Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедры веб-технологий и компьютерного моделирования

А. С. Кравчук, А. И. Кравчук, Е. В. Кремень

Язык C++. Императивное программирование

Учебные материалы
для студентов специальности 1-31 03 08
«Математика и информационные технологии
(по направлениям)»

Минск
2023

УДК 004.432.045С++(075.8)
К 772

Решение о депонировании вынес:
Совет механико-математического факультета
28 февраля 2023 г., протокол № 6

Авторы:
А. С. Кравчук, доктор физико-математических наук профессор кафедры экономической информатики БГЭУ,
А. И. Кравчук, кандидат физико-математических наук доцент кафедры веб-технологий и компьютерного моделирования БГУ,
Е. В. Кремень, кандидат физико-математических наук доцент кафедры веб-технологий и компьютерного моделирования БГУ.

Рецензенты
Кафедра информационных технологий Белорусского государственного экономического университета (заведующая кафедрой Садовская М.Н., кандидат технических наук, доцент);
Медведев С.В., заведующий лабораторией синтеза технических систем Объединенного института проблем информатики НАНБ, доктор технических наук.

Кравчук, А. С. Язык C++. Императивное программирование : учебные материалы для студентов специальности: 1-31 03 08 «Математика и информационные технологии (по направлениям)» / А. С. Кравчук, А. И. Кравчук, Е. В. Кремень ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2023. – 389 с. : ил. – Библиогр.: с. 389.

Издание ориентировано на развитие и закрепление у студентов профессиональных навыков использования императивного программирования, как одного из основных инструментов решения прикладных задач. Это объясняется тем, что студент обязан хорошо освоить методологию использования функций, прежде чем он начнет понимать, как используются методы при создании и работе с объектами классов. В пособие включен такой методический материал, как таблицы ручного счета простейших алгоритмов, так и сопоставление кода простейших программ на C++ и Pascal. Издание адресуется прежде всего студентам, а также всем, кто хотел бы научиться приемам программирования при решении стандартных задач.

Оглавление

ОСНОВЫ СИНТАКСИСА **10**
Алфавит. Идентификаторы. Служебные слова 10
Идентификаторы 10
Лексемы 11
Структура программы 11
Начальные сведения о директивах препроцессора 12
Комментарии 13
Общий вид программы, оформленной в одном файле 13
Типы данных 14
Базовые типы 15
Модификаторы типов 16
Переменные 17
Объявление переменной 17
Копирующая и прямая инициализация переменных 18
`Uniform`-инициализация 19
Спецификатор `auto` 20
Простейшие средства ввода-вывода 21
Код простейшей программы 21
Общие сведения о константах 23
Неименованные константы (литералы) 23
Именованные константы 27
Простейший вид макроподстановки 29
Операции и выражения с использованием переменных и констант базовых типов 30
Основные унарные операции 30
Бинарные операции 32
Операция «запятая» 42
Операция `sizeof()` 43
Приоритет рассмотренных операций 44
Приведение типов 46
Приведение типов в выражениях 46

Приведение типов при присваивании 49
Явное приведение типов.. 51
Использование операции static_cast 52
Определение оператора 53
Пустой оператор.. 53
Требования к синтаксису операций и операторов 53
Хранение и обработка двоичного кода 54
Системы счисления. Перевод из десятичной в двоичную систему счисления и обратно.. 54
Основы представления информации в памяти компьютера 57
Стандартные математические функции 66
Средства форматирования ввода/вывода.. 67
Форматирующие методы.. 67
Некоторые флаги форматирования 68
Манипуляторы форматирования 70
Использование средств форматированного вывода языка С в программах на С++... 71
Функция printf() 72
Функция scanf() 74
Дополнительные функции символьного ввода/вывода языка С 75
ОПЕРАТОРЫ УПРАВЛЕНИЯ ПРОГРАММОЙ 75
Составной оператор 76
Область видимости переменных... 76
Условные операторы.. 77
Оператор if() 77
Оператор выбора switch() 90
Требования к оформлению условных операторов....................... 94
Операторы циклов.. 95
Оператор цикла for() 95
Оператор цикла while() 109
Оператор цикла do while() 111
Схема бесконечного цикла... 112

Вложенные циклы ... 113
Требования к оформлению операторов циклов ... 113
Операторы перехода ... 114
Оператор `break` ... 114
Оператор `continue` ... 115
Оператор безусловного перехода `goto` ... 115
Организация вычислений с точностью ... 116
Суммирование отрезка степенного ряда с точностью ... 116
Точность вычислений согласно рекуррентным уравнениям ... 118
АВТОМАТИЧЕСКИЕ МАССИВЫ ... 120
Одномерные автоматические массивы ... 121
Примеры простейших действий над одномерными автоматическими массивами ... 123
Операции линейной алгебры для векторов в контексте их применения к одномерным массивам ... 146
Использование в программе меньшего количества элементов массива, чем задано при объявлении ... 149
Строки ... 155
Многомерные автоматические массивы ... 162
Примеры элементарных действий с двумерным массивом ... 163
Заполнение двумерного массива по шаблону ... 167
Транспонирование квадратного двумерного массива ... 171
Ввод с клавиатуры «усеченного» двумерного массива ... 172
Заполнение двумерного массива псевдослучайными числами ... 174
ВВЕДЕНИЕ В ФУНКЦИИ ... 174
Объявление функций (прототип функции) ... 175
Определение функции ... 176
Вызов функции ... 177
Возвращение результата в точку вызова функции ... 178
Некоторые примеры функций, возвращающих значение ... 180
Возврат значений функцией `main()` ... 182
Тип возврата `void`. Игнорирование значений, возвращаемых функциями ... 183

Пример оформления функции без прототипа ... 184
Допустимые варианты оформления функций разработчика ... 185
Последовательность простейших действий по созданию функции разработчика, выделением алгоритмической части из главной функции ... 185
Примеры простейших функций ... 189
Инициализация значений формальных параметров по умолчанию... 191
Использование квалификатора `const` в параметрах функций... 193
Требования к синтаксису функций, создаваемых программистом ... 194
Элементы ООП в императивном программировании. Перегрузка функций ... 195
Параметры функции `main()` ... 196
СТЕК ВЫЗОВОВ ФУНКЦИЙ И РЕКУРСИЯ ... 197
Стек, как структура данных ... 197
Стек вызовов функций... 198
Стек на практике ... 199
Переполнение стека ... 199
Рекурсия ... 200
СОЗДАНИЕ ИСПОЛНЯЕМОГО КОДА ... 203
Трансляция программы: компиляция и интерпретация ... 203
Этапы создания исполняемого кода программ на языке C++ ... 204
Средства управления препроцессорной обработкой программы... 206
Подключаемые файлы ... 207
Директива `#include`... 209
Дополнительные сведения о директиве `#define`... 212
К вопросу об области видимости директивы `#define` ... 214
Директива `#undef` ... 215
Директивы условной компиляции... 216
Проблема дублирования объявлений... 218
УКАЗАТЕЛИ И ССЫЛКИ ... 221
Указатели... 221
Особый тип указателя `void` ... 226

Нулевые указатели 227
Указатели на константы 228
Константный указатель 229
Константный указатель на константу 230
Размер указателей 231
Операции над указателями 231
Адрес указателя. Указатель на указатель 238
Указатели и автоматические массивы 239
Взаимосвязь указателей и одномерных автоматических массивов 239
Одномерные автоматические массивы указателей 243
Указатель-на-указатель и двумерные автоматические массивы 245
Куча. Средства работы с кучей 247
Работа со скалярными переменными в куче 248
Взаимодействие стека и кучи при работе со скалярной переменной в куче 249
Одномерные массивы в куче 251
Двумерные массивы в куче 258
Ссылки 264
Основные сведения о ссылке 264
Ссылки на неконстантные значения 264
Ссылки vs указатели 266
Ссылки на константные значения 267
Константные ссылки 270
Ссылки на указатели 270
УКАЗАТЕЛИ И ССЫЛКИ В ПАРАМЕТРАХ ФУНКЦИЙ 271
Ссылки и указатели на переменные в параметрах функций 271
Указатели в параметрах функции 271
Ссылки в качестве параметров функций 273
Ссылка и указатель в качестве возвращаемого функцией значения 275
Ссылки и указатели на константы в параметрах функций 276
Указатели на константы в параметрах функций 276

Ссылки на константы в параметрах функций 277
Обработка в функциях массивов с помощью указателей 278
Функции и одномерные массивы 278
Обработка двумерных массивов в функциях 319
Передача в функцию автоматических массивов по ссылке 325
Использование ссылки на одномерный автоматический массив 325
Ссылка на двумерный автоматический массив 328
Использование константных ссылок на массив 332
Использование в параметрах функций ссылки на указатель 332
УКАЗАТЕЛИ И ССЫЛКИ НА ФУНКЦИИ 336
Синтаксис создания указателя на функцию 337
Выбор функции, решающей задачу 339
Передача указателя на функцию другой функции через параметры. Выбор направления сортировки 342
Массив указателей на функцию 348
Создание псевдонимов типов указателей на функцию 350
Ключевое слово `typedef` 350
Инструкция `using` 352
Ссылки на функцию 353
МАКРОСЫ ПРОВЕРКИ ОШИБОК 353
Инструкция `assert()` 353
Макрос `NDEBUG` 359
Инструкция `static_assert()` 359
ПРОСТЕЙШИЕ ЧИСЛЕННЫЕ МЕТОДЫ 360
Правило округления чисел 360
Элементы теории погрешностей 361
Виды погрешностей 362
Погрешность арифметических действий над приближенными числами 362
Элементарные методы решения нелинейных уравнений 363
Отделение корней уравнения 363
Метод деления отрезка пополам (метод дихотомии) 364
Метод простых итераций 366

Метод Ньютона (метод касательных) 368
Вычисление определенных интегралов 373
Квадратурные формулы левых, правых и средних прямоугольников 375
Формула трапеций... 376
Нижняя и верхняя суммы Дарбу 377
Пример вычисления нижней суммы Дарбу................................ 377
Правило Рунге (правило двойного пересчета) 378
Пример программной реализации вычисления определенного интеграла с заданной точностью....................... 380
Особенности вычисления определенного интеграла с точностью с помощью сумм Дарбу.. 381
Численное решение обыкновенного дифференциального уравнения первого порядка 384
Описание метода Эйлера.. 385
Пример программной реализации метода Эйлера..................... 386
ЛИТЕРАТУРА.. 389

Основы синтаксиса

Алфавит. Идентификаторы. Служебные слова

В алфавит С++ входят:

1. прописные и строчные буквы латинского алфавита: `A, B, C, …, a,b,c, … .`
2. цифры: `0, 1, 2, … , 9`;
3. специальные знаки: `", {, }, [, ], ', (, ), +, ….`
4. неотображаемые символы (обобщенные пробельные символы): пробел, табуляция, переход на новую строку.

Идентификаторы

Идентификатор – последовательность букв, цифр и символов подчеркивания (`'_'`), начинающихся с буквы или с символа подчеркивания.

Пример.

```
Kom_16,   _MIN,   TIME,   time
```

Замечание.

Прописные и строчные буквы различаются, таким образом два последних идентификатора различны.

Множество идентификаторов разделяется на:

- ключевые слова;
- свободно выбираемые идентификаторы (не могут совпадать с ключевыми словами и именами стандартных функций С++).

Ключевые слова – это зарезервированные идентификаторы, которые наделены определенным смыслом. Трансляторы языков С++, соответствующие требованиям стандарта ANSI, воспринимают только служебные слова, записанные строчными буквами.

Пример.

```
double,  else,   enum,       extern
```

Ключевые (служебные) слова сообщают компилятору о типе данных, способе их организации, последовательности выполнения операторов и пр.

Нет необходимости перечислять все служебные слова сразу, будем знакомиться с ними по мере освоения языка.

Требования к написанию ***свободно*** выбираемых ***идентификаторов***:

- ограничений на длину идентификатора не задокументировано;
- в идентификаторах по возможности следует использовать стандартные (общеупотребительные) сокращения слов;
- идентификатор должен нести смысл, поясняющий назначение именованного объекта в программе.

Замечание.

В дальнейшем термин «идентификатор» будет применяться исключительно в смысле «свободно выбираемый идентификатор». Т.е. словосочетание «свободно выбираемый» будет опускаться.

Следует ***избегать***:

- идентификаторов, которые различаются только одним или двумя символами;
- идентификаторов, использующих цифры;
- сокращений идентификаторов до одного символа (однако в некоторых случаях использование односимвольных идентификаторов не только допускается, но и полностью оправдано, например, имена, используемые для обозначения индексов элементов массивов).

Лексемы

Лексема – единица текста программы, которая не может быть разделена на более мелкие элементы и воспринимается как единое целое.

В языке С++ обычно выделяют ***шесть классов лексем:***

- свободно выбираемые идентификаторы;
- служебные (ключевые) слова;
- константы;
- строки (строковые константы);
- знаки операций;
- разделители (знаки пунктуации).

Структура программы

Программа на С++ – совокупность одного или нескольких модулей. Модулем является самостоятельно транслируемый файл, который обычно содержит одну или несколько функций.

Функция состоит из операторов языка. Термин «функция» в языке C++ охватывает понятия «подпрограмма», «процедура» и «функция», используемые в других языках программирования. Программа на C++ может содержать любое количество функций.

Однако все программы на C++ должны содержать ***одну*** главную функцию `main()` (собственно исполняемую программу). Остальные функции определяют потенциально возможные алгоритмические действия и вызываются из функции `main()` или из какой-либо другой функции в процессе выполнения программы. Эти функции могут находиться в том же текстовом файле, что и функция `main()`, или в других текстовых файлах.

Замечания:

- *программа, состоящая из нескольких файлов, обычно называется проектом;*
- *любой из текстовых файлов проекта разбит на строки;*
- *в конце каждой строки есть управляющий символ ее окончания (не отображается), а кроме того, управляющий символ перехода на новую строку (также не отображается);*
- *при просмотре текстового файла на экране видна только последовательность строк без управляющих символов.*

Отметим, что программа на C++ может содержать определение ***глобальных объектов***. Определения глобальных объектов размещаются в строках текстовых файлов достаточно произвольно за пределами функций, в том числе за пределами функции `main()`.

Начальные сведения о директивах препроцессора

Любая программа на C++ обязательно содержит раздел ***препроцессорных директив***.

В самом начале преобразования текста программы в исполняемый код он обрабатывается ***препроцессором*** – специальной программой, которая модифицирует этот текст в соответствие с ***директивами***.

Для ***препроцессорных директив*** существует ограничение: символ # являющийся признаком директивы препроцессора, должен быть первым отличным от пробела символом в строке описания директивы.

Простейшей и практически универсальной директивой является директива `#include`. Наиболее часто встречающийся в учебной литературе формат директивы:

```
#include <имяЗаголовочногоФайла>
```

Например, если `имяЗаголовочногоФайла` является `iostream`, то директива приобретает вид:

```
#include <iostream>
```

Это запись позволяет разработчику ПО использовать средства потокового ввода/вывода в программе, написанной на C++.

Комментарии

Комментарий – это набор символов, которые игнорируются компилятором.

Начало ***многострочного комментария*** начинается с символов `/*` и заканчивается символами `*/`. Все, что помещено между ними, игнорируется компилятором.

Замечание.

Многострочные комментарии `/*  */` не могут быть вложенными.

Хотя символами `/*  */` можно оформлять и однострочный комментарий, однако в C++ для этого используется специальная пара символов `'/'` (`//`), указывающая начало строки комментария. В этом случае концом комментария считается конец строки, так что нет необходимости отмечать его специальным символом. Этот способ наиболее полезен для коротких комментариев и является наиболее распространенным.

Замечание.

Обычно именно с помощью ***однострочных комментариев*** *оформляются и* ***многострочные комментарии****. Для этого в каждой строке многострочного комментария (в ее начале) ставится пара символов `'/'`.*

Общий вид программы, оформленной в одном файле

Программа – это последовательность команд (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке. ***Программа*** записывается на языке высокого уровня, наиболее удобном для реализации алгоритма решения определенного класса задач.

Обычно ***программа*** на C++, расположенная в одном текстовом файле, имеет вид:

```
#директивы препроцессора
//описание прототипов функций (например, с именами a и b)
//определение глобальных переменных

int main ( ) { //исполняемая программа
    //объявления и инициализация переменных и констант;
    //операторы управления программой;
    //вызов функции a;
    //операторы управления программой;
    //вызов функции b;
    //операторы управления программой;
    return 0;
}
описание функции a ( ) {
    //операторы
}

описание функции b ( ) {
    //операторы
}
```

В примере служебное слово `int` (выделено фиолетовым) перед `main()` означает, что главная функция должна вернуть операционной системе целочисленный код завершения программы с помощью оператора `return` (расположен ниже и также выделен фиолетовым). Возвращаемое значение нуль в операторе `return` обозначает «по умолчанию» нормальное завершение программы.

Типы данных

Известно, что числа могут быть натуральными, целыми, и вещественными. Так как объем памяти у компьютера ограничен, то при решении реальных задач используются числа различных типов, не превышающие какого-то определенного значения.

Во всех языках программирования типы данных не только указывают формат представления чисел, но и устанавливают определенный диапазон возможных значений. Этот диапазон обычно определяется аппаратной организацией компьютера.

Базовые типы

Тип – это атрибут, определяющий не только диапазон возможных значений, но и правила обработки числа этого типа.

*В С++ следует различать **тип** данных и **модификаторы** типа.*

Имеются следующие ***базовые типы*****:**

- `char` – символьный;
- `int` - целый;
- `bool` – логический тип;
- `float` – вещественный;
- `double` – вещественный с двойной тучностью;
- `void` – пустой тип;
- `wchar_t` – символьный для символов из расширенного набора (*не стандартизированный по размеру выделяемой области памяти тип, лучше не использовать*).

В таблице приведены ориентировочные размеры выделяемой памяти для некоторых ***базовых типов*** (Таблица 1).

Таблица 1 – Некоторые типы, размеры выделяемой памяти, а также минимальные и максимальные значения

Тип данных	Размер, байт	Диапазон значений
`char`	1	`-128…+127`
`int`	4	`-2 147 483 648…` `+2 147 483 647`
`bool`	1	`false, true`

Приведем пример объяснения происхождения, а также перевод некоторых служебных слов, обозначающих ***тип***:

- `char` (`CHARacter`: буква, символ);
- `int` (`INTeger`: целое число).

В заключение отметим, что ***тип*** `void` в С++ имеет три основных назначения:

- указание о невозвращении значения функцией;
- указание о неполучении параметров функцией;
- создание не типизированных указателей.

Каждое из этих назначений будет обсуждаться позже в соответствующих разделах.

Параметры ***логического типа*** (`bool`) могут принимать значения `false` и `true`, которые являются служебными (ключевыми) словами.

При преобразовании к целому типу:

- значения `false` – это целочисленный `0` (нуль);
- значение `true` – это целочисленное значение `1`.

Обратно:

- любое числовое значение отличное от нуля (в том числе содержащее дробную часть) может интерпретироваться как `true`;
- нуль интерпретируется как `false`.

Это дает возможность использовать арифметические выражения в качестве условий выполнения операторов управления программой.

Модификаторы типов

Существуют четыре ***модификатора*** типа, уточняющие внутреннее представление и диапазон значений стандартных типов:

- `unsigned` (без знака);
- `signed` (со знаком);
- `short` (короткий);
- `long` (длинный).

Модификатор типа уточняет диапазон значений, в пределах которого может изменяться переменная. Его применение может уменьшать или увеличивать размер области памяти для хранения числового значения, а может оставить его без изменения.

Кроме того, изменение диапазона хранения данных может осуществляться за счет изменения внутреннего представления (при изменении типа данного).

Варианты использования ***модификаторов***:

- все перечисленные ***модификаторы*** могут применяться к типу `int`;
- к типу `char` могут применяться `signed` и `unsigned`;
- `long` кроме типа `int` может примениться к типу `double`.

Пример.

`signed char` или `unsigned int`

Использование `signed int` избыточно т.к. целые числа по умолчанию числа со знаком. «По умолчанию» запись `short int` эквивалентна одному слову `short`, а `long int` одному слову `long`.

Для целочисленных типов данных обычно соблюдается следующее правило диапазонов в любых компиляторах:

short ~~int~~ <= int <= long ~~int~~,

но всегда выполнено

short < long.

Для типа `int` допускаются и более сложные логические конструкции, использующие ***модификаторы***.

Пример.

```
unsigned short int,
unsigned long int,
unsigned long long.
```

Переменные

Объявление переменной

Переменная - именованная область памяти, содержание которой меняется во время выполнения программы.

Доступ к значению ***переменной*** возможен через ее имя, а доступ к участку памяти – по ее адресу. Каждая ***переменная*** перед использованием в программе должна быть объявлена, т.е. ей должна быть выделена память.

Размер участка памяти, выделяемой для ***переменной***, и интерпретация его содержимого зависят от типа, указанного в объявлении переменной. Тип ***переменной*** изменить нельзя.

Общая форма объявления ***переменной***:

~~Модификатор~~ ~~Модификатор~~ тип **имяПеременной;**

Также допускается обявлять переменные списком, перечисляя имена перемнных через запятую:

~~Модификатор~~ ~~Модификатор~~ тип **список, именПеременных;**

Замечание.

В последнем случае переменные из списка будут иметь один и тот же тип.

Имена переменных – это свободно выбираемые программистом идентификторы, ***не*** совпадающие со служебными (ключевыми) словами.

В соответсвтии с выше изложенными требованиями для индентификаторов будем придерживаться наиболее распротраненного в настоящее время формата `camelCase`.

Это означает, что ***имена переменных*** должны состоять из ***нескольких слов***, при этом первое слово начинается с буквы нижнего регистра, а каждое следующее слово внутри фразы пишется с заглавной буквы.

Пример.

```
unsigned short int finalTime;
unsigned long int soundSpeed;
```

Замечания:

- *Допускается использование односимвольных идентификаторов для обозначения целых чисел, в частности, количества элементов в массивах и их индексов. В сосотвесвтии с принятыми в математике стандатрными обозначениями переменные* `i`, `j`, `k`, `l`, `m` *и* `n` *не должны иметь тип отличный от* `int`.
- *С другой стороны переменные с именами* `x`, `y`, `z`, *ассоциирующиеся с координатами в декатровой системе координат не должны иметь иных типов чем* `float` *или* `double`.

Копирующая и прямая инициализация переменных

Переменные могут быть не только объявлены, но и определены (инициализированы). Существует ***три*** вида (способа) инициализации:

- ***копирующая*** (с помощью символа '=', где '=' – операция присваевания);
- ***прямая*** (с использованием круглых скобок аналогично вызову конструктора при работе с классами);
- `uniform`-инициализация.

Формат ***копирующей*** инициализации:

~~`Модификатор`~~ ~~`Модификатор`~~ `тип имяПеременной` **`= значение`**`;`

Формат ***прямой*** инициализации:

```
~~Модификатор Модификатор~~ тип имяПеременной(значение);
```

<u>*Замечания:*</u>

- *объявление и инициализация переменных может осуществляться в любом месте программы;*
- *если переменная не проинициализирована отличным от нуля значением, то «по умолчанию» она инициализируется нулем.*

Uniform-инициализация

Прямая или копирующая инициализация работают не со всеми типами данных (например, разработчик не сможет использовать эти способы для инициализации массива значений).

В попытке обеспечить единый механизм инициализации, который будет работать со всеми типами данных, в C++ добавили новый способ инициализации, который называется `uniform`-инициализация, т.е. инициализация с «фигурными» скобками:

```
~~Модификатор Модификатор~~ тип имяПеременной{значение};
```

<u>*Пример.*</u>

```
int value{5};
```

Инициализация переменной с пустыми фигурными скобками указывает на инициализацию «по умолчанию» (переменной присваивается нуль).

<u>*Пример.*</u>

```
//инициализация переменной значением 0 (нуль)
int value{};
```

<u>*Замечание.*</u>

В отличие от других типов инициализации при `uniform`*-инициализации тип значения должен совпадать с типом переменной иначе компилятор выдаст ошибку.*

Спецификатор auto

Иногда бывает трудно определить тип выражения. В этом случае можно предоставить компилятору самому выводить тип объекта. И для этого применяется спецификатор `auto`. При этом если переменная определяется со спецификатором `auto`, эта переменная должна быть обязательно инициализирована каким-либо значением:

Пример.

```
auto number = 5;         //number имеет тип int
auto sum {1234.56};      //sum имеет тип double
auto distance {267UL}; //distance имеет тип unsigned long
```

На основании присвоенного значения компилятор выведет тип переменной. Использование неинициализированных переменных со спецификатором `auto` не допускаются.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      auto number; //синтаксическая ошибка
6      return 0;
7  }
```

Результат компиляции:

Compilation failed due to following error(s).

```
main.cpp: In function ‘int main()’:
main.cpp:5:5: error: declaration of ‘auto number’ has no initializer
    5 |     auto number; //синтаксическая ошибка
      |     ^~~~
```

Простейшие средства ввода-вывода

В C++ кроме средств ввода-вывода, унаследованных от языка C была создана своя система ввода-вывода. Для ее использования следует подключить заголовочный файл с именем `iostream` с помощью директивы `#include`.

После нее в начале обучения для упрощения дальнейшего текста программ следует добавить оператор:

```
using namespace std;
```

Он определяет то, что разработчик будет использовать стандартное пространство имен `std`. После этого далее в программе можно воспользоваться стандартными потоками ввода (`cin`) и вывода (`cout`).

Формат ввода с консоли значения уже объявленной переменной:

```
cin>> ИмяПеременной;
```

Формат вывода на экран значения уже объявленной переменной:

```
cout<< ИмяПеременной;
```

Замечание.

Некоторые авторы учебной литературы считают, что навязывание с первых шагов программирования на C++ использования инструкции `using`… наносит определенный вред будущему пониманию студентом правил работы с пространством имен. Они предлагают обходиться без этой инструкции. В этом случае ввод/оформляется следующим образом:

```
std::cin>> ИмяПеременной;
std::cout<< ИмяПеременной;
```

Код простейшей программы

Программа – это последовательность команд (инструкций), которые помещаются в памяти и выполняются процессором в указанном порядке.

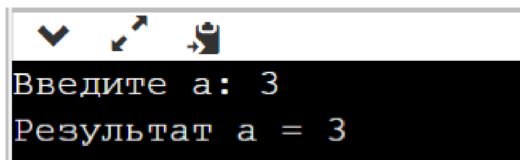
Пример.

main.cpp

```
1 #include <iostream>
2 #include <clocale> //для функции setlocale()
3 using namespace std;
```

```
 4
 5  int main() {
 6      //корректное отображение русского языка в Visual-е
 7      setlocale(LC_ALL, "Russian"); // не нужно в online сред.
 8
 9      int a;
10      cout << "Введите a: ";
11      cin >> a;
12
13      cout<< "Результат a = "<< a;
14      return 0;
15  }
```

Результат выполнения программы:



Перепишем эту программу в онлайн интегрированной среде без использования инструкции `using`… и без вызова функции `setlocale()` (работающую с тем же результатом) и сопоставим с программой, написанной на языке Pascal (Таблица 2)

Таблица 2 – Сравнение простейших программ на C++ и Pascal

<table>
<tr><th>Язык C++</th><th>Pascal</th></tr>
<tr><td><pre>#include <iostream>

int main () {
    int a;
    std::cout<< "Введите a: ";
    std::cin>> a;

    std::cout<< "Результат a = "<< a;
    return 0;
}</pre></td><td><pre>program main;
var a : integer;
begin   // в C++ это {
    // в C++ это std::cout
    write('Введите a: ');
    // в C++ это std::cin
    readln(a);

    // в C++ это std::cout
    write('Результат a = ', a);
end.   // в C++ это }</pre></td></tr>
</table>

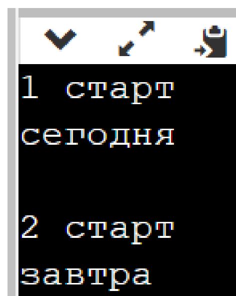
Отметим, что разработчик может управлять выводом сообщения на экран. В частности, перевод на новую строку можно выполнить несколькими способами. Они будут рассмотрены позже, но самым простым является использование строки, содержащей управляющий неотображаемый символ `'\n'`. Этот символ может использоваться как самостоятельное выражение, так и входить в состав строк.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      cout<< "1 старт \nсегодня"
6          << "\n\n";
7      cout<< "2 старт" << "\n"
8          << "завтра";
9      return 0;
10 }
```

Результат работы программы



Общие сведения о константах

Константы – данные неизменяемые в процессе выполнения программы.

Константы, в отличие от переменных, являются фиксированными значениями, которые можно вводить и использовать в языке C++.

Различают четыре типа ***констант***:

- целые;
- с плавающей точкой;
- символьные;
- строковые литералы.

Неименованные константы (литералы)

Целые ***неименованные константы*** не имеют дробной части и не содержат десятичной точки. Они представляют целую величину в одной из

следующих форм: десятичной, двоичной, восьмеричной, или шестнадцатеричной (Таблица 3).

Десятичная ***константа*** состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное).

Таблица 3 - Примеры неименованных целых констант

Десятичная константа	Двоичная константа	Восьмеричная константа	Шестнадцатеричная константа
16	0b10000	020	0x10
127	0b1111111	0177	0x7F
240	0b11110000	0360	0xF0

Двоичная ***константа*** начинается с обязательной последовательности `0b` или `0B` и содержит `0` или `1`.

Восьмеричная ***константа*** состоит из обязательного нуля и одной или нескольких восьмеричных цифр.

Шестнадцатеричная ***константа*** начинается с обязательной последовательности `0x` или `0X` и содержит одну или несколько шестнадцатеричных цифр.

Если необходимо задать значение неименованной отрицательной константы в любом из перечисленных кодов, то необходимо явно указать знак «минус» перед соответствующим кодом числа (Таблица 4).

Таблица 4 - Примеры записи отрицательных неименованных целых констант

Десятичная константа	Двоичная константа	Восьмеричная константа	Шестнадцатеричная константа
-16	-0b10000	-020	-0x10
-127	-0b1111111	-0177	-0x7F

Неименованная ***константа с плавающей точкой*** – десятичное число, представленное в виде действительной величины с десятичной точкой:

~~знак~~ ~~цифры~~.~~цифры~~

или в экспоненциальной форме:

~~знак~~ ~~цифры~~.~~цифры~~ E|e ~~знак~~ ~~цифры~~

Число с плавающей точкой состоит из целой и дробной части и (или) экспоненты. «По умолчанию» ***константы*** с плавающей точкой имеют тип `double`.

Пример.

`115.75, 1.5E-2, -0.025, .075, -0.85E2.`

Замечание.

*Если число записывается **без** точки, то эта запись воспринимается компилятором как неименованная **целочисленная** константа.*

Если программист хочет изменить размеры области памяти для хранения ***неименованных констант***, то с С++ он может использовать суффиксы:

- `F` (или `f`) – `float` (для ***вещественных констант***);
- `U` (или `u`) – `unsigned` (для ***целых***);
- `L` (или `l`) – `long` (для ***целых*** и ***вещественных***).

Пример.

`3.14F` (***константа*** типа `float`),
`3.14L` (***константа*** типа `long double`).

Неименованные символьные константы можно разделить на две группы:

- печатные символы;
- непечатные символы.

Неименованная символьная константа в языке С++ состоит либо из одного печатного символа, заключенного в апострофы (`'  '`, `'Q'`) , либо специального управляющего кода (`Esc`-последовательности), заключенного в апострофы (`'\n'`, `'\t'`, `'\b'`, `'\x47'`).

Символьная ***константа*** рассматривается как символьный беззнаковый тип данных с одним из возможных диапазонов значений:

- от `-128` до `127` для типа «по умолчанию» `signed char` в онлайн интегрированных средах;
- от `0` до `255` для типа `unsigned char`.

Замечание.

В некоторых интегрированных средах можно установить, что тип `char` «по умолчанию» соответствует `unsigned char`.

Константа (`Esc`-последовательность) `'\0'` является признаком окончания строки.

Любая символьная ***константа*** может быть заменена своим восьмеричным или шестнадцатеричным кодом вида:

`'\0ddd', '\xhh'.`

<u>*Пример.*</u>

`'\017', '\x2A'.`

Символьная ***константа***, которой непосредственно предшествует буква *L*, является широкой символьной ***константой***.

<u>*Пример.*</u>

`L'a'`

Такие ***константы*** имеют тип `wchar_t`.

Неименованная строковая константа – последовательность символов (включая строчные и прописные буквы русского и латинского алфавитов, а также цифры), заключенная в двойные кавычки (`""`).

<u>*Пример.*</u>

`"Образец строки"`

Для запоминания строковых ***констант*** используется по одному байту на каждый символ строки и автоматически добавляется к ней признак конца строки, которым служит символ `'\0'`. Для составления строковых ***констант*** можно использовать любые печатные символы или управляющие коды.

Строка-***литерал***, перед которой непосредственно идет символ `L`, считается широкосимвольной строкой.

<u>*Пример.*</u>

`L"asdf"`

Такая строка имеет тип «массив элементов типа `wchar_t`», где `wchar_t` – целочисленный тип.

Именованные константы

Именованные константы – это именованная область памяти, содержание которой не изменяется во время выполнения программы.

Замечание.

Имена (идентификаторы) ***именованных констант*** *должны содержать только заглавные буквы. Слова в сложном имени должны разделяться символом подчеркивания.*

Объявление констант

Определение именованных констант использует следующей конструкции со служебным словом **const**:

```
const ~~Модификатор~~ ~~Модификатор~~ тип ИМЯ_КОНСТ = знач;
```

Служебное слово **const** указывает, что объект с указанным именем не может быть изменен и доступен только для чтения (Таблица 5).

Пример.

```
const int MAX_SIZE = 2;
```

Замечание.

Объявление константы без ее инициализации вызывает ошибку компиляции.

Таблица 5 – Сравнение работы с константами в C++ и Pascal

<table>
<tr><th>Программа на С++</th><th>Программа на Pascal</th></tr>
<tr><td><pre>#include <iostream>

int main () {
    const int MAX_SIZE = 2;

    std::cout<< "Константа = "
            <<MAX_SIZE;
    return 0;
}</pre></td><td><pre>program main;
const MAX_SIZE = 2;
begin
    write('Константа = ',
          MAX_SIZE);
end.</pre></td></tr>
</table>

Целочисленные константы перечислимого типа

Вторую возможность определения констант дает служебное слово enum. Это целочисленные константы так называемого перечислимого типа:

```
enum ИдентификаторПеречисления {
    СПИСОК,
    ПЕРЕЧИСЛИТЕЛЕЙ
};
```

Идентификаторы перечислений обычно начинаются с заглавной буквы, а ***имена перечислителей*** состоят только из заглавных букв. Имена перечислителей не могут повторяться.

Каждому перечислителю автоматически присваивается целочисленное значение в зависимости от его позиции в списке перечисления. «По умолчанию», первому перечислителю присваивается целое число 0, а каждому следующему — на единицу больше, чем предыдущему.

Пример.

```
enum Colors {
    COLOR_YELLOW, // присваивается 0
    COLOR_WHITE,  // присваивается 1
    COLOR_ORANGE  // присваивается 2
};
```

Можно и самому определять значения перечислителей. Они могут быть как положительными, так и отрицательными, или вообще иметь значение, совпадающие со значением другого перечислителя. Любой перечислитель с неопределенным разработчиком значением будет иметь значения на единицу больше, чем значения предыдущего перечислителя.

