством оператора взятия адреса «&», не типизированы, следовательно, для сохранения этих адресов в типизированных переменных массива их необходимо явно преобразовать к типу `float *`, например:

`(float *)&B[0].`

Размерность матрицы будем передавать через специальные параметры n и m:

```
void pereform(int n,int m, float *p[]);
```

Полный текст программы:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include "mod.h"
void pereform(int n,int m,float * p[])
{
     for(int i=0;i<n;i++)
     for(int j=0;j<m;j++) if (p[i][j]<0) p[i][j]=0;
}
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float B[3][4]={1.2,-4.9 ,5.0,-8.1,
                    -3,6.1,-8.5,9.6,
                    3.3,-6.7,-1.2,7.8};
     float *ptr[]={(float*)&B[0],(float *)&B[1],(float *)&B[2]};
     pereform(3,4,ptr);
     puts("Результирующая матрица:");
     for(int i=0;i<3;i++)
     {
          for(int j=0;j<4;j++)
               printf("%5.2f",B[i][j]);
          printf("\n");
     }
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
    }
```

**Пример 4.11.** Разработать две подпрограммы, формирующую матрицу переменного размера и меняющую отрицательные элементы матрицы на их абсолютное значение. Тестировать эти подпрограммы.

Для увеличения степени универсальности подпрограмм объявим матрицу с дополнительным вектором адресов строк (см. рисунок 4.3)

Указатель `int **M` указывает на массив указателей `int *M`, каждый из элементов которого, в свою очередь адресует одномерный массив элементов целого типа.

Так как размеры массивов нигде не указаны, память под массивы статически не выделяем. Выделение памяти осуществим во время выполнения программы, когда размеры массива станут известны.

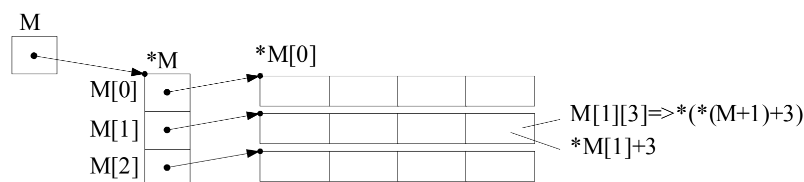



**Рисунок 4.3** – Структура данных примера

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
// Функция формирования и ввода матрицы
int **matr(int &k,int &p)
{
     int **m,i,j;
     printf("Введите размеры матрицы:\n");
     scanf("%d %d",&k,&p);
     printf("Введите %2d строки по %2d елементов:\n",k,p);
     m=new int *[k];         // выделение памяти под массив указателей
     for (i=0;i<k;i++)
     {
          m[i]=new int[p]; // выделение памяти под каждую строку
          for (j=0;j<p;j++)scanf("%3d",*(m+i)+j);
     }
     return m; // возврат сформированной матрицы
}
// Функция замены отрицательных элементов абсолютной величиной
void sortmas(int **m,int n,int k)
{
     int i,j;
     for(i=0;i<n;i++)
          for(j=0;j<k;j++)
               if (m[i][j]<0) m[i][j]=abs(m[i][j]);
}
```

```
// Основная программа
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int n,l,**mat,i,j;
     mat=matr(n,l);  // вызов функции формирования матрицы
     printf("\n===== Введеная матрица =====\n");
     for(i=0;i<n;i++)
          for(j=0;j<l;j++)
               printf("%4d%c",mat[i][j],(j==l-1)?'\n':' ');
     sortmas(mat,n,l);  // вызов функции изменения матрицы
     printf("\n=== Результирующая матрица ==\n"); for(i=0;i<n;i+
+)
          for(j=0;j<l;j++)
               printf("%4d%c",mat[i][j],(j==l-1)?'\n':' ');
     // удаление матрицы и освобождение памяти
     for (i=0;i<n;i++) delete[] mat[i];
     delete[] mat;
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

### 4.3.2 Параметры-функции

Как уже отмечалось ранее, функция характеризуется типом возвращаемого значения, именем и сигнатурой. Имя функции – это указатель, хранящий ее адрес, который может быть присвоен другому указателю. Однако при объявлении для нового указателя должен быть задан тот же тип, что и возвращаемое функцией значение, и тот же список параметров с точностью до типов формальных параметров (имена параметров могут различаться).

Указатель на функцию объявляется следующим образом:

**<Тип функции>(* <Имя>)(<Спецификация параметров>);**

Например:

```
int (*ptrfun)(int,int);
```

При объявлении указатель на функцию может быть инициализирован, но в качестве значения должен быть указан адрес функции, тип и сигнатура которой соответствуют определяемому указателю, например:

Описание функций:

```
char f1(char){…}
char f2(int){…}
void f3(float){…}
int f4(float){…}
int f5(int){…}
```

Объявление указателей на функции:

```
void (*ptr1)(float)=f3; // инициализированный указатель
int (*ptr2)(int);
char (*ptr3)(int);
```

Присваивание указателей:

```
ptr2=f5; ptr3=f2; // корректное
prt2=f4; ptr3=f1; // некорректное: несоответствие типов или спецификаций
```

**Пример 4.12.** Разработать подпрограммы выполнения элементарных операций.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int add(int n,int m) {return n+m;}
int sub(int n,int m) {return n-m;}
int mul(int n,int m) {return n*m;}
int div(int n,int m) {return n/m;}
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     int (*ptr)(int,int);  // указатель на функцию
     int a=6, b=2; char c='+';
     while (c!=' '){
          printf("%d%c%d=",a,c,b);
          switch (c)
          {
               case '+': ptr=add; c='-';break;
               case '-': ptr=sub; c='*';break;
```

```
                case '*': ptr=mul; c='/';break;
                case '/': ptr=div; c=' ';
            }
            printf("%d\n",a=ptr(a,b));     // вызов функции по указателю
        }
        puts("Нажмите любую клавишу для завершения...");
        _getch();
        return 0;
}
```