Пример.

```
enum Animals {
    ANIMAL_PIG = -4,
    ANIMAL_LION, // присваивается -3
    ANIMAL_CAT, // присваивается -2
    ANIMAL_HORSE = 6,
    ANIMAL_ZEBRA = 6, // имеет то же значение
    ANIMAL_COW // присваивается 7
};
```

Следует обратить внимание, что ANIMAL_HORSE и ANIMAL_ZEBRA имеют одинаковые значения. Хотя C++ это не запрещает, присваивать одно значение нескольким перечислителям в одном перечислении, не рекомендуется.

Простейший вид макроподстановки

В некоторых пособиях использование ***макроподстановки***, созданной с помощью директивы препроцессора #define относят к способу определения именованных констант, но это неверная трактовка. Собственно, к именованным константам этот инструмент не имеет никакого отношения. Детально особенности работы с данной директивой будут рассмотрены позже. Простейший же вариант создания макроподстановки имеет формат:

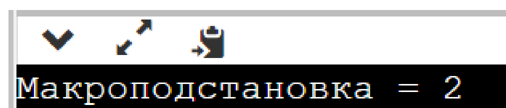
#define **ИМЯ_ПОДСТАНОВКИ** ЗначениеНеименКонстанты

После этого определения везде в тексте программ можно использовать ИМЯ_ПОДСТАНОВКИ, точно также, как и имя именованной константы, объявленной с использованием слова **const**.

Пример.

main.cpp
```
1  #include <iostream>
2  #define MAX_SIZE 2
3
4  int main () {
5      std::cout<< "Макроподстановка = " << MAX_SIZE;
6      return 0;
7  }
```

Результат работы программы:



Операции и выражения с использованием переменных и констант базовых типов

Операндом называется переменная, константа или выражение, участвующее в операции.

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется ***выражением***.

Каждый операнд в ***выражении*** сам может быть ***выражением***.

Операции бывают:

- унарные (участвует один операнд),
- бинарные (участвуют два операнда)
- условные (тернарные - три операнда).

Основные унарные операции

Унарные операции, в основном, имеют префиксный формат, т.е. знак операции `O` предшествует операнду:

```
O операнд
```

Однако для некоторых унарных операций используется и постфиксный формат, т.е. знак операции `O` следует за операндом:

```
операнд O
```

В таблице ниже (Таблица 6) можно ознакомиться со списком унарных операций.

Таблица 6 – Список унарных операций

Знак	Назначение
`+`	Унарный плюс, не выполняет никаких действий с арифметическим операндом
`–`	Унарный минус, меняет знак арифметического операнда
`!`	Логическое отрицание (НЕ). Меняет значение `true` или `false` логической переменной на противоположное (т.е. на `false` или `true`).
`++`	Инкремент (увеличение на единицу): префиксная форма увеличивает операнд перед использованием, а постфиксная – после использования в выражении;
`--`	Декремент (уменьшение на единицу): префиксная форма уменьшает операнд до его использования, а постфиксная – после использования в выражении;

Примеры.

```
++i;          j++;          !n
```

Операции ***инкремента*** и ***декремента*** (`++`, `--`) относятся к унарным арифметическим операциям, которые служат соответственно для увеличения или уменьшения значения, хранимого в переменной.

Операции инкремента и декремента не только изменяют значения переменных, но и возвращают значения. Таким образом, их можно сделать частью более сложного выражения.

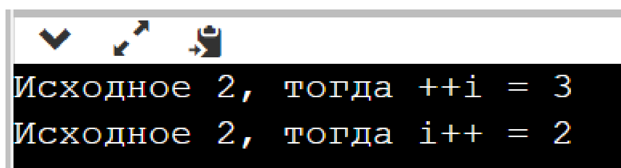
Имеется постфиксная и префиксная формы операций инкремента и декремента. В постфиксной форме записи значение переменной, к которой применена операция, увеличивается (или уменьшается) только после того, как ее значение будет использовано в контексте.

Пример.

main.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main () {
5   int i = 2;
6       cout<< "Исходное " << i
7           << ", тогда ++i = "<< ++i <<"\n";
8       i = 2;
9       cout<< "Исходное " << i
10          << ", тогда i++ = "<< i++ <<"\n";
11      return 0;
12  }
```

Результат работы программы:



Переменной `i` будет присвоена двойка. После операции `++i` значение переменной `i` немедленно измениться на `1` и станет равным `3`, и после этого оно напечатается оператором вывода.

Далее переменной `i` будет присвоена двойка, затем в следующей строке во время выполнения операции вывода в переменной `i` будет по-прежнему находится значение `2`, а после вывода значение `i` станет равным `3`. Это можно назвать ***отложенным*** изменением переменной на одно обращение (одно использование в контексте).

Типичной ошибкой является попытка использовать в операции инкремента или декремента операнд, в виде выражения.

Пример.

```
++(x + 1); //ошибка
```

Замечание.

Операции инкремента и декремента могут применяться и к переменным типа `float`, `double`.

Бинарные операции

Бинарные операции имею формат:

```
операнд1 O операнд2
```

где `O` – знак операции.

Арифметические операции

В таблице ниже (Таблица 7) можно ознакомиться со списком бинарных арифметических операций.

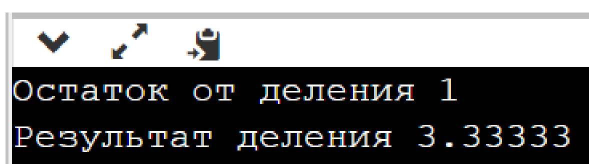
Таблица 7 – Список бинарных арифметических операций

Знак	Назначение	Группы операций
+	Бинарный плюс. Сложение арифметических операндов	Аддитивные операции
-	Бинарный минус. Вычитание арифметических операндов	
*	Умножение двух операндов арифметического типа	Мультипликативные операции
/	Деление операндов арифметического типа (если операнды целочисленные, то дробная часть результата отбрасывается)	
%	Получение остатка от деления целочисленных операндов	

Пример.

main.cpp
```cpp
#include <iostream>

int main() {
    std::cout<<"Остаток от деления " << 10 % 3
            <<"\nРезультат деления " << 10.0 / 3;
    return 0;
}
```

Результат работы программы:



Операции сравнения

В таблице ниже (Таблица 8) можно ознакомиться со списком операций сравнения.

Таблица 8 – Список операций сравнения

Знак	Название
<	Меньше, чем
>	Больше, чем
<=	Меньше или равно
>=	Больше или равно
==	Равно (сравнение)
!=	Не равно

Результат операции сравнения – это значение типа `boolean`, то есть `true` или `false`.

Замечание.

Хотя результат операций сравнения имеет булевский тип, но в C++ при выводе на экран этого результата будет осуществляться автоматическое преобразование к целочисленному значению, т.е. на экране вместо `true`

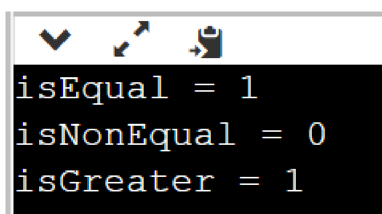
будет `1`*, а вместо* `false` *–* `0`*. Каким образом можно вывести на экран именно булево значение будет разъяснено позже.*

Пример.

main.cpp

```cpp
#include <iostream>

int main() {
    //инициализация переменных целого типа
    int x1 = 5, x2 = 5, x3 = 3, x4 = 7;
    //объявление переменных булевского типа
    bool isEqual, isNonEqual, isGreater;
    //возможные выражения
    isEqual = x1 == x2;       // isEqual = true
    isNonEqual = x1 != x2;  // isNonEqual = false
    isGreater = x1 > x3;     // isGreater = true

    std::cout<<"isEqual = " << isEqual
             <<"\nisNonEqual = " << isNonEqual
             <<"\nisGreater = " << isGreater;
    return 0;
}
```

Результат работы программы:



Хотя операции сравнения применимы к значениям всех базовых типов, однако применение операции == к двум значениям типа `double` (или `float`) является «небезопасным» в связи со спецификой хранения чисел с плавающей точкой.

Эта специфика приводит к тому, что числа с плавающей точкой никогда абсолютно точно не совпадают. Значение двух переменных типа `double` (или `float`) могут сравниваться только в смысле совпадения их значений с некоторой точностью.

Логические бинарные операции

Основные логические операции (в программировании и математике) можно применять к логическим аргументам (операндам), а также составлять более сложные выражения, подобно арифметическим действиям над числами.

В таблице ниже (Таблица 9) можно ознакомиться со списком логических бинарных операций.

Таблица 9 – Список логических бинарных операций

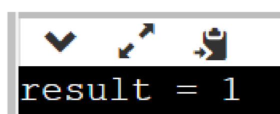
Знак	Назначение
`&&` или слово `and`	Условное `И` (сокращенное логическое `И`). Если операнд, находящийся слева от `&` является `false`, данная операция возвращает `false` без проверки второго операнда.
`||` или слово `or`	Условное `ИЛИ` (сокращенное логическое `ИЛИ`). Если операнд слева является `true`, то операция возвращает `true` без проверки второго операнда.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    //объявление переменных булевского типа
    bool a = true, b = false, result;
    //инициализация переменных целого типа
    int c = 5, q = 5;
    //допустимое логическое выражение
    result = (a||b) || (c<100) && (! (q == 5));
    cout<<"result = " << result; //преобразованное true
    return 0;
}
```

Результат работы программы:



Побитовые операции

Операции `&`, `|` и `^` применяются к операндам целого типа (`char`, `short`, `int`, `long`) и выполняют побитовые действия для двоичных представлений этих операндов.

Кроме того, в C++ существуют также операции битового сдвига (<< и >>). В таблице ниже (Таблица 10) можно ознакомиться со списком бинарных побитовых операций.

Замечание.

Примеры использования побитовых операций и их результаты будут рассмотрены позже после разъяснений о представлении целочисленных значений в памяти компьютера.

Таблица 10 – Список побитовых операций

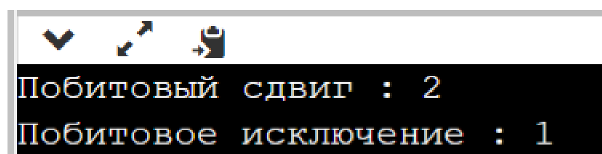
Операции	Назначение	Группы операций
`&`	Поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов	Поразрядные операции
`|`	Поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов	
`^`	Поразрядное исключающее или битовых представлений значений целочисленных операндов	
`<<`	Сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда	Операции сдвига
`>>`	Сдвиг вправо битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого целочисленного операнда	

Следует отметить, что все побитовые операции могут быть применены и к операндам булевского типа. При этом перед выполнением этих побитовых операций сначала будет вычислено значение булевских операндов, участвующих в операции, а затем полученное значение `false` или `true` преобразуется к целочисленному `0` или `1`.

Пример.

```cpp
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout<< "Побитовый сдвиг : "
6          <<(true << true) //эквивалентно 1 << 1
7          <<"\n";
8      cout<< "Побитовое исключение : "
9          <<(false ^ true); //эквивалентно 0 ^ 1
10     return 0;
11 }
```

Результат работы программы:



Операции присваивания и составного присваивания

В таблице ниже (Таблица 11) можно ознакомиться со списком операций присваивания.

Таблица 11 – Список операций присваивания

<table>
<tr><th>Знак</th><th>Назначение</th></tr>
<tr><td>=</td><td>Операция «присваивание», с ее помощью значение правого операнда присваивается левому.</td></tr>
<tr><td>+=</td><td rowspan="11">Выполнить соответствующую операцию с левым операндом и присвоить результат левому операнду</td></tr>
<tr><td>-=</td></tr>
<tr><td>*=</td></tr>
<tr><td>/=</td></tr>
<tr><td>%=</td></tr>
<tr><td>|=</td></tr>
<tr><td>&=</td></tr>
<tr><td>^=</td></tr>
<tr><td><<=</td></tr>
<tr><td>>>=</td></tr>
</table>

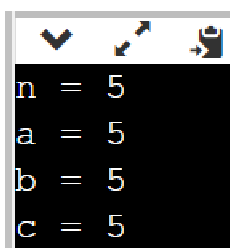
Операция присваивания «=» рассматривается как выражение, имеющее значение правого операнда после ее выполнения. Присваивание может включать несколько операций присваивания, изменяя значения нескольких операндов.

Пример.

main.cpp

```cpp
#include <iostream>

int main() {
    int a, b, c = 5;
    //допустимый синтаксис
    int n = a = b = c;
    //эквивалентно следующему набору
    //операторов: b = c; a = b; n = a;

    std::cout<<"n = " << n
             <<"\na = " << a
             <<"\nb = " << b
             <<"\nc = " << c;
    return 0;
}
```

Результат работы программы:



Замечание.

Недопустимыми являются: присваивание константе или присваивание выражению.

Пример.

```cpp
2 = a + b;          //ошибка
(a + 1) = 2 + b;  //ошибка
```

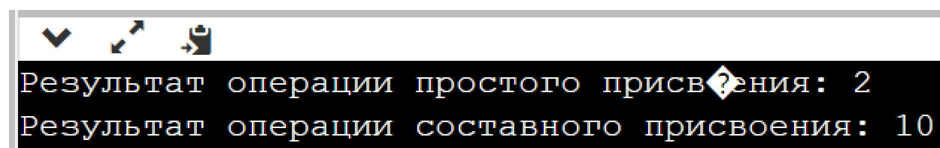
Отметим следующую особенность любой операции присвоения (простого или составного) – она, как и любая бинарная операция имеет результат, и ее результатом является значение, присваиваемое левому операнду.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int c;
6      cout<<"Результат операции простого присвоения: "
7          << (c = 2) <<"\n";
8      cout<<"Результат операции составного присвоения: "
9          << (c *= 5);
10     return 0;
11 }
```

Результат работы программы:



Тернарная (условная) операция

В отличие от унарных и бинарных операций в тернарной условной операции используется три операнда:

```
Выражение1 ? Выражение2 : Выражение3;
```

Первым вычисляется значение `Выражения1`. Если оно истинно, то вычисляется значение `Выражения2`, которое становится результатом. Если при вычислении `Выражения1` получится `false`, то в качестве результата берется значение `Выражения3`.

Замечание

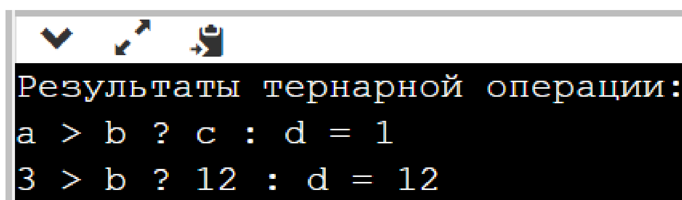
В качестве выражений могут использоваться любые арифметические и логические выражений, а в качестве операндов в этих выражениях - константы, переменные или другие выражения.

Пример.

main.cpp

```cpp
#include <iostream>

int main() {
    int a = 1, b = 2, c = 7;
    bool d = true;

    std::cout<<"Результаты тернарной операции:\n"
             <<"a > b ? c : d = " << (a > b ? c : d)
             <<"\n3 > b ? 12 : d = " << (3 > b ? 12 : d);
    return 0;
}
```

Результат работы программы:



Замечание.

Напомним, что значение `1` *в первом результате объясняется тем, что это* `true` *(результат операции) преобразуется к целому типу.*

Рассмотрим дополнительные примеры применения тернарной операции. Одним из самых простых является вычисление с помощью тернарной операции модуля переменной.

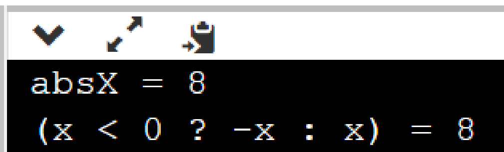
Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    double x = -8.;
    double absX = x < 0 ? -x : x;

    cout<<" absX = "<< absX << endl;
```

```
9       cout<<" (x < 0 ? -x : x) = " << (x < 0 ? -x : x);
10
11      return 0;
12  }
```

Результат работы программы:



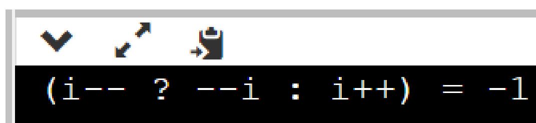
Более сложным примером является совместное использование операций инкремента или декремента и тернарной операции.

Пример.

main.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int i = 0;
6
7       cout<<" (i-- ? --i : i++) = " << (i-- ? --i : i++);
8
9       return 0;
10  }
```

Результат работы программы:



Результат операции в данном примере объясняется следующим образом:

- в первой секции тернарной операции постфиксный декремент `i` (`i--`) означает отложенное уменьшение значения `i` на единицу, таким образом к моменту преобразования результата `i--` к булевскому типу выражение имеет целочисленное значение `0`, которое преобразуется к `false`, а после этого значение переменной `i` становиться `-1`;

- управление передается к третьей секции тернарной операции, т.е. к постфиксному (отложенному) инкременту (`i++`) и к моменту первичного обращения в этой секции к переменной `i` для последующего увеличения на единицу ее значение равно `-1`. Именно оно выводится на экран, а после этого увеличивается на единицу.

Операция «запятая»

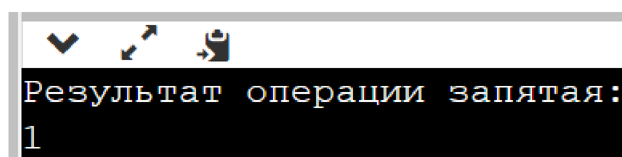
Операция запятая выполняет каждый из его операндов (слева направо) и возвращает значение последнего операнда. Формат операции:

```
expr1, expr2, expr3...
```

Пример.

```
main.cpp
#include <iostream>

int main() {
    int c;
    bool d;

    std::cout<<"Результат операции запятая:\n"
            //true при выводе преобразуется к 1
            <<(3, 2, c = 7, d = true);
    return 0;
}
```

Результат работы программы:



Можно использовать операция «запятая», когда необходимо включить несколько выражений в место, которое принимает только одно выражение. Наиболее частый пример использования этого операции - это перечисление нескольких операторов в секциях заголовка цикла `for()` (будет рассматриваться позже).

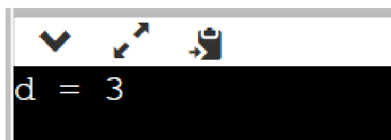
Другой важный пример использования операции «запятая» – это вычисления перед выводом значения на экран или перед возвратом значения функцией.

Пример.

main.cpp

```
#include <iostream>

int main() {
    int c = 0;

    std::cout<<"d = " << (c += 3, c);
    return 0;
}
```

Результат работы программы:



Замечание.

***Не** является оператором «запятая» ее использование в следующих синтаксических конструкциях:*

- *объявлении переменных или констант, так как в данном случае она находится не внутри выражения;*
- *инициализации массивов перечислением значений, аргументах и параметрах функций и конструкторов классов.*

Операция `sizeof()`

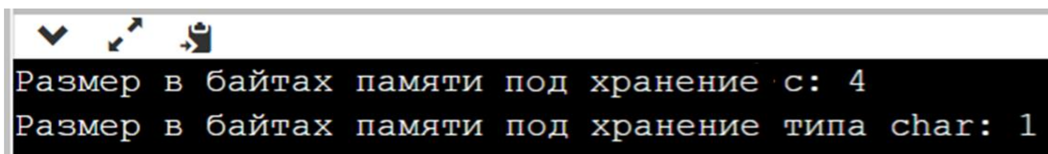
Для того чтобы определить размер памяти, выделяемой для переменной (или объекта) данного типа, можно использовать операцию `sizeof()`.

Значением этой операции является размер в байтах любой переменной (или объекта), а также спецификации типа. Кроме переменной объектом может быть также массив, структура, объект класса или просто спецификация типа.

Пример.

```cpp
#include <iostream>
using namespace std;

int main() {
    int c = 0;
    cout<<"Размер в байтах памяти под хранение c: "
        <<sizeof(c)<<"\n";
    cout<<"Размер в байтах памяти под хранение типа char: "
        <<sizeof(char);
    return 0;
}
```

Результаты работы программы:



Приоритет рассмотренных операций

Последовательность вычисления значения выражения зависит от:

- круглых скобок в выражении (() – операция группировка);
- приоритета выполнения рассмотренных операций в выражении (Таблица 12).

Замечание.

Операций в С++ больше, но не целесообразно включать в таблицу операций еще не известные операции, т.к. это может существенно путать начинающих программистов.

В качестве особенностей применения приоритета операций следует рассмотреть вывод на экран результатов тернарной операции. Как следует из таблицы выше (Таблица 12). Эта операция имеет один из низших приоритетов.

Рассмотрим, например, вывод на экран результата следующего выражения:

```cpp
5 % 2 ? 9 : 5
```

Теоретически его результатом является значение 9. Однако посмотрим на отображаемый на экране результат.

Таблица 12 – Приоритет операций

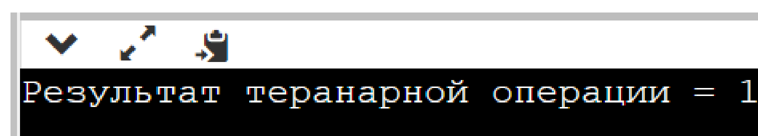
Ранг	Операции
1	( ) (группировка)
2	!, ~, -, ++,-- (унарные арифметические), sizeof(тип) (Получение размера объекта указанного типа)
3	*, /, % (мультипликативные бинарные операции)
4	+, – (аддитивные бинарные операции)
5	<<, >> (операция побитового или поразрядного сдвига)
6	>, <, >=, <= (операции сравнения)
7	==, != (операции сравнения)
8	& (побитовая или поразрядная конъюнкция «И»)
9	^ (побитовое или поразрядное исключающее «ИЛИ»)
10	| (побитовая или поразрядная дизъюнкция «ИЛИ»)
11	&& (условное или сокращенное «И»)
12	|| (условное или сокращенное «ИЛИ»)
13	? : (тернарная или условная операция)
14	=, *=, /=, %=, -=, &=, ^=, |=, <<=, >>= (операции присваивания)
15	…, … (операция запятая)

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      std::cout<< "Результат теранарной операции = "
5                << 5 % 2 ? 9 : 5;
6      return 0;
7  }
```

Результат работы программы:



При поверхностном рассмотрении кода программы этот результат вызывает некоторые сомнения в адекватности. Однако в данном случае все совершенно нормально, т.к. следует рассматривать не только тернарную

операцию, но и операцию вставки в поток << (перегруженная операция поразрядного сдвига влево).

Бинарная операция сдвига влево << имеет ранг 5, а тернарная операция имеет ранг 14, т.е. выполняется после операции вставки в поток. При этом в поток вставляется результат операции взятия целочисленного остатка (5 % 2), и именно он (значение 1) выводится на экран.

Для того, чтобы получить значение именно тернарной операции ее следует заключить в скобки.

Пример.

```
std::cout<< "Результат теранарной операции = "
        << (5 % 2 ? 9 : 5);
```

Приведение типов

Приведение типов в выражениях

В операциях могут участвовать операнды различных типов, в этом случае они неявно преобразуются к общему типу в порядке увеличения их объема памяти, необходимого для хранения их значений. Поэтому неявные (автоматические) приведения типов всегда идут от «меньших» объектов к «большим». Схема выполнения приведений операндов ***арифметических*** и ***логических*** операций (первая и вторая строка ниже):

```
bool -> int

char, short, int, unsigned, long, float, double -> bool

char, short -> int -> unsigned -> long

float -> double
```

Следует отметить, следующее правило: если оба операнда имеют одинаковый тип, то и результат операции преобразуется к этому типу.

Например, результат выражения 3 / 2 равен 1, т.к. 3 и 2 - целые числа, следовательно результат 3 / 2 это целое число, полученное отсечением дробной части от значения 1.5.

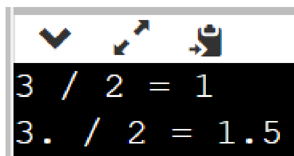
В данном случае потерю точности легко предотвратить: достаточно в качестве одного из операндов использовать тип `double` или `float`, т.е. после одного из операндов поставить десятичную точку.

Пример.

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      cout<<"3 / 2 = " << 3 / 2 <<"\n"
7          <<"3. / 2 = " << 3. / 2;
8      return 0;
9  }
```

Результат работы программы:



Более детально рассмотрим унарную операцию отрицания (НЕ). Это широко используемая операция в языке С++. Для начинающего программиста одним из неожиданных вариантов синтаксиса этой операции является ее применение как к обычным переменным, так и к арифметическим выражениям, что, казалось бы, невозможно в связи с тем, что сама операция «по определению» применяется к операндам логического типа.

Пусть `a`, `b` и `n` – переменные целого типа. Тогда результат операции:

$$!\,n = \begin{cases} true, \text{если } n = 0, \\ false, \text{если } n \neq 0. \end{cases}$$

$$!\,(a+b) = \begin{cases} true, \text{если } a+b = 0, \\ false, \text{если } a+b \neq 0. \end{cases}$$

Возможность применения логической унарной операции `!` (НЕ) к числовым операндам объясняется автоматическим приведением типа `int` к типу `bool`, в соответствии с уже упомянутым правилом: любое ***ненулевое***

значение при необходимости автоматически приводится к значению `true`, а ***нулевое*** к значению `false`.

Замечание.

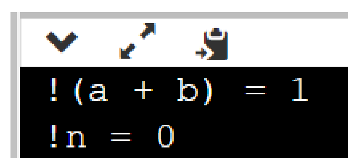
Переменные `a`, `b` *и* `n` *могут принимать как положительные значения и нуль, так и отрицательные.*

Следует подчеркнуть, что при вставке в поток (перегруженная операция сдвига влево <<) произойдет обратное преобразование из типа `bool` в `int` и на экране будет отображаться либо `0`, либо `1`.

Пример.

main.cpp
```
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main() {
 5      int a = 5, b = -5;
 6      int n = -11;
 7
 8      cout<<" !(a + b) = " << !(a + b) << endl;
 9      cout<<" !n = " << !n;
10
11      return 0;
12  }
```

Результат работы программы:


```
!(a + b) = 1
!n = 0
```

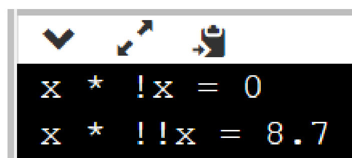
Следует отметить, что совершенно аналогично операция отрицания применяется и к переменным с плавающей точкой.

Пример.

main.cpp
```
 1  #include <iostream>
```

```
 2  using namespace std;
 3
 4  int main() {
 5      double x = 8.7;
 6
 7      cout<<" x * !x = " << x * !x << endl;
 8      cout<<" x * !!x = " << x * !!x;
 9
10      return 0;
11  }
```

Результат работы программы:



Остановимся на более детальном рассмотрении выражения `x * !x`. Поскольку `x` не нуль, то он преобразуется к значению `true`, в этом случае выражение `!x` приобретает значение `false`. Далее поскольку умножение — это бинарная арифметическая операция, то происходит автоматическое преобразование `false` к целочисленному значению `0` (***нуль***), т.е. `x` (`типа double`) умножается на ***нуль*** (`типа int`) и результатом этой операции будет ***нуль*** типа `double`.

Совершенно аналогично рассматривается выражение `x * !!x`.

Приведение типов при присваивании

При присваивании значение правой части преобразуется к типу левой, который и является типом результата. При некорректном использовании операций присваивания могут возникнуть ошибки выполнения кода.

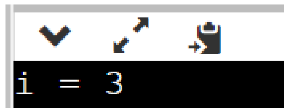
Пример.

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
```

```
 6      float x = 3.14;
 7      int i;
 8      i = x;  //float преобразуется в int, дробная часть
 9              //отбрасывается
10      cout<<"i = " << i;
11      return 0;
12  }
```

Результат выполнения программы:



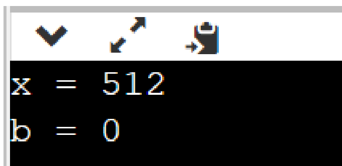
В строке `i = x;` приведен пример так называемого сужающего преобразования. В данном случае сужающим оно называется из-за того, что переменная `x` типа `float` содержит дробную часть, а целое `i` после присваивания содержит только целую часть `x` (дробная уже утеряна).

Таким образом одним из вариантов сужения является отбрасывание дробной части, но сужение можно продемонстрировать и более детально. Например (Таблица 1) известно, что максимальное число, которое может вместить переменная типа `char` (1 байт) это `127`, а переменная типа `float` имеет несравненно больший диапазон хранения. Тогда можно рассмотреть следующий пример.

Пример.

```
main.cpp
 1  #include <iostream>
 2
 3  using namespace std;
 4
 5  int main() {
 6      float x = 512.0;
 7      char a = x; //сужающее пробразование
 8      int b = a;  //расширяющее преобразование
 9
10      cout<<"x = " << x <<"\n"
11          <<"b = " << b;
12      return 0;
13  }
```

Результат работы программы:



Явное приведение типов

В любом выражении приведение типов может быть осуществлено явно, в C++ для этого достаточно перед выражением поставить в скобках идентификатор соответствующего типа:

```
(тип) выражение;
```

Также допускается следующий вариант синтаксиса явного приведения типов:

```
тип(выражение);
```

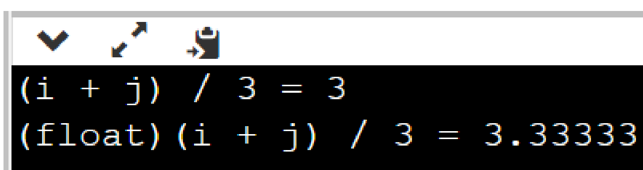
В результате значение выражения преобразуется к заданному `тип`-у.

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      int i = 6, j = 4;
5
6      std::cout<<"(i + j) / 3 = " << (i + j) / 3 <<"\n"
7               <<"(float)(i + j) / 3 = " << (float)(i + j) / 3;
8      return 0;
9  }
```

Результат выполнения программы:



Дополнительными примерами явного приведения типов является вычисление целой и ***дробной*** частей ***положительного*** значения с плавающей точкой (для отрицательного значения вычисление осуществляется иначе, т.к. целая часть – это ***меньшее*** целое отрицательное число).

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double x = 3.57;
6
7      cout<<" Целая часть = " << (int) x << endl;
8      cout<<" Дробная часть = " << x - (int) x;
9
10     return 0;
11 }
```

Результат работы программы:

```
Целая часть = 3
Дробная часть = 0.57
```

Использование операции `static_cast`

Операция `static_cast` в языке C++ осуществляет явное допустимое приведение типа данных. Формат операции:

```
static_cast < ТипПриведения > ( ОбъектДляПреобраз )
```

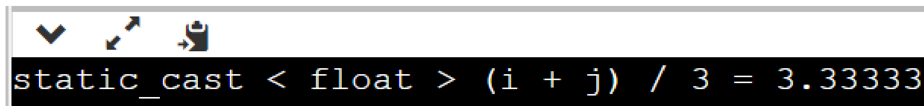
Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
```

```
4      int i = 6, j = 4;
5
6      std::cout<<"static_cast < float > (i + j) / 3 = "
7              << static_cast < float > (i + j) / 3;
8      return 0;
9  }
```

Результат выполнения программы:



Определение оператора

Оператор – это выражение, после которого стоит точка с запятой.

Таким образом, отличие оператора от операции или записи последовательности операций и операндов заключается в наличии разделителя «точка с запятой» в конце выражения.

Запись действий, которые должен выполнить компьютер, состоит из операторов. При выполнении программы операторы выполняются один за другим, за исключением операторов управления, которые могут изменить последовательное выполнение программы.

Пустой оператор

Простейшей формой оператора является ***пустой оператор***:

```
;
```

Он ничего не делает. Однако он может быть полезен в тех случаях, когда синтаксис требует наличия оператора, а он не нужен.

Требования к синтаксису операций и операторов

- все операнды и знаки операций разделяются пробелами;
- каждый оператор пишется в отдельной строке (даже объявления переменных одного типа следует писать в разных строках для удобства компилирования;
- желательно, чтобы оператор не превосходил ширины экрана;

- если длинное выражение разбивается на несколько строк, то табуляцией следующие строки выравниваются отступом слева до знака присваивания, открывающей скобки функции или операции вставки в поток.

Хранение и обработка двоичного кода

Системы счисления. Перевод из десятичной в двоичную систему счисления и обратно

Теоретические сведения

Под ***системой счисления*** понимается способ записи чисел с помощью символов (цифр, букв и т.д.). Системы счисления бывают ***позиционные*** и ***непозиционные***. ***Непозиционной*** является, например, римская система счисления. В ***позиционных*** системах счисления любое число записывается в виде последовательности символов, количественное значение («вес») которых зависит от местоположения в числе, т.е. позиции в записи числа. ***Основанием*** позиционной системы счисления называется целое число, определяющее количество символов, используемых в ней (обозначим его через p). В ***позиционной*** системе счисления с основанием p (p-ичной системе счисления) ***целое*** число R может быть представлено в виде:

$$R_{(p)} = a_N p^N + a_{N-1} p^{N-1} + \ldots + a_1 p^1 + a_0 p^0, \qquad (1)$$

где коэффициенты a_i – символы в изображении числа R, которые принимают значения от 0 до $p-1$. Обычно ***целое*** число $R_{(p)}$ представляется записью коэффициентов a_i:

$$R_{(p)} = a_N a_{N-1} \ldots a_1 a_0 .$$

Пример.

$$265_{(10)} = 2\cdot 10^2 + 6\cdot 10^1 + 5\cdot 10^0,$$

$$10001_{(2)} = 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0.$$

Величина p показывает, во сколько раз численное значение единицы данного разряда больше численного значения единицы предыдущего разряда.

В вычислительной технике широко используются позиционные системы счисления (двоичная, восьмеричная, десятичная, шестнадцатеричная).

В памяти компьютера, на уровне аппаратной реализации, информация представляется в двоичной системе счисления, а на уровне операционной системы используются системы счисления большей разрядности в зависимости от времени выпуска компьютера.

Двоичная система счисления

В двоичной системе счисления используются две – цифры 0 и 1. Основание двоичной системы счисления записывается в виде $2_{(10)} = 2 \cdot 10^0 = 1 \cdot 2^1 + 0 \cdot 2^0 = 10_{(2)}$.

Арифметические операции в двоичной системе счисления выполняются с помощью таблиц 13, 14 по тем же правилам, что и в десятичной системе счисления.

Таблица 13 - Двоичная таблица сложения

+	0	1
0	0	1
1	1	10

Таблица 14 - Двоичная таблица умножения

×	0	1
0	0	0
1	0	1

Приведем некоторые примеры выполнения основных операций с целыми числами в двоичной системе счисления (Таблица 15).

Пример.

а) сложение:

$$\begin{array}{r} 110000011 \\ +\quad 11101 \\ \hline 110100000 \end{array}$$

б) вычитание:

$$\begin{array}{r} 110010101 \\ -\ 100001011 \\ \hline 10001010 \end{array}$$

в) умножение:

$$\begin{array}{r} 11001 \\ \times\quad 11 \\ \hline 11001 \\ 11001\ \\ \hline 1001011 \end{array}$$

г) деление:

$$\begin{array}{l|l} 1111 & 101 \\ \underline{101} & \overline{\quad 11} \\ \ \ 101 & \\ \ \ \underline{101} & \\ \quad 0 & \end{array}$$

Таблица 15 - Запись чисел в десятичной и двоичной системах счисления

Десятичное число	Двоичное число	Десятичное число	Двоичное число

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Перевод целого десятичного числа в двоичную систему счисления

Целую часть числа, записанную в ***десятичной*** системе счисления, делят на основание ***двоичной*** системы счисления, записанное в ***десятичной*** системе счисления (все операции производятся по правилам ***десятичной*** системы счисления). Полученное в остатке число является младшей (последней) цифрой в ***двоичной*** записи числа.

Полученное частное снова делят на основание ***двоичной*** системы счисления; остаток этой операции – предпоследняя цифра в искомой записи числа и т.д.

Операцию деления проводят до тех пор, пока в частном не получат число, меньшее 2; частное – старшая (первая) цифра в двоичной записи числа.

Пример.

$$24_{(10)} = 11000_{(2)}.$$

```
24|2
24 12|2
 0 12  6|2
    0  6 3|2
       0 2 1
         1
```

Перевод двоичного числа в десятичную систему счисления

Перевод чисел в ***десятичную*** систему счисления рекомендуется выполнять суммированием с учетом «веса» цифры в числе по формуле (1):

$$a_N a_{N-1} \dots a_1 a_0 \, . \, a_{-1} \dots a_{-K} = a_N p^N + a_{N-1} p^{N-1} + \dots + a_1 p^1 + a_0 p^0$$

Пример.

$$1101_{(2)} = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{(10)}.$$

Основы представления информации в памяти компьютера

Информация в памяти компьютера хранится и обрабатывается в двоичном виде. Форма записи данных в памяти машины называется ***внутренним представлением информации*** в памяти компьютера. Единицей хранения информации является один бит, т.е. двоичный разряд, который может принимать значения 0 или 1. Применение двоичной системы счисления позволяет использовать для хранения информации элементы, имеющие всего два устойчивых состояния. Одно состояние служит для изображения единицы соответствующего разряда числа, другое – для изображения нуля. Обычно биты памяти группируются в более широкие структуры. Группа из восьми битов называется байтом. Все байты пронумерованы, начиная с нуля.

Адресом любой информации считается адрес (номер) самого первого байта поля памяти, выделенного для ее хранения.

Существует два основных способа представления чисел, называемых представлением с ***фиксированной*** и с ***плавающей*** точкой. В начале рассмотрим представление в памяти компьютера числа с фиксированной точкой, т.е. целого числа.

Двоичные числа с фиксированной точкой (целые числа)

Для чисел с фиксированной точкой положение точки зафиксировано после младшей цифры числа, дробная часть отсутствует, точка в изображении числа опускается. Таким образом, в виде с фиксированной точкой могут храниться только целые числа (в памяти компьютера они записываются в двоичном виде). Следует отметить, что в алгоритмических языках для записи числа в память используются специальные слова – ***атрибуты***.

В языке C++, например, для указания того, что значение переменной должно быть записано с фиксированной точкой, применяют такие атрибуты как типы (или типы с модификаторами), например: `int`, `short`, `long`, `unsigned`, `signed char` (или обычно просто `char`), а также `unsigned char`.

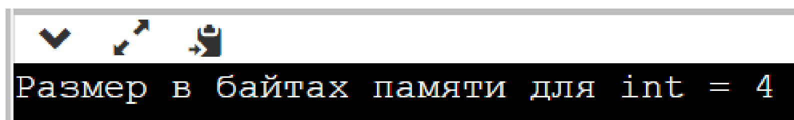
Будем предполагать, что целое двоичное число занимает в памяти 32 двоичных разряда (бита), т.е. 4 байта. Длина участка памяти, выделяемая под

хранение каждого из типов данных, зависит от компилятора и операционной системы. В каждом отдельном случае можно узнать размер выделяемой памяти с помощью операции `sizeof()`.

Пример.

main.cpp
```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      cout<< "Размер в байтах памяти для int = "
6          << sizeof(int) << "\n";
7      return 0;
8  }
```

Результат работы программы.



Восемь бит составляют байт, два подряд идущих байта – слово, четыре подряд идущих байта – двойное слово. Итак, рассмотрим, как записывается число в двойном слове. Седьмой бит первого байта является служебным при хранении чисел со знаком, и его содержимое не оказывает влияния на абсолютную величину числа. Отметим, что для положительных чисел содержимым служебного бита, является нуль. Младший двоичный разряд числа записывается в 0-ой бит, т.е. число заполняют справа налево. Если число положительно, то оставшиеся биты заполняются нулями.

Например, число $127_{(10)}$ в 4 байтах будет представлено следующим образом:

$$127_{(10)} = 1111111_{(2)},$$

31 24	23 16	15 8	7 6 5 4 3 2 1 0	биты
0000 0000 1 – ый байт	**0000 0000** 2–ой байт	**0000 0000** 3–ий байт	**01111111** 4–ый байт	байты

Из приведенного выше представления видно, что в младшем разряде записан коэффициент при 2^0, в следующем – при 2^1 и т.д.

Если во всех битах с 0-го по 30-ый находятся 1, то максимальное целое положительное число, которое можно записать в четырех байтах, имеет вид:

$$01111111\ 11111111\ 11111111\ 11111111 = 2^{31} - 1 = 2147483647_{(10)}.$$

Форму записи положительных целых двоичных чисел называют ***прямым кодом***. В этом случае их запись в отведенной памяти полностью соответствует символической записи в двоичной системе счисления.

Отрицательные числа записываются в ***дополнительном коде***. Использование этого кода позволяет упростить аппаратную реализацию операции вычитания, которая заменяется операцией сложения уменьшаемого, представленного в прямом коде, и вычитаемого, представленного в дополнительном коде.

Дополнительный код получается из прямого путем:

- инвертирования каждого бита (***обратный код***);
- добавления `1` к младшему биту числа.

Представим число $-95_{(10)}$ как двоичное число с фиксированной точкой.

$$-95_{(10)} = -1011111_{(2)}$$

1. Запишем прямой код ***модуля*** числа в четыре байта:
   00000000 00000000 00000000 01011111
2. Запишем обратный код числа:
   11111111 11111111 11111111 10100000
3. Прибавив к младшему биту 1, получим дополнительный код числа:

$$\begin{array}{r} 11111111\quad 11111111\quad 11111111\quad 10100000 \\ + \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 1 \\ \hline \underbrace{\mathbf{11111111}}_{\text{1-ый байт}}\quad \underbrace{\mathbf{11111111}}_{\text{2-ой байт}}\quad \underbrace{\mathbf{11111111}}_{\text{3-ий байт}}\quad \underbrace{\mathbf{10100001}}_{\text{4-ый байт}} \end{array}$$

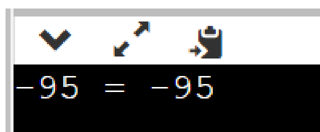
Пример.

main.cpp

```
1 //Программа получения дополнительного кода для
2 //числа -95
3 #include <iostream>
4
```

```
 5 int main () {
 6     int a = -95;
 7     //прямой код модуля числа a
 8     unsigned b = -a;
 9     //Обратный код b (пример унарной бит. опер.)
10     b = ~b;
11     //дополнительный код
12     b = b + 1;
13     //сравнение исходного значения и допол. кода
14     std::cout<< a << " = " << (int) b;
15     return 0;
16 }
```

Результат работы программы:



Замечание.

При хранении целых положительных чисел существует возможность вдвое увеличить числовой диапазон. Это достигается за счет использования служебного 31-го бита первого байта для хранения старшего коэффициента в двоичном представлении числа (`unsigned`*).*

Стандартная ошибка «превращения» отрицательного целого числа в очень большое положительное целое

Эта ошибка характерна не только для начинающих программистов, но и для людей с опытом. Просто в случае наличия опыта разработчик сразу исправляет ее понимая, где локализована данная ошибка, а начинающий программист долгое время воспринимает ее как «мистику».

Все дело в преобразовании типов отрицательного целого числа в беззнаковое (`unsigned`) целое число. Рассмотрим детально почему это происходит:

- отрицательное число храниться в дополнительном коде, поэтому его `31`-ый бит занят единицей;
- если же еще отрицательное целое число ***по модулю*** относительно маленькое, то практически все последние биты в двоичной записи дополнительного кода тоже будут единицами;

* при указанном преобразовании типов (`signed` в `unsigned`) 31-й бит (как и вся группа единиц дополнительного кода) становятся ***значащим кодом положительного числа***, что и приводит к преобразованию небольшого по модулю отрицательного числа в огромное положительное.

Продемонстрируем это на простейшем примере, выполнив явное преобразование типов из `int` в `unsigned` в `cout`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = -95;
6      cout<< "Исходное a = " << a << ".\n"
7          <<"Результат явн. прив. тип.="<<(unsigned) a;
8      return 0;
9  }
```

Результат работы программы:

```
Исходное a = -95.
Результат явн. прив. тип.=4294967201
```

Замечание.

Результат будет таким же и любом другом случае приведения отрицательного целого значения к беззнаковому.

Логические побитовые операции

Для работы с отдельными битами в С++ предусмотрены ***побитовые операции***. Эти операции нельзя применять к переменным с плавающей точкой. Операндами операций над битами могут быть только выражения, приводимые к целому типу.

В примере предыдущего раздела уже была продемонстрирована унарная операция, называемая «инверсией» или «побитовым НЕ» (~) для создания обратного кода при преобразовании двоичного представления целого числа в дополнительный код.

Кроме того существуют бинарные операции (`&`, `|`, `^`), которые носят названия: «побитовое умножение» («побитовое И», Таблица 16), «побитовое

сложение» (побитовое ИЛИ, Таблица 17) и побитовое вычитание («побитовое исключающее ИЛИ», Таблица 18) выполняются поразрядно над всеми битами в двоичном представлении значений операндов (знаковый разряд также участвует в операции).

Таблица 16 - Двоичная таблица побитового И

&	0	1
0	0	0
1	0	1

Таблица 17 - Двоичная таблица побитового ИЛИ

|	0	1
0	0	1
1	1	1

Таблица 18 - Двоичная таблица побитового исключающего ИЛИ

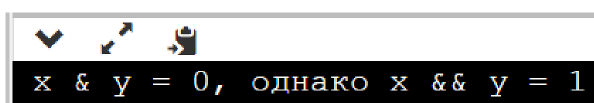
^	0	1
0	0	1
1	1	0

Необходимо **отличать побитовые операции** `&` и | от соответствующих **логических бинарных операций** `&&` и `||`.

Пример.

main.cpp
```
1  #include <iostream>
2  using namespace std;
3
4  int main () {
5      int a = 1, b = 2;
6
7      int ResA = a & b,   //ResA=0, т.к. 0001 & 0010=0000
8          ResAA = a && b; //ResAA= 1, т.к. в операц. логич.
9                          // умн. оба операнда истина
10     cout<< " x & y = "  << ResA
11         << ", однако x && y = "  << ResAA;
12     return 0;
13 }
```

Результат работы программы:



Операции сдвига << и >>.

Формат операций:

```
a << k,
a >> k,
```

где `a` – целочисленный операнд, у которого осуществляется сдвиг в его двоичном представлении в памяти компьютера, `k` – целое число, указывающие на сколько бит осуществляется сдвиг.

При этом должно выполняться цепочка неравенств:

```
0 < k <= 8 * sizeof(a).
```

В другом случае результат операции сдвига не определен.

Операции сдвига выполняются для всех разрядов двоичного представления числа `a` с потерей выходящих за границы ячейки памяти значащих бит.

Операции сдвига ***вправо*** на `k` разрядов можно использовать для деления, а сдвиг ***влево*** – для умножения целых чисел на `2` в степени `k`.

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main () {
4      unsigned a = 30;
5
6      std::cout<<"Исх. a="<< a<<" сдвиг влево на 1 разряд ="
7                << (a << 1) << "\n";
8      std::cout<<"Исх. a ="<< a<<" сдвиг вправо на 1 разряд ="
9                << (a >> 1) << "\n";
10     std::cout<<"Исх. a ="<< a<<" сдвиг влево на 3 разряда ="
11               << (a << 3) << "\n";
12     return 0;
13 }
```

Результат работы программы:

```
Исх. a=30 сдвиг влево на 1 разряд =60
Исх. a =30 сдвиг вправо на 1 разряд =15
Исх. a =30 сдвиг влево на 3 разряда =240
```

Двоичные числа с плавающей точкой

Десятичное число с дробной частью также можно перевести в двоичный код. Это преобразование осуществляется отдельно для целой и дробной частей.

Для использования вещественных значений, содержащих в своей записи десятичную точку, используется специальный способ представления в памяти компьютера - формат числа с плавающей точкой.

В общем виде любое число A в системе счисления с основанием p можно представить в виде $A = m \cdot p^k$, где m - мантисса числа A, k - порядок числа. При этом если мантисса числа удовлетворяет неравенству $1 \le |m| < p$, то число ***нормализованное***.

Размеры областей, выделяемых под числа с плавающей точкой, существенно зависят от аппаратной реализации вычислительной техники, а также операционной системы. Например, числа с плавающей точкой, в зависимости от объявленных атрибутов `float`, `double`, `long double`, могут располагаться в ячейках памяти размером `32`, `64` и `80` бит соответственно.

Кратко и не вдаваясь в подробности рассмотрим пример представления числа с плавающей точкой.

float	±	характеристика	нормализованная мантисса
биты	31	30 23	22 0

double	±	характеристика	нормализованная мантисса
биты	63	62 52	51 0

long double	±	характеристика	нормализованная мантисса
биты	79	78 64	63 0

Мантисса числа записывается в нормализованном виде (в двоичном представлении числа перед точкой сохраняется один значащий двоичный бит). Чтобы увеличить диапазон представления чисел в форматах `float` и `double`, единица перед точкой в двоичной нормализованной записи числа в память не записывается; в формате `long double` – единица сохраняется. Знак мантиссы записывается в последнем бите первого (слева по порядку) байта.

С целью упрощения аппаратной реализации арифметических операций в представлении числа с плавающей точкой знак порядка явно не хранится. Вместо порядка в память записывается величина, называемая ***характеристикой***. ***Характеристика*** получается из порядка путем прибавления поправочного коэффициента, который для чисел типа `float` равен $127_{(10)}$, для `double` – $1023_{(10)}$, для `long double` – $16383_{(10)}$.

Представим число $-18.2_{(10)} = -10010.(0011)_{(2)}$, нормализованная запись которого $-1.0010(0011)_{(2)} \cdot 2^{100}$, где нормализованная мантисса $m = -1.0010(0011)_{(2)}$, а порядок $k = 4_{(10)} = 100_{(2)}$. Перейдем от порядка к его характеристике: $4 + 127 = 131_{(10)} = 83_{(16)} = 1000\,0011_{(2)}$. Получим представление числа $18.2_{(10)}$ в формате `float`:

1	**10000011**	**00100011001100110011010**
31	30 23	22 0

Если количество цифр в числе, записываемом в памяти компьютера или полученном в результате вычислений, больше выделенного поля памяти, то избыточные цифры отбрасываются, а число, записываемое в память, округляется с избытком.

Таким образом размеры выделенных для хранения данных областей определяют интервал допустимых для этого атрибута значений – диапазон (Таблица 19).

Таблица 19 – Примеры диапазонов чисел с плавающей точкой

Тип данных	Кол-во бит	Диапазон	
		`fabs(min)`	`fabs(max)`
`float`	32	$3.4 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$
`double`	64	$1.7 \cdot 10^{-308}$	$1.7 \cdot 10^{308}$
`long double`	80	$3.4 \cdot 10^{-4932}$	$3.4 \cdot 10^{4932}$

Замечание.

Как уже отмечалось ранее диапазон хранимых в переменной значений, а также правила их арифметических преобразований определяется указанным программистом типом при объявлении (или инициализации) переменной.

Всякое число, меньшее по абсолютной величине положительного минимального числа, представленного в соответствующем формате, будет записано в память в виде нуля. Для данного формата это так называемый ***машинный нуль***.

Кроме того, числа данного формата не должны превышать по абсолютной величине максимальное число, представленное в этом формате. В противном случае старшие биты числа будут потеряны, а результат не верен. Описанная ситуация называется переполнением разрядной сетки, а сами числа – машинной бесконечностью.

Кодирование символьной информации

В памяти компьютера для внутреннего представления символьных данных используется так называемый ASCII код, согласно которому каждому символу соответствует 8-разрядный код, т.е. в один байт записывается один символ.

Доступ к ASCII-таблице символов можно получить, используя `Esc`-последовательности.

Стандартные математические функции

Математические функции языка С++ декларированы в файлах `<cmath>` и `<stdlib.h>`. В большинстве приведенных здесь функций аргументы `x`, `y` и результат выполнения имеют тип `double` (Таблица 20).

Замечание.

Заголовочный файл `<cmath>` также содержит определения некоторых математических констант. Они не стандартизированы для языка С++. Однако в онлайн интегированных средах после подключения указанного заголовочного файла ими можно пользоваться без ограничений.

Таблица 20 – Список математических функций

Прототип	Выполняемое действие

`double ceil(double);`
`float ceilf(float);`
`long double ceill(double);`
	Функции округления числа с плавающей точкой до наименьшего целого

`double cos(double);`	Функция вычисляет значение косинуса, формальный параметр – значение в радианах
`double exp(double);`	Функция вычисляет экспоненту

`double fabs(double);`
`float fabsf(float);`
`long double fabsl(long double);`
	Функция вычисляет абсолютное значение числа с плавающей точкой

`double floor(double);`
`float floorf(float);`
`long double floorl(long double);`
	Функции округления числа с плавающей точкой в большую сторону

`double fmod(double, double);`	Функция получения остатка от деления первого аргумента с плавающей точкой на второй
`double log(double);`	Функция вычисляет натуральный логарифм аргумента
`double log10(double);`	Функция вычисляет десятичный логарифм аргумента
`double pow(double, double);`	Функция вычисляет первый аргумент в степень, указанную во втором аргументе
`double sin(double);`	Функция вычисляет значение синуса, формальный аргумент – значение в радианах
`double sqrt(double);`	Функция вычисляет значение квадратного корня аргумента
`double tan(double);`	Функция вычисляет значение тангенса, формальный аргумент – значение в радианах

Средства форматирования ввода/вывода

Форматирующие методы

Основными форматирующими методами в C++ являются:

- **`fill`**`('символ')` – устанавливает символ заполнитель, где `символ` – символ-заполнитель, он передается в одинарных кавычках, или в виде числа (код символа). Формат вызова метода:

```
cout.fill('символ');
```

где `.` – операция уточнения имени и будет рассмотрена позже.

- **width**(ширинаПоля) - задает ширину поля, где ширинаПоля – количество позиций(одна позиция вмещает один символ). Формат вызова:

```
cout.width(ширинаПоля);
```

- **precision**(число) - задает количество знаков после десятичной точки, где число - количество знаков после десятичной точки. Формат вызова:

```
cout.precision(число);
```

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double x = 3.145679;
6      cout.precision(3);
7      cout.width(30);
8      cout.fill('%');
9      //esc-последовательность \t отвечает за табуляцию
10     //т.е. перенос курсора сразу на несколько позиций
11     cout<<"\t\t"<< x<<"\n";
12     return 0;
13 }
```

Результат работы программы:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%                3.15
```

Некоторые флаги форматирования

Одних функций недостаточно для форматирования потоков ввода/вывода, поэтому в C++ предусмотрен еще один способ форматирования — флаги.

Флаги форматирования позволяют включить или выключить один из параметров ввода/вывода. Для того чтобы установить флаг ввода/вывода, необходимо вызвать метод `setf()`. Метод `setf()` принимает один аргумент — имя флага. Флаги вывода объявлены в классе `ios`, поэтому, перед тем как обратиться к флагу, необходимо написать имя класса — `ios`, после которого с помощью операции уточнения области действия («`::`») вызвать нужный флаг. Если необходимо отключить флаг вывода, то используется функция `unsetf()`. Пусть `имяФлага` является именем флага, тогда формат установки и снятия флагов вывода имеет вид (в случае если в программе ***не*** используется инструкция `using namespace std;`):

- установка флага вывода:

```
std::cout.setf(std::ios::имяФлага);
```

- снятие флага вывода:

```
std::cout.unsetf(std::ios::имяФлага);
```

Если при вводе/выводе необходимо установить (снять) несколько флагов, то можно воспользоваться ***поразрядной*** логической операцией `ИЛИ` «`|`». В этом случае конструкция языка С++ будет такой:

- установка нескольких флагов:

```
std::cout.setf(std::ios::имяФлага | std::ios::имяФлага);
```

- снятие нескольких флагов:

```
std::cout.unsetf(std::ios::имяФлага | std::ios::имяФлага);
```

Перечислим некоторые ***флаги***:

- `boolalpha` - вывод логических величин в текстовом виде (`true`, `false`);
- `scientific` - вывод чисел с плавающей точкой в экспоненциальной форме;
- `fixed` - вывод чисел с плавающей точкой в фиксированной форме(по умолчанию) ;
- `right` - выравнивание по правой границе (по умолчанию), сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки);
- `left` - выравнивание по левой границе, сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки).

Манипуляторы форматирования

Еще один способ форматирования — форматирование с помощью манипуляторов. Манипулятор — объект особого типа, который управляет потоками ввода/вывода, для форматирования передаваемой в потоки информации. Отчасти манипуляторы дополняют функционал, для форматирования ввода/вывода. Но большинство манипуляторов выполняют точно, то же самое, что и функции с флагами форматирования.

Пример некоторых манипуляторов:

- `endl` - переход на новую строку при выводе;
- `boolalpha` - вывод логических величин в текстовом виде (`true`, `false`);
- `scientific` - вывод чисел с плавающей точкой в экспоненциальной форме.

Пример.

main.cpp

```
1   #include <iostream>
2   #include <cmath> //для использования константы M_PI
3
4   int main() {
5       std::cout<< std::scientific
6                << std::boolalpha
7                << M_PI<<"   "<< true
8                << std::endl;//манипулятор - аналог "\n";
9       return 0;
10  }
```

Результаты выполнения программы:

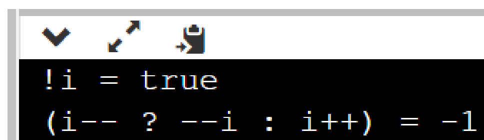
Приведем примеры того, как будут отличаться результаты вывода на экран значений операций, рассмотренных ранее при использовании манипулятора форматирования.

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    double i = 0;

    cout<< boolalpha
        << " !i = " << !i
        << endl
        <<" (i-- ? --i : i++) = " << (i-- ? --i : i++);

    return 0;
}
```

Результат работы программы:



Как правило начинающий программист ожидает вывода результата операции `!i` в целочисленном виде, т.е. `1`, но как объяснялось ранее результат `1` получается из-за автоматического промежуточного преобразования `true` к целому типу. В данном случае манипулятор `boolalpha` отключает это автоматическое преобразование и на экран выводится значение `true`.

Для второго выражения использование манипулятора `boolalpha` не имеет никакого значения, т.к. его значение изначально является целочисленным.

Использование средств форматированного вывода языка C в программах на C++

Обмен данными средствами форматированного ввода языка C вывода можно реализовать реализуется с помощью библиотеки функций ввода-вывода языка C `<stdio.h>`. Она, как и другие библиотеки подключается директивой `#include`:

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

Замечание.

Для всех библиотек языка С, используемых в С++ в названиях следует писать расширение `.h` *(*`header file`*). Однако необходимо помнить, что средства форматированного ввода/вывода языка С также описаны в уже известном заголовочном файле* `iostream` *и достаточно воспользоваться подключением только этого заголовочного файла без дополнительного подключения* `<stdio.h>`.

Функция printf()

Функция `printf()` позволяет выводить на дисплей данные всех типов, работать со списком из нескольких аргументов и определять способ форматирования данных. Простейший формат использования:

```
printf(форматнаяСтрока, списокАргументов);
```

где `форматнаяСтрока` – строка символов, заключенных в кавычки, которая показывает, каким образом должны быть напечатаны аргументы, `списокАргументов` – это перечисление имен переменных, которые будут выведены в форматированной строке.

Пример.

main.cpp

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    printf(" Значение E = %f\n", M_E);
    cout << " Значение E = " << M_E << endl;
    return 0;
}
```

Результат работы программы:

```
Значение E = 2.718282
Значение E = 2.71828
```

Параметр `форматнаяСтрока` может содержать символы, печатаемые в виде текста, ***спецификации преобразования*** (начинается с символа '%' + следующий символ, указывающая тип данных), а также управляющие символы (Esc-последовательности).

Используются следующие спецификации преобразования:

`%d` - десятичное целое число;

`%f` - вещественное число типа `float` или `double`, иногда в старых версиях интегрированных сред для вывода значения переменной типа `double` используется спецификация преобразования `%lf`;

`%c` - символ;

`%s` - строка;

`%p` - указатель;

`%u` – беззнаковое целое число;

`%o` – целые числа в восьмеричной системе счисления;

`%x` – целые числа в шестнадцатеричной системе счисления;

`%e` – вещественное число в экспоненциальной форме.

Буква после '`%`' называются ***спецификаторами формата***. Однако между символом '`%`' и буквой могут располагаться ***модификаторы формата*** - числовые значения, определяющие максимальную длину поля (измеряемую в символах), в котором будет расположено значение из списка аргументов. Более того для чисел с плавающей точкой, можно указать два числовых значения `общаяДлина.количЦифрПослеЗапятой`. Например, для вещественных чисел запись `%4.2f` означает, что вывод на экран числа будет осуществлен в поля общей длиной четыре символа, при двух цифрах после запятой точка также включается в расчет длины поля вывода значения.

Замечание.

Если ширина поля превосходит количество значащих символов переменной, то ***перед*** *числом добиваются пробелы.*

Можно управлять перемещением курсора при выводе информации на экран и выполнять некоторые другие действия, используя управляющие коды, называемые esc-последовательностями. Последовательность начинается с символа обратной слеша (`\`):

- `\n` - перемещает курсор в начальную позицию следующей строки;
- `\t` – перемещает курсор в следующую позицию табуляции экрана;
- `\r` – выполняет «возврат каретки», перемещая курсор к началу той же строки без перехода на следующую;
- `\b` – передвигает курсор только на одну позицию влево.

Функция scanf()

Функция scanf() является многоцелевой функцией, дающей возможность вводить данные любых типов. Указатели формата почти полностью соответствуют, используемым функцией printf(). Общий вид функции:

```
scanf(форматнаяСтрока, списокАргументов);
```

В качестве ***аргументов*** используются адреса (указатели) объектов т.е. запись &имяПеременной.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double x;
6      cout << "Введите x: ";
7      scanf("%lf", &x);
8      cout << " x = " << x << endl;
9      cout << "Еще раз введите x: ";
10     cin>> x;
11     cout << " x = " << x << endl;
12     return 0;
13 }
```

Результат работы программы:

```
Введите x: 3.5
 x = 3.5
Еще раз введите x: 5.1
 x = 5.1
```

Замечание.

Если нужно ввести значение строковой переменной, то использовать символ & не нужно. Как будет показано дальше: строка – массив символов, а имя массива является адресом его первого элемента.

Дополнительные функции символьного ввода/вывода языка С

Функция `putchar()` осуществляет ***вывод*** символа (значения переменной или символьной макроподстановки) на экран. Требует подключения `#include <stdio.h>`.

Пример.

```
…
#define INITIAL 'H'
putchar(INITIAL);
…
```

Функция `getchar()` ***вводит*** с клавиатуры единичный символ.

Пример.

```
…
int letter;
letter = getchar();
…
```

Операторы управления программой

Все операторы управления программой могут быть условно разделены на следующие категории:

- составные операторы;
- операторы выбора, к которым относятся оператор условия `if()` и оператор выбора `switch();`
- операторы цикла (`for(), while(), do while()`);
- операторы перехода (`break, continue, return, goto`).

Составной оператор

Любая последовательность операторов, заключенная в фигурные скобки `{}`, может выступать в любой синтаксической конструкции как один составной оператор (блок):

Пример.

```
{
    a = b + 2;
    b++;
}
```

Он позволяет рассматривать группу операторов как единое выражение.

Область видимости переменных

Областью действия объекта (данного) называется та часть программы, в которой можно пользоваться этим объектом. В частности, областью действия может быть:

- блок операторов (составной оператор - `{…}`);
- модуль (файл);
- вся программа в целом.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main ( ) {
5      {
6          int a = 3, b = 0;
7          a = b + 2;
8          b++;
9          cout << "В блоке a = " << a
10             << ", b ="<< b << "\n";
11     }
12 // за пределами блока переменные a и b не видны -
13 //ошибка компиляции
```

```
14      cout<< "За пределами блока a = " << a
15          << ", b ="<< b << "\n";
16      return 0;
17  }
```

Результатом выполнения программы будут ошибки при компиляции:

```
input
Compilation failed due to following error(s).
main.cpp: In function ‘int main()’:
main.cpp:14:57: error: ‘a’ was not declared in this scope
   14 |     cout<< "За пределами блока a = " << a
      |                                                         ^
main.cpp:15:22: error: ‘b’ was not declared in this scope
   15 |         << ", b ="<< b << "\n";
      |                      ^
```

Условные операторы

Оператор `if()`

Формат оператора:

- полная форма:

```
if (условие-выражение) оператор1;
else оператор2;
```

- сокращенная форма:

```
if (условие-выражение) оператор1;
```

Выполнение оператора `if()` начинается с вычисления `условия-выражения`. В качестве `условия-выражения` может использоваться:

- числовая или логическая переменная,
- арифметическое выражение,
- операции отношения (сравнения),
- логическое выражение.

Далее выполнение осуществляется по следующей схеме:

- если `условие-выражение` ***истинно*** (имеет значение `true` или арифметическое значение отлично от `0`), то выполняется `оператор1`,

- если `условие-выражение` ***ложно*** (имеет значение `false` или арифметическое значение равно `0`), то выполняется `оператор2`.

После этого управление передается на следующий после `else` оператор.

В сокращенной форме оператора если `условие-выражение` ***ложно***, то выполняется следующий за `if()` оператор.

Пример.

```
if (i < j) i++;
else {
    j = i - 3;
    i++;
}
```

Простейший пример с ветвлением программы

Проверим меньше ли заданной константы `LIMIT` введенное нами число. Определение константы заменим макроподстановкой (директива `#define`).

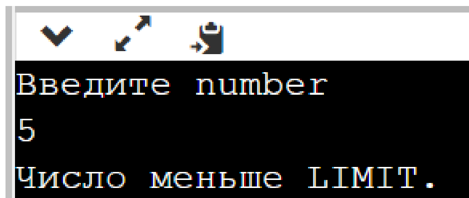
Пример.

main.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   #define LIMIT 10
5
6   int main() {
7       int number;
8
9       std::cout << "Введите number"<< std::endl;
10      std::cin >> number;
11
12      if (number < LIMIT) {
13          // Если введенное число меньше 10.
14          std::cout << "Число меньше LIMIT."
15                    << std::endl;
16      }
```

```
17      else {
18          // иначе
19          std::cout << "Число больше либо равно LIMIT."
20                    << std::endl;
21      }
22      return 0;
23  }
```

Результат выполнения программы:



Приведем аналог этой же программы на Pascal-е.

Пример.

main.pas

```
 1  program main;
 2  //константы и переменные в Pascal-е всегда
 3  //глобальные по отношению к телу программы
 4  //и любой процедуре или функции, объявленной
 5  //внутри программы
 6  const LIMIT : integer = 10;
 7  var number : integer;
 8  //тело программы
 9  begin              // в C++ это {
10      //writeln() в C++ это cout + endl
11      writeln('Введите number');
12      //readln() в C++ это cin
13      readln(number);
14
15      if Number < LIMIT then begin // в C++ begin это {
16          //writeln() в C++ это cout + endl
17          writeln('Число меньше LIMIT');
18      end            // в C++ это }
19      else begin     // в C++ begin это {
20          //writeln() в C++ это cout + endl
21          writeln('Число больше либо равно LIMIT.');
22      end;           // в C++ это }
23  end.               // в C++ это }
```

Результат работы этой программы такой же как в предыдущем случае (как у программы на С++).

Одними из простейших примеров использования оператора `if()` является вычисление модуля переменной любого типа, а также определения попадает ли введение числовое значение в заданный интервал `]-a, a[`.

Пример.

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    double a = 4;
    double x;

    std::cout<< "Введите x ";
    std::cin>> x;

    if (x < 0) x = -x;
    std:: cout<< "Модуль x = "<< x <<std::endl;

    bool isBelong = false;
    if (-a < x && x < a) isBelong = true;

    std::cout<< "x принадлежит интервалу? "
              << std::boolalpha
              << isBelong;

    return 0;
}
```

Результат работы программы:

```
Введите x -8
Модуль x = 8
x принадлежит интервалу? false
```

Задача о попадании в круг

Написать программу, определяющую попадают ли введенные координаты в область круга с центром в начале координат и радиусом, определенным константой RADIUS (Рисунок 1), объявленной с помощью служебного слова **const**.

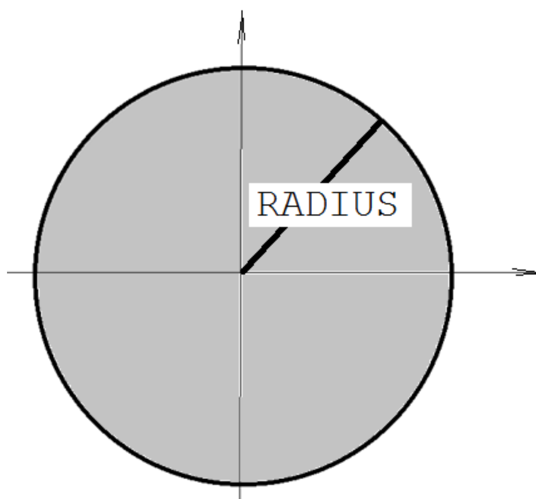


Рисунок 1 – Круг радиуса RADIUS с центром в начале координат

Зачастую при программировании под консоль в профессиональных интегрированных средах разработки возникает проблема отображения русского языка в сообщениях, выводимых на экран. Ниже продемонстрирован один из самых прямолинейных способов избежать данной проблемы – писать сообщения на английском языке.

Замечания:

- *использование транслитерации (замена русских букв на латинские) считается непрофессиональным, поэтому рекомендуется этого не делать;*
- *для того, чтобы писать сообщения на русском языке следует использовать функцию* `setlocale()` *внутри функции* `main()` *перед выводом первого сообщения в формате* `setlocale(LC_ALL, "Russian")`, *которое можно заменить на* `setlocale(0, "")` *либо* `setlocale(LC_ALL, "ru_RU")`*;*
- *при программировании в онлайн средах разработки проблем с выводом русскоязычных сообщений нет.*

Пример.