**Пример 4.13.** Написать подпрограмму вычисления значения интеграла произвольных функций одной переменной на отрезке a,b с точностью eps.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
float (* funuk)(float);  // указатель на функцию
// Функция расчета интеграла методом прямоугольников
float integral(float(*funuk)(float),float a,float b,float eps)
{
        int i,n=5,k=0; float s1,s2=1.0e+10,x,d;
        d=(b-a)/n;
        do
        {
            s1=s2;  s2=0;n=n*2; d=d/2;
            x=a;  k++;
            for(i=1;i<=n;i++)
            {
                s2=s2+funuk(x);
                x=x+d;
            }
            s2=s2*d;
        }
        while(fabs(s2-s1)>eps);
        return s2;
}
```

```
// Описание функций, для которых считается интеграл
float f1(float x){return x*x-1;}
float f2(float x){return 2*x;}
int main(int argc, char* argv[])
{
     setlocale(0,"russian");
     float a,b,eps;
     puts("Введите a,b,eps для функции y=x^2-1:");
     scanf("%f %f %f",&a,&b,&eps);
     //вызов функции расчета интеграла для функции f1
     printf("Значение интеграла=%10.5f\n",integral(f1,a,b,eps));
     puts("Введите a,b,eps для функции y=2*x:");
     scanf("%f %f %f",&a,&b,&eps);
     // вызов функции расчета интеграла для функции f2
     printf("Значение интеграла=%10.5f\n",integral(f2,a,b,eps));
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

Результаты работы программы (вводимые данные выделены полужирным):

```
Введите a,b,eps для функции y=x^2-1:
1 2 0,001
Значение интеграла   1,33285
Введите a,b,eps для функции y=2*x:
1 2 0,001
Значение интеграла=   2,99928
Нажмите любую клавишу для завершения...
```

**Контрольные вопросы к главе 4**

1. Что такое объявление и описание функции и чем они отличаются? Что такое прототип функции?

Ответ.

2. Назовите способы передачи данных в функции.

Ответ.

3. Как вернуть результат работы функции? Какую роль в этом играет оператор возврата?

Ответ.

4. Что такое класс памяти и что он определяет? Какие классы памяти Вы знаете?

Ответ.

5. Чем отличаются статические переменные от автоматических?

Ответ.

6. Как передать в подпрограмму параметр массив или строку?

Ответ.

7. Что такое модуль в C++ и какова его структура?

Ответ.

8. Что такое заголовочный файл и как его подключить к программе?

Ответ.

9. Как передать в подпрограмму многомерный массив произвольного размера?

Ответ.

10. Что такое указатель на функции и как его можно использовать при работе подпрограмм?

Ответ.

# 5 ФАЙЛОВАЯ СИСТЕМА

## *5.1 Механизм выполнения операций ввода/вывода. Типы файлов*

*Файл* – последовательность информационных записей. Такая последовательность может быть записана на дисках *внешней памяти* (магнитных, оптических и т.д.) или появляться в результате обмена информацией с внешними устройствами (клавиатура, устройство вывода на печать, дисплей и т.д.). Соответственно различают *дисковые файлы* и *логические устройства.*

Логически файлы представляют собой длинную строку байтов, которая разбивается на части (записи) в зависимости от типа файла. В C++ определены два типа файлов: текстовые и двоичные.

*Текстовые файлы* состоят из символьных строк переменной длины, заканчивающихся маркером «Конец строки» (двухбайтовый код: 13, 10).

*Двоичные файлы* представляют собой последовательность компонентов фиксированного размера, например, целых чисел, структур и т. д., записанных в файл *без преобразования к символьному виду.*

Операции ввода/вывода с файлами, как правило, выполняются через специальные области оперативной памяти – *буферы*. При первом обращении к операции ввода файловая система переписывает в буфер блок информации размером с буфер, и затем ввод осуществляется уже из буфера. При выводе – в буфере накапливается выводимая информация, а операция вывода выполняется при заполнении буфера. Это позволяет существенно сократить время выполнения операций ввода/вывода с медленными устройствами. Размер буфера, определяемый типом устройства, вместе с адресом буфера и прочей необходимой информацией хранятся в специальной таблице, формат которой определен в структуре типа **FILE**.

## *5.2 Объявление, открытие и закрытие файлов*

Программирование операций ввода/вывода с файлами начинается с *объявления файловой переменной*:

`FILE` *<Файловая переменная>; // объявляется указатель на таблицу `FILE`

Оглавление

Перед работой файл должен быть открыт. При открытии *файла* выделяется память под таблицу, на которую указывает файловая переменная, и частично заполняются ее поля. Операцию выполняет специальная функция:

<Файловая переменная>=`fopen`(<Имя файла>,<Операция [+] [Тип]>);

где <Имя файла> – строковая переменная или константа – полное имя файла с путем – при указании имени файла без пути файл ищется только в текущем каталоге;

<Операция[+]> – кодируется следующим образом:

r – ввод из существующего файла;

w – вывод с очисткой файла или создание нового файла для вывода;

a – добавление к существующему файлу или создание файла для вывода;

r+ – ввод/вывод в существующий файл;

w+ – ввод/вывод в существующий или создание нового файла для ввода/вывода;

a+ – ввод/добавление к существующему или создание файла для ввода/вывода.

<тип> – тип файла:

t – текстовый файл (принимается по умолчанию);

b – двоичный файл.

Примеры.