```
main.cpp
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  //объявление глобальной константы
6  const float RADIUS = 1.5;
7
8  int main() {
9      float xPoint, yPoint;
10
11     cout << "Input X and Y coordinates"<<endl;
12     cin >> xPoint>> yPoint;
13
14     if ( xPoint * xPoint + yPoint * yPoint <= pow(RADIUS, 2)) {
15         cout << "The point belongs to the circle"
16              << endl;
17     }
18     else {
19         cout <<"The point is out of the circle"
20              << endl;
21     }
22     return 0;
23 }
```

Замечание.

Пропуск одной или обеих фигурных скобок, ограничивающих составной оператор (блок) – типичная ошибка программирования. Для того, чтобы избежать ошибок, нужно сначала записать открывающую и закрывающую скобки составного оператора (блока), а потом вписать требуемые операторы.

Попадание в четверть круга

Обычным для определения условий попадания в геометрическую область любой конфигурации является использование простейших подобластей, пересечение которых дает искомую область. В данном случае четверть круга (Рисунок 2) является пересечением круга (Рисунок 1) и двух полуплоскостей `x < 0` и `y > 0`.

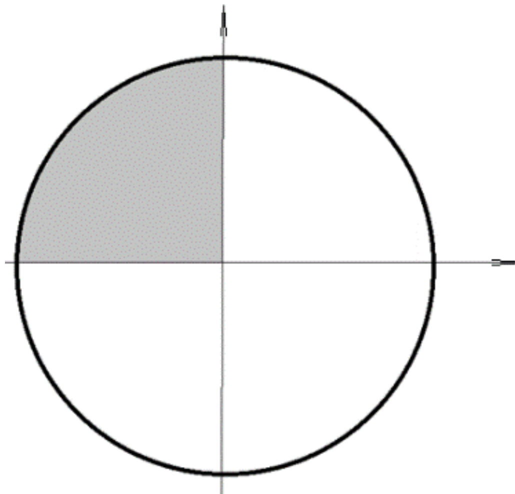


Рисунок 2 – Четверть круга с центром в начале координат

Решение этой усложненной задачи сводится к замене в предыдущей программе оператора `if()` следующим кодом:

```
...
if (xPoint*xPoint + yPoint*yPoint <= pow(RADIUS,2))
    if (yPoint > 0)
        if (xPoint < 0) {
            cout<<" hit the figure " << endl;
        }
        else {
            cout<<" don't hit the figure "<< endl;
        }
...
```

Последний участок программы можно переписать в частично сокращенном варианте со сложным условием:

```
...
if (xPoint*xPoint + yPoint*yPoint <= pow(RADIUS,2))
    if ( 0 < yPoint && xPoint < 0)  {
        cout<<" hit the figure "<<endl;
    }
    else {
        cout<<" don't hit the figure "<< endl;
    }
...
```

Рассмотрим еще одни вариант представления сложного условия с одним `if()`:

```
...
if (xPoint * xPoint + yPoint * yPoint <= pow(RADIUS,2)
        && 0 < yPoint
        && xPoint < 0) {
    cout<<" hit the figure "<<endl;
}
else {
    cout<<" don't hit the figure "<< endl;
}
...
```

Для повышения читаемости сложного условия в последнем примере стоит использовать вспомогательные переменные булевского типа:

```
...
//Переменная принимающая значение true, когда точка
//принадлежит кругу
bool isCircle = xPoint * xPoint + yPoint * yPoint <=
            pow(RADIUS,2);
//Переменная, принимающая значение true, когда точка
//второй четверти координатной плоскости
bool isQuarterPlane = xPoint < 0 && 0 < yPoint;

if (isCircle && isQuarterPlane) {
    cout<<" hit the figure "<<endl;
}
else {
    cout<<" don't hit the figure "<< endl;
}
...
```

Общее представление о кластеризации

Кластерный анализ (англ. `Cluster analysis`) – многомерная (Рисунок 3) статистическая процедура, выполняющая сбор данных, содержащих информацию о выборке объектов, и затем упорядочивающая объекты в сравнительно однородные группы (Рисунок 4).

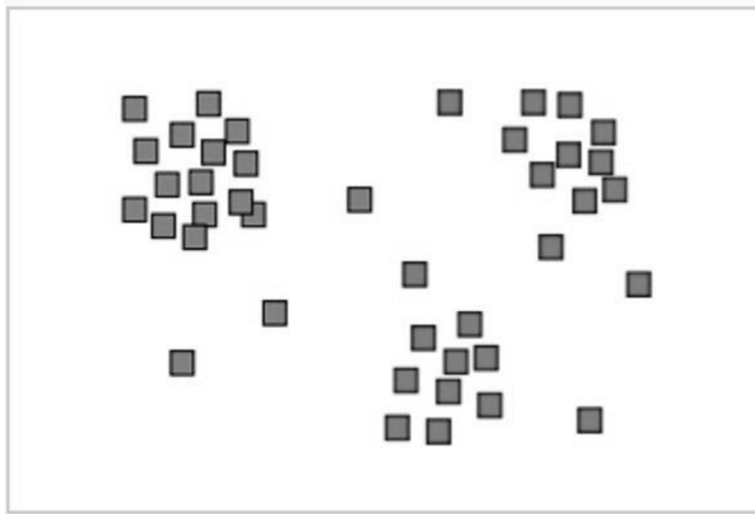


Рисунок 3 - Исходная выборка объектов, имеющих две характеристики

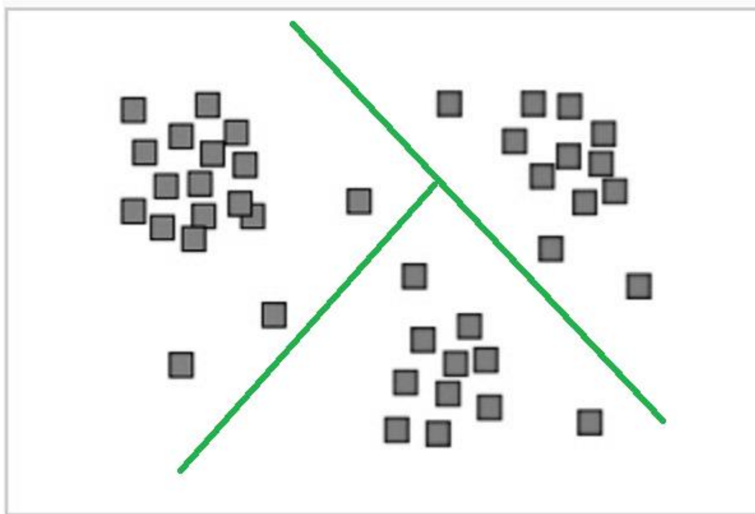


Рисунок 4 - Построение границ кластеров в виде линий с использованием статистических методов

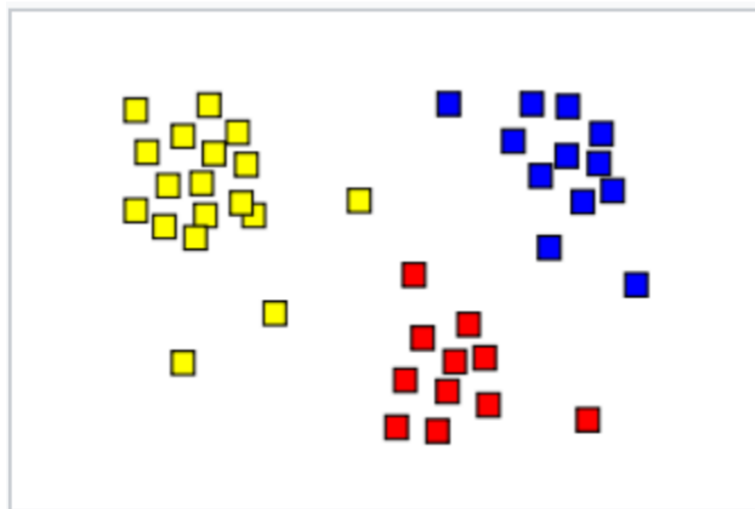


Рисунок 5 – Пример результата кластерного анализа

К каждому выделенному кластеру могут применяться в дальнейшем разные группы функций, обрабатывающие соответствующие ему множества точек. Фактически целью кластеризации является возможность индивидуализации для каждой группы применяемых средств статистического анализа и разработке на этой основе более точных методов предсказания поведения объектов, принадлежащих разным кластерам.

Пример кластеризации на основе решения задачи о попадании двумерной точки в треугольник

Рассмотрим треугольник, ограниченный линиями: `y = x;` `y = 0.3 * x;` `y = -x + 2` (Рисунок 6).

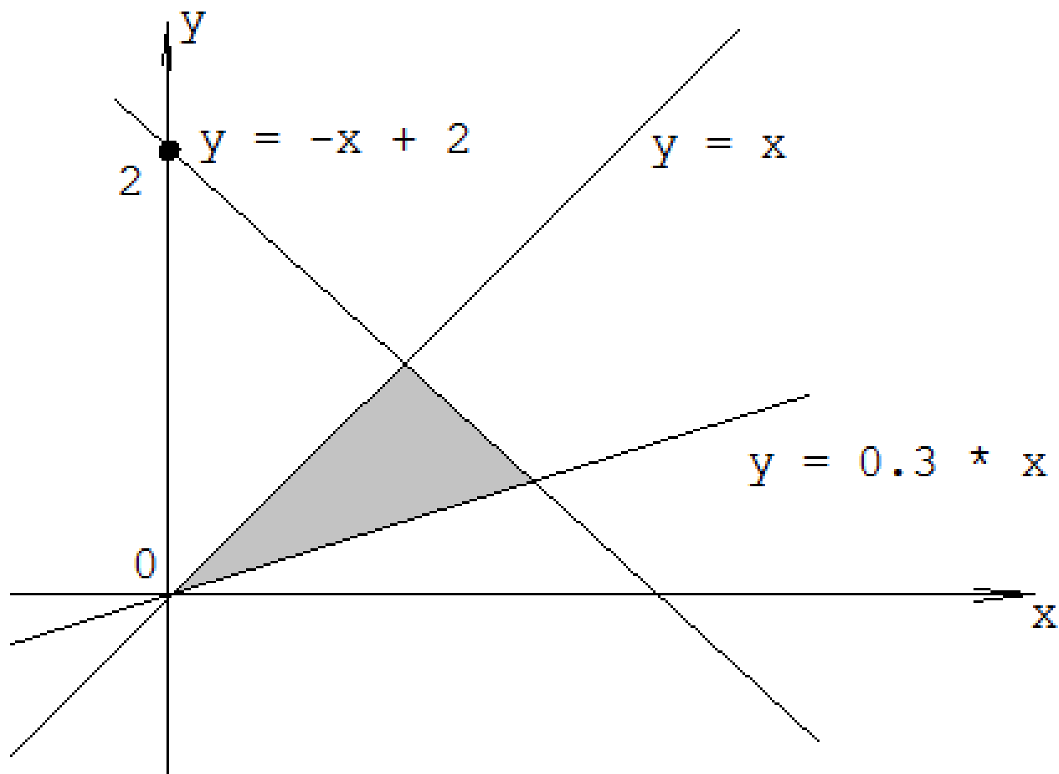
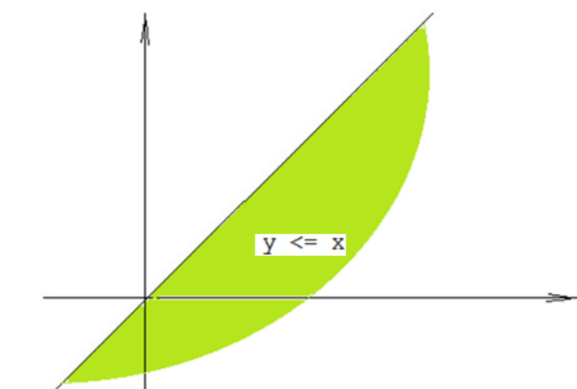
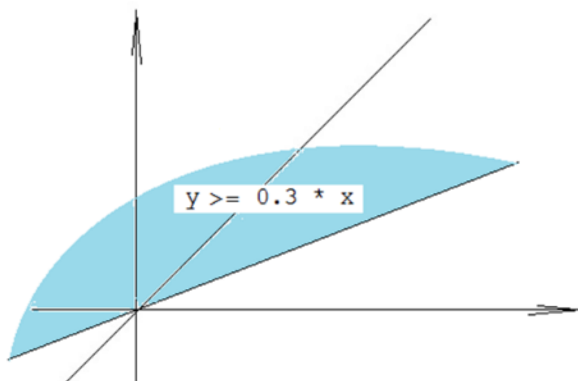


Рисунок 6 – Треугольная область

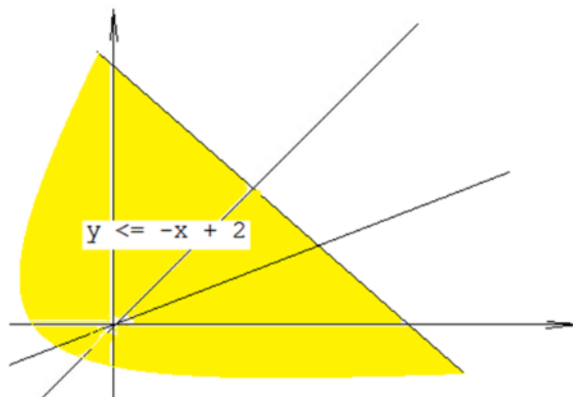
Для того, чтобы понять каким образом программируются условия принадлежности объекта к заданной области (например, треугольник), необходимо ее разбить на простейшие подобласти (полуплоскости), которые в пересечении дают искомый треугольник (Рисунок 7).



а)



б)



в)

Рисунок 7 - Разбиение на вспомогательные области

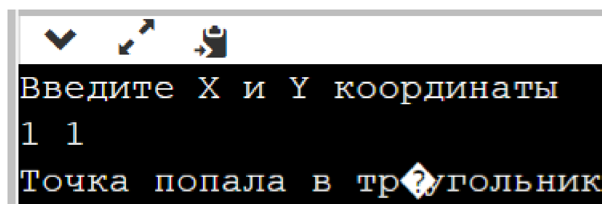
Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      float x, y;
5      std::cout << "Введите X и Y координаты"
6                <<std::endl;
```

```
    std::cin >> x >> y;

    if ((y <= x) && (y >= 0.3) && (y <= -x + 2)) {
        std::cout << "Точка попала в треугольник"
                  << std::endl;
    }
    else {
        std::cout <<"Точка не попала в треугольник"
                  << std::endl;
    }
    return 0;
}
```

Результат работы программы:



Лестница `if-else-if`

Одной из распространенных конструкций языка С++ является лестница `if-else-if`.

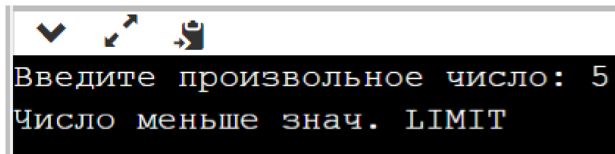
Пример.

main.cpp

```
#include <iostream>
using namespace std;
#define LIMIT 10

int main() {
    double x;

    cout << "Введите произвольное число: ";
    cin >> x;

    if (x < LIMIT)
        cout <<"Число меньше знач. LIMIT"<<endl;
```

```
13      else if (x == LIMIT)
14          cout <<"Число равно знач. LIMIT."<<endl;
15      else  // иначе
16          cout <<"Число больше знач. LIMIT."<<endl;
17      return 0;
18  }
```

Результат работы программы:


```
Введите произвольное число: 5
Число меньше знач. LIMIT
```

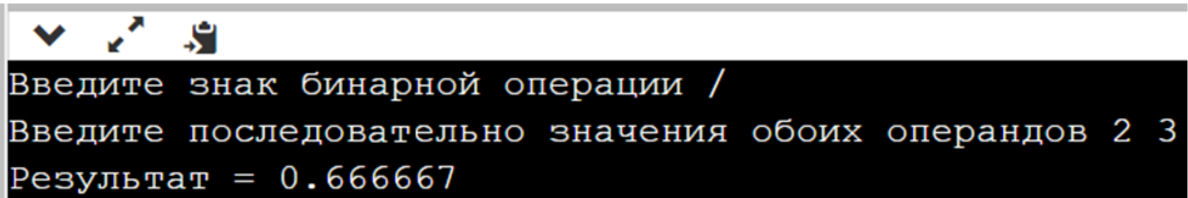
Следующий пример лестницы `if-else-if` имеет более сложный вид. Это элементарный пример организации консольного калькулятора.

Пример.

main.cpp
```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       char OperationSing;
6       double x, y, z;
7
8       cout<< "Введите знак бинарной операции ";
9       cin>> OperationSing;
10      cout<< "Введите последовательно значения "
11          << "обоих операндов ";
12      cin >> x;
13      cin >> y;
14
15      if (OperationSing == '-') z = x - y;
16      else if (OperationSing == '+') z = x + y;
17      else if (OperationSing == '*') z = x * y;
18      else if (OperationSing == '/') z = x / y;
19      else {
20          cout<<boolalpha << false;
```

```
21            return 0; //окончание программы
22        }
23        cout <<"Результат = "<< z;
24        return 0;
25    }
```

Результат работы программы:



Оператор выбора switch()

Другим способом организации выбора из множества различных вариантов является использование оператора выбора `switch()`. Оператор имеет следующий формат:

```
switch (выражение) {
     case константное-выражение1:
               список-операторов1;
          break;
     case константное-выражение2:
               список-операторов2;
          break;
     …
     default:
               список операторов;
          break;
}
```

Выражение, следующее за ключевым словом `switch()` в круглых скобках, может быть любым выражением, допустимым в языке C++, ***значение которого будет исключительно целым***. Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора `switch` состоит из нескольких операторов, помеченных ключевым словом `case` с последующим константным выражением.

Обычно в качестве константного выражения используются целые или символьные константы (но ***не*** переменные или вызовы функций). Все константные выражения в операторе `switch()` должны быть уникальны

(т.е. не должны повторяться). Кроме операторов, помеченных ключевым словом `case`, может быть только один фрагмент, помеченный ключевым словом `default`.

Список операторов может быть пустым либо содержать один оператор или более. В операторе `switch()` не требуется заключать в фигурные скобки последовательность операторов. ***Для окончания*** действий по одному из вариантов `case` присутствует оператор `break`**.**

Схема выполнения оператора switch()

Схема выполнения оператора `switch()` следующая (Таблица 21):

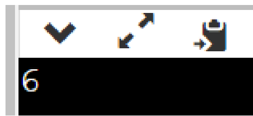
- вычисляется выражение в круглых скобках;
- вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами `case`;
- если одно из константных выражений совпадает со значением выражения, то управление передается на оператор (или список операторов), соответствующий совпавшему варианту значения `case`;
- если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом `default`, а в случае его отсутствия управление передается на следующий после `switch()` оператор.

Таблица 21 – Сопоставление синтаксисов операторов выбора в C++ и Pascal

main.cpp	main.pas

<pre>1 #include <iostream>
2 using namespace std;
3
4 int main() {
5   int i = 2;
6   switch (i) {
7   case 1: i += 2; break;
8   case 2: i *= 3; break;
9   case 0: i %= 2; break;
10   case 4: i -= 5; break;
11   default: ; break;
12   }
13   cout<< i << endl;
14   return 0;
15 }</pre>
	<pre>1 program main;
	2 var i : integer = 2;
	3 begin
	4 case i of // switch()
	5 1: i:= i + 2;
	6 2: i := i * 3;
	7 0: i := i mod 2;
	8 4: i := i - 5;
	9 else ; //default
	10 end;
	11 writeln(i);
	12 end.</pre>

Результат выполнения программы на языке С++ (на Pascal результат тот же):

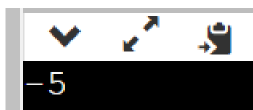


Операторы `break` могут отсутствовать, тогда действия выполняются по списку операторов подряд без проверки `case` до появления следующего оператора `break`.

Пример.

main.cpp
```
#include <iostream>
using namespace std;

int main() {
    int i = 2;
    switch (i) {
        case 1: i += 2; break;
        case 2: i *= 3;
        case 0: i %= 2;
        case 4: i -= 5; break;
        default: ; break;
    }
    cout<< i << endl;
    return 0;
}
```

Результат работы программы:



Пример использования switch() совместно с enum

Рассмотренный в предыдущем разделе пример использования оператора выбора `switch()` явно демонстрирует, что с ним следует как можно чаще использовать перечисления.

Это, собственно говоря, следует из требований к оформлению констант в программе: необходимо использовать именованные константы и максимально сократить применение неименованных «волшебных» констант в тексте программы, а также того, что после каждого `case` в операторе `switch()` стоит одна из уникальных констант, обычно изменяющаяся согласно какому-то порядку (возрастания/убывания).

В качестве примера напишем небольшую программу элементарного переводчика цифр 0…9 в словесное англоязычное представление.

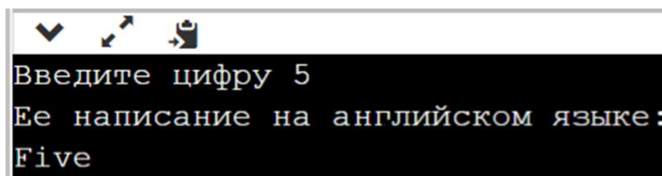
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      enum {
6          ZERO = 0, ONE, TWO, THREE, FOUR,
7          FIVE, SIX, SEVEN, EIGHT, NINE
8      };
9      unsigned i;
10     cout<< "Введите цифру ";
11     cin>> i;
12
13     cout<< "Ее написание на английском языке:\n";
14
15     switch (i) {
16         case ZERO:  cout << "Zero"; break;
17         case ONE:   cout << "One"; break;
18         case TWO:   cout << "Two"; break;
19         case THREE: cout << "Three"; break;
20         case FOUR:  cout << "Four"; break;
21         case FIVE:  cout << "Five"; break;
22         case SIX:   cout << "Six"; break;
23         case SEVEN: cout << "Seven"; break;
24         case EIGHT: cout << "Eight"; break;
```

```
25            case NINE:   cout << "Nine"; break;
26            default:     cout << "false"; break;
27        }
28        return 0;
29    }
```

Результат работы программы:



Требования к оформлению условных операторов

Для `if()`:

- по ветке `true` в `if()` должен идти наиболее вероятный вариант действий, а после `else` (ветка `false`) – наименее вероятный;
- отсутствие `else` следует комментировать;
- очень сложные условия следует разделять на отдельные булевские переменные или выносить в функции;
- в случае вложенных `if()` первыми должны проверяться наиболее общие условия;
- следует рассмотреть не имеет ли смысл заменить лестницу `if-else-if` на `switch()`;
- операторы, выполняющиеся по условию, не следует располагать в одной строке с `if()` или `else`.

В условии-выражении оператора `if()` ***не*** рекомендуется:

- использовать операции «`==`» и «`!=`» для сравнения значения с плавающей точкой с нулем или любым целым числом;
- ставить одну переменную (или вызов функции), не сравнивая ее с каким-либо другим выражением (`if (a)` – плохо, `if (a!=0)` – хорошо).

Для `switch()`:

- все перечисляемые значения должны быть выстроены в логическом порядке (например, по возрастанию);
- действия для любого из пунктов (любого `case`) должны быть простыми;

- если необходимо, то следует вынести последовательность действий в функцию разработчика, а в соответствующем `case` осуществить ее вызов;
- `default` следует использовать для вывода сообщения об ошибках (другие применения не желательны);
- `default` рекомендуется использовать всегда даже если кроме пустого оператора никакой последовательности действий нет.

Операторы циклов

При выполнении программы нередко возникает необходимость неоднократного повторения однотипных вычислений над различными данными. Для этих целей используются циклы. Цикл – участок программы, в котором одни и те же вычисления реализуются неоднократно над различными значениями одних и тех же переменных (объектов).

Для организации циклов в C++ используются следующие операторы:

- `for()`,
- `while()`,
- `do while()`.

Оператор цикла for()

Цикл `for()` является циклом с параметрами и обычно используется в случае, когда известно точное количество повторов вычислений. Однако синтаксис этого оператора в C++ настолько гибок, что позволяет также использовать его и для организации вычислений с заранее ***не***известным числом повторений, например, для вычислений с точностью.

Таким образом, оператор `for()` – это наиболее общий способ организации цикла в языке C++.

Он имеет следующий формат:

```
for(выражение1; условие-выражение; выражение2) оператор;
```

где:

- `выражение1` обычно используется для установления начального значения переменных, управляющих циклом (может содержать несколько операторов, разделенных запятыми);
- `условие-выражение` – это конструкция, определяющая условие, при котором ***тело цикла*** (`оператор;`) будет выполняться;

- `выражение2` обычно определяет изменение переменной, управляющей циклом после каждого выполнения ***тела цикла*** (также может содержать несколько операторов, разделенных запятыми).

Схема выполнения оператора `for()`:

- вычисляется `выражение1`;
- вычисляется `условие-выражение`;
- если значение `условия-выражения` равно `true` в смысле булевского типа (либо отлично от нуля в смысле арифметического типа), то выполняется ***тело цикла*** (`оператор;`);
- далее вычисляется `выражение2` и осуществляется переход к повторной проверке `условия-выражения`;
- если `условие-выражение` равно `false` в смысле булевского типа (либо равно нулю в смысле арифметического типа), то управление передается на оператор, следующий за оператором `for()`.

Проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Вывод на экран квадратов натуральных чисел до 10 включительно

Рассмотрим, каким образом работает оператор цикла при выводе квадратов чисел на экран (Таблица 22).

Таблица 22 – Сопоставление программ с циклами for на C++ и Pascal

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int i;
    cout << "Квадраты "
         <<"натуральных "
         << "чисел до 10 \n";
    for(i = 1; i <= 10; i++) {
        cout<<' '<<i * i;
    }
    return 0;
}
```

main.pas

```pascal
program main;
var i : integer;
begin
    writeln('Квадраты',
        'натуральных ',
        'чисел до 10');
    for i := 1 to 10 do
    begin
        write(' ', i*i);
    end;
end.
```

Результат работы программы на C++:

```
Квадраты натуральных чисел до 10
 1 4 9 16 25 36 49 64 81 100
```

Продемонстрируем так называемую таблицу ручного счета алгоритма предыдущей программы на C++ (Таблица 23).

Замечание.

В таблице ручного счета (Таблица 23) вместо пробела для наглядности используется нижнее подчеркивание.

Таблица 23 – Вывод в цикле значений квадратов на экран (программа на C++)

Переменная/ действие	Значения				
i	1	2	3	4	...
i < 10	true	true	true	true	...
cout	1*1 = 1	2*2 = 4	3*3 = 9	4*4 = 16	...
Результат на экране	_1	_1_4	_1_4_9	_1_4_9_16	...

Поскольку таблица ручного счета может вызвать вопросы и читателю непонятно, почему каждый раз в строке вывода добавляется конструкция «пробел-число в квадрате», то перепишем предыдущий пример слегка его видоизменив. Изменения будут касаться вывода на экран квадратов натуральных чисел.

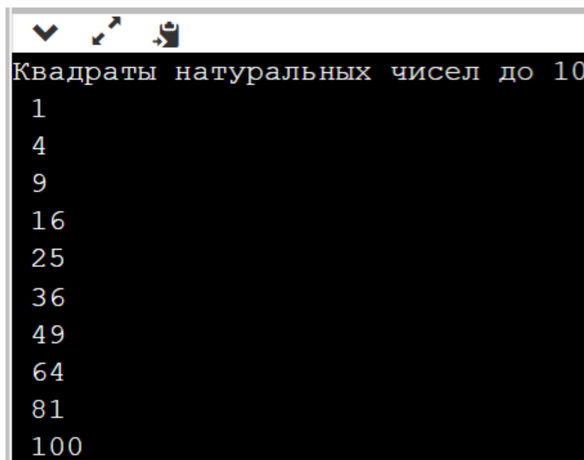
Пример.

main.cpp

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6      int i;
7      cout<< "Квадраты натуральных чисел до 10 \n";
```

```
 8      for(i = 1; i <= 10; i++) {
 9          cout << ' '<< i*i << endl;
10      }
11      return 0;
12  }
```

Результаты работы программы:



В данном случае начинающий программист может убедиться, что в рассмотренном алгоритме и в первой программе, и во второй выводимое в цикле сообщение представляет из себя конструкцию «`пробел-число в квадрате`», только в первом примере каждое сообщение печатается в одну строку за предыдущим, а во втором - эти сообщения выстроены в столбец.

Кроме того, отметим, что цикл в приведенном выше примере не выполниться вовсе если изменить условие-выражение.

<u>Пример.</u>

```
int i;
for(i = 1; i > 10; i++) cout<<' '<<i * i;
```

<u>Замечание.</u>

По своей сути предыдущий пример не только демонстрирует каким образом формируется строка вывода, но и представляет из себя пример простейшего тестирования (отладки) циклического алгоритма с помощью отладочной печати (`cout`).

Суммирование квадратов натуральных чисел до 10 включительно

Приведем формальное (математическое) описание требуемого результата с помощью формулы:

$$\sum_{i=1}^{10} i^2 = 1 + 4 + \cdots + i^2 + \cdots + 100.$$

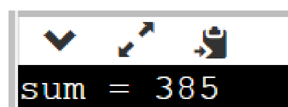
Перейдем к демонстрации реализации алгоритма с помощью оператора `for()`.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    for(int i = 1; i <= 10; i++) {
        sum = sum + i * i;
    }
    cout << "sum = " << sum;
    return 0;
}
```

Результат работы программы:



Работа оператора цикла при вычислении суммы $\sum_{i=1}^{10} i^2$ описывается таблицей (Таблица 24).

Таблица 24 – Вычисление суммы квадратов целых чисел

Переменная/ действие	До входа в цикл	Значения в цикле			
i	---	1	2	3	…
i < 10	---	true	true	true	…
sum	0	0 + 1*1 = 1	1 + 2*2 = 5	5 + 3*3 = 14	…

Вернемся к обсуждению таблицы ручного счета. Обычно начинающему программисту хотелось бы увидеть эту таблицу собственными глазами, сформированную в ходе выполнения программы.

Воспользуемся тем же приемом, что и в случае с выводом на консоль квадратов натуральных чисел – немного изменим программу суммирования квадратов.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int sum = 0;
6      cout << "Таблица ручного счета\n" << boolalpha;
7      //формирование шапки таблицы
8      cout << "i" << " " << "i<=10" << " " << "sum" << endl;
9      //вывод значений парамеров до входа в цикл
10     cout << "-" << " " << "----" << "  " << sum << endl;
11     for(int i = 1; i <= 10; i++) {
12         sum = sum + i * i;
13         //формирование столбцов в цикле
14         cout << i << " " << (i<=10) << "  " << sum << endl;
15     }
16     cout << "sum = " << sum;
17     return 0;
18 }
```

Замечания:

- *внесенные в программу изменения выделены красным;*
- *никаких изменений в алгоритме не производилось*

Результат работы программы:

```
Таблица ручного счета
i i<=10 sum
- ----  0
1 true  1
2 true  5
3 true  14
4 true  30
```

```
5 true  55
6 true  91
7 true  140
8 true  204
9 true  285
10 true  385
sum = 385
```

Если сопоставить этот результат и значения, приведенные в таблице ручного счета (Таблица 24), то с очевидностью можно заметить, что строки указанной таблицы являются столбцами в последнем выводе на консоль результатов работы измененной программы.

Замечание.
Таблица ручного счета – это аналитическое исполнение кода программы программистом.

Синтаксисом допускаются и другие конструкции, организующие вычисление суммы квадратов целых чисел до 10.

- инициализация нескольких переменных в первой секции `for()` (тело программы):

```
int i, sum;
for(sum = 0, i = 1; i <= 10; i++) {
    sum = sum + i * i;
}
```

- использование пустого оператора в качестве тела цикла:

```
int sum = 0;
for(int i = 1; i <= 10; sum += i*i, i++);
```

- отсутствуют `выражение1` и `выражение2` в заголовке оператора (тело программы):

```
int sum = 0, i = 1;
for( ; i <= 10; ) {
    sum += i*i;
    i++;
}
```

Вычисление в цикле факториала целого числа

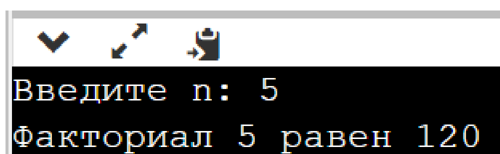
Вычислим в цикле факториал натурального числа n:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n\,.$$

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int n, product;
6      cout << "Введите n: ";
7      cin >> n;
8
9      product = 1;
10     for(int i = 2; i <= n; i++) {
11         product = product * i;
12     }
13     cout<< "Факториал "<< n
14         << " равен " << product<< endl;
15     return 0;
16 }
```

Результат работы программы:



Работа оператора цикла при вычислении факториала $n!$ описывается таблицей (Таблица 25).

Таблица 25 – Вычисление факториала целого числа

Переменная/ действие	До входа в цикл	Значения в цикле			
i	---	2	3	4	…
i <= n	---	true	true	true	…
product	1	1*2	1*2*3	1*2*3*4	…

Двойной факториал целого числа

Вычислим в цикле $n!!$ (двойной факториал натурального n). Его определение различается для четного и нечетного n:

- для нечетного - $n!! = 1 \cdot 3 \cdot 5 \cdot \ldots \cdot n$;
- для четного - $n!! = 2 \cdot 4 \cdot 6 \cdot \ldots \cdot n$.

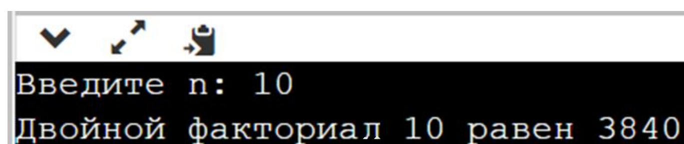
Для решения поставленной задачи следует использовать не увеличение параметра цикла `i`, а его уменьшение причем уменьшение не на `1`, а на `2`.

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int n, product;
    cout << "Введите n: ";
    cin >> n;

    product = 1;
    for(int i = n; i > 0; i = i - 2) {
        product = product * i;
    }
    cout << "Двойной факториал " << n
         << " равен " << product;
    return 0;
}
```

Результат работы программы:



```
Введите n: 10
Двойной факториал 10 равен 3840
```

Работа оператора цикла при вычислении двойного факториала $n!!$ числа `10` описывается таблицей (Таблица 26).

Таблица 26 – Вычисление двойного факториала числа 10

Переменная/ действие	До входа в цикл	Значения в цикле			
i	---	10	8	6	...
i > 0	---	true	true	true	...
product	1	1*10	1*10*8	1*10*8*6	...

Возведение в целую степень n переменной x в цикле

Возведем в цикле число с плавающей точкой x в целую степень n:

$$x^n = 1 \cdot \underbrace{x \cdot x \cdot x \cdot \ldots \cdot x}_{n \text{ раз}}.$$

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    int n;
    double product, x;
    cout << "Введите значения x, n "
         << "через пробел: ";
    cin >> x >> n;

    product = 1;
    for(int i = 1; i <= n; i++) {
        product = product * x;
    }
    cout<< n << "-я степень "<< x
        << " равна " << product;
    return 0;
}
```

Результат выполнения программы:

```
Введите значения x, n через пробел: 2 3
3-я степень 2 равна 8
```

Работа оператора цикла при вычислении выражения x^n описывается таблицей (Таблица 27).

Таблица 27 – Вычисление целой степени переменной x

Переменная/ действие	До входа в цикл	Значения в цикле			
i	---	1	2	3	…
i <= n	---	true	true	true	…
product	1	1*x	1*x*x	1*x*x*x	…

Вычисление в цикле выражения с меняющимся знаком в зависимости от четности/нечетности степени

Вычислим в цикле выражение $(-1)^n \cdot x^n$.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int n;
6      double product, x;
7      cout << "Введите значения x, n "
8           << "через пробел: ";
9      cin >> x >> n;
10
11     product = 1;
12     for(int i = 1; i <= n; i++) {
13         product = product * (-1) * x;
14     }
15     cout<<"Результат = "<< product;
16     return 0;
17 }
```

Результат выполнения программы:

```
Введите значения x, n через пробел: 2 3
Результат = -8
```

Работа оператора цикла при вычислении выражения $(-1)^n \cdot x^n$ описывается таблицей (Таблица 28).

Таблица 28 – Вычисление целой степени переменной с меняющимся в зависимости от четности или нечетности степени знаком

Переменная/ действие	До входа в цикл	Значения в цикле			
i	---	1	2	3	…
i <= n	---	true	true	true	…
product	1	1·(-1)·x	(-1)·x·(-1)·x	$(-1)^2 \cdot x^2 \cdot (-1) * x$	…

Пример использования в цикле параметра не целого типа

Пусть задана некоторая функция `f(x)` на отрезке `[a, b]`, а также целое число `n` равное количеству интервалов разбиения исходного отрезка `[a, b]` узловыми точками с шагом `h = (b - a) / n`. Процесс замены непрерывной функции набором ее значений в узлах разбиения исходного отрезка `[a, b]` будем называть дискретизацией значений непрерывной функции `f(x)` (Рисунок 8).

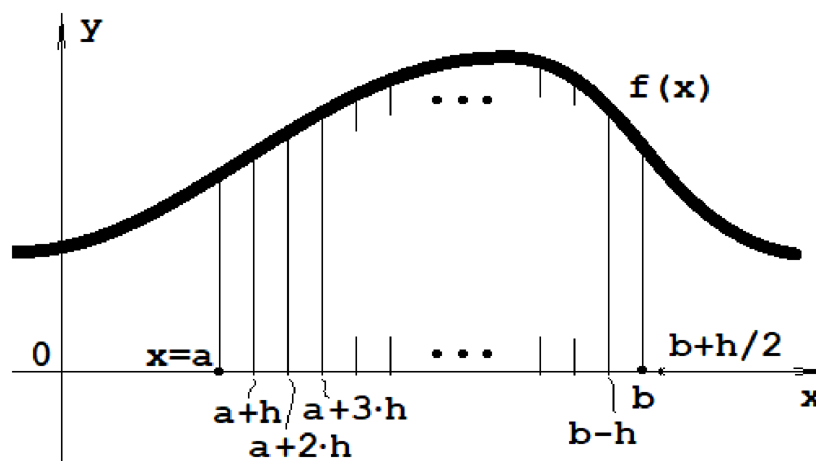


Рисунок 8 – Геометрическая интерпретация дискретизации функции

Характерной чертой всех рассмотренных в этом разделе примеров является то, что в цикле `for()` параметр цикла будет иметь типа `double`.

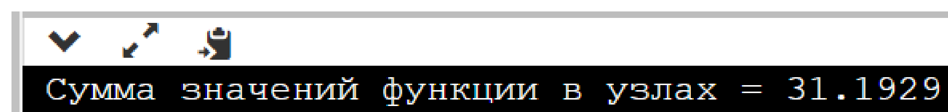
Пусть необходимо просуммировать дискретные значения функции `exp(x)` в равноотстоящих с шагом `h` узлах на интервале `[a, b]`.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3
4  int main() {
5      double a = 0.7, b = 3.5;
6      //n - количество интервалов, на которые
7      //будет разбит отрезок [a,b]
8      int n = 15;
9      //h - длина интервалов
10     double h = (b - a) / n;
11
12     double sum = 0;
13     for(double x; x < b + h/2; x++) {
14         sum = sum + exp(x);
15     }
16     std::cout<< " Сумма значений функции в узлах = "
17             << sum;
18
19     return 0;
20 }
```

Результат работы программы:



Кроме того, что впервые рассмотрен пример, в котором демонстрируется использование параметра не целого типа, второй характерной особенностью примера является использование выражения в качестве условия продолжения цикла:

```
x < b + h/2
```

Казалось бы, условие продолжения цикла точнее и проще записать в виде:

```
x <= b
```

Но, к сожалению, если в условии задачи указан замкнутый интервал (отрезок) `[a, b]` и возможна ситуация, когда будет необходимо проверить равенство `x == b`, то ***вероятным*** (но не обязательным) результатом проверки этого условия для двух чисел с плавающей точкой будет `false` из-за специфики представления чисел с плавающей точкой в памяти компьютера. Таким образом узел `b` и значение `exp(b)` ***вероятнее*** всего будет проигнорировано.

Поэтому к `b` следует добавить эмпирическое значение `h / 2`, гарантирующее, что значение `x` достигнет `b`, но не достигнет `b + h`.

С другой стороны, если в алгоритме задан ***не***замкнутый ***справа*** интервал `[a, b[` или достижение значения `b` не имеет существенного влияния на результат вычислений, то следует использовать более простое условие `x < b`.

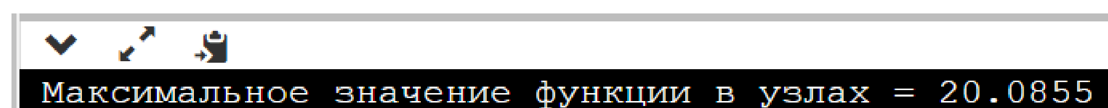
Далее рассмотрим программу, ***определяющую*** максимальное среди дискретных значений функции на заданном отрезке `[a, b]`.

Поскольку рассматриваемый пример сильно перекликается с предыдущим (операторы программы с `1` по `11` и с `18` по `20` полностью совпадают), то целесообразно привести в качестве примера только ту часть, которая касается изменений, т.е. участок программы с `12` по `17` операторы.

Пример

```
//участок программы с 12 по 17 операторы
12      double max = exp(a);
13      for(double x; x < b + h/2; x++) {
14          if(max < exp(x)) max = exp(x);
15      }
16      std::cout<< " Максимальное значение функции в узлах = "
17                  << max;
```

Результат работы программы:



В качестве последнего примера рассмотрим программу, которая определяет являются ли не убывающими дискретные значения функции `cos()` вычисленные в равноотстоящих узлах на отрезке `[a, b]`. Как и в

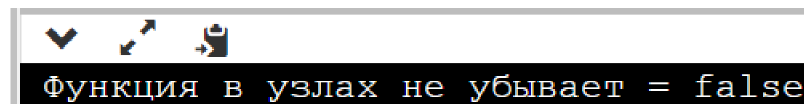
предыдущем случае приведем код только участка программы с 12 по 17 операторы.

Напомним, что последовательность значений называется не убывающей тогда, когда она на разных интервалах (или на всем интервале целиком) является возрастающей или постоянной.

Пример.

```
//участок программы с 12 по 17 операторы
12      bool increase = true;
13      for(double x; x < b - h/2; x++) {
14          if(cos(x) > cos(x + h)) increase = false;
15      }
16      std::cout<< " Функция в узлах не убывает = "
17                  <<std::boolalpha << increase;
```

Результат работы программы:



```
Функция в узлах не убывает = false
```

Замечания:

- *в разделе рассмотрены алгоритмы перебора узловых значений с плавающей точкой;*
- *все приведенные примеры в данном разделе демонстрируют, что нет необходимости в использовании вспомогательных массивов, составленных из узловых значений функции для выполнения рассмотренных действий.*

Оператор цикла while()

Оператор цикла while() называется циклом с предусловием и имеет следующий формат:

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

В качестве условия-выражения допускается использовать любое выражение языка C++, в качестве тела (инструкция оператор;) – любой оператор, в том числе пустой или составной (блок).

Схема выполнения оператора while() следующая:

- вычисляется условие-выражение:

- если `условие-выражение` ложно или нуль, то выполнение оператора `while()` заканчивается и выполняется следующий за `while()` по порядку оператор;
    - если выражение истинно или не нуль, то выполняется тело оператора (инструкция `оператор;`).
- далее управление передается на проверку `условия-выражения`.

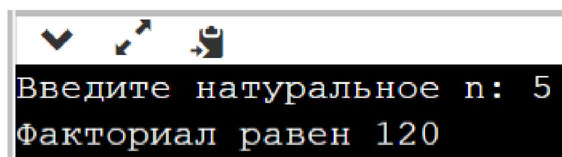
Перепишем пример вычисления $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ с помощью оператора цикла `while()`.

Пример.

main.cpp

```
 1 #include <iostream>
 2 using namespace std;
 3 
 4 int main() {
 5     int n;
 6     cout<<"Введите натуральное n: ";
 7     cin>>n;
 8 
 9     int i = 2, product = 1;
10     while(i <= n) {
11         product = product * i;
12         i++;
13     }
14     cout<< "Факториал равен "<< product;
15     return 0;
16 }
```

Результат работы программы:



```
Введите натуральное n: 5
Факториал равен 120
```

Оператор цикла do while()

Формат оператора имеет следующий вид:

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

Схема выполнения оператора `do while()`:

- выполняется тело цикла (инструкция `оператор;`), которое может быть составным оператором, т.е. блоком;
- вычисляется `условие-выражение`;
    - если `условие-выражение` ложно или нуль, то выполнение оператора `do while()` заканчивается и выполняется следующий по порядку оператор;
    - если `условие-выражение` истинно или не нуль, то выполняется тело цикла (инструкция `оператор;`).

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int n;
    cout << "Введите натуральное n: ";
    cin >> n;

    int i = 2, product = 1;
    do {
        product = product * i;
        i++;
    }
    while(i <= n);
    cout<< "Факториал равен "<< product<< endl;
    return 0;
}
```

Результат выполнения программы совпадает с предыдущим примером.

Замечание.

Тело цикла `do while()` *всегда выполниться хотя бы один раз даже при ложном или нулевом значении* `условия-выражения`. *Это и отличает* `do while()` *от* `while()`, *в котором тело цикла может не выполниться ни разу.*

Схема бесконечного цикла

С помощью цикла `for()`:

```
…
for(;true;) {
    //тело цикла
}
…
```

С помощью цикла `while()`:

```
…
while(true) {
    //тело цикла
}
…
```

Организация бесконечного цикла может понадобиться разработчику для создания защиты программы от неверного ввода данных. Она заключается в бесконечном продолжении цикла, пока пользователь не введет значения, которые удовлетворяют ограничениям на значения переменных, используемых далее в программе.

Конкретные примеры реализации защиты будут рассмотрены ниже после того, как будут обсужден оператор `break`.

Замечания:

В смысле простейшей организации защиты использование в ней оператора `do while()` *не целесообразно, т.к. этот цикл из-за неизбежного однократного выполнения его тела может допустить некорректный ввод данных.*

Вложенные циклы

Операторы `while()`, `for()` и `do while()` могут быть вложенными.

Пример.

```
while(условие-выражение1) {
    списокОператоров1;
    while(условие-выражение2) {
        списокОператоров2;
    }
    списокОператоров3;
}
```

Требования к оформлению операторов циклов

- даже для циклов с телом в виде одного оператора рекомендуется использовать операторные скобки (оформлять в виде блока);
- циклы без тела (с пустым оператором) не желательны;
- в циклах с неизвестным числом повторений операторы, имеющие отношение к управлению значением параметра цикла, следует собирать в конце или начале тела цикла;
- следует следить, чтобы цикл завершался при любых обстоятельствах;
- условия, при которых завершается выполнение цикла должны быть максимально простыми;
- количество уровней вложенности циклов не должно превышать трех;
- желательно, чтобы цикл был такой длины, чтобы целиком помещался в поле экрана монитора;
- в заголовке оператора `for()` должны участвовать только операторы, определяющие начальное значение и порядок изменения параметра цикла;
- в теле цикла `for()` нежелательно изменять счетчик (для этого предназначена третья секция в заголовке `for()`);
- присвоение начальных значений переменных цикла `while()` должно выполняться непосредственно перед заголовком цикла.

Операторы перехода

Оператор break

Оператор `break` изменяет порядок управления, он прерывает выполнение последовательности действий в составном операторе, в частности, в теле цикла. Его целесообразно использовать, когда выполнение операторов цикла или `switch()` следует прервать.

Формат оператора:

```
break;
```

Управление передается на оператор, следующий за блоком (например, телом оператора цикла), в котором используется этот оператор.

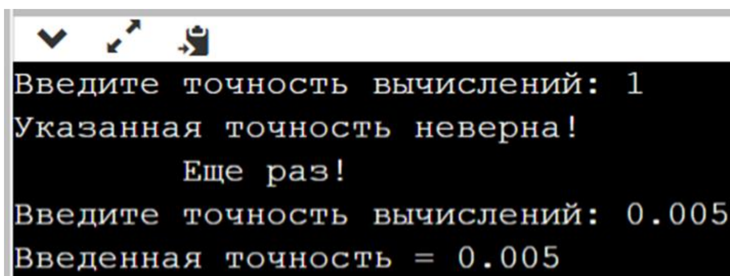
Рассмотрим пример использования `break` в бесконечном цикле проверки условия корректности ввода данных.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      double epsilon; //точность вычислений
6
7      while(true) {
8          cout<<"Введите точность вычислений: ";
9          cin>> epsilon;
10
11         if (0 < epsilon && epsilon < 0.05) break;
12
13         cout<<"Указанная точность неверна! "
14             << endl <<"\tЕще раз!" << endl;
15     }
16     cout<<"Введенная точность = "<< epsilon;
17     return 0;
18 }
```

Результат работы программы:



Оператор continue

Оператор `continue`, в отличие от `break` передает управление из тела цикла на проверку `условия-выражения` для определения необходимости продолжения итерационных вычислений.

Формат оператора:

```
continue;
```

Оператор безусловного перехода goto

Использование оператора безусловного перехода `goto` в практике программирования С++ ***настоятельно не рекомендуется***, так как он затрудняет понимание программы и возможность ее модификации.

Оператор имеет следующий формат:

```
          goto имя-метки;
          ...
имя-метки:     оператор;
```

Оператор `goto` передает управление на оператор, помеченный меткой `имя-метки`. Помеченный оператор должен находиться в той же функции, что и оператор `goto`, а используемая метка перед оператором, на который передается управление, должна быть уникальной, т.е. одно и тоже `имя-метки` не может указываться перед разными операторами.

Организация вычислений с точностью

Суммирование отрезка степенного ряда с точностью

При реализации многих численных методов точность вычислений зависит от числа шагов. Однако за какое именно число шагов будет достигнута приемлемая точность, заранее сказать трудно и желательно, чтобы программа сама определяла, когда следует остановиться. Например, это касается алгоритма вычисления значения любой из математических функций (например, sin, cos, …), использующего разложение в ряд Тейлора.

Рассмотрим один из простейших вариантов ряда Тейлора:

$$\frac{1}{1-x} = \sum_{0}^{\infty} x^i = 1 + x^2 + x^3 + x^4 + \cdots$$

Чем большее количество членов ряда будет просуммировано, тем точнее будет вычислено значение функции, представленной этим рядом. Пусть требуется вычислить до 2-го знака после запятой, т.е. приемлемая погрешность $\varepsilon = 5 \cdot 10^{-3}$. Для этого достаточно суммировать члены ряда до тех пор, пока очередной член ряда не окажется меньше ε, т.е. вычисления суммы проводить пока $x^i > \varepsilon$.

Достаточным условием сходимости практически всех рядов Тейлора является малость модуля аргумента x относительно 1, т.е. должно выполняться условие $|x| < 1$.

В отличии от ранее рассмотренных примеров в цикле выполняется не одно действие (накопление суммы или возведение в степень), а сразу два: число x возводится в степень и полученный таким образом член ряда суммируется. Этот прием существенно уменьшает как количество операций, так и время вычисления значения функции.

Пример.

main.cpp

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4
```

```
 5 int main() {
 6     double  x,       //значение x
 7             eps;     //значение точности
 8 
 9     while(true) {
10         cout<<"Введите eps: ";
11         cin>> eps;
12         cout<<"Введите x: ";
13         cin>> x;
14         //проверка двух переменных eps и x
15         if (0 < eps && eps < 1
16             && -1 < x && x < 1) break;
17     }
18 
19     double sum =0,        //сумма отрезка ряда
20            xPower = 1;    //степень члена ряда x^i
21     while( fabs(xPower) > eps) {
22         sum = sum + xPower;
23         xPower = xPower * x;
24     }
25     cout<< "sum = " << sum << endl;
26     return 0;
27 }
```

Результат работы программы:

```
Введите eps: 0.005
Введите x: 0.7
sum = 3.31751
```

Вычисление значения ряда с точностью описывается таблицей (Таблица 29).

Таблица 29 – Вычисление отрезка ряда с точностью

Переменная/ действие	До входа в цикл	Значения в цикле		
XPower > eps	---	true	true	true
Sum	0	0 + 1 = 1	1 + x	1+x + x^2
XPower	1	1 · x = x	x · x = x^2	x^2 · x = x^3

Точность вычислений согласно рекуррентным уравнениям

Рекуррентными соотношениями или уравнениями называются уравнения вида:

$$x_n = F(x_{n-1}, \dots, x_0)$$

где $F(\quad)$ - некоторая функция, x_i – значение, вычисленное на i-ом шаге ($i = \overline{1, n}$). Таким образом, в рекуррентной записи каждое следующее значение вычисляется по предыдущим значениям. Поэтому одним из основных практических вопросов является вопрос выбора как начального x_0, так и других необходимых значений для запуска алгоритма.

Если вычисления производятся в соответствии с рекуррентными соотношениями, то используется другой способ поставить связанное с точностью условие прекращения вычислений. Оно заключается в том, что вычисления прекращаются, если изменение вычисляемой величины на очередном шаге $n+1$ меньше заданной величины ε:

$$|x_{n+1} - x_n| < \varepsilon.$$

Также условие можно наложить на относительное изменение:

$$\left|\frac{x_{n+1} - x_n}{x_n}\right| < \varepsilon.$$

При этом необходимо особо подчеркнуть, что за вычисленное значение принимается x_n (но не x_{n+1}).

В качестве примера следует рассмотреть рекуррентную формулу Ньютона вычисления корня k-ой степени из числа a (т.е. $\sqrt[k]{a}$):

$$x_{n+1} = \frac{k-1}{k} x_n - \frac{a}{k \cdot {x_n}^{k-1}},$$

где $x_0 = a$.

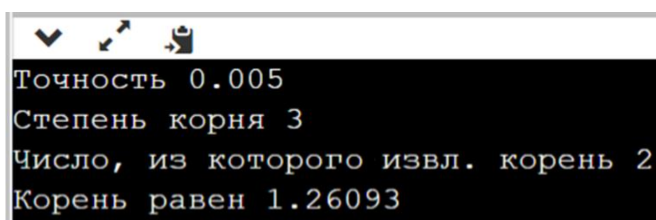
Пример.

main.cpp

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
```

```
 4
 5  int main() {
 6      double  a,      //значение из которого
 7                      //извлекается корень
 8              eps;    //значение точности
 9              int k;  //целочисленное значение степени
10                      //корня
11
12      while(true) {
13          cout<<"Точность ";
14          cin>> eps;
15          cout<<"Степень корня ";
16          cin>> k;
17          cout<<"Число, из которого извл. корень ";
18          cin>> a;
19          //проверка достаточных условий решения задачи
20          if (0< eps && eps< 1 && 0< k && 0< a) break;
21          cout<<"Данные введены не верно!" << endl
22              <<"Еще раз!"<< endl << endl;
23      }
24
25      //значение предыдущей итерации
26      double xn = a;
27      //значение следующей итерации
28      double xnn = (k - 1) / (double) k * xn +
29                   a / (k * pow(xn, k - 1));
30      //реализация итерационного алгоритма
31      while(fabs(xnn - xn) > eps) {
32          xn = xnn; // переход к следующей итерации
33          xnn = (k - 1) / (double) k * xn +
34                   a / (k * pow(xn, k - 1));
35      }
36      cout<<"Корень равен " << xn;
37      return 0;
38  }
```

Результат работы программы:


```
Точность 0.005
Степень корня 3
Число, из которого извл. корень 2
Корень равен 1.26093
```

Автоматические массивы

Массив - именованная структура данных одного типа с прямым доступом (т.е. по номерам индексов).

Массивы широко используются при разработке различного рода приложений. Они полезны как при сохранении таблиц данных, так и для выполнения многих вычислительных задач. С понятием «массив» приходится сталкиваться при решении научно-технических и экономических задач, связанных с обработкой большого объема однотипных данных.

Массив позволяет сохранять и манипулировать большими объемами данных посредством следующих идентификаторов:

- имени массива,
- индекса (для многомерных массивов – нескольких индексов).

Имя массива – произвольно выбираемый разработчиком идентификатор.

Индекс или индексы элемента массива - одно или несколько целых значений, указанных в виде константы, переменной или выражения, заключённых в квадратные скобки (например, для одномерного: `d[i]`, для двумерного: `a[3][5]`).

Объединяя две концепции «массив» и «цикл» можно, используя небольшое число операторов, обрабатывать большой объем данных.

Память под массив может:

- выделяться автоматически – подобное выделение памяти используют, когда размер массива известен на этапе компиляции (например, задан в виде константы);
- выделяться в куче – этот вариант используется, когда размер массива неизвестен на этапе компиляции (допустим, запрашивается у пользователя в процессе выполнения программы).

<u>Замечания:</u>

К настоящему времени сложилась традиция именовать «динамическими» только те массивы (или структуры данных), для которых к моменту запуска программы не только не определено количество содержащихся в нем элементов, но и возможно физическое изменение их числа (удаление/добавление) во время работы программы. В C++ строго под это определение подпадают исключительно коллекции (будут рассмотрены далее).

Одномерные автоматические массивы

Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет формат:

```
спецификаторТипа имяМассива[количЭлементов];
```

где:

- **имяМассива** – это идентификатор массива;
- спецификаторТипа задает тип элементов объявляемого массива (элементами массива не могут быть элементы типа void).
- Выражение **количЭлементов** – именованная или неименованная ***константа***, определяющая количество элементов в одномерном массиве, может отсутствовать, если массив при объявлении инициализируется.

Индексация элементов массива начинается с ***нуля*** и заканчивается ***n – 1***, где n – число элементов массива (значение параметра **количЭлементов**).

Допускается инициализировать массив значениями при его объявлении. Формат:

```
спецификаторТипа имяМассива[количЭлементов] =
                                        {список, значений};
```

где типы элементов списка значений должны совпадать с параметром спецификаторТипа, параметр ~~**количЭлементов**~~ может отсутствовать.

Пример.

```
//Объявление
int arr[5];

//Инициализация
double d[] = {1., 2., 3., 4., 5.};

int a[5] = {11};//инициализация только нулевого
                //элемента массива числом 11, а
                //остальные элементы будут нулями
```

При инициализации массива и отсутствии константы ~~**количЭлементов**~~ длина ленты памяти, выделяемой под его хранение, вычисляется компилятором по количеству значений, перечисленных в фигурных скобках (элементов инициализирующего списка) и их типу.

Если ~~**количЭлементов**~~ массива задано, нельзя задать больше инициализирующих значений, чем объявлено в массиве (синтаксическая ошибка), но меньше указать можно (минимально должен быть определен один элемент).

Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем значение ~~**количЭлементов**~~, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) инициализируются нулем.

Очевидно, что для того, чтобы инициализировать массив нулями достаточно использовать конструкцию:

```
int a[5] = {0};
```

В этом случае нулевому элементу массива присваивается нулевое значение при инициализации, а далее до конца массива (элементам `a[1]`, `a[2]`, `a[3]`, `a[4]`) нули будут присвоены «по умолчанию».

После инициализации массива, например:

```
int arr[] = {12, 23, 1, 345, 63};
```

можно выполнять алгоритмические действия обращаясь к значению элемента массива по имени массива и его индексу (`arr[i]`, где `i` – значение от `1` до `4`, Рисунок 9).

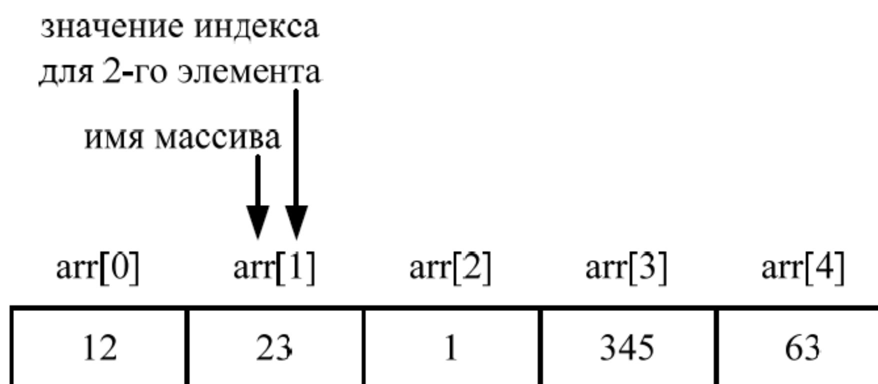


Рисунок 9 – Лента, выделяемая для хранения массива

<u>Замечание.</u>

*При объявлении автоматических массивов для задания их размеров следует использовать **именованные** константы.*

Примеры простейших действий над одномерными автоматическими массивами

Определение длины инициализированного массива

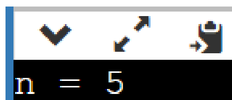
Определение длины инициализированного массива выполняется с помощью операции `sizeof()`.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    float array[] = {1., 21., 13., 64., 5.};
    int n = sizeof(array) / sizeof(float);

    cout<<"n = "<< n;
    return 0;
}
```

Результат работы программы:



Вывод на экран в одну строку одномерного массива

Рассмотрим программу (Таблица 30) вывода на экран одномерного целочисленного массива (массив выводится в виде строки).

Таблица 30 – Сравнение кода на разных языках программирования

<table>
<tr><th>Код на С++</th><th>Код на Pascal-е</th></tr>
<tr><td><pre>#include <iostream>
using namespace std;

int main() {
    int array[] = {1, 21, 13, 64, 5};
    int n = sizeof(array) / sizeof(float);
    cout<<"Инициализированный массив"
        << endl;
    for(int i = 0; i < n; i++) {
        cout<<' '<< array[i];
    }
    return 0;
}</pre></td><td><pre>program main;
const n = 5;
var a : array[1..n] of integer =
                    (1, 21, 13, 64, 5);
i : integer;
begin
    writeln('Инициализированный массив');
    for i:=1 to n do
    begin
        write(' ', a[i]);
    end;
end.</pre></td></tr>
</table>

Результат выполнения программы на С++:

```
Инициализированный массив
 1 21 13 64 5
```

Ручной счет работы алгоритма на языке С++ приведен в таблице (Таблица 31).

Таблица 31 – Вывод на экран одномерного массива в строку

Переменная/ операция	До входа в цикл	Значения параметров в цикле			
n	5	5	5	5	...
i	---	0	1	2	...
i < n	---	true	true	true	...
cout	---	_array[0]	_array[1]	_array[2]	...
Результат на экране	---	_1	_1_21	_1_21_13	...

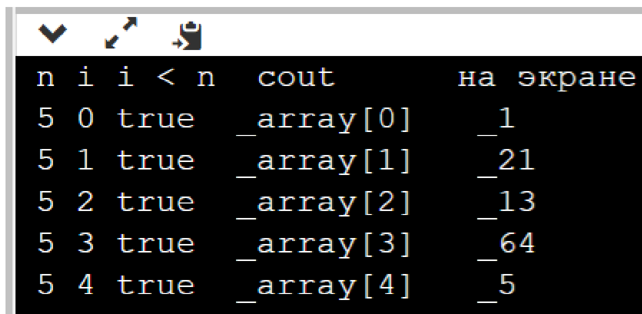
Как и ранее таблицу ручного счета можно проконтролировать, несколько изменив предыдущий код. В частности, организовав ***вывод массива в столбец*** (а не в строку, как в предыдущем примере). Кроме того, для наглядности пробел перед `array[i]`, заменен на символ подчеркивания.

Пример.

```
main.cpp
1  #include <iostream>
```

```
using namespace std;

int main() {
    int array[] = {1, 21, 13, 64, 5};
    int n = sizeof(array) / sizeof(float);
    //шапка таблицы ручного счета
    cout <<" n"<< " "<< "i" << " " << "i < n"
         <<"  cout  " <<"    на экране"
         << endl;
    for(int i = 0; i < n; i++) {
        cout<<boolalpha
            <<' ' << n <<' '<< i <<' '<< (i < n)
            <<"  _array["<< i <<"]   "
            <<'_' << array[i]
            << endl;
    }
    return 0;
}
```

Результат работы программы:



<u>*Замечание.*</u>

Сравнив столбцы результата работы программы со строками таблицы (Таблица 31) можно легко установить ее корректность.

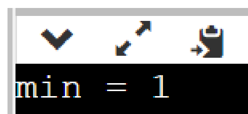
Выбор минимального значения из элементов массива

Организация цикла `for()` при выборе минимума *с помощью* оператора `if()` в одномерном массиве.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float array[] = {35., 1., 15., 43., 7.};
6      int n = sizeof(array) / sizeof(float);
7
8      float min = array[0];
9      for(int i = 1; i < n; i++) {
10         if (min > array[i]) min = array[i];
11         //ветка else отсутствует,
12         //или else; - имеет пустой оператор
13     }
14     cout<< "min = "<< min;
15     return 0;
16 }
```

Результат работы программы:



Ручной счет работы алгоритма приведен в таблице (Таблица 32).

Таблица 32 – Выбор минимума в одномерном массиве

Переменная / операция	До входа в цикл	Значения параметров в цикле			
n	5	5	5	5	5
i	---	1	2	3	4
i < n	---	true	true	true	true
min > array[i]	---	true	false	false	false
min	---	array[1]	array[1]	array[1]	array[1]

На основе того же алгоритма выбора минимума простым перебором можно написать программу выполнения этой операции с помощью ***тернарной операции***.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    float array[] = {35., 1., 15., 43., 7.};
    int n = sizeof(array) / sizeof(float);

    float min = array[0];
    for(int i = 1; i < n; i++) {
        min = min > array[i] ? array[i] : min;
    }
    cout<< "min = "<< min;
    return 0;
}
```

В связи с тем, что исходный массив в данной и в предыдущей программе одинаковы, то и ручной счет алгоритма, и результат работы программы совпадают.

Определение минимального значения из элементов одномерного массива, имеющих четные/нечетные индексы

Важным в программировании на любых С-подобных языках является понимание организации обращения к элементам одномерного массива с четными/нечетными индексами. Это делается элементарно за счет изменения индекса не на единицу, а на двойку (`i = i + 2`).

Фактически результат обращения к элементам с четным или нечетным индексом зависит только от того с какого индекса программист начинает перебор. Если с нуля (или любого четного числа), то это будет перебор элементов с четными индексами, а если «хождение» по массиву начинается с элемента с индексом равным единице (или любого нечетного числа), то, очевидно, поиск будет происходить по нечетным индексам.

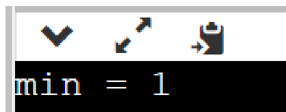
Приведем пример выбора минимума из элементов массива, имеющих нечетные индексы.

Пример.

main.cpp

```
#include <iostream>

int main() {
    float array[] = {35., 1., 15., 43., 7.};
    int n = sizeof(array) / sizeof(float);

    float min = array[1];
    for(int i = 3; i < n; i = i + 2) {
        if (min > array[i]) min = array[i];
    }
    std::cout<< "min = "<< min;
    return 0;
}
```

Результат работы программы:



Ручной счет работы алгоритма приведен в таблице (Таблица 33).

Таблица 33 – Выбор минимума по значению элементов с нечетными индексами в одномерном массиве

Переменная/ операция	До входа в цикл	Значения параметров в цикле	
n	5	5	5
i	---	3	5
i < n	---	true	true
min > array[i]	---	false	false
min	array[1]	array[1]	array[1]

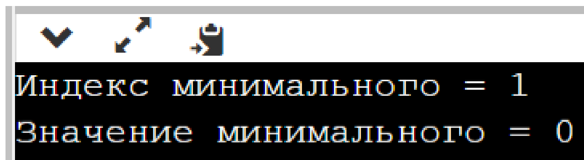
Определение индекса минимального значения

Как показал предыдущий пример выбор минимума в одномерном массиве можно организовать с помощью выбора непосредственно значения, но в некоторых случаях необходимо ***определить индекс*** минимального значения.

Пример.

```cpp
main.cpp
1  #include <iostream>
2
3  int main() {
4      float array[] = {5., 0., 1., 0., 11.};
5      int n = sizeof(array) / sizeof(float);
6
7      int minInd = 0;
8      for(int i = 1; i < n; i++) {
9          if (array[minInd] > array[i]) minInd = i;
10     }
11     std::cout<< "Индекс минимального = "
12             << minInd << std::endl
13             << "Значение минимального = "
14             << array[minInd];
15     return 0;
16 }
```

Результат работы программы:



Ручной счет работы алгоритма приведен в таблице (Таблица 34).

Таблица 34 – Выбор индекса минимального элемента в одномерном массиве

Переменная/ операция	До входа в цикл	Значения параметров в цикле			
n	5	5	5	5	5
i	---	1	2	3	4
i < n	---	true	true	true	true
min > array[i]	---	true	false	false	false
minInd	0	1	1	1	1

Отметим, что алгоритм в случае наличия в массиве нескольких минимальных выдаст индекс первого по порядку минимального значения. Однако если внести незначительные изменения в условие оператора `if()`, а именно написать условие `(array[minInd] >= array[i])`, то программа всегда будет выдавать индекс последнего минимального.

Проверка существования в массиве хотя бы одного элемента, удовлетворяющего определенному условию

Рассмотрим несколько вариантов алгоритма, определяющих существование в одномерном массиве хотя бы одного элемента с заданным значением (равным нулю, больше/меньше нуля, четного/нечетного и т.д.)

Для определенности (не уменьшая общности) будем рассматривать алгоритм существования в массиве хотя бы одного положительного элемента.

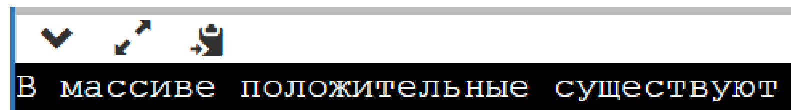
Первый вариант алгоритма является наиболее простым. Этот вариант заключается в подсчете количества положительных элементов (целочисленная переменная `count`) в операторе `if()`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int array[] = {-5, -1, 11, 0, -3};
6      int n = sizeof(array) / sizeof(float);
7
8      int count = 0;
9      for(int i = 0; i < n; i++) {
10         if(array[i] > 0) count++;
11     }
12
13     if (count == 0) cout<<"Положительных нет";
14     else cout<< "В массиве положительные существуют";
15     return 0;
16 }
```

Результат работы программы:



Очевидно, предыдущий пример должен восприниматься слушателями наиболее естественно, т.к. алгоритм рассмотренного примера универсален и переносим безо всяких изменений в любой язык программирования с точностью до замены синтаксиса соответствующих операторов.

В следующем примере продемонстрируем, что предыдущий вариант кода можно сократить (убрать `if()`), используя особенности языка C++, в частности, возможность автоматического преобразования типов (например, `bool` в `int`).

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int array[] = {-5, -1, 11, 0, -3};
    int n = sizeof(array) / sizeof(float);

    int count = 0;
    for(int i = 0; i < n; i++) {
        count = count + (array[i] > 0);
    }

    if (count == 0) cout<<"Положительных нет";
    else cout<< "В массиве положительные существуют";
    return 0;
}
```

Результат работы программы не изменился.

Предыдущий пример сохранил переменную `count` для подсчета числа положительных. Однако можно полностью отказаться от этой переменной, но взамен сохранить `if()`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int array[] = {-5, -1, 11, 0, -3};
6      int n = sizeof(array) / sizeof(float);
7
8      int i = 0;
9      for( ; i < n; i++) {
10         if(array[i] > 0) break;
11     }
12
13     if (i == n) cout<<"Положительных нет";
14     else cout<< "В массиве положительные существуют";
15     return 0;
16 }
```

Результат работы программы не изменился.

Последний алгоритм (как и первый) является по сути дела универсальным, переносимым в любой язык программирования с точностью до замены синтаксических конструкций. Из-за свой универсальности он ***не*** может в достаточной степени учитывать синтаксические возможности языка С++. В данной программе использован тот факт, что если после проверки все элементы окажутся отрицательными (т.е. `i` равно `n`), то, очевидно, условие существования хотя бы одного положительного будет нарушено и выхода из цикла по `break` не случиться.