а) объявление и открытие файла на текущем диске для записи:

```
FILE *f;
f=fopen("abc.txt","w"); // файл в текущем каталоге
```

б) открытие файла для чтения с проверкой существования:

```
if((f=fopen("f:\\iva\\text.txt","r"))!=NULL) ...
```

Открытый файл обязательно должен быть закрыт. При закрытии файла вывода оставшиеся записи из буфера переписываются в файл, и если файл новый, то формируется соответствующий элемент оглавления. Поэтому, прежде чем переименовывать или удалять файл, его обязательно нужно закрыть:

```
fclose(<Файловая переменная>); // функция возвращает 0, если закрытие
                               // прошло успешно, и -1, если обнаружена ошибка.
```

### *5.3 Работа с файловым указателем*

Особую роль при программировании операций с файлом играет *файловый указатель,* который определяет, в какое место файла выводится запись или откуда она вводится. При последовательной обработке файла файловый указатель является счетчиком уже

прочитанных или записанных байт. Имеются специальные функции, которые позволяют работать с файловым указателем:

а) определение значения файлового указателя:

```
long ftell(FILE *stream); // возвращает текущее значение файлового
                          // указателя;
```

б) установка файлового указателя на начало файла:

```
int rewind(FILE *stream); // файловый указатель устанавливается
                          // на начало файла (см. пример в разделе 5.4.3);
```

в) установка файлового указателя в указанное место:

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

Параметр `whence` определяет, откуда отсчитывается смещение указателя `offset`: 0 – от начала файла; 1 – от текущей позиции; 2 – от конца файла (см. пример в разделе 5.4.1).

### *5.4 Текстовые файлы. Стандартные текстовые файлы*

Текстовый файл представляет собой последовательность символьных записей переменной длины. Каждая программа, написанная на Си или С++, по умолчанию имеет доступ к трем текстовым файлам:

`stdin` – файловая переменная стандартного файла ввода, связанного с клавиатурой;

`stdout` – файловая переменная стандартного файла вывода, связанного с дисплеем;

`stderr` – файловая переменная стандартного файла вывода сообщений об ошибках, связанного с дисплеем.

Первые два файла имеют тип «логическое устройство» и могут быть переназначены при использовании в командной строке специальных записей вида:

< – для стандартного файла ввода;

> – для стандартного файла вывода с перезаписыванием;

>> – для стандартного файла вывода с добавлением.

Стандартные файлы имеют номера 0, 1, 2, которые можно использовать вместо имен файлов соответственно.

Например:

```
cat.exe  <file1.dat  >file2.dat
cat2.exe  <file3.dat >>file2.dat 2>&1
```

В данном примере для программы cat1.exe ввод данных будет осуществляться из файла `file1.dat`, а вывод – в файл `file2.dat`, для программы cat2.exe ввод – из файла `file3.dat`, а вывод – дописываться в конец файла `file2.dat`, вместе с потоком ошибок (`2>&1`) (если они будут).

Программирование операций ввода/вывода для текстовых файлов выполняется с использованием специальных функций.

### 5.4.1 Ввод/вывод символов

Ввод символа:

`int getc(FILE *stream);` // введенный символ возвращается как результат функции. При достижении конца файла функция возвращает константу `EOF`.

Вывод символа:

`int putc(int c,FILE *stream);` // выводимый символ передается через параметр `c`, он же возвращается в качестве результата функции.

**Пример 5.1.** Посимвольный ввод из тестового файла и вывод считанных данных на экран.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[ ])
{
     setlocale(0,"russian");
     FILE *in;
     unsigned char ch;
     char filename[20];
     printf("Введите имя файла:");
     scanf_s("%s",filename,19);
     if ((in=fopen(filename,"r"))!=NULL)
     {
          while ((ch=getc(in))!=EOF) putc(ch,stdout);
          fclose(in);
     }
     else puts("Файл с указанным именем не существует.\n");
     puts("Нажмите любую клавишу для завершения...");
```

```
    _getch();
    return 0;
}
```

В рассмотренном примере вывод на экран осуществляется как вывод в стандартный файл `stdout`. Однако обычно для работы со стандартными файлами ввода/вывода используют специальные функции, рассмотренные в разделе 1.10, но возможно и использование стандартных файлов:

```
getchar( )  == getc(stdin)
putchar(ch) == putc(ch,stdout)
```

**Пример 5.2.** Демонстрация «прямого» доступа к компонентам файла.

Программа, чередующая печать символов из файла с начала и с конца, например, если в файле хранится строка `АВСД`, то должно быть напечатано `АДВССВДА`.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main()
{
    setlocale(0,"russian");
    FILE *f;  long offset=0L;    char ch;
    f=fopen("text.txt","r");
    if(f){
        while (!fseek(f,offset++,0) && (ch=getc(f))!=EOF){
            putchar(ch);
            if (!fseek(f,-(offset+2),2)) putchar(getc(f));
        }
        putchar('\n');
        fclose(f);
    }
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Операции ввода/вывода при использовании функций ввода/ вывода символов выполняются с использованием буферов, а при вводе – выполняется «эхо»-вывод.

Эти функции не следует путать с функциями `getch()` и `putch(c)`, прототипы которых определены в `conio.h` и которые выполняют операции ввода с клавиатуры и вывода на экран без использования буферов, причем при вводе с клавиатуры отсутствует «эхо»-вывод.

Использование буферов при выполнении операций ввода/вывода иногда приводит к плохо предсказуемым эффектам. Рассмотрим фрагмент:

```
putchar(getchar());
```

Ожидается, что при его выполнении программа должна тут же выводить введенный символ, но в действительности при выполнении программы вывод произойдет после завершения ввода с помощью нажатия клавиши `Enter`. Причем до нажатия клавиши `Enter` может быть введено несколько (до 128 – размер буфера ввода с клавиатуры) символов.

Соответственно будет работать и фрагмент:

```
while((n=getchar())!=EOF)putchar(n);
```

Ожидается, что программа должна посимвольно выводить введенные символы, однако в действительности вывод работает построчно (также из-за использования буферов).

Если в указанных фрагментах заменить функции на `getch()` и `putch()`, то фрагменты будут работать, как ожидалось.

### 5.4.2 Ввод/вывод строк

Ввод строки:

**`char *fgets(char *s,int n,FILE *stream)`**`;` // вводит строку длиной до n-1 или до маркера «Конец строки» в буфер по адресу **s.** Возвращает дубликат указателя на введенную строку. При достижении конца файла возвращает `NULL`.

**Пример 5.3.** Вывод на экран содержимого текстового файла.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[ ])
{
     setlocale(0,"russian");
     FILE *f1;
     char string[80];
     f1=fopen("main.cpp","r");
```

```
    if(f1)
    {
        while (fgets(string, 80, f1)!=NULL) puts(string);
        fclose(f1);
    }
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Вывод строки:

**int fputs(const char *s,FILE *stream)**; // выводит строку из буфера по адресу **s**. Если вывод прошел нормально, то возвращает 0, иначе возвращает -1.

**Пример 5.4.** Создание текстовый файл из 10 строк, содержащих буквы ABCD.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[])
{
    setlocale(0,"russian");
    FILE *f;  int n;    char *s="ABCD";
    f=fopen("test.dat","w");
    if(f){
        for (n=0;n<10;n++){
            fputs(s,f);        // вывод строки в файл
            fputs("\n",f);     // вывод маркера «Конец строки»
        }
        fclose(f);
    }
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

### 5.4.3 Форматный ввод/вывод

Операции форматного ввода/вывода предполагают, что при выполнении этих действий осуществляется: при вводе – преобразование символьного представления данных во внутреннее машинное представление в соответствии с заданным форматом и, наоборот, при выводе – преобразование из внутреннего формата в символьный. Символьное представление данных в отличие от внутреннего может быть воспринято человеком, и преобразование к символьному виду, как правило, сопутствует выполнению операций ввода информации от пользователя или выводу результатов пользователю.

Форматный ввод:

**`int fscanf(FILE *stream,const char *format[,adress,...]);`**

Функция работает аналогично `scanf` (и `scanf_s`), но при этом осуществляется ввод не с клавиатуры, а из указанного файла. Возвращает количество введенных полей. Если при выполнении операции достигается конец файла, то функция возвращает `EOF`.

Форматный вывод:

**`int fprintf(FILE *stream,const char *format[,argument,...]);`**

Функция работает аналогично `printf`, но при этом вывод осуществляется в указанный файл.

**Пример 5.5.** Разработать программу, которая вводит данные: размер матрицы и саму матрицу из текстового файла, затем выводит в другой текстовый файл матрицу и суммы элементов строк.

Пример исходного файла:

```
3 4          - размеры матрицы
1 2 3 4      - 0-я строка
2 3 4 5      - 1-я строка
3 4 5 6      - 2-я строка
```

Текст программы:

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[])
{
     setlocale(0,"russian");
     int mas[10][10],i,j,n,m,s;
     char namein[20],nameout[20];
     FILE *f1,*f2;
```

```
    puts("Введите имя файла исходных данных:\n");
    gets_s(namein,sizeof(namein)-1);
    puts("Введите имя файла результата:\n");
    gets_s(nameout,sizeof(nameout)-1);
    f1=fopen(namein,"r");          // открытие файлов
    f2=fopen(nameout,"w+");
    if( f1 && f2 ){
        fscanf_s(f1,"%d %d",&n,&m);
        puts("Исходная матрица: Суммы:");
        for (i=0;i<n;i++){
            for (j=0,s=0;j<m;j++){
                fscanf_s(f1,"%d",&mas[i][j]);
                printf("%4d",mas[i][j]);
                fprintf(f2,"%4d",mas[i][j]);
                s+=mas[i][j];
            }
            printf("%6d\n",s); fprintf(f2,"%6d\n",s);
        }
    }
    _fcloseall();                  // закрытие всех файлов
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

Резльтирующий файл для приведенного выше исходного файла:

```
  1   2   3   4    10
  2   3   4   5    14
  3   4   5   6    18
```

## *5.5 Двоичные файлы*

В двоичных файлах информация хранится во внутренних форматах без преобразования к символьному виду. Запись в файл и чтение из файла осуществляется через специальный буфер, причем программист должен указывать количество обрабатываемых байт. Разделители компонентов в файле отсутствуют. Если в двоичном дисковом файле все компо-

ненты одинаковой длины, то можно вычислить значение файлового указателя для каждого компонента, и, соответственно, осуществить *прямой доступ* к элементам.

Двоичный файл, содержащий текстовую информацию, практически ничем не отличается от текстового. Как уже говорилось в разделе 5.1, принципиально возможно создать файл как текстовый, а затем прочитать его как двоичный, элементами которого являются строки (в этом случае необходимо знать их длину) или символы. При этом нами будут прочитаны также символы, образующие маркер «Конец строки» (см. пример 30). Соответственно, и наоборот, можно создать файл как двоичный, а прочитать – как текстовый.

*Примечание*. Двоичный файл, содержащий несимвольную информацию, при просмотре текстовыми редакторами выглядит как нагромождение букв и символов псевдографики. Создать такой файл текстовыми редакторами нельзя.

При программировании операций ввода/вывода с двоичными файлами используют специальные функции, которые работают с двоичными образами данных, сгруппированными в блоки.

Функция ввода из двоичного файла:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *stream);
```

Функция вводит информацию из файла в буфер по адресу `ptr` объемом `size*n`, где `size` – размер вводимого блока, а `n` – количество блоков**.** В качестве результата функция возвращает действительно считанное количество блоков. При достижении конца файла функция возвращает 0, а при обнаружении ошибок – -1.

Функция вывода в двоичный файл:

```
size_t fwrite(void *ptr,size_t size,size_t n,FILE *stream);
```

Функция выводит информацию из буфера по адресу `ptr` объемом size*n.

Указанные функции в основном используются в двух вариантах:

а) при работе со структурами:

```
fread  (&myrec,sizeof(myrec),1,f1);
fwrite (&myrec,sizeof(myrec),1,f1);
```

где `myrec` – переменная типа структура;

б) при работе с файлом в целом, когда обрабатываемая информация не детализируется, например, при копировании файла:

```
size_t bufSize=0x200;
char *buffer = new char[bufSize];
fread (buffer,1,bufSize,f2);
fwrite(buffer,1,bufSize,f2);...
delete[] buffer;
```

Размер буфера при этом, как правило, задается кратным 512 байт.