В следующем примере выполним все те же действия, однако удалим из кода не только оператор `if()`, но и `break`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int array[] = {-5, -1, 11, 0, -3};
6      int n = sizeof(array) / sizeof(float);
```

```
 7
 8      int i = 0;
 9      for(; i < n && array[i] <= 0; i++);
10
11      if (i == n) cout << "Положительных нет";
12      else cout<< "В массиве положительные существуют";
13      return 0;
14  }
```

Результат работы программы не изменился.

В предыдущих примерах неоднократно использовалась гипотеза о том, что определенное значение присутствует в массиве. Однако это может и не выполняться и в этом случае программа отработает некорректно.

Для того, чтобы гарантировать себя от подобной ситуации в подавляющем числе случаев недостаточно абстрактного ответа, что элемент с необходимым значением где-то в массиве существует. Обычно необходимо знать индекс первого или последнего элемента с заданным свойством.

Если более внимательно посмотреть на текст последней программы, то несложно заметить, что когда положительные элементы в массиве существуют, то индекс `i` после работы цикла `for()` будет иметь значение индекса первого по порядку найденного положительного числа.

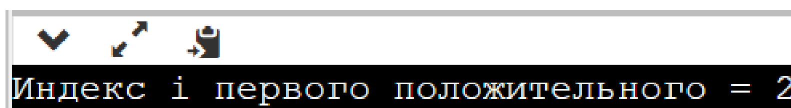
Заменим в предыдущем примере оператор `if()`:

```
if (i == n) cout << "Положительных нет";
else cout<< "В массиве положительные существуют";
```

На вывод текстового сообщения:

```
cout<< "Индекс i первого положительного = "<< i;
```

Этой замены будет достаточно, чтобы увидеть интересующий результат:



Замечание.

В случае отсутствия положительного значения в массиве на экран в качестве значения `i` *выведется 5 – невозможное значения для индекса, т.к. он в примере может принимать значения не больше 4.*

Определение минимального значения только из положительных элементов одномерного массива

Объединим две программы:

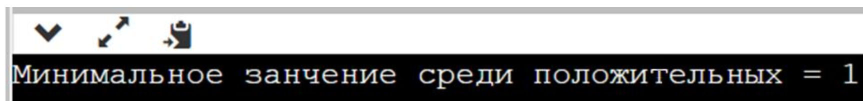
- одна – это определение индекса первого положительного элемента в массиве (предыдущий пункт);
- второе – выбор минимального значения (его придется немного откорректировать: добавить в `if()` дополнительное условие проверки положительности элемента массива).

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    float array[] = {-5., 0., 1., 0., 11.};
    int n = sizeof(array) / sizeof(float);

    //определение индекса первого положительного
    int i = 0;
    for(; i < n && array[i] <= 0; i++);

    //условие аварийного завершения программы:
    //отсутствие положительных в массиве
    if (i == n) {
        cout << "false";
        return 0;
    }

    //выбор минимума по положительным, начиная с
    //первого положительного
    float min = array[i];
    //в цикле j начинается со следующего, т.е. i + 1
    for(int j = i + 1; j < n; j++) {
        if (min > array[j] && array[j] > 0) {
            min = array[j];
        }
    }
    cout<< "Минимальное занчение среди положительных = "
        << min;
    return 0;
}
```

Результат работы программы



Вычисление средних значений массива

Пусть необходимо определить ***среднее арифметическое*** значение целочисленного массива $\{a_i\}_{i=0}^{n-1}$. В статистике эта операция называется вычислением математического ожидания.

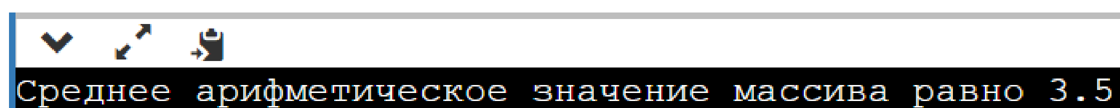
Среднее значение `average` определяется формулой:

$$average = \frac{\sum_{i=0}^{n-1} a_i}{n}.$$

Пример.

main.cpp
```cpp
#include <iostream>

int main() {
    int a[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(a) / sizeof(int);

    double sum = 0;
    for(int i = 0; i < n; i++) {
        sum += a[i];
    }
    std::cout<< "Среднее арифметическое "
             << "значение массива равно "
             << sum / n;

    return 0;
}
```

Результат работы программы:



Замечание.

Хотя массив целочисленный, но переменная `sum` *имеет тип* `double`*, т.к. среднее значение в подавляющем числе случае должно иметь дробную часть. В этом случае результат выражения* `sum / n` *также будет иметь тип* `double`*, что позволит избежать отсечение дробной части.*

Рассмотрим вычисление среднего геометрического значения того же целочисленного массива. Оно определяется формулой:

$$geometricAverage = \sqrt[n]{\prod_{i=0}^{n-1} a_i}\,.$$

Пример.

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    int a[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(a) / sizeof(int);

    double product = 1;
    for(int i = 0; i < n; i++) {
        product *= a[i];
    }
    std::cout<< "Среднее геометрическое "
             << "значение массива равно "
             << pow(product, 1./n);

    return 0;
}
```

Результат работы программы:

```
Среднее геометрическое значение массива равно 2.9938
```

Реверс одномерного массива

Реверсом называется перестановка первого элемента массива с последним, второго элемента с предпоследним и т.д.

Напишем программу, которая:

- вначале выводит исходный массив в одну строку на экран;
- выполняет реверс массива;
- в заключение выводит на экран результат реверса.

Очевидно, что в данном случае придется комбинировать коды как уже рассмотренных примеров программ (например, вывод на экран одномерного массива в строку), так и создавать новые (реверс).

Пример.

main.cpp

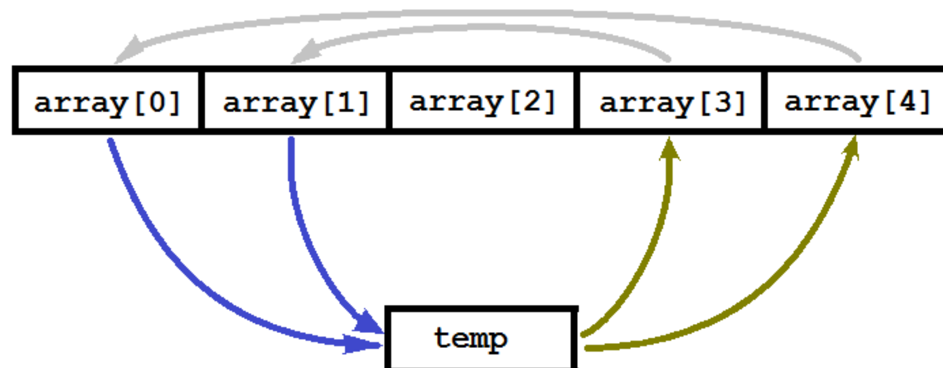
```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float array[] = {1., 21., 13., 64., 5.};
6      int N = sizeof(array) / sizeof(float);
7
8      // блок вывода в строку на экран исх. массива
9      cout<<"Исходный массив: "<< endl;
10     for(int i = 0; i < N; i++) {
11         cout<<' '<< array[i];
12     }
13
14     // реверс
15     float temp;
16     for(int i = 0; i < N/2; i++) {
17         temp = array[i];
18         array[i] = array[N - 1 - i];
19         array[N - 1 - i] = temp;
20     }
21
22     // вывод на экран преобразов. массива
23     cout<< endl << "Массив после реверса" << endl;
```

```
24     for(int i = 0; i < N; i++) {
25         cout<<' '<< array[i];
26     }
27     return 0;
28 }
```

Результат работы программы:

```
Исходный массив:
 1 21 13 64 5
Массив после реверса
 5 64 13 21 1
```

Поскольку таблица ручного счета реверса будет иметь исключительно громоздкий и нечитаемый вид, то в данном случае представляется более информативным изобразить графически схему алгоритма (Рисунок 10).



N/2 = 5/2 = 2 (перобразование результата к целому числу)

Рисунок 10 - Схема алгоритма реверса (цвета линий соответствуют цветам фона в присваиваниях при перестановке)

Забой элемента массива сдвигом влево

В алгоритмизации часто встречается задача о «забое» в массиве (удалении сдвигом) одного или нескольких элементов определенного значения (Рисунок 11), при этом кроме удаления необходимо сократить активное количество элементов массива, т.к. при забое сдвигом влево образуется два одинаковых элемента в конце массива. Поскольку при работе с автоматическими массивами нет средств для того, чтобы укоротить длину ленты, выделенной под массив, то последний элемент необходимо просто

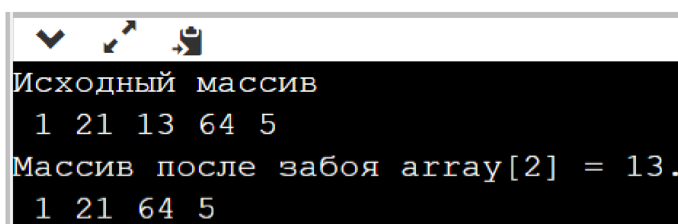
игнорировать. Это достигается простым уменьшением значения переменной, хранящей количество элементов на единицу (`n = n - 1;`).

Пример.

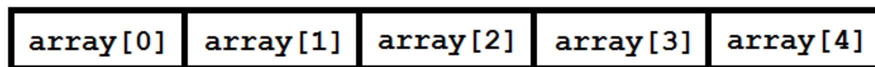
main.cpp

```
#include <iostream>
using namespace std;

int main() {
    float array[] = {1., 21., 13., 64., 5.};
    int n = sizeof(array) / sizeof(float);

    // блок вывода в строку на экран исх. массива
    cout<<"Исходный массив"<< endl;
    for(int i = 0; i < n; i++) {
        cout<<' '<< array[i];
    }

    //забой элемента array[2]
    for(int i = 2; i < n - 1; i++) {
        array[i] = array[i + 1];
    }

    n = n - 1; //игнорирование последнего повторяющ.
               //элемента массива array

    // вывод на экран преобразов. массива
    cout<< endl<< "Массив после забоя array[2] = 13."
        << endl;
    for(int i = 0; i < n; i++) {
        cout<<' '<< array[i];
    }
    return 0;
}
```

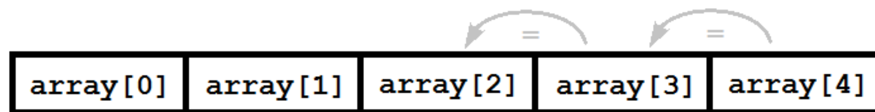
Результат работы программы:



Исходное состояние массива из 5-ти элементов



Забой элемента array[2] сдвигом влево



Результат выполнения алгоритма

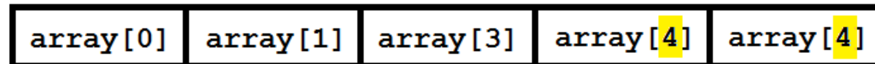


Рисунок 11 – Схема алгоритма забоя одного элемента сдвигом влево

Циклический сдвиг влево/вправо на один элемент массива

Рассмотрим циклический сдвиг влево. Данный алгоритм идеологически очень близок к предыдущей задаче. Сдвиг влево всего массива осуществляется также как в предыдущем случае (Рисунок 12). Однако в данном алгоритме значение с нулевым индексом не должно быть уничтожено, а перенесено в конец существующего массива (Рисунок 12).

Пример.

main.cpp

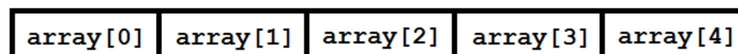
```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float array[] = {1., 21., 13., 64., 5.};
6      int n = sizeof(array) / sizeof(float);
7
8      // блок вывода в строку на экран исх. массива
9      cout<<"Исходный массив"<< endl;
10     for(int i = 0; i < n; i++) {
11         cout<<' '<< array[i];
12     }
13
14     //сохранение значения array[0]
15     float temp = array[0];
16     //забой элемента array[0]
```

```
17      for(int i = 0; i < n - 1; i++) {
18          array[i] = array[i + 1];
19      }
20      //перестановка значения array[0] в конец массива
21      array[n - 1] = temp;
22
23      // вывод на экран преобразов. массива
24      cout<< endl<<"Массив после циклического сдвига влево"
25          << endl;
26      for(int i = 0; i < n; i++) {
27          cout<<' '<< array[i];
28      }
29      return 0;
30  }
```

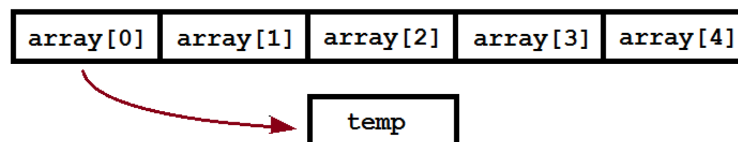
Результат работы программы:

```
Исходный массив
 1 21 13 64 5
Массив после циклического сдвига влево
 21 13 64 5 1
```

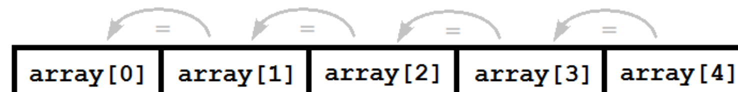
Исходное состояние массива из 5-ти элементов



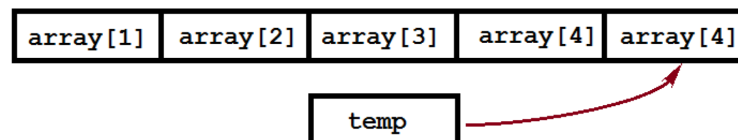
Подготовительное действие при циклическом сдвиге влево
(копирование элемента массива с нулевым индексом)



Забой элемента array[0] сдвигом влево



Присвоение сохраненного в temp значения array[0]
последнему элементу массива



Результат выполнения алгоритма

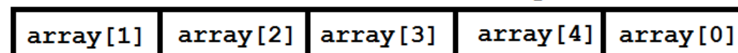


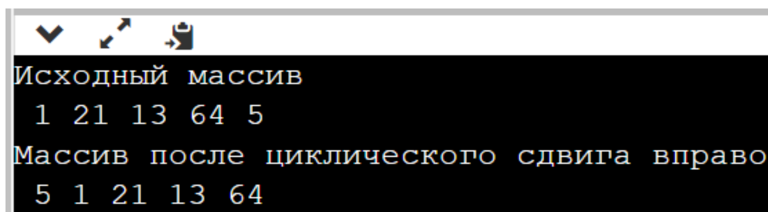
Рисунок 12 – Этапы выполнения циклического сдвига влево

Заменим выделенный фрагмент кода в предыдущей программе на следующий:

```
14     //сохранение значения array[n - 1]
15     float temp = array[n - 1];
16     //забой элемента array[n - 1]
17     for(int i = n - 1; i > 0; i--) {
18         array[i] = array[i - 1];
19     }
20     //перестановка значения array[n - 1] в начало массива
21     array[0] = temp;
```

Таким образом получаем программу, осуществляющую циклический сдвиг массива ***вправо***.

После корректировки последнего заголовка, выводимого с помощью `cout` результат работы программы, приобретает вид:



Сортировки одномерных массивов

В этом разделе рассмотрены два самых популярных алгоритма сортировки массивов, применяемых в учебных целях. Следует подчеркнуть, что все рассмотренные алгоритмы медленнее, чем известный рекурсивный алгоритм быстрой сортировки, однако в данном курсе последний рассматриваться не будет.

Сортировка выбором

Сортировка выбором является одним из самых простых алгоритмов сортировки массива. На небольших массивах данный алгоритм может оказаться даже эффективнее, чем более сложные алгоритмы сортировки, но в любом случае проигрывает на больших массивах.

Будем предполагать, что сортируемый массив уже инициализирован. Рассмотрим сортировку ***по возрастанию***.

Идея сортировки выбором по возрастанию (Рисунок 13): смысл в том, чтобы идти по массиву, и каждый раз в оставшейся неотсортированной его части искать минимальный элемент, обменивая его с начальным элементом

неотсортированной части. Следует подчеркнуть, что в примере используется выбор минимального по индексу, а не по значению.

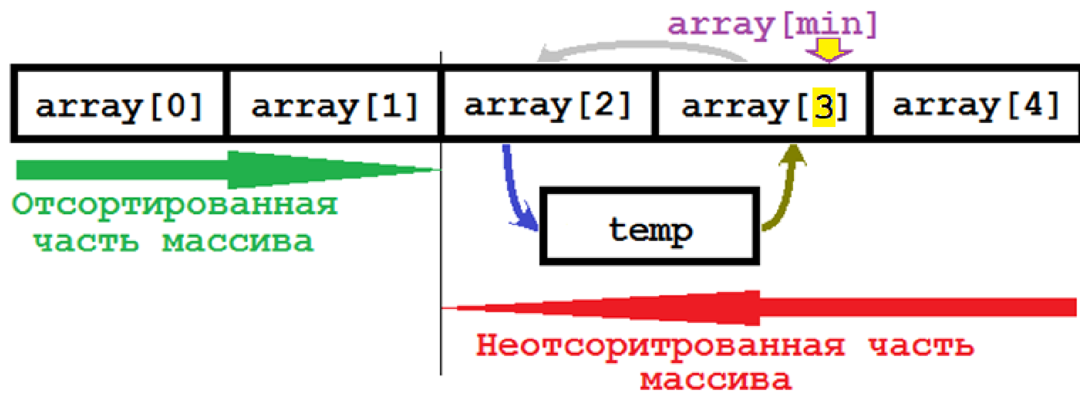


Рисунок 13 – Схема сортировки выбором

Словесное описание алгоритма:

1. находим индекс минимального элемента в массиве;
2. меняем местами минимальный и нулевой (по индексу) элемент местами;
3. снова ищем индекс минимального элемента в неотсортированной части массива;
4. меняем местами вновь найденный по индексу минимальный из неотсортированной части и первый элемент массива (напомним, что после предыдущего шага нулевой (по индексу) элемент массива является уже элементом отсортированной части);
5. ищем далее индексы минимальных значений и последовательно меняем соответствующие элементы массива местами с начальными элементами постоянно уменьшающейся неотсортированной части массива.

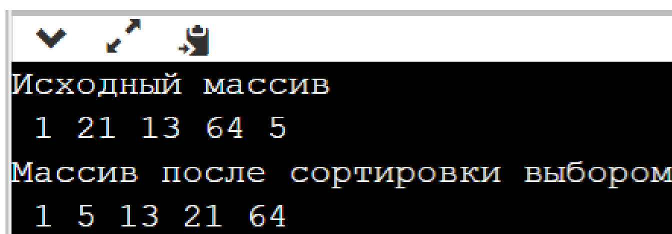
<u>Пример.</u>

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float array[] = {1., 21., 13., 64., 5.};
6      int n = sizeof(array) / sizeof(float);
7
8      // блок вывода в строку на экран исх. массива
9      cout<<"Исходный массив"<< endl;
```

```
10      for(int i = 0; i < n; i++) {
11          cout<<' '<< array[i];
12      }
13  
14      //сортировка выбором
15      for(int i = 0; i < n - 1; i++) {
16          //выбор индекса минимального
17          int min = i;
18          for(int j = i + 1; j < n; j++) {
19              if (array [j] < array [min]) {
20                  min = j;
21              }
22          }
23          //производим перестановку элементов массива
24          float temp = array[i];
25              array[i] = array[min];
26              array[min] = temp;
27      }
28  
29      //вывод на экран преобразов. массива
30      cout<< endl<< "Массив после сортировки выбором"
31          << endl;
32      for(int i = 0; i < n; i++) {
33          cout<<' '<< array[i];
34      }
35      return 0;
36  }
```

Результат работы программы:


```
Исходный массив
 1 21 13 64 5
Массив после сортировки выбором
 1 5 13 21 64
```

Сортировка пузырьком

Этот алгоритм также является одним из самых простых алгоритмов сортировки одномерного массива и, пожалуй, самым известным алгоритмом, применяемый в учебных целях. Для практического применения является слишком медленным. Однако данный алгоритм лежит в основе более

сложных алгоритмов: шейкерная сортировка (сортировка перемешиванием), пирамидальная сортировка, быстрая сортировка.

Примечательно то, что один из самых быстрых алгоритмов алгоритм быстрой сортировки был разработан путем модернизации одного из самых худших алгоритмов ***сортировки пузырьком***.

Пусть необходимо отсортировать одномерный массив с помощью алгоритма ***сортировка пузырьком***. Для изложения идеи и алгоритма сортировки будем предполагать, что сортируемый массив уже инициализирован.

Идея сортировки пузырьком по возрастанию: смысл алгоритма заключается в том, что самые «легкие» элементы массива как бы «всплывают», а самые «тяжелые» «тонут». Отсюда и название «сортировка пузырьком».

Словесное описание алгоритма:

1. каждый элемент массива сравнивается с последующим и, если `элемент[i] > элемент[i + 1]` происходит взаимная перестановка. Таким образом, элементы имеющие наименьшие значения перемещаются к началу списка, а имеющие наибольшие - к концу.
2. Повторяем пункт `1` данного алгоритма `n - 1` раз, где `n` - количество элементов в массиве.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      float array[] = {1., 21., 13., 64., 5.};
6      int n = sizeof(array) / sizeof(float);
7
8      // блок вывода в строку на экран исх. массива
9      cout<<"Исходный массив"<< endl;
10     for(int i = 0; i < n; i++) {
11         cout<<' '<< array[i];
12     }
13
14     //сортировка пузырьком
15     for(int i = 0; i < n; i++) {
16         for(int j = 0; j < n - i - 1; j++) {
17             if (array[j] > array[j + 1]) {
18                 //производим перестановку элементов
```

```
19              float temp = array[j];
20                  array[j] = array[j + 1];
21                  array[j + 1] = temp;
22              }
23          }
24      }
25
26      //вывод на экран преобразов. массива
27      cout<< endl<< "Массив после сортировки пузырьком"
28          << endl;
29      for(int i = 0; i < n; i++) {
30          cout<<' '<< array[i];
31      }
32      return 0;
33  }
```

Результат работы программы:

```
Исходный массив
 1 21 13 64 5
Массив после сортировки пузырьком
 1 5 13 21 64
```

Операции линейной алгебры для векторов в контексте их применения к одномерным массивам

Скалярное произведение двух векторов

Скалярное произведение двух n-мерных векторов в контексте одномерных массивов $a = \{a_i\}_{i=0}^{n-1}$ и $b = \{b_i\}_{i=0}^{n-1}$ определяется формулой:

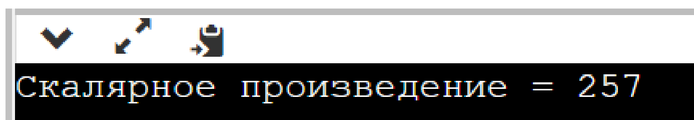
$$a \cdot b = a_0 \cdot b_0 + a_1 \cdot b_1 + \cdots + a_{n-1} \cdot b_{n-1} = \sum_{i=0}^{n-1} a_i \cdot b_i \,.$$

Пример.

```
main.cpp
1  #include <iostream>
2
```

```
 3 int main() {
 4     const int N = 5;
 5     float a[N] = {1., 21., 13., 64., 5.};
 6     float b[N] = {5., 2., 1., 3., 1.};
 7 
 8     //скалярное произведение вычисл. в переменной sum
 9     float sum = 0;
10     for(int i = 0; i < N; i++) {
11         sum = sum + a[i] * b[i];
12     }
13 
14     //вывод на экран скалярного произведения
15     std::cout<< "Скалярное произведение = "
16               << sum;
17     return 0;
18 }
```

Результат работы программы:



```
Скалярное произведение = 257
```

Вычисление длины радиус-вектора n-мерной точки

Длина радиус-вектора точки x в n-мерном пространстве в контексте одномерного массива $x = \{x_i\}_{i=0}^{n-1}$ вычисляется по формуле:

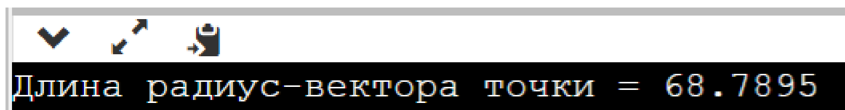
$$d = \sqrt{{x_0}^2 + {x_1}^2 + \cdots + {x_{n-1}}^2} = \sqrt{\sum_{i=0}^{n-1} {x_i}^2}\,.$$

Пример.

```
main.cpp
1 #include <iostream>
2 #include <cmath>
3 
4 int main() {
5     const int N = 5;
6     float x[N] = {1., 21., 13., 64., 5.};
```

```
 7
 8      //промежуточное вычисление суммы в переменной sum
 9      float sum = 0;
10      for(int i = 0; i < N; i++) {
11          sum = sum + x[i] * x[i];
12      }
13
14      //вывод на экран длины радиус-вектора
15      std::cout<< "Длина радиус-вектора точки = "
16          << sqrt(sum);
17      return 0;
18  }
```

Результат работы программы:



```
Длина радиус-вектора точки = 68.7895
```

Умножение вектора на скаляр

Умножение на скаляр λ n-мерного вектора в контексте одномерного массива $a = \{a_i\}_{i=0}^{n-1}$ определяется выражением:

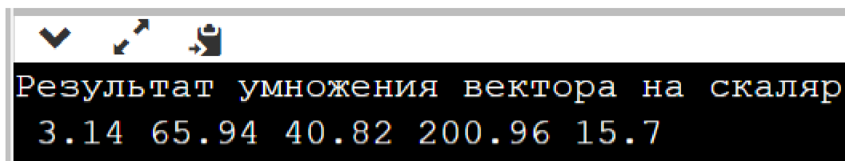
$$\lambda \cdot a = \{\lambda \cdot a_i\}_{i=0}^{n-1}.$$

Пример.

```
main.cpp
 1  #include <iostream>
 2  using namespace std;
 3
 4  int main() {
 5      const int N = 5;
 6      float lambda = 3.14;
 7      float x[N] = {1., 21., 13., 64., 5.};
 8
 9      //умножение вектора на скаляр
10      for(int i = 0; i < N; i++) {
11          x[i] = lambda * x[i];
12      }
13
```

```
14        //вывод на экран преобразов. массива
15        cout<< "Результат умножения вектора на скаляр"
16            << endl;
17        for(int i = 0; i < N; i++) {
18            cout<<' '<< x[i];
19        }
20        return 0;
21    }
```

Результат работы программы:


Результат умножения вектора на скаляр
 3.14 65.94 40.82 200.96 15.7

Использование в программе меньшего количества элементов массива, чем задано при объявлении

Интерактивный ввод значений массива

Обычно от работающей программы требуется обработка массивов с различным количеством элементов, а также различных значений. Использовать для этого инициализацию неудобно.

Более того непонятно почему разработчик обязан передать открытый код своей программы всем желающим, а по-другому невозможно, т.к. другой пользователь не сможет инициализировать массив своими собственными данными.

Рассмотренные до сих пор примеры имеют строго определенное количество элементов и не предполагают никакой гибкости при вводе интерактивном вводе данных, т.к. интерактивный ввод просто отсутствует.

Данную проблему легко исправить:

- первое что должен сделать программист, это определить из требований заказчика или будущих пользователей максимальное число элементов одномерного массива, для которого данная программа является рабочей;
- второе – это объявить массив в разрабатываемой программе с этим максимальными числом;
- третье – организовать интерактивный ввод целого числа равного меньшему (или равному максимальному значению) количества

элементов массива для обработки (обычно имя этой переменной n);

- четвертое - организовать интегративный ввод усеченной части массива с помощью введенного на предыдущем шаге количества активных элементов (n);
- пятое - ограничить во всех циклах по всему тексту программы количество обрабатываемых элементов с помощью введенного на третьем шаге целого числа (n).

Приведем пример реализации этих шагов. Предположим, что заказчик будет обрабатывать одномерные массивы чисел с плавающей точкой, в которых количество элементов ***не*** превосходит 100 (обозначим это число с помощью имени MAX_DIM используя макроподстановку), кроме того, заведем переменную n для ввода числа активных элементов массива.

Пример.

main.cpp

```
1  #include <iostream>
2  #define MAX_DIM 100
3
4  int main() {
5      double array[MAX_DIM];
6      int n;
7
8      while(true) {
9          std::cout<<  "Введите число элементов <="<< MAX_DIM
10                     << std::endl;
11         std::cin>> n;
12         if(0 < n && n <= MAX_DIM) break;
13     }
14
15     std::cout<< "Интерактивный ввод массив"
16                << std::endl;
17     for(int i = 0; i < n; i++) {
18         std::cout<<"array["<< i<<"]=";
19         std::cin>> array[i];
20     }
21
22     std::cout<<"Проверка. Исходный массив"
23                << std::endl;
24     for(int i = 0; i < n; i++) {
25         std::cout<< ' '<< array[i];
26     }
27     //Далее можно вставлять в эту программу любые
```

```
28      //алгоритмы для обработки одномерного массива и
29      //вывода на экран результата
30      return 0;
31  }
```

Результат выполнения программы:

```
Введите число элементов <=100
5
Интерактивный ввод массив
array[0]=1
array[1]=2
array[2]=3
array[3]=4
array[4]=5
Проверка. Исходный массив
 1 2 3 4 5
```

Применение генератора случайных чисел для автоматического заполнения одномерного массива

Компьютеры неспособны генерировать случайные числа. Вместо этого они могут имитировать случайность, что достигается с помощью генератора псевдослучайных чисел, а также вспомогательных функций.

Функции `srand()` *и* `rand()`

Язык C++ имеет свои собственные встроенные средства генерации псевдо случайных чисел. Они реализованы в двух отдельных функциях, которые находятся в заголовочном файле `cstdlib`:

- функция `srand()` устанавливает передаваемое разработчиком значение в качестве стартового. `srand()` следует вызывать только один раз — в начале программы (обычно в верхней части функции `main()`);
- функция `rand()` генерирует следующее случайное число в последовательности. Оно будет находиться в диапазоне от `0` до `RAND_MAX` (константа в `cstdlib`, значением которой является `32767`).

Пример.

main.cpp

```cpp
#include <iostream>
#include <cstdlib>  //для функций rand() и srand()
#define SEED 4541   //начальное значение для srand()
#define NUMBER 12   //количество псевдосл. чисел
#define NUM_PRINT_COLUMNS 4 //кол. колонок на экране
using namespace std;

int main() {
    srand(SEED); //устанавл. старт. значение 4541

    //выводим NUMBER случайных чисел
    for (int count = 1; count <= NUMBER; ++count) {
        cout << rand() << "\t";
        //через 5 чисел вставляем символ новой строки
        if (count % NUM_PRINT_COLUMNS == 0) cout<<"\n";
    }
    return 0;
}
```

Результат работы программы:

```
822351550    1622942556    1721308543    188933821
858593265    1584136113    790961218     377942596
486417893    2090487109    1000318466    402762403
```

Стартовое число и последовательности в ГПСЧ

Запуская вышеприведенную программу несколько раз, то заметите, что в результатах всегда находятся одни и те же числа. Хотя каждое число в последовательности кажется случайным относительно предыдущего, вся последовательность не является случайной. В свою очередь это, означает, что результат генерации предсказуем (одни и те же значения ввода приводят к одним и тем же значениям вывода).

Более подробно рассмотрим, почему это происходит и как это можно исправить. Следует понимать, что при генерации каждое новое число в последовательности ГПСЧ (генератора псевдослучайных чисел) генерируется исходя из предыдущего определенным способом. Таким образом, при постоянном начальном числе ГПСЧ всегда будет генерировать

одну и ту последовательность. В программе, приведенной выше, последовательность чисел будет всегда одинакова, так как стартовое число всегда равно 4541.

Для того, чтобы это исправить необходимо автоматически каждый запуск программы указывать другое число. Общепринятым решением является использование системных часов. Каждый раз, при запуске программы, время будет другое.

Для реализации этого подхода в языке C++ есть функция time(), которая возвращает в качестве времени общее количество секунд, прошедшее от полуночи 1 января 1970 года. Чтобы использовать эту функцию, нам просто нужно подключить заголовочный файл ctime, а затем инициализировать функцию srand() вызовом функции time(0).

Внесем на первый взгляд незначительные исправления в уже приведенную выше программу:

- добавим подключение еще одного заголовочного файла (ctime) (#include <ctime>).
- вместо вызова srand(**SEED**) выполним вызов функции srand() со значением системных часов в качестве в качестве стартового числа (srand(**time(0)**)).

Генерация случайных чисел в заданном диапазоне

Пусть разработчику необходимо сченерировать псевдослучайные числа в заданном диапазоне от нуля до некоторой целочисленной константы MAX_RAND_VALUE.

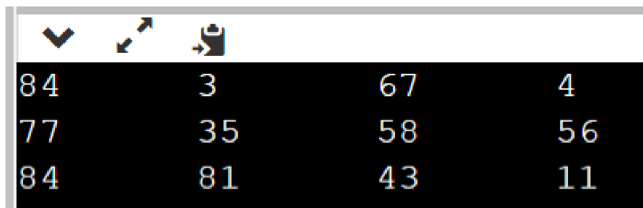
Для этого достаточно использовать операцию вязтия целочисленного остатка: rand() % MAX_RAND_VALUE. Кроме того, следует использовать указанное в предыдущем пункте начальное значение в виде time(0).

Пример.

main.cpp
```
1  #include <iostream>
2  #include <cstdlib>  //для функций rand() и srand()
3  #include <ctime>        //для функции time
4  #define NUMBER 12   //количество псевдосл. чисел
5  #define NUM_PRINT_COLUMNS 4 //кол. колонок на экране
6  #define MAX_RAND_VALUE 100
7  using namespace std;
8
9  int main() {
10     srand(time(0));
```

```
    for (int count = 1; count <= NUMBER; ++count) {
        cout << rand() % MAX_RAND_VALUE << "\t";
        if (count % NUM_PRINT_COLUMNS == 0) cout<<"\n";
    }
    return 0;
}
```

Результат работы программы:



Пример заполнения случайными числами одномерного массива

Заполним одномерный массив чисел с плавающей точкой случайными значениями.

Пример.

main.cpp

```
#include <iostream>
#include <cstdlib>  //для функций rand() и srand()
#include <ctime>    //для функции time()
#define MAX_DIM 100
#define MAX_RAND_VALUE 100

int main() {
    double array[MAX_DIM];
    int n;

    while(true) {
        std::cout<<"Введите число элементов <="
                 << MAX_DIM
                 << std::endl;
        std::cin>> n;
        if(0 < n && n <= MAX_DIM) break;
    }
```

```
18
19      //заполнение массива псевдослуч. числами
20      srand(time(0));
21      for(int i = 0; i < n; i++) {
22          array[i] = rand() % MAX_RAND_VALUE;
23      }
24
25      std::cout<< "Псевдо случ. массив"
26                << std::endl;
27      for(int i = 0; i < n; i++) {
28          std::cout<<' '<< array[i];
29      }
30      //далее можно вставлять в эту программу любые
31      //алгоритмы для обработки одномерного массива и
32      //вывода на экран результата
33      return 0;
34  }
```

Результат работы программы:

```
Введите число элементов <=100
7
Псевдо случ. массив
 7 14 14 76 9 44 17
```

Строки

Для представления символьной информации можно использовать литералы-символы, символьные переменные и символьные именованные константы. В C++ поддерживаются два типа строк – встроенный тип, доставшийся «в наследство» от C, и класс `string` из стандартной библиотеки C++. Класс `string` предоставляет гораздо больше возможностей обработки строк и удобнее в применении. Но он будет рассматриваться позже, как часть библиотеки STL.

Встроенный строковый тип

Встроенный строковый тип перешел в C++ от C. Строка символов хранится в памяти как одномерный массив. Количество элементов в таком

массиве на один элемент больше, чем изображение строки на экране, т.к. в конец строки добавлен '\0'.

Формат объявления и инициализации строк:

```
char имяСтроки[] = "текст";
```

или

```
char имяСтроки[] = {'т', 'е', 'к', 'с', 'т', '\0'};
```

Пример.

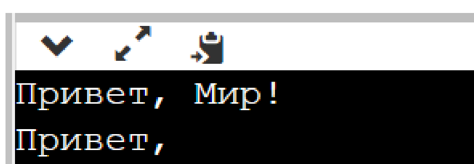
```
char str[] = "String";
char strArray[]={'S', 't', 'r', 'i', 'n', 'g', '\0'};
```

Кроме инициализации строку можно также ввести с помощью потокового ввода cin. Однако следует помнить, что ввод строки с помощью cin осуществляется до первого пробела, появившегося в строке.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      //введенная строка не должна превышать 19 симв.
6      char str[20];
7      cin >> str;
8      int a = 5;
9      cout<< str << endl;
10     return 0;
11 }
```

Результат работы программы:



Дополнительные функции языка С для строкового ввода/вывода

Ввод/вывод строк с помощью функций форматированного ввода/вывода `scanf()` и `printf()` уже рассматривался ранее. Однако не рассмотренными остались функции `puts()`, `gets()` и `fgets()` описанных в заголовочном файле `stdio.h` (или `cstdio.h` для С++), с помощью которых можно вести обмен строковыми данными.

Функция `puts()`

Функция `puts()` осуществляет вывод информации на экран. Функция `puts()` выводит на экран строку, завершая вывод переходом на новую строку. Передаваемым ей аргументом является имя строки.

Пример.

main.cpp

```
1  #include <stdio.h>
2
3  int main () {
4      const char str[] ="Проверка работы функции puts.";
5      //Вывод строки
6      puts(str);
7
8      return 0;
9  }
```

Результат работы программы:

```
Проверка работы функции puts().
```

Функция `gets()`

Использование `gets()` связано с проблемой, о которой следует знать. Применяя `gets()`, можно перейти границы массива символов, который передавался в качестве параметра. Например, если вызвать `gets()` с строкой длиной в сорок символов, а затем ввести сорок или более символов, то ошибки не будет (вся большая строка будет введена без потерь). Это

возможно, поскольку не существует способа указать для `gets()` максимальное количество символов для ввода.

Функция `gets()` не проверяет буферное пространство. В дополнение к случайным проблемам с переполнением, эта слабость может быть использована злоумышленниками для создания всевозможных разрушений. Один из первых широко распространенных червей, выпущенный в 1988 году, использовал `gets()` для распространения в Интернете.

В настоящее время практически все современные интегрированные среды (в том числе и онлайн IDE) сообщают о невозможности использования данной функции.

Функция `fgets()`

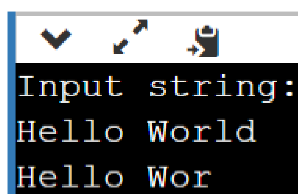
В замену `gets()` была создана практически аналогичная функция `fgets()`. Она отличается тем, что в ней необходимо уже указывать размер буфера для вводимой информации, т.е. наибольшее ожидаемое количество символов для ввода.

Пример.

main.cpp

```
1  #include<iostream>
2
3  int main() {
4      const int BUFFER_SIZE = 10; //включая '\0'
5      char str[BUFFER_SIZE];
6      puts("Input string: ");
7      //вводим строку с пробелами,
8      //stdin - имя стандартного входного потока
9      fgets(str, BUFFER_SIZE, stdin);
10
11     puts(str);
12     return 0;
13 }
```

Результат работы программы:



```
Input string:
Hello World
Hello Wor
```

Некоторые функции языка С, применяемые для обработки строк

Стандартная библиотека С предоставляет набор функций для манипулирования строками (Таблица 35). Эта библиотека является частью библиотеки С++. Для ее использования необходимо подключить одни из заголовочных файлов `cstring` или `stdlib.h` директивой `#include`.

Пример.

```
#include <cstring>
```

Таблица 35 – Некоторые функции, используемых в С для обработки строк

Функция	Прототип и краткое описание результатов применения
`strcmp()`	`int strcmp(const char *str1, const char *str2);` Сравнивает строки `str1` и `str2`. Возвращаемое значение: • `0` – если сравниваемые строки идентичны; • положительное число – если строки отличаются и код первого отличающегося символа в строке `str1` больше кода символа на той же позиции в строке `str2`; • отрицательное число – если строки отличаются и код первого отличающегося символа в строке `str1` меньше кода символа на той же позиции в строке `str2`.
`strcpy()`	`char *strcpy (char *target, const char *join);` где `target` – строка, в которую будут скопированы данные; `join` – строка источник копируемых данных. Возвращаемое значение: строка, в которую скопированы данные.
`strdup()`	`char *strdup(const char *str);` где `str` – копируемая строка. Возвращаемое значение: • дублирующая строка; • `NULL` – если не удалость выделить память под новую строку или скопировать строку на которую указывает аргумент `str`.
`strlen()`	`int *strlen (const char *str);` где `str` – строка, длина которой вычисляется. Возвращаемое значение: количество символов в строке до первого вхождения символа конца строки `'\0'`.
`strncat()`	`char *strncat (char* target, const char* join, int n);` где `target` – строка, в конец которой будет добавлена строка `join`, `n` – максимальное количество символов добавляемых к `target` из `join`. Возвращаемое значение: объединение строки destination и `n` символов из `join`.

Замечание.

В некоторых пособиях упоминаются также функции `atoi()` *и* `atof()` *для преобразования строки в целое число и число с плавающей точкой (так называемый парсинг строк). Однако это не стандартизированные функции и, например, в онлайн интегрированных средах они не доступны.*

Примеры обработки строк

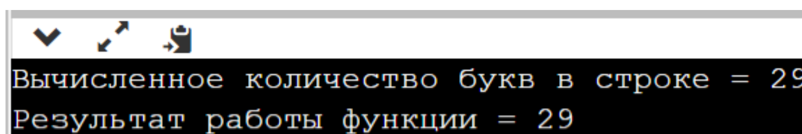
Рассмотрим алгоритм определения длины строки. Также сравним результат его работы с работой стандартной функцией `strlen()`.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstring>
3
4  int main() {
5      char str[] = "Тестовая строка";
6      int n;
7
8      for(n = 0; str[n] != '\0'; n++); // ищем '\0'
9
10     std::cout<< "Вычисленное количество букв в строке = "
11              << n
12              << std::endl
13              << "Результат работы функции = "
14              << strlen(str);
15     return 0;
16 }
```

Результат работы программы:



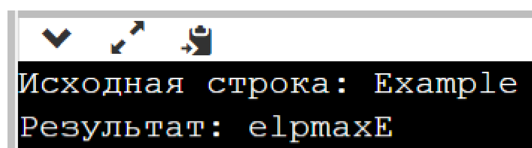
Перейдем к рассмотрению реверса строки. Поскольку по своей сути строка встроенного типа и одномерный массив - это абсолютно эквивалентные понятия в C++, то реверс строки выполняется точно также как и реверс одномерного массива.

Пример.

main.cpp

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char str[] = "Example";
    cout<<"Исходная строка: " <<str <<endl;

    int n = strlen(str);
    for(int i = 0; i < n / 2; i++) {
        char buff = str[i];
        str[i] = str[n - 1 - i];
        str[n - 1 - i] = buff;
    }
    cout<<"Результат: "<< str <<endl;
    return 0;
}
```

Результат работы программы:



Одним из самых распространенных алгоритмических действий со строками является выполнение сравнения строк. Это действие можно выполнить как с использованием самостоятельно написанного кода, так и с помощью стандартной функции `strcmp()`. В данном примере воспользуемся упомянутой функцией из заголовочного файла `cstring`.

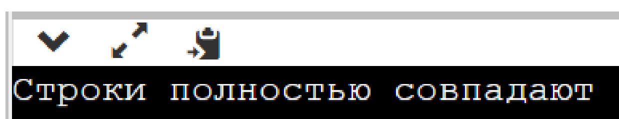
Пример.

main.cpp

```cpp
#include <iostream>
#include <cstring>
using namespace std;
```

```
4
5  int main() {
6      char str[] = "String";
7      char strArray[] = {'S','t','r','i','n','g','\0'};
8
9      if (strcmp(str, strArray) == 0) {
10         cout<< "Строки полностью совпадают";
11     }
12     else {
13         cout<< "Различные строки";
14     }
15     cout<< endl;
16     return 0;
17 }
```

Результат работы программы:



Многомерные автоматические массивы

Как будет указано позже формально многомерных массивов в C++ не существует и в качестве многомерного массива используется конструкция массива указателей на массивы меньшей размерности. Так, например, одномерный массив, элементами которого являются другие одномерные массивы значений, называется «двумерным».

Подробно физическое расположение в памяти многомерных, в частности двумерных массивов, будет рассмотрено позже после изучения указателей.

Замечание.

В дальнейшем в рамках данного учебного курса для упрощения объяснений одномерные массивы, состоящие из одномерных массивов, будут называться многомерными.

Многомерными массивами в C++ называют массивы, которые имеют два и более индексов. Формат объявления массива:

```
МодифСпецТипа ИмяМассива[КоличЭлем1]…[КоличЭлемN];
```

Многомерные массивы допускают инициализацию. В этом случае нет необходимости указывать количество элементов по столбцам, строкам,

листам и т.д. Разбитие на строки, листы и т.д. при инициализации осуществляется с помощью фигурных скобок «{ }».

В случае его неполной инициализации (не всем элементам массива присваиваются инициализирующие значения), остальной части элементов «по умолчанию» присваиваются нулевые значения.

Пример.

```
//объявление двумерного массива
int a[5][7];

//построчная инициализация двумерного массива:
//можно не указывать только количество строк
int a[][3] = {{1, 2, 3}, {4, 5, 6}};

//неполная инициализация
int a[5][7] = {{1, 2}, {1, 2}};

//инициализация нулями
    int a[5][7] = {0};
//либо
    int a[5][7] = {{0}};
```

Примеры элементарных действий с двумерным массивом

Пусть перед разработчиком стоит задача инициализировать двумерный массив с количеством элементов `n x m` (`n` – количество столбцов, а `m` – количество строк) (Рисунок 14) и выполнить какие-либо действия с данным массивом.

$$\begin{pmatrix} a_{0,0} \cdots a_{0,j} \cdots a_{0,m-1} \\ a_{i,0} \cdots a_{i,j} \cdots a_{i,m-1} \\ a_{n-1,0} \cdots a_{n-1,j} \cdots a_{n-1,m-1} \end{pmatrix}$$

Рисунок 14 – Прямоугольный двумерный массив `n x m`

Замечания:

- *следует избегать использования термина «матрица» при работе с двумерными массивами, т.к. по определению матрицы индексы должны начинаться с единицы, а в двумерном массиве в C++ они начинаются с нуля;*
- *в примерах, приведенных в этом разделе при объявлении или инициализации двумерных массивов, будут использоваться:*
    - *для прямоугольных массивов - две заглавные буквы N и M для обозначения констант определяющих количество строк и столбцов;*
    - *для квадратных массивов - одна заглавная буква N для обозначения количество строк и столбцов.*

Вывод на экран инициализированного массива

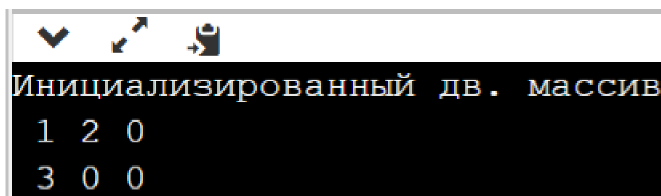
Очевидно, следует воспользоваться результатами решения задачи вывода одномерного массива в одну строку и построить это вывод `n` раз (количество строк двумерного массива), а также не забыть, что количество элементов в строках `m` (Рисунок 14).

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    const int N = 2;
    const int M = 3;
    //неполная инициализация
    int a[N][M] = {{1, 2}, {3}};

    cout<<"Инициализированный дв. массив"<< endl;
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            cout<<' '<< a[i][j];
        }
        cout<<endl; //переход к след. строке дв. массива
    }
    return 0;
}
```

Результат работы программы:



Продемонстрируем, что двумерный массив является одномерным массивом, состоящим из одномерных массивов. Для этого модифицируем предыдущую программу.

Пример.

main.cpp
```
#include <iostream>
using namespace std;

int main() {
    const int M = 3;
    int a[][M] = {{1, 2, 3}, {3, 2, 1}};
    cout<<"Инициализированный дв. массив"<< endl;
    //определяем количество строк (одномерных массивов)
    int N = sizeof(a)/sizeof(a[0]);
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            cout<<' '<< a[i][j];
        }
        cout<<endl; //переход к след. строке массива
    }
    return 0;
}
```

Результат работы программы не отличается от предыдущего.

Выбор минимального значения в двумерном массиве

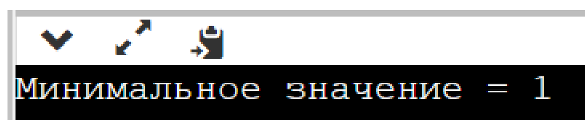
Рассмотрим алгоритм выбора минимального значения в прямоугольном инициализированном массиве `N x M`.

Пример.

main.cpp

```
#include <iostream>

int main() {
    const int N = 2;
    const int M = 3;
    int a[N][M] = {{1, 2, 3}, {3, 2, 1}};

    int min = a[0][0];
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < M; j++) {
            if( min > a[i][j] ) min = a[i][j];
        }
    }
    std::cout<< "Минимальное значение = " << min;
    return 0;
}
```

Результат работы программы:



Определение индексов первого по порядку перебора встретившегося минимального

Несложно обобщить предыдущий алгоритм на случай выбора минимума по индексам, также как это делалось для одномерного массива. Однако в этом случае, очевидно, необходимо использовать для хранения значений индексов уже две переменные (`iMin` и `jMin`).

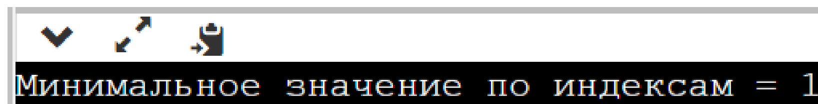
Пример.

main.cpp

```
#include <iostream>
using namespace std;

```

```
 4 int main() {
 5     const int N = 2;
 6     const int M = 3;
 7     int a[N][M] = {{1, 2, 3}, {3, 2, 1}};
 8 
 9     int iMin = 0, jMin = 0;
10     for(int i = 0; i < N; i++) {
11         for(int j = 0; j < M; j++) {
12             if(a[iMin][jMin] > a[i][j] ) {
13                 //операция запятая
14                 (iMin = i, jMin = j);
15             }
16         }
17     }
18     cout<< "Минимальное значение по индексам = "
19         << a[iMin][jMin];
20     return 0;
21 }
```

Результаты работы программы:



Заполнение двумерного массива по шаблону

Далее в примерах будет рассматриваться только квадратный двумерный массив, т.е. массив N x N (N столбцов и N строк). Обычно задание на заполнение по шаблону содержат два варианта заполнения

- линейного элемента двумерного массива (строки столбца, главной или побочной диагонали);
- всего двумерного массива по определенному «узору», обычно связанному с частями двумерного массива (половине, четверти и пр.).

Заполнение главной диагонали квадратного двумерного массива единицами

В соответствии с заданием необходимо расставить единицы на главной диагонали квадратного двумерного массива. Все остальные элементы должны быть нулевыми. Данное задание обычно демонстрирует элементарные знания синтаксиса C++ и понимание основ алгоритмизации.

Пример.

main.cpp

```
#include <iostream>

int main() {
    const int N = 3;
    int a[N][N] = {0};

    //заполнение глав. диагонали
    for(int i = 0; i < N; i++) {
        a[i][i] = 1;
    }

    //вывод на экран результатов заполнения
    std::cout<<"Дв. массив с заполненой гл. диаг."
              << std::endl;
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            std::cout<< ' '<< a[i][j];
        }
        std::cout<< std::endl;
    }
    return 0;
}
```

Результат работы программы:

```
Дв. массив с заполненой гл. диаг.
 1 0 0
 0 1 0
 0 0 1
```

Присваивание номера строки элементам побочной диагонали

Выполнение задание для побочной диагонали призвано продемонстрировать четкость восприятия поставленной задачи, а также лаконичность ее реализации в коде.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int main() {
    const int N = 3;
    int a[N][N] = {0};

    //побочная диагональ
    for(int i = 0; i < N; i++) {
        a[i][N - 1 - i] = i + 1;
    }

    //вывод на экран результатов заполнения
    cout<<"Заполнение побочной диаг."<< endl;
    for(int i = 0; i < N; i++) {
        for(int j = 0; j < N; j++) {
            cout<<' '<< a[i][j];
        }
        cout<<endl;
    }
    return 0;
}
```

Результат работы программы:

```
Заполнение побочной диаг.
 0 0 1
 0 2 0
 3 0 0
```

Заполнение единицами главной диагонали и верхнего треугольника квадратного двумерного массива

Рассмотрим пример заполнения единицами каких-либо частей двумерного массива, в частности, ее верхнего треугольника, расположенного над главной диагональю (включая диагональ). Поскольку в данном случае заполняемая область представляет собой двумерный объект, то, очевидно, при заполнении не удастся обойтись одним циклом.

Особенность же заполнения должна продемонстрировать точное понимание учащимся взаимосвязи определения индексов. При этом будет большой ошибкой, демонстрирующей практическое отсутствие опыта в стандартных задачах алгоритмизации, если учащийся воспользуется оператором `if()` внутри вложенного цикла, заполняющего массив. Это связано с тем, что применение данного оператора увеличит время заполнения двумерного массива почти в два раза.

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      const int N = 3;
5      int a[N][N] = {0};
6      //заполнение верхнего треугольника
7      for(int i = 0; i < N; i++) {
8          for(int j = i; j < N; j++) {
9              a[i][j] = 1;
10         }
11     }
12     //вывод на экран результатов заполнения
13     std::cout<<"Верхний треугольник"
14                 << std::endl;
15     for(int i = 0; i < N; i++) {
16         for(int j = 0; j < N; j++) {
17             std::cout<< ' '
18                         << a[i][j];
19         }
20         std::cout<< std::endl;
21     }
22     return 0;
23 }
```

Результат работы программы:

```
Верхний треугольник
 1 1 1
 0 1 1
 0 0 1
```

Транспонирование квадратного двумерного массива

Одной из наиболее известных операций с двумерными массивами является транспонирование, т.е. перестановке строк и столбцов. Данный пример также является по сути дела знаковым, т.к. учащийся должен продемонстрировать не только понимание работы с индексами (аналогично предыдущему примеру), но и знание элементарных правил перестановок значений.

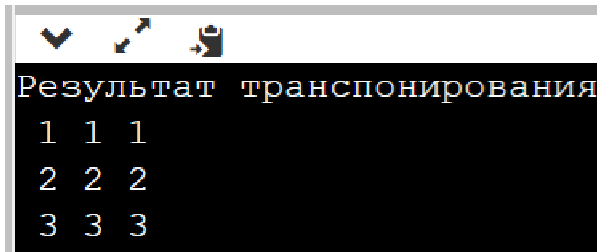
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      const int N = 3;
6      int a[N][N] = {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}};
7      //транспонирование
8      for(int i = 0; i < N; i++) {
9          for(int j = i + 1; j < N; j++) {
10             int buff = a[i][j];
11                 a[i][j] = a[j][i];
12                 a[j][i] = buff;
13         }
14     }
15     //вывод на экран результатов заполнения
16     cout<<"Результат транспонирования"<< endl;
17     for(int i = 0; i < N; i++) {
18         for(int j = 0; j < N; j++) {
19             cout<< ' '<< a[i][j];
20         }
```

```
21            cout<<endl;
22        }
23        return 0;
24    }
```

Результат выполнения программы:



Ввод с клавиатуры «усеченного» двумерного массива

Собственно, как и в случае с одномерным массивом в практических приложениях всегда необходимо создать возможность интерактивного ввода размерности двумерного массива, а также интерактивного ввода значений самого массива для выполнения действий, требуемых по алгоритму.

Как и ранее будем предполагать, что заказчик заявил, что двумерный массив размерностью более чем `20 x 30` использоваться не будет. Это нисколько не снижает общности рассматриваемой задачи, а конкретные значения, определяющие размерности, всегда могут быть увеличены.

Пример.

main.cpp

```
1   #include <iostream>
2   #define I_MAX_DIM 20
3   #define J_MAX_DIM 30
4
5   int main() {
6       double a[I_MAX_DIM][ J_MAX_DIM];
7       //активное количество строк и столбцов
8       int n, m;
9       //проверка корректности введеных значений n и m
10      while(true) {
11          std::cout<<"Введите число строк <="
12                  << I_MAX_DIM << std::endl;
```

```
13          std::cin>> n;
14          std::cout<<"Введите число столбцов <="
15                  << J_MAX_DIM << std::endl;
16          std::cin>> m;
17          if(0 < n && n <= I_MAX_DIM &&
18             0 < m && m <= J_MAX_DIM) break;
19      }
20      std::cout<<"Интерактивный ввод дв. массива"
21              << std::endl;
22      for(int i = 0; i < n; i++) {
23          for(int j = 0; j < m; j++) {
24              std::cout<< "a["<< i<< "]["<<j<<"]=";
25              std::cin>> a[i][j];
26          }
27      }
28      std::cout<< "Проверка. вывод введеного массива"
29              << std::endl;
30      for(int i = 0; i < n; i++) {
31          for(int j = 0; j < m; j++) {
32              std::cout<< ' '<< a[i][j];
33          }
34          std::cout<<std::endl;
35      }
36      //Далее можно вставлять в эту программу любые
37      //алгоритмы для обработки двумерного массива и
38      //вывода на экран результата
39      return 0;
40  }
```

Результат работы программы:

```
Введите число строк <=20
2
Введите число столбцов <=30
3
Интерактивный ввод дв. массива
a[0][0]=1
a[0][1]=2
a[0][2]=3
a[1][0]=4
a[1][1]=5
a[1][2]=6
Проверка. вывод введеного массива
 1 2 3
 4 5 6
```

Заполнение двумерного массива псевдослучайными числами

Заполнение двумерного массива псевдослучайными числами требует:

- использования дополнительных директив препроцессора, выделенных цветом:

```
#include <iostream>
#include <cstdlib>  //для функций rand() и srand()
#include <ctime>    //для функции time
#define MAX_RAND_VALUE 10 //диапазон гинерации [0, 9]
#define I_MAX_DIM 20
#define J_MAX_DIM 30
```

- замены блока, выделенного цветом в предыдущем примере на код:

```
    //заполнение псевдосл. числами
    srand(time(0));
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            a[i][j] = rand() % MAX_RAND_VALUE;
        }
    }
```

Введение в функции

Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе С++ должна быть функция с именем `main` (главная функция), именно эта функция является исполняемой программой.

С использованием функций связаны четыре понятия:

- объявление функции (прототип функции);
- определение функции (описание действий, выполняемых функцией);
- вызов функции;
- возврат значения в точку вызова.

Функция – это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи.

Одно из главных преимуществ, предоставляемых функцией, состоит в том, что она может быть выполнена столько раз, сколько необходимо, в различных точках программы (функции с именем `main`).

Без такой возможности программы были бы намного больше, поскольку один и тот же код повторялся бы много раз. Необходимость в функциях вызвана также тем, что для независимой разработки и тестирования программу можно разбивать на фрагменты.

Объявление функций (прототип функции)

Если требуется вызвать функцию до ее определения в рассматриваемом файле или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением (прототипом) этой функции перед функцией `main()`.

Объявление (прототип) функции имеет следующий формат (для сокращения записи модификаторы не используются):

`типВозврЗнач` **`имяФункции`**`(`~~`список, форм., парам.`~~`);`

где:

- `типВозврЗнач` функции обязателен и задает тип возвращаемого ею значения в точку вызова (в качестве типа возвращаемого значения может использовать любой из допустимых типов, в частности, любой базовый);
- **`имяФункции`** –
    - либо `main` для основной функции,
    - либо произвольный идентификатор, не совпадающий со служебными словами и именами других объектов программы;
- ~~`список, форм., парам`~~`.` – это разделенная запятыми последовательность объявлений формальных параметров, которая:
    - может содержать перечисление через запятую записей вида:

      ~~`модификатор`~~ `тип` ~~`имяПараметра`~~

    - может содержать только перечисление через запятую списка типов (с/без модификаторов), но без конкретизации имен параметров (~~`имяПараметра`~~ отсутствует);
    - может полностью отсутствовать при этом после **`имяФункции`** пишутся пустые круглые скобки;
    - если программист хочет подчеркнуть, что список формальных параметров пуст, то в этом случае следует написать `void` в круглых скобках.
- прототип функции всегда завершается точкой с запятой, т.е. пустым оператором (выделено желтым).

Пример.

```
//полное объявление функции rus() с именами параметров
long rus(unsigned char c, double x);

//эквивалентное пред. объявление rus() без имен парам.
long rus(unsigned char, double);

//объявление функции belarus() даже без типов
long belarus();

//еще одно эквивалентное объявление функции belarus()
long belarus(void);
```

Определение функции

Определение функции имеет следующую форму:

`типВозврЗнач` **`имяФункции`** `(`~~`список, форм., парам.`~~`)` **`{`**
    `последовательность операторов;`
`}`

где `последовательность операторов` определяет алгоритм работы функции.

При этом ***тело функции*** – это `последовательность операторов`, заключенная в фигурные скобки `{ }`. Таким образом, ***тело функции*** – это блок кода (или составной оператор).

Определение отличается от объявления (***прототипа***):

- наличием ***тела функции*** и отсутствием `точки с запятой` (т.е. пустого оператора) после заголовка;
- в ***прототипе*** могут отсутствовать имена формальных параметров, а в ***определении*** все формальные параметры должны быть поименованы.

В остальном определение функции в заголовке и ее прототип ***должны совпадать***. Перечислим совпадающие элементы прототипа и описания функции:

- тип возвращаемого значения (с учетом модификаторов),
- типы и последовательность формальных параметров (с учетом модификаторов).

Замечание.

Компилятор сообщит об ошибке, если прототип и определение функции не полностью согласуются.

Область действия имен ***формальных параметров*** распространяется только до конца тела функции. Даже когда функция не использует ***формальных параметров***, то наличие круглых скобок обязательно.

Вызов функции

Формат вызова функции определится следующей синтаксической конструкцией:

ИмяФункции(~~список, фактич., парам.~~);

Точка вызова функции — это запись, содержащая имя функции с указанным списком ***фактических*** параметров (уже вычисленных значений).

Эта запись предписывает процессору прервать выполнение текущей функции (например, `main()`) и приступить к выполнению другой функции (***вызываемой***), имя которой указано в точке вызова.

Процессор «оставляет закладку» в текущей точке выполнения, и приступает к выполнению вызываемой функции. Когда выполнение вызываемой функции завершено, процессор возвращается к закладке и возобновляет выполнение прерванной (***вызывающей***) функции (например, `main()`).

Каждый ***фактический аргумент*** к моменту вызова функции ***должен быть*** либо константой, либо переменной с уже вычисленным значением, либо результатом вычисленного выражения. Таким образом, если фактический аргумент представлен в виде выражения, то его значение сначала вычисляется, а затем передается в вызываемую функцию.

Пример.

```
…
//прототип функции с одним формальным параметром
double SetCoordinateX(double v);
…
y = 5;
s = 7;
z = 1.5;
//пример точки вызова функции и передача одного
//значения одного фактического параметра (v = z * s)
x = y*SetCoordinateX(z * s);
…
```

Если в функцию требуется передать несколько уже вычисленных ***фактических*** значений, то они записываются через запятую. При этом формальные параметры инициализируются значениями фактических параметров в порядке их следования в заголовке функции.

Пример.

```
…
// прототип с тремя формальными параметрами
float min(float x, float y, float z);
…
s = 1; f = 3; p = 0;// значения фактических
                    // параметров должны быть уже определено
v = min(s, f, p);   // точка вызова x = s, y = f, z = p
…
```

Возвращение результата в точку вызова функции

Для передачи результата из функции в точку вызова используется оператор `return`, записанный в теле функции. Он может использоваться в двух формах:

- `return;` – завершает функцию, не возвращающую никакого значения (т.е. перед именем функции указан тип `void`);
- `return переменная/выражение;` – возвращает значение `переменной` или `выражения`, тип которого будет приведен к типу, указанному перед именем функции.

Если не писать в программе оператор `return` явно, то компилятор автоматически дописывает пустой `return;` в конец тела функции перед закрывающей фигурной скобкой. При этом не важно возвращает эта функция значение или нет.

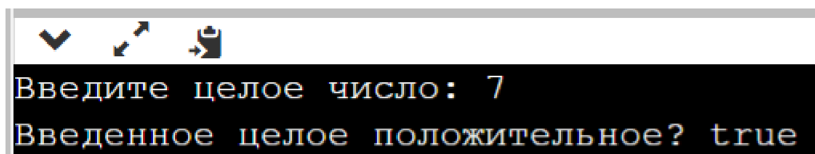
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  //прототип (объявление) функции
5  bool isPositive(int n);
6
```

```
7  int main() {
8      int a;
9      cout<< "Введите целое число: ";
10     cin>>a;
11     cout<<boolalpha;
12     cout<<"Введенное целое положительное? "
13         //точка вызова функции isPositive()
14         << isPositive(a);
15     return 0;
16 }
17
18 //определение (описание) функции
19 bool isPositive(int n) {
20     return 0 < n; //возвращаемое значение
21 }
```

Результат работы программы:



Все переменные, объявленные в теле функции «по умолчанию» являются локальными. В примере переменная `n` (формальный параметр) существует только до конца работы функции, и обратиться к ней можно только внутри тела функции.

При вызове функции локальным переменным отводится память в ***стеке*** и производится их инициализация передаваемыми при вызове фактическими значениями параметров (в примере, `n = a`).

Далее начинается выполнение тела функции и управление передается первому оператору, которым является оператор `return` (он же последний оператор тела функции). Однако перед выполнением этого оператора и завершением работы функции сперва вычисляется значение логического выражения `0 < n` и уже его результат возвращается оператором `return` в качестве результата работы всей функции.

Управление программой при этом возвращается в точку вызова выполненной функции, а ее локальные переменные удаляются из памяти.

При новом вызове функции для локальных переменных память выделяется вновь, и поэтому старые значения локальных переменных теряются.

Некоторые примеры функций, возвращающих значение

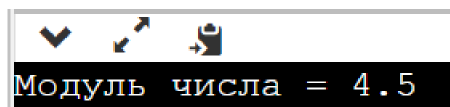
Рассмотрим простейший вариант функции `absValueIf()`, вычисляющей модуль вещественного числа. Она написана с использованием оператора `if()`.

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

//прототип (объявление) функции
double absValueIf(double);

int main() {
    float z = -4.5;
    cout<<"Модуль числа = "
        //точка вызова функции
        << absValueIf(z);
    return 0;
}

//определение (описание) функции
double absValueIf(double x) {
    if(x < 0) x = -x;
    return x; //возвращаемое значение
}
```

Результат работы программы:



Модуль числа можно также вычислить в функции `ternaryAbsValue()` с помощью тернарной операции.

```
//определение (описание) функции
double ternaryAbsValue(double x) {
    return x < 0 ? -x : x; //возвращаемое значение
}
```

Замечание.

Необходимо подчеркнуть, что при использовании тернарной операции в операторе `return` *ее* ***не*** *следует выделять скобками как в* `cout`.

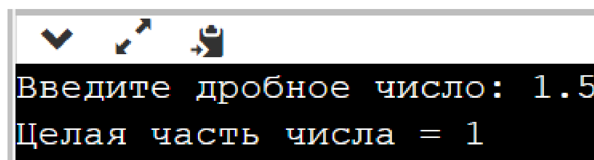
Рассмотрим функцию `integerPart()`, возвращающую целое число, равное целой части ***положительного вещественного*** числа `x`, переданного в функцию как параметр. В данном случае «отрубание» дробной части происходит при приведении типа переданного параметра `x` (тип `double`) к типу возвращаемого функцией значения (тип `int`). Это случай приведения типа операндов при присваивании.

Пример.

main.cpp

```
1  #include <iostream>
2
3  //прототип функции
4  int integerPart(double);
5
6  int main() {
7      float z;
8      std::cout<< "Введите дробное число: ";
9      std::cin>>z;
10     std::cout<<"Целая часть числа = "
11         //точка вызова функции
12         << integerPart(z);
13     return 0;
14 }
15
16 //описание функции
17 int integerPart(double x) {
18     return x; //возвращаемое значение
19 }
```

Результат работы программы:



Еще одним примером эффективного использования приведения типов является функция `fractionalPart()`, вычисляющая дробную часть ***положительного*** числа с плавающей точкой. В данном случае в операторе `return` используется явное приведение типов.

Пример.

```
//определение (описание) функции
double fractionalPart(double x) {
    return x - (int)x; //возвращаемое значение
}
```

Возврат значений функцией main()

На основе общего понимания как работают функции можно применить эти знания к функции `main()`. При вызове `main()` операционная система начинает ее выполнение. Алгоритмические блоки в `main()` выполняются последовательно. В конце `main()` возвращает целочисленное значение (обычно `0`) обратно в операционную систему. Поэтому `main()` объявляется как `int main()`.

Возвращать значения обратно в операционную систему нужно потому, что возвращаемое значение функции `main()` является кодом состояния, который сообщает операционной системе об успешном или неудачном выполнении программы. Обычно, возвращаемое значение `0` (нуль) означает то, что все прошло успешно, тогда как любое другое значение означает неудачу/ошибку.

Обратите внимание, что по стандартам языка C++ функция `main()` должна возвращать целочисленное значение. Однако, если не указать `return` в конце функции `main()`, то компилятор возвратит `0` автоматически, если иных ошибок не будет.

Замечание.

Рекомендуется ***обязательно явно*** *в конце писать оператор* `return 0;`.

Тип возврата void. Игнорирование значений, возвращаемых функциями

Функции могут и не возвращать значения. Чтобы сообщить компилятору, что функция не возвращает значение, нужно использовать в качестве типа возвращаемого значения тип `void`.

Пример.

main.cpp

```cpp
#include <iostream>

//прототип (объявление) функции
void doPrint(void);

int main() {
    //вызов функции, не возвращающей результат
    doPrint();
    return 0;
}

//объявление (описание) функции
void doPrint(void) {
    std::cout<<"Сообщение"; // нет возвращаемого зн.
}
```

Результат работы программы:

Замечание.

Если функция не объявлена как `void`, то она может (по желанию программиста) использоваться в качестве операнда в любом корректном арифметическом или логическом выражении.

Хотя все функции, кроме функций, объявленных как `void`, возвращают значения, никто ***не*** заставляет разработчика использовать эти значения. Если в процессе выполнения программы не указана переменная, которой присваивается возвращаемое значение, то это значение:

- либо может быть выведено на экран, если вызов функции был осуществлен в составе `cout` (см. примеры выше),
- либо просто проигнорировано (т.е. утеряно).

Например, обычно в учебных программах игнорируются значения, возвращаемые функции форматированного ввода `printf()` и `scanf()`. Хотя обе эти функции возвращают целое число корректно выведенных на экран или введенных с клавиатуры параметров.

Пример оформления функции без прототипа

Как уже отмечено в языке С++ допускается определение (описание) функций без прототипа. Но в этом случае описание функции должно быть оформлено перед функцией `main()`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3  
4  //определение (описание) функции без прототипа
5  bool isPositive(int n) {
6      return 0 < n;
7  }
8  
9  int main() {
10     int a;
11     cout<< "Введите целое число: ";
12     cin>>a;
13     cout<<boolalpha;
14     cout<<"Введенное целое положительное? "
15         << isPositive(a);
16     return 0;
17 }
```

Результат работы программы такой же, как и раньше.

Допустимые варианты оформления функций разработчика

Перечислим возможные варианты оформления функций разработчика:

- следуют за функцией `main()` (если перед `main()` перечислены прототипы);
- расположены перед функцией `main()` (в этом случае прототипы не нужны);
- находятся в другом файле (список прототипов будет подключаться директивой `#include`).

Хотя игнорирование прототипов допустимо в начале изучения языка, но в профессиональной деятельности это считается определенной «грубостью».

Объясняется такая оценка довольно просто: профессиональное владение языком C++ неразрывно связано с созданием многофайловых проектов, а в этом случае просто необходимо умение работать с прототипами.

Последовательность простейших действий по созданию функции разработчика, выделением алгоритмической части из главной функции

Пусть стоит задача с помощью функции оформить определение четности введенного целого числа. В начале на первом этапе следует решить эту задачу в `main()`.

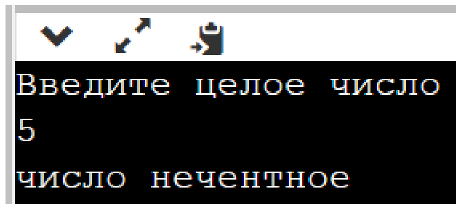
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a;
6      cout << "Введите целое число" << endl;
7      cin >> a;
```

```
 8      if (a % 2) cout<<"число нечетное";
 9      else cout<<"число четное";
10      return 0;
11  }
```

Результат работы программы:



На втором этапе следует заменить имя `main` на любой осмысленный идентификатор (например, `isOdd`). Это уже функция, созданная разработчиком, но для работы алгоритма ее уже необходимо вызвать в новом `main()`.