**Пример 5.6.** Программа создания файла, содержащего ассортимент игрушек в магазине с указанием цены.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
struct toys
{
     char name[20];  float cost;
} toy;
int main(int argc,char *argv[ ])
{
     setlocale(0,"russian");
     FILE *f;  char s_cost[20];
     f=fopen("test.dat","w+b");
     if(f){
          while(puts("Введите название и стоимость:"),
               strcmp(gets(toy.name),"end")!=0)
          {
               toy.cost=atof(gets(s_cost));
               fwrite(&toy,sizeof(toy),1,f);
          }
          fclose(f);
     }
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

Результаты работы программы (вводимые данные выделены полужирным):

```
Введите название и стоимость:
```
**Кукла**

**307,67**

```
Введите название и стоимость:
Зайка
476,89
Введите название и стоимость:
end
Нажмите любую клавишу для завершения...
```

**Пример 5.7.** Чтение двоичного файла (компонент - структура).

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
struct toys
{
     char name[20];      float cost;
} toy;
int main(int argc,char *argv[ ])
{
     setlocale(0,"russian");
     FILE *f;
     f=fopen("test.dat","r+b");
     while(fread(&toy,sizeof(toy),1,f)>0)
          printf("Игрушка %s - цена %f\n",toy.name,toy.cost);
     fclose(f);
     puts("Нажмите любую клавишу для завершения...");
     _getch();
     return 0;
}
```

**Пример 5.8.** Программе создания текстового файл, который затем читается как двоичный, состоящий из компонентов размером байт.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[ ])
{
     setlocale(0,"russian");
     char c;   FILE *f;
```

```
    f=fopen("ddd.dat","w"); // открытие файла как текстового
    fputs("ABCDEF\n",f);    // запись строки в текстовый файл
    fclose(f);              // закрытие текстового файла
    f=fopen("ddd.dat","rb"); // открытие файла как двоичного
    while (fread(&c,1,1,f)!=0) printf("%x ",c); // побайтное чтение
    printf("\n");
    fclose(f);
    puts("Нажмите любую клавишу для завершения...");
    _getch(); return 0;
}
```

Результаты программа выводит в шестнадцатеричной системе счисления:

```
41 42 43 44 45 46 d a
Нажмите любую клавишу для завершения...
```

Последние две буквы – шестнадцатеричный маркер конца строки (13, 10).

### *5.6 Удаление и переименование файлов*

Для удаления файлов можно использовать одну из двух функций, названия которых являются синонимами:

**`int unlink(const char *filename)`**; // функция удаляет указанный файл;

**`int remove(const char *filename)`**; // функция удаляет указанный файл.

Переименование файлов осуществляется с помощью функции:

**`int remove(const char *oldname, const char *newname);`**.

**Пример 5.9.** Вставка фрагмента в середину текстового файла. Для осуществления вставки программа использует копирование в другой файл, который после удаления исходного файла получает его имя.

```
#include <locale.h>
#include <stdio.h>
#include <conio.h>
int main(int argc,char *argv[]){
    setlocale(0,"russian");
    int n,m;  FILE *f,*g;
    f=fopen("rand.dat","r");      // открытие исходного файла
    g=fopen("$$$$xxx.tmp","w");   // открытие рабочего файла
```

```
    for (n=0;n<5;n++) {                // цикл копирования начала файла
        fscanf(f,"%d",&m);
        fprintf(g,"%d\n",m);
    }
    for (n=0;n<4;n++) fprintf(g,"%d\n",n); // вставка фрагмента
    while ((n=fgetc(f))!=EOF) fputc(n,g);  // копирование остатка
    fcloseall();                           // закрытие обоих файлов
    unlink("rand.dat");                    // удаление исходного файла
    rename("$$$$xxx.tmp","rand.dat");      // переименование файла
    puts("Нажмите любую клавишу для завершения...");
    _getch();
    return 0;
}
```

## Контрольные вопросы к главе 5

1. Что такое файл, и какие типы файлов вы знаете?

Ответ.

2. Что такое файловая переменная и как ее объявит в программе на С++?

Ответ.

3. Как открыть файл на чтение или запись с контролем наличия файла на диске? Приведите пример.

Ответ.

4. Что такое файловый указатель? Каково его назначение?

Ответ.

5. Какие Вы знаете функции работы с файловым указателем?

Ответ.

6. Что такое стандартные текстовые файлы? Где они используются?

Ответ.

7. Каковы особенности работы с текстовыми файлами?

Ответ.

8. Что такое форматный ввод-вывод? Приведите основные функции работы с ним?

Ответ.

9. Что такое двоичные файлы и чем они отличаются от текстовых?

Ответ.

10. Какие Вы знаете функции работы с двоичными файлами?

Ответ.

Оглавление

# 6 ПРЕПРОЦЕССОР ЯЗЫКА С

Препроцессор используется для обработки текста программы *до ее компиляции* и вызывается автоматически при обращении к компилятору. Обработка заключается в выполнении специальных команд. Рассмотрим наиболее часто используемые команды препроцессора.

## *6.1 Команда #include*

Команда `#include` используется для включения в исходный текст программы файлов, содержащих прототипы стандартных функций и части исходных текстов программы. Данное средство языка позволяет хранить в отдельных файлах описания констант, типов и функций и затем использовать их в разных программах.

Команда встречается в двух вариантах:

`#include <Имя файла>` – в этом случае файл ищется в каталогах среды – эта форма используется при подключении стандартных файлов Си и С++;

`#include "Имя файла"` – в этом случае файл сначала ищется в текущем каталоге проекта, а потом уже в каталогах среды – такая форма используется при подключении файлов конкретного проекта.

## *6.2 Команды #define и #undef*

Команда `#define` используется для описания макроопределений – текстовых строк, вставляемых препроцессором вместо указанных имен (макрокоманд). Вставляемые фрагменты можно настраивать с использованием специально задаваемых параметров. Компиляция строк, как уже упоминалось ранее, выполняется после настройки и подстановки фрагментов.

Команда записывается в следующем виде:

**`#define` <Идентификатор>[(<Список параметров>)] <Строка>**

где <Идентификатор> – имя, используемое для идентификации строки (макрокоманда);

<Список параметров> – список идентификаторов, обозначающих заменяемые элементы строки – может отсутствовать;

<Строка> – строка-макроопределение, заменяющая имя в тексте программы.

Примеры:

а) `#define nil 0` // определяет замену идентификатора `nil` нулем

б) `#define N 5` // определяет замену идентификатора `N` числом 5

```
int main()
{
    int A[N][N];  ⇒  A[5][5]
  …
```

в) `#define MSG "Это строка - пример"` // определяет замену
// идентификатора `MSG` заданной строкой

г) `#define P(X) printf("X равен %d.\n",X)` // определяет замену
// идентификатора с параметром X на строку вывода данного параметра,
// например, вместо P(alf) будет вставлено printf("alf равно %d.\n",alf)

Следует осторожнее использовать форму с параметрами, так как в отличии от inline-функций, препроцессор осуществляет *формальную* подстановку параметров как символьных строк без проверки типов. Например, рассмотрим два варианта определения функции поиска максимального значения:

а) `inline int MAX(int X, int Y){return X>Y?X:Y;}`

б) `#define MAX(X,Y) (X>Y?X:Y)`

Вариант а не способен работать с типами, отличными от int, в то время как вариант б может принимать аргументы любых типов.

*Примечание.* Следует заметить, что подобное использование Си-команды #**define** для С++ не рекомендуется, поскольку С++ имеет другие механизмы, например шаблоны (template), позволяющие выполнять контроль типов:

```
template <typename T> T MAX(T x, T y){ return x>y?x:y;}
```

Вызов функции при использовании шаблона можно осуществлять либо с неявным указанием типа, например `MAX(1,4)`, либо если аргументы имеют разный тип, с явным указанием типа `MAX<float>(1.03, 4)`.

Формальные подстановки, выполняемые посредством #define, могут привести к неожиданным ошибкам, например:

```
#define ABS(X) (X>0?X:-X)
void main()
{
  int i,j; … i= ABS(j++); // Ошибка! j++ будет выполнено дважды
}
```

Если подстрока в команде `#define` не указана, то она получает значение «пусто» и при обработке описанное таким образом имя макроса из текста удаляется. Однако соответствующая константа считается определенной и при проверке командой **`if defined`** (или **`ifdef`**) или **`if !defined`** (или **`ifndef`**) возвращает «Константа определена».

Команда `#undef` используется для отмены команды `#define`. Формат команды:

**#undef <Имя>**

После выполнения этой команды замена идентификатора на строку прекращается.

### *6.3 Команды условной компиляции*

Специальные команды препроцессора позволяют осуществлять условную генерацию текста программы. При этом используются две конструкции:

**#if <Константное выражение>**

**<Операторы языка и команды препроцессора>**

**[#else**

**<Операторы языка и команды препроцессора>]**

**#endif**

или

**#if <Константное выражение>**

**<Операторы языка и команды препроцессора >**

**#elif <Константное выражение>**

**<Операторы языка и команды препроцессора>**

**[#else**

**<Операторы языка и команды препроцессора>]**

**#endif**

При этом в обоих случаях сначала осуществляется проверка заданного константного выражения. Если результат этого выражения отличен от нуля, то в текст вставляются следующие за `#if` операторы языка и команды препроцессора. Если результат выражения равен 0, то в первом варианте в текст вставляются операторы языка и команды препроцессора, следующие за `#else`, а во втором варианте – осуществляется дополнительная проверка второго константного выражения. И если результат проверки второго выражения отличен от нуля, то в текст программы вставляются операторы языка и команды препроцессора, следующие за `#elif`, иначе в текст вставляются операторы языка и команды пре-

процессора, следующие за #else. Альтернатива #else в обоих вариантах может быть опущена.

Таким образом, команды условной генерации позволяют изменять исходный текст программы в зависимости от значений некоторых констант.

Команда #error используется для вывода сообщений об особенностях генерации.

Формат команды

**#error <Сообщение>**

Команды #ifdef и #ifndef используются как альтернативы соответственно:

#if defined(<Идентификатор>) и #if !defined(<Идентификатор>), где defined – специальная функция препроцессора, возвращающая 1, если заданный идентификатор определен командой #define и это определение не отменено командой #undef.

С использованием этих команд выполняется, например, защита от повторной компиляции подключаемых файлов:

```
#ifndef MyTerm_h        // если не определена константа
#define MyTerm_h        // определяем константу как пустую
… Подключение файлов и исходные тексты …
#endif
```

Первый раз константа не определена, и текст внутри фрагмента подключается. Однако уже при первом входе во фрагмент константа определяется посредством #define. Соответственно следующий раз константа уже будет определена, и фрагмент подключаться не будет.

Ниже приведены еще два примера использования условной макрогенерации:

а)
```
#if !define(MODEL)
  #error Building model not defined
 #endif
```

б)
```
#if defined(NEARPOINTERS)
       space=farcoreleft();
 #elif defined(FARPOINTERS)
       space=coreleft();
 #else
 #error Unsupported memory model
 #endif
```

### *6.4 Некоторые предопределенные макроопределения*

**DATE** – текстовая строка, содержащая дату компиляции в формате Mmm dd yyyy, где Mmm – название месяца, dd – число и yyyy – год.