Пример.

```
main.cpp
 1  #include <iostream>
 2  using namespace std;
 3
 4  int isOdd() {
 5      int a;
 6      cout << "Введите целое число" << endl;
 7      cin >> a;
 8      if (a % 2) cout<<"число нечетное";
 9      else cout<<"число четное";
10      return 0;
11  }
12
13  //новая главная функция
14  int main() {
15      isOdd(); //вызов функции isOdd()
16      return 0;
17  }
```

Далее необходимо ввод переменных перенести в `main()`, а введенное значение передать в функцию через формальный параметр.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int isOdd(int a) {
5      if (a % 2) cout<<"число нечетное";
6      else cout<<"число четное";
7      return 0;
8  }
9
10 int main() {
11     //операторы перенесенные из isOdd()
12     int a;
13     cout << "Введите целое число" << endl;
14     cin >> a;
15     //значение a передается в функцию isOdd()
16     isOdd(a);
17     return 0;
18 }
```

На следующем шаге следует переделать функцию таким образом, чтобы не только ввод значения, но и результирующие сообщения также выводились на экран из `main()`. Для этого:

- из функции `isOdd()` следует перенести оператор `if()` в `main()`;
- вместо проверки параметра `a` на четность (выражение `a % 2`) вставить вызов функции `isOdd()`;
- в функции `isOdd()` в операторе `return` написать `a % 2`.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int isOdd(int a) {
5      return a % 2;
6  }
```

```
 7
 8 int main() {
 9     int a;
10     cout << "Введите целое число" << endl;
11     cin >> a;
12 
13     if ( isOdd(a) ) cout<<"число нечетное";
14     else cout<<"число четное";
15 
16     return 0;
17 }
```

На финальной стадии необходимо перенести функцию после `main()`, а перед `main()` оформить прототип из заголовка, поставив точку с запятой:

Пример.

```
main.cpp
 1 #include <iostream>
 2 using namespace std;
 3 
 4 int isOdd(int);
 5 
 6 int main() {
 7     int a;
 8     cout << "Введите целое число" << endl;
 9     cin >> a;
10 
11     if ( isOdd(a) ) cout<<"число нечетное";
12     else cout<<"число четное";
13 
14     return 0;
15 }
16 
17 int isOdd(int a) {
18     return a % 2;
19 }
```

Результат работы программы такой же, как и в исходном варианте кода.

Примеры простейших функций

Рассмотрим каким образом можно с помощью функций реализовать следующие простейшие действия:

- вычисление модуля вещественного числа с помощью оператора `if();`
- использование тернарной операции при вычислении модуля вещественного числа;
- принадлежит ли вещественное число заданному интервалу `]-a, a[;`
- вычислить целую часть вещественного числа (как положительного, так и отрицательного) с помощью вложенной тернарной операции;
- вычислить дробную часть числа с помощью вызова функции, вычисляющей целую часть числа.

Замечание.

Для простоты восприятия код программы оформлен без использования прототипов.

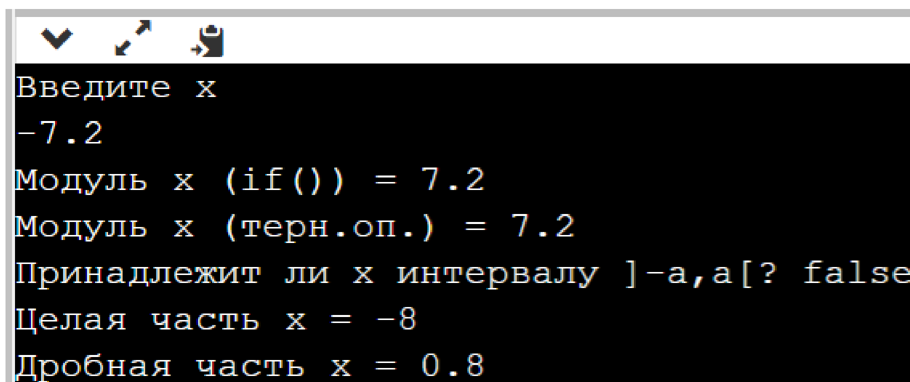
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  //вычисление модуля с помощью if()
5  double ifUserAbs(double x) {
6     if (x < 0) x = -x;
7     return x;
8  }
9
10 //вычисление модуля с помощью тернар. оп.
11 double ternUserAbs(double x) {
12     return x < 0 ? -x : x;
13 }
14
15 //определение принадлежит ли x интервалу ]-a,a[
16 bool isBelong(double a, double x) {
17     return -a < x && x < a;
18 }
19
```

```
20 //вычисление целой части x (вложенная терн. оп.)
21 int integerPart(double x) {
22     return x > 0 ? x : (x - (int) x < 0 ? x - 1 : x);
23 }
24
25 //вычисление дробной части x
26 double fractionalPart(double x) {
27     return x - integerPart(x);
28 }
29
30 int main() {
31     double a = 4;
32     double x;
33     cout<< "Введите x" << endl;
34     cin>> x;
35
36     cout<< "Модуль x (if()) = "
37         << ifUserAbs(x) <<endl
38         << "Модуль x (терн.оп.) = "
39         << ternUserAbs(x)<< endl
40         << boolalpha
41         << "Принадлежит ли x интервалу ]-a,a[? "
42         << isBelong(a,x) << endl
43         << "Целая часть x = "
44         << integerPart(x) << endl
45         << "Дробная часть x = " << fractionalPart(x);
46
47     return 0;
48 }
```

Результат работы программы:



Приведем еще один пример функции, который уже имеет «развитое» алгоритмическое тело в том смысле, что впервые в теле функции

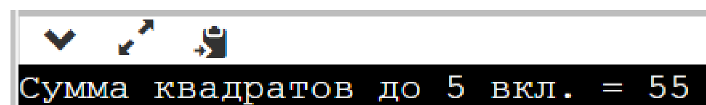
используется оператор цикла. Напишем функцию суммирования квадратов натуральных чисел до заданного натурального `n` включительно.

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

//вычисление суммы квадратов нат. чисел
int sum(int n) {
    int result = 0;
    for(int i = 1; i <=n ; i++) {
        result = result + i * i;
    }
    return result;
}

int main() {
    cout<< "Сумма квадратов до 5 вкл. = "
        << sum(3);
    return 0;
}
```

Результат работы программы:



Инициализация значений формальных параметров по умолчанию

Допускается инициализировать значения формальных параметров в функции. Это выполняется ***единожды*** перед `main()` (Таблица 36):

- если у функции существует прототип, то в прототипе;
- если функция полностью определяется (описывается) перед `main()` без прототипа, то в заголовке функции.

Таблица 36 – Варианты инициализации формальных параметров функции

<table>
<tr><th>Инициализация в прототипе</th><th>Инициализация в заголовке</th></tr>
<tr><td><pre>#include <iostream>
using namespace std;

//прототип функции
bool isPositive(int n = 10);

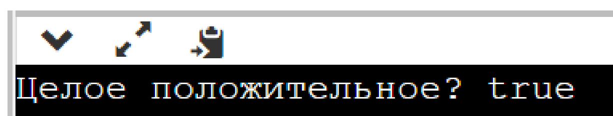
int main() {
    cout<<boolalpha;
    cout<<"Целое положительное? "
        //вызов без передачи знач.
        << isPositive();
    return 0;
}

//опеределение (описание) функции
bool isPositive(int n) {
    return 0 < n;
}</pre></td><td><pre>#include <iostream>
using namespace std;

//опеределение (описание) функции
bool isPositive(int n = 10) {
    return 0 < n;
}

int main() {
    cout<<boolalpha;
    cout<<"Целое положительное? "
        //вызов без передачи знач.
        << isPositive();
    return 0;
}</pre></td></tr>
</table>

Результат работы программы один и тот же в обоих случаях:



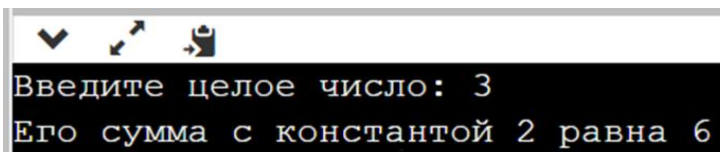
Допускается инициализация не всех параметров в функциях, имеющих несколько параметров.

Пример.

main.cpp
```
1  #include <iostream>
2  using namespace std;
3
4  int Sum(int a, int b = 1, int c = 1) {
5      return a + b + c;
6  }
7
8  int main() {
9      int a;
10     cout<< "Введите целое число: ";
```

```
11        cin>>a;
12        cout<<boolalpha;
13        cout<<"Его сумма с константой 2 равна "
14            << Sum(a, 2); //два фактических параметра
15        return 0;
16    }
```

Результат выполнения программы отличается от психологически ожидаемого, т.к. неявно в суммировании участвует еще третий параметр `c`, который равен `1`:



Использование квалификатора `const` в параметрах функций

Для того, чтобы сообщить компилятору, что при работе программы невозможно изменить значение формальных параметров функции, можно применить квалификатор **`const`**. Константному параметру можно передать в качестве аргумента, как константу, так и переменную.

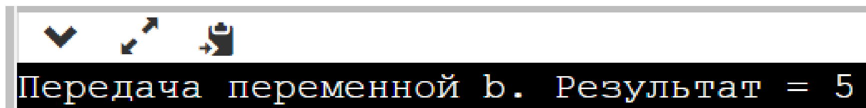
Пример.

main.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int incr(const int n) {
5       //инструкцию return n++; не пропустит
6       //компилятор, n - это константа
7       //однако в выражениях константа участв.
8       //может
9       return n + 1;
10  }
11
12  int main() {
13      const int A = 10;
14      int b = 4;
```

```
15      cout<<"Передача константы A. Результат = "
16          << incr(A)<< endl;
17      cout<<"Передача переменной b. Результат = "
18          << incr(b)<< endl;
19      return 0;
20  }
```

Результат работы программы:



Требования к синтаксису функций, создаваемых программистом

В языке C++ функции не могут быть вложенными. Т.е. тела функций – это всегда внешние блоки кода, как по отношению друг к другу, так и по отношению к функции `main()`.

Требования к ***именам функций***:

- рекомендуется писать имена функций английскими словами, а не транслитерацией;
- если имя состоит из нескольких слов, то первое слово со строчной буквы, а каждое следующее слово в имени следует начинать с заглавной буквы;
- не рекомендуется использовать цифры в названиях функций;
- если функция возвращает значение, то, обычно, ее имя должно отражать что она возвращает. Например: `getMaxValue();`
- если функция не возвращает значения, то ее имя должно отражать то, что она делает. Например: `setCoordinates();`
- имена функций, возвращающих логическое значение, должны означать, истинность какого именно утверждения проверяется в функции, например: `isRed()`, `hasLicense()`);
- все функции должны иметь прототипы.

Требования к оформлению ***тела*** функции

- как правило, рекомендуется, чтобы функция не занимала более одной страницы текстового редактора интегрированной среды;
- как правило, функция должна выполнять одну задачу, это не касается выполнения некоторых алгоритмов вычисления с точностью;
- если есть смысл, то необходимо выделять составные части функций в другие функции.

Требования к оформлению ***формальных параметров***:

- необходимо стремиться к тому, чтобы количество формальных (входных) параметров не превышало пяти;
- необходимо следить за тем, чтобы все входные параметры использовались;
- если есть необходимость использовать более чем пять входных параметров, а также в случае, если их типы с модификаторами и/или имена не помещаются в один разворот экрана по ширине, то их следует располагать в столбец с возможным дополнением построчными комментариями.

Элементы ООП в императивном программировании. Перегрузка функций

В программе возможно определение нескольких функций с одинаковым именем, но с разными типами и количеством ***формальных*** параметров. При этом на этапе компиляции выбирается соответствующая функция по типу значений формальных параметров и/или их количеству, определенных к моменту вызова функции.

Замечание.

Компилятор сообщит об ошибке, если функции с одинаковым именем будут различаться исключительно типом возвращаемого значения.

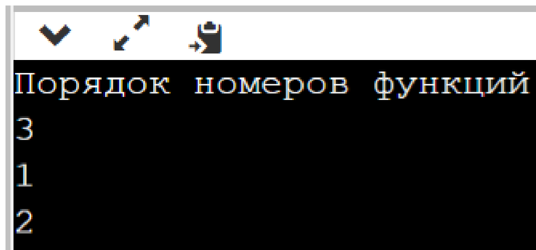
Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  //перегруженные функции
6  int flag(int a, int b)           {return 1;}
7  int flag(int a, int b, int c)    {return 2;}
8  double flag(double a, double b) {return 3;}
9
10 int main() {
11     int c =1, d= 2, e =3;
12     double x = 1., y = 2.;
```

```
13      std::cout<<"Порядок вызовов функций\n"
14              << flag(x, y)<<std::endl
15              << flag(c, d)<<std::endl
16              << flag(e, c, d)<<std::endl;
17      return 0;
18  }
```

Результат работы программы:



Назначение перегрузки – разрешить выполнять одно и то же либо различные действия с разными типами и наборами операндов.

Замечание.

Полиморфизм (один из базовых принципов ООП) в языках программирования и теории типов - способность функции обрабатывать данные разных типов. Таким образом, перегрузку функций можно рассматривать как пример полиморфизма (хотя и так называемого статического, т.к. коллизия разрешается еще на этапе компилирования).

Параметры функции main()

Функция `main()` имеет два ***необязательных предопределенных*** аргумента. Формат функции `main()` с необязательными параметрами:

```
int main(int argc, char *argv[]) {...}
```

где все имена формальных параметров `main()` предопределены. Аргумент `argc` типа `int` содержит в себе количество аргументов командной строки.

Аргумент `argv` – указатель на массив строк. Каждый элемент массива указывает на отдельный строковый аргумент командной строки. Один параметр отделяется от другого пробелами.

- `argv[0]` – полное имя запущенной программы;
- `argv[1]` – первая строка записанная после имени программы;
- `argv[2]` – вторая строка записанная после имени программы;

- `argv[argc-1]` – последняя строка записанная после имени программы;
- `argv[argc]` – `NULL`.

В средах разработки фирмы Borland (Borland Builder C++, Borland C++ и т.п.) предусмотрен еще и третий аргумент `env`, который, так же, как и `argv` является указателем на массив строк, но содержит установки среды.

Замечание.

Данные параметры можно использовать исключительно для запуска exe-файла консольной программы через командную строку для передачи текстовой информации в `main()`.

Стек вызовов функций и рекурсия

Память, которую используют программы, состоит из нескольких частей – сегментов. В частности, одним из основных является ***стек вызовов функций*** (или просто ***стек***) – участок оперативной памяти, где хранятся параметры функции, локальные переменные и другая информация, связанная с функциями.

Стек, как структура данных

Структура данных в программировании — это механизм организации данных для их эффективного использования.

Для разъяснения сущности работы структуры данных типа ***стек***, рассмотрим стопку тарелок на столе. Поскольку каждая тарелка тяжелая, а они еще и сложены друг на друге, то можно сделать лишь что-то одно из следующего:

- посмотреть на поверхность тарелки, которая находится на самом верху (вершина стека);
- взять верхнюю тарелку из стопки (обнажая таким образом следующую тарелку, которая находится под верхней, если она вообще существует);
- положить новую тарелку поверх стопки (спрятав под ней самую верхнюю тарелку, если она вообще была).

Таким образом ***стек*** (Рисунок 15) — это структура данных типа LIFO (англ. «Last In, First Out» = «Последним пришел, первым ушел»). Последний элемент, который находится на вершине ***стека***, первым и уйдет из него. Если положить новую тарелку поверх других тарелок, то именно эту тарелку

первой и возьмете. По мере того, как элементы помещаются в ***стек*** — ***стек*** растет по мере того, как элементы удаляются из ***стека*** — ***стек*** уменьшается.

Стек вызовов функций реализуется как одноименная структура данных типа ***стек***.

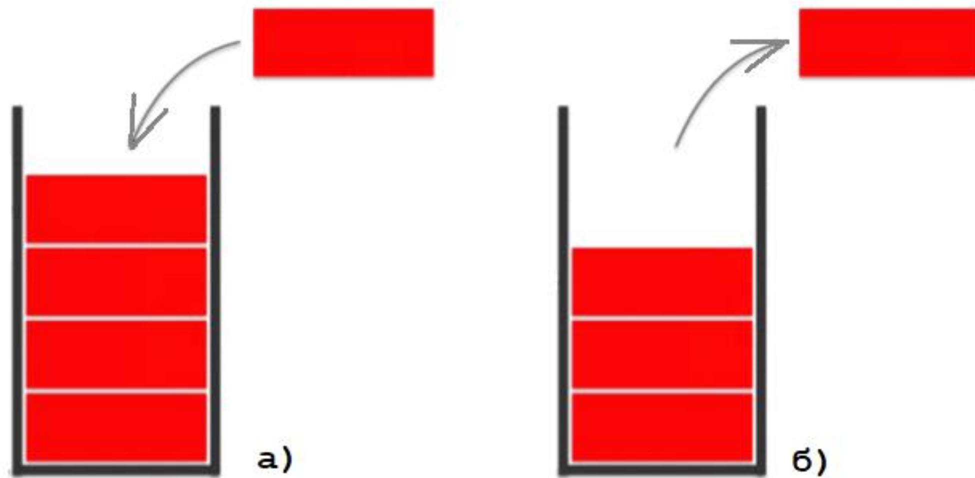


Рисунок 15 - Схема работы стека: а) добавление элемента стека; б) удаление элемента из стека

Стек вызовов функций

Стек вызовов функций — это область оперативной памяти, в которой выделяется память, при вызове каждой функции. Выделяемая при вызове функции память (графически это один красный прямоугольник, Рисунок 15) содержит все формальные параметры и локальные переменные, используемые в функции. Эта добавляемая при вызове функции или удаляемая при ее завершении память (красный прямоугольник, Рисунок 15) называется ***фреймом стека***.

Внутри ***фрейма*** при выполнении инструкций тела соответствующей функции, когда переменные пропадают из области видимости (например, заканчивается выполнение соответствующего блока тела функции), то они автоматически удаляются. Однако память под формальные параметры вызываемой функции удаляются из ***фрейма*** только после завершения ее выполнения.

После завершения работы функции и возвращения ею результата происходит удаление самого фрейма из ***стека***.

Так при запуске программы, фрейм функции `main()` помещается операционной системой в ***стек*** вызовов, и программа начинает свое выполнение.

Когда программа встречает вызов функции, то фрейм этой функции также помещается в ***стек*** вызовов, но уже над фреймом `main()`. При завершении выполнения функции, ее фрейм удаляется из ***стека*** вызовов.

Таким образом, просматривая функции, добавленные в ***стек***, можно видеть все функции, которые были вызваны до текущей точки выполнения.

Стек вызовов имеет фиксированное количество адресов памяти (фиксированный размер), что может привести к его переполнению.

Стек на практике

Пусть программа доходит до вызова определенной функции.

- создается ***фрейм стека***, он состоит из:
  - адреса инструкции, которая находится за вызовом функции (так называемый «обратный адрес»). Так процессор запоминает, куда ему возвращаться после выполнения функции;
  - аргументов функции;
  - памяти для локальных переменных;
  - и др. информации;
- процессор переходит к точке начала выполнения функции;
- инструкции внутри функции последовательно выполняются;

После завершения функции, выполняются следующие шаги:

- обрабатывается возвращаемое значение;
- ***фрейм стека*** удаляется, освобождается память, которая была выделена для всех локальных переменных и аргументов;
- регистры ***стека*** вызовов вновь открываются для записи;
- центральный процессор возобновляет выполнение кода, вызвавшего функцию.

Возвращаемые значения могут обрабатываться разными способами, в зависимости от архитектуры компьютера. Некоторые архитектуры считают возвращаемое значение частью фрейма ***стека***, другие используют регистры процессора.

Переполнение стека

Стек имеет ограниченный размер и, следовательно, может содержать только ограниченный объем информации. В операционной системе Windows размер ***стека*** по умолчанию составляет 1МБ. На некоторых Unix-системах этот размер может достигать и 8МБ. Если программа пытается поместить в ***стек*** слишком много информации, то это приведет к переполнению ***стека***. Переполнение ***стека*** (англ. «stack overflow») происходит, когда запрашиваемой памяти нет в наличии (вся память уже занята).

Переполнение ***стека*** является результатом добавления слишком большого количества переменных в ***стек*** и/или создания слишком большого

количества вложенных вызовов функций (например, при рекурсии). Переполнение ***стека*** обычно приводит к сбою в программе.

Стек имеет свои преимущества и недостатки:

- выделение памяти в ***стеке*** происходит сравнительно быстро;
- память, выделенная в ***стеке***, остается в области видимости до тех пор, пока находится в ***стеке***, она уничтожается при выходе из ***стека***;
- вся память, выделенная в ***стеке***, обрабатывается во время компиляции, следовательно, доступ к этой памяти осуществляется напрямую через имена переменных;
- поскольку размер ***стека*** является относительно небольшим, то не рекомендуется делать что-либо, что «съест» много памяти ***стека*** (например, передача по значению или создание локальных больших массивов или других затратных в смысле потребляемого объема памяти структур данных).

Рекурсия

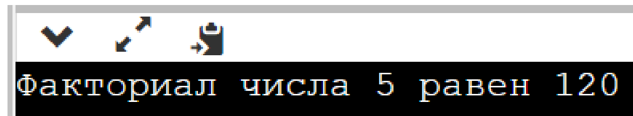
Любая функция в программе может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. Пример рекурсии – это вычисление факториала (`n!`).

Пример.

main.cpp

```
1  #include <iostream>
2
3  long factorial(int n) {
4      return n == 1 ? 1 : n * factorial(n - 1);
5  }
6
7  int main() {
8      std::cout<< "Факториал числа 5 равен "
9              << factorial(5);
10     return 0;
11 }
```

Результат работы программы:



Например, при вызове `factorial(5)` получится следующая цепь вызовов (Рисунок 16). При этом можно также рассмотреть схему заполнения ***стека*** соответствующими фреймами при организации рекурсии (Рисунок 17).

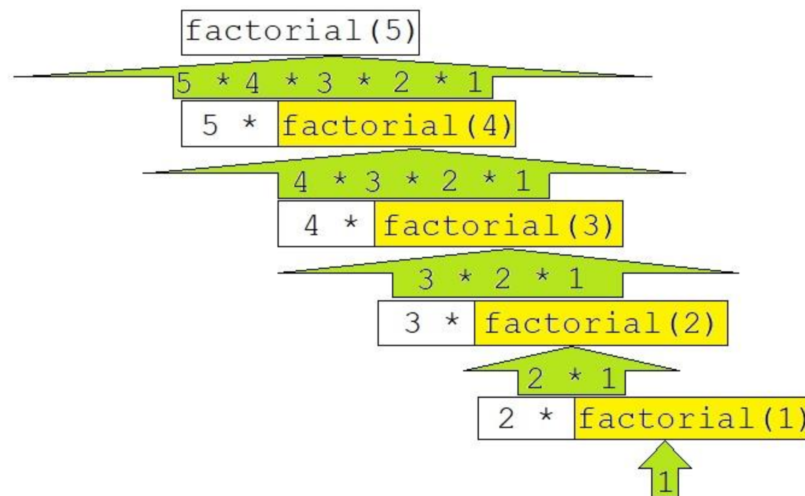


Рисунок 16 – Последовательность отложенных вызовов

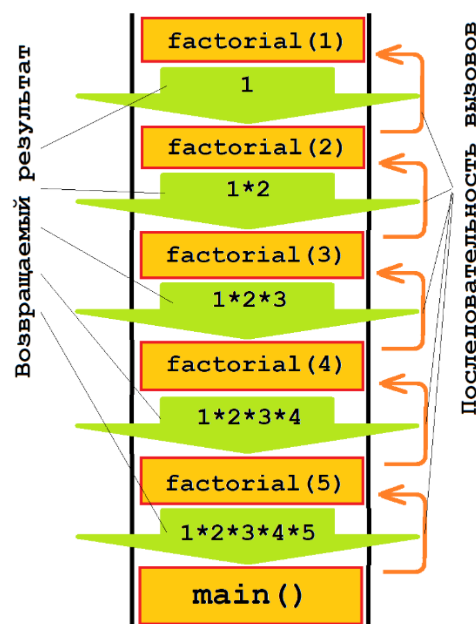


Рисунок 17 – Схема функционирования стека при рекурсивных вызовах

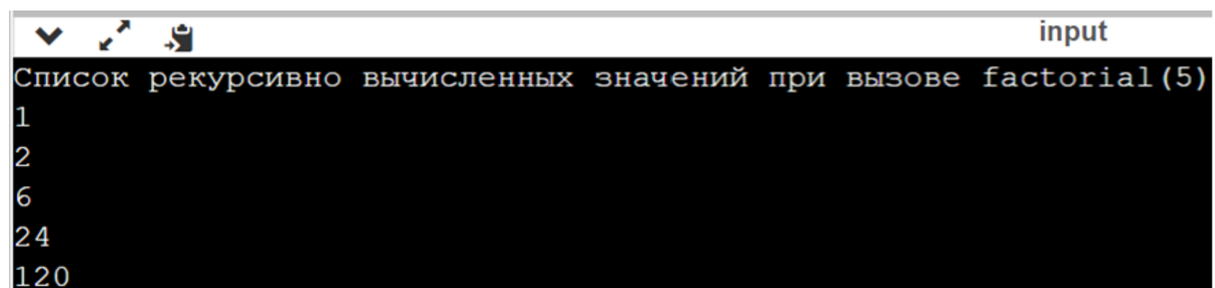
Для того чтобы продемонстрировать, что данная цепь вызовов действительно существует немного изменим код программы, вычисляющей факториал.

Пример.

main.cpp

```cpp
#include <iostream>

long factorial(int n) {
    if (n == 1) return 1;
    else {
        long a;
        std::cout<< (a = factorial(n - 1))
                    << std::endl;
        return n * a;
    }
}

int main() {
    std::cout<<"Список рекурсивно вычисленных "
                << "значений при вызове factorial(5)"
                <<std::endl << factorial(5);
    return 0;
}
```

После этого можно продемонстрировать последовательность возвращаемых результатов, полностью соответствующих рассмотренной схеме функционирования ***стека*** (Рисунок 17)



Рекурсивная функция обязательно должна иметь некоторый базовый вариант, который использует оператор `return` и к которому сходится выполнение остальных вызовов этой функции. В случае с факториалом базовый вариант представлен ситуацией, при которой `n` равно `1`. В этом

случае сработает инструкция `return 1;`, после которой начинается «сворачивание» вызовов и последовательное выполнение собственно вычислений.

Рекурсивные версии большинства алгоритмов могут выполняться несколько медленнее, чем их итеративные эквиваленты (выполняемые с помощью операторов цикла), поскольку к выполнению необходимых по алгоритму действий добавляются еще и время необходимое на обработку вызовов функций (выделение памяти под ожидаемый результат, присвоение ему значения и пр.).

На примере рекурсивного вычисления факториала еще раз напомним, что местом для хранения формальных параметров и локальных переменных функции является ***фрейм стека***. При этом при каждом новом вызове выделяется новая память под новый ***фрейм*** (Рисунок 17), соответственно если рекурсивных вызовов слишком много, то пространство ***стека*** может исчерпаться и возникнет ошибка - переполнение ***стека***.

Основным преимуществом применения рекурсивных функций является наглядность – многие алгоритмы выглядят в рекурсивном изложении проще, чем версии некоторых алгоритмов в итеративном представлении. Например, сортирующий алгоритм `Quicksort`.

Следует отметить, что практически любой циклический алгоритм можно представить в виде рекурсии (Таблица 37).

Таблица 37 - Суммирование целых чисел до заданного `n` (определение результата выражения `1 + 2 + … + n`)

<table>
<tr><th>Циклический алгоритм</th><th>Рекурсия</th></tr>
<tr><td><pre>int sum(int n) {
    int s = 0;
    for(int i = 1; i <= n; i++) {
        s = s + i;
    }
    return s;
}</pre></td><td><pre>int sum(int n) {
    if (n == 0) {
        return 0;
    }
    else {
        return n + sum(n - 1);
    }
}</pre></td></tr>
</table>

Создание исполняемого кода

Трансляция программы: компиляция и интерпретация

Большая часть работы программистов связана с написанием исходного кода, тестированием и отладкой программ на одном из языков программирования. Различные языки программирования поддерживают

различные стили программирования. Единственный язык, напрямую выполняемый процессором - это называемый машинный код. Изначально все программисты создавали программы в машинном коде, но сейчас эта трудная работа уже не делается. Вместо этого программисты пишут исходный код на языке программирования высокого уровня, и компьютер (используя компилятор или интерпретатор) транслирует его, в один или несколько этапов, уточняя все детали, в машинный код, готовый к исполнению на процессоре. Запись исходных текстов программ при помощи языков программирования облегчает понимание и редактирование человеком.

Трансляция программы - преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определенном смысле, равносильную первой. При трансляции выполняется перевод программы, понятной человеку, на язык, понятный компьютеру. Это выполняется специальными программными средствами - ***трансляторами***.

Трансляторы реализуются в виде ***компиляторов*** или ***интерпретаторов***. С точки зрения выполнения работы ***компилятор*** и ***интерпретатор*** существенно различаются. Если цель ***трансляции*** - преобразование ***всего*** исходного текста во внутренний язык компьютера, то такая ***трансляция*** называется также ***компиляцией***. Исходный текст называется также исходной программой или исходным модулем (кодом), а результат компиляции - ***объектным кодом*** или ***объектным модулем***.

Если же трансляции подвергаются отдельные операторы исходных текстов и при этом полученные коды сразу выполняются, такая трансляция называется ***интерпретацией***.

Таким образом можно дать следующие определения:

Компиляция - преобразование программой-компилятором исходного текста программы, написанного на языке высокого уровня либо в машинный язык, либо в язык, близкий к машинному, либо в объектный модуль. Обычно результатом компиляции является объектный файл с необходимыми внешними ссылками для компоновщика.

Интерпретация - процесс непосредственного покомандного (построчного) выполнения программы «на ходу» без предварительной компиляции.

Этапы создания исполняемого кода программ на языке С++

Объединенная единым алгоритмом совокупность описаний и операторов образует программу на алгоритмическом языке. Процесс перевода программы в машинный код состоит из нескольких этапов. Рисунок 18 иллюстрирует эти этапы для языка С++.

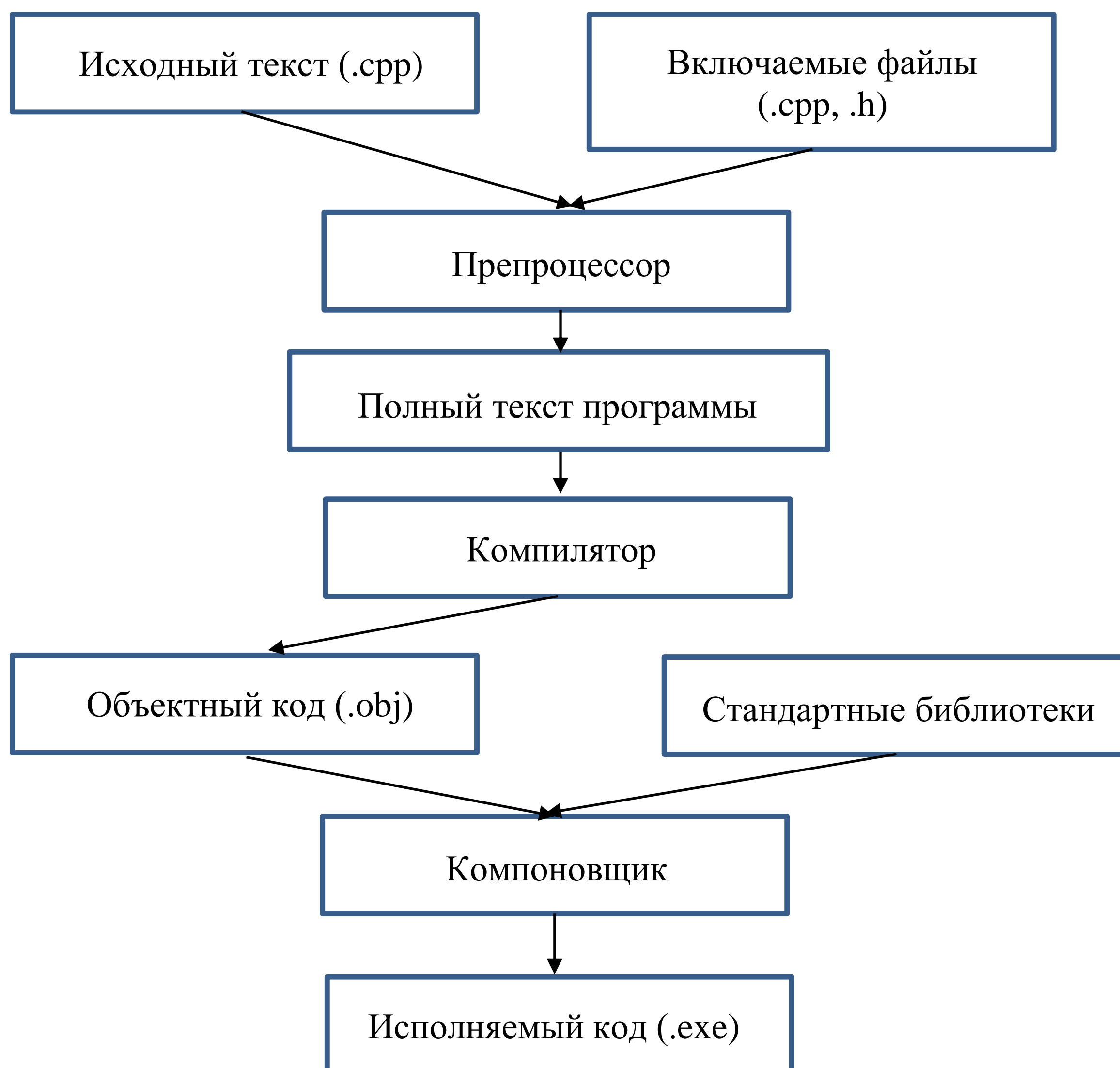


Рисунок 18 - Этапы создания исполняемого кода

Сначала исходная программа (текстовый файл с расширением `.cpp`) передается ***препроцессору***, который выполняет ***директивы***, содержащиеся в ее тексте (например, включение в текст так называемых ***заголовочных файлов*** — текстовых файлов, в которых содержатся описания используемых в программе элементов, в частности, это могут быть файлы с расширением `.h`).

Получившийся полный текст программы поступает на вход компилятора, который выделяет лексемы, а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом ***компилятор*** выявляет ***синтаксические ошибки*** и в случае их отсутствия строит объектный модуль.

Компоновщик (или ***редактор связей***), формирует ***исполняемый модуль