**FILE** – текстовая строка, содержащая имя компилируемого файла, включая полный путь к нему.

**LINE** – номер текущей строки в десятичном формате. Может быть изменен директивой #line.

**TIME** – текстовая строка, содержащая время компиляции в формате hh:mm:ss, где hh – часы, mm – минуты, ss - секунды.

**TIMESTAMP** – текстовая строка, содержащая дату и время последнего изменения компилируемого файла, в формате Ddd Mmm Date hh:mm:ss yyyy, где Ddd – сокращение названия дня недели, Date – число.

**FUNCTION** – текстовая строка, содержащая название функции, внутри которой вставлено макроопределение.

**Пример.** Создание макрокоманды отладочной печати:

```
#ifdef _DEBUG
   #define DEBUG_OUTPUT \
      printf( "%s\t%s\t%d\n",__FILE__,__FUNCTION__,__LINE__ );
#else
      #define DEBUG_OUTPUT
#endif
```

При наличии такой макрокоманды в тексте программы, где необходим отладочный вывод, достаточно вставить строку вида:

```
DEBUG_OUTPUT("file error…");
```

чтобы было сгенерировано детальное сообщение об ошибке.

**STDC** – наличие макроопределения сообщает о том, что компилятор удовлетворяет требованию стандарта ANSI C.

**cplusplus** – определено в том случае, если компилируется программа на C++.

**Пример.** Подключение заголовочного файла **bdb.h**, написанного на языке Си, независимо от того, компилируется ли Си или C++ программа:

```
#ifdef __cplusplus
 extern "C"{
#endif
#include <bdb.h>
```

```
#ifdef __cplusplus
 };
#endif
```

*Примечание.* Необходимость отдельно обрабатывать Си-заголовочные файлы связана с тем, что в C и C++ используются различные схемы преобразования имен функций при компиляции. Поэтому библиотека, полученная компилятором Си, может быть подключена к C++ программе только в том случае, если имена функций совпадают, что и делает обеспечивает директива `"C"`.

**`DEBUG`** – определено в том случае, если компилируется отладочная версия программы.

**`MSC_VER`** – содержит версию компилятора. Microsoft Visual C++ .NET 2003 – **1310**, где 13 – номер версии, 1.0 - реализация. Visual C++ 2005 – 1400, Visual C++ 2008 - 1500.

**`WIN32`** – определено при компиляции программ для Win32 и Win64.

**`WIN64`** – определено при компиляции программ для Win64.

**Контрольные вопросы к 6 главе**

1. Что такое препроцессор C++?

Ответ.

2. Каково назначение команды #include?

Ответ.

3. Какая команда используется для описания макроопределений?

Ответ.

4. Что такое условная генерация?

Ответ.

5. Какие команды применяются для условной генерации программы?

Ответ.

6. Какие команды позволяют выполнить защита от повторной компиляции подключаемых файлов?

Ответ.

7. Какие предопределенные макроопределения вы помните?

Ответ.

# ЛИТЕРАТУРА

1. Агабеков Л.Е., Иванова Г.С. Программирование на С++. Учебное пособие по курсам «Системное программирование», «Вычислительная техника и информационная технология», «Программирование на С++». Часть 1. Средства процедурного программирования.– М.: МГТУ им. Н.Э. Баумана, 1997.

2. Подбельский В.В. Язык С++: Учеб. пособие. – М.: Финансы и статистика, 2006.

# ПРИЛОЖЕНИЕ А *ОПТИМИЗАЦИЯ КОДА ПРОГРАММЫ*

Современные компиляторы включают средства оптимизации кода программ для уменьшения размера исполняемого файла или ускорения работы программы.

При оптимизации выполняются:

- размножение и сверка констант – многократная подстановка значения переменной вместо обращения к ней, если в программе она не меняется, а также замена в тексте программы результата арифметических операций над значениями таких переменных вместо выполнения этих операций при запуске программы;
- вычисление константных выражений – подстановка результата вместо последовательности действий. Например, если в программе необходимо подставить число секунд в году, то целесообразно написать выражение 3600*24*365, но не 31536000, поскольку это упрощает понимание программистом текста программы и избавляет от лишних ошибок, связанных с непониманием значения написанной константы;
- удаление неиспользуемых присваиваний, лишних выражений – позволяет устранить из текста программы присвоения значений переменным, которые нигде не используются. Если в программе нет последующего обращения к значению переменной, она не будет создана в результирующем коде;
- замена операций целочисленных деления и умножения на операции сдвига по возможности;
- удаление лишних условий и удаление фрагментов кода со всегда невыполняемым условием, например if (false){…};
- оптимизация ветвлений switch и последовательностей if(){}else if(){}….– позволяет уменьшить количество необходимых операций сравнения за счет преобразования линейной последовательности сравнений в дерево;
- оптимизация циклов, включая замену предусловий на постусловия и замену циклов с инкрементом на циклы с декрементом для приведения к сравнению переменной-счетчика с нулём;
- замена функций вставками inline по возможности.

Среда разработки MS Visual Studio по умолчанию предоставляет два режима сборки программы: Debug и Release. В первом случае программа собирается с отладочной ин-

формацией и отключенной оптимизацией. Во втором – отладочная информация не подключается, поэтому невозможно просмотреть состояние переменных при пошаговой отладке, однако включена оптимизация.

При необходимости режим оптимизации может быть включен и в отладочном режиме, однако следует помнить, что в этом случае текст программы не будет однозначно соответствовать сгенерированному компилятором коду (как по причине исключения, так и по причине добавления нового кода и изменения порядка выполнения исходных операторов программы), поэтому пошаговая отладка будет затруднена.

# ПРИЛОЖЕНИЕ Б *НЕКОТОРЫЕ ОПЦИИ КОМПИЛЯТОРА И КОМПОНОВЩИКА*

Сборка программы в MS Visual Studio невозможна без создания проекта. Управление опциями компилятора и компоновщика при использовании MS Visual Studio возможно непосредственно из среды разработки (свойства проекта), либо путем их задания в командной строке.

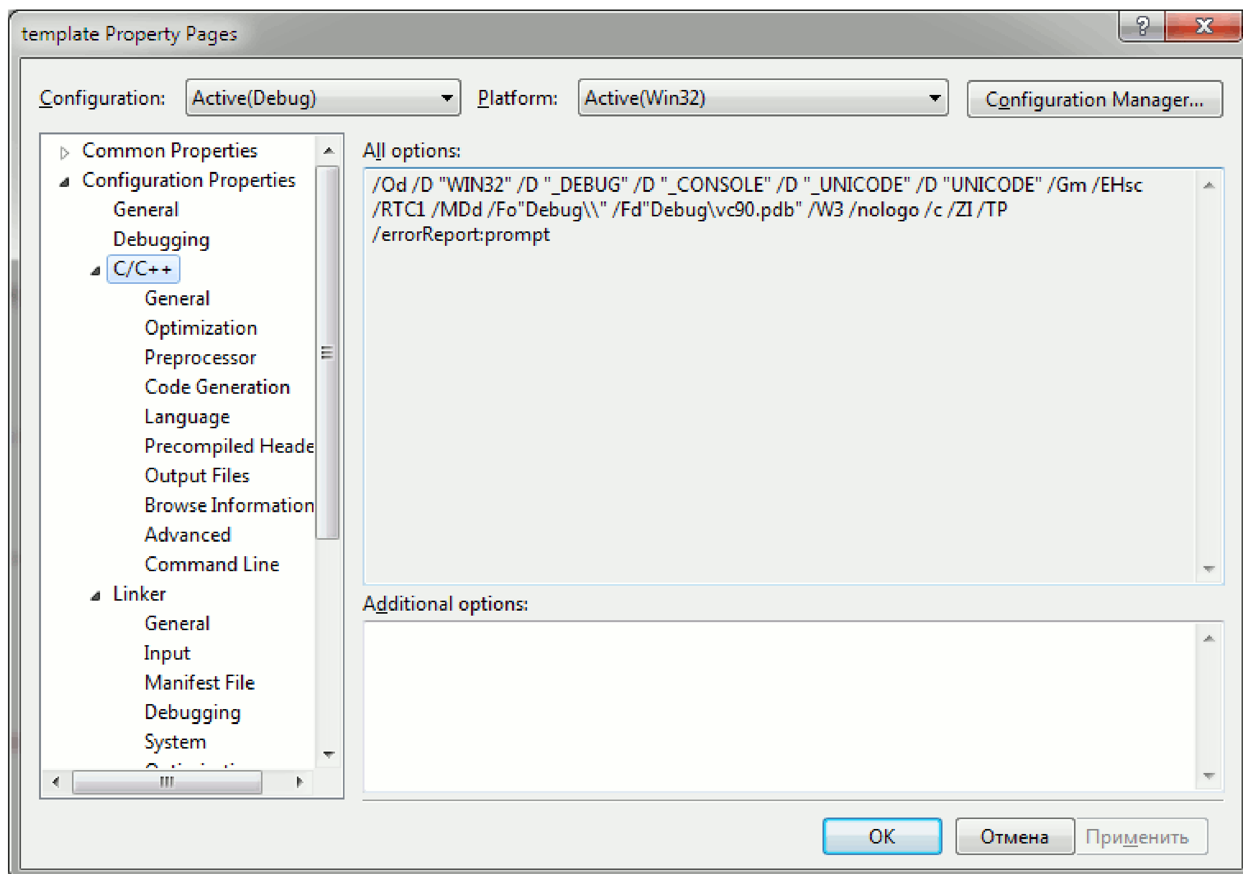



**Рисунок П2.1** – Настройка свойств проекта

На рисунке П2.1 показана часть дерева параметров проекта, в котором ветвь C/C++ обеспечивает управление опциями компилятора, а ветвь Linker – опции компоновщика. Помимо специально выделенных разделов опций, как для компилятора, так и для компоновщика имеется раздел «Command Line», где представлены параметры, которые фактически будут подставлены компилятору или компоновщику при сборке проекта. Здесь же возможно вписать любые дополнительные опции.

Рассмотрим некоторые важные разделы.

1. Раздел C/C++/**Precompiled header** – механизм ускорения компиляции проектов с большим количеством компилируемых модулей. В MS VC++ по умолчанию предполагается создание специального заголовочного файла stdafx.h. Суть механизма Precompiled head-

er заключается в том, что заголовочный файл, указанный как кэшируемый, компилируется один раз для всего проекта. Внутри этого файла может быть включено произвольное количество заголовочных файлов (обычно наиболее часто используемые в cpp-файлах). Тогда, компилятор помещает результат его компиляции в файл с расширением .pch и в дальнейшем использует только этот результат. В проекте, в котором используется Precompiled header необходимо, чтобы **во всех** компилируемых cpp-файлах первым подключался заголовочный файл, указанный как precompiled header.

Например опции компилятора /Yc"stdafx.h" /Fp"Debug\prg1.pch" означают, что в качестве precompiled header используется файл stdafx.h, результат компиляции которого помещается в Debug\prg1.pch.

2. Опция **C/C++/Preprocessor/Preprocessor definitions** позволяет указать список макроопределений, которые будут применены ко всем файлам проекта. Таким способом например определяют макроопределение _DEBUG. Пример: /D”_DEBUG”

3. Опция **C/C++/Preprocessor/Generate Preprocessed File** – позволяет указать препроцессору сформировать файл, являющийся результатом его работы, с выполненными подстановками, включениями файлов и разрешения фрагментов условной компиляции. Предусмотрено два варианта: /P – сформировать файл с номерами исходных строк, /EP /P – без вывода номеров строк.

4. Опция **C/C++/Optimization/ Optimization** – позволяет указать степень оптимизации программы или отключить её вообще (/Od)

5. Опция **C/C++/Advanced/Enable Code Analysis for C/C++ on Build** включает режим анализа наиболее распространенных ошибок программирования и обеспечивает вывод соответствующих предупреждений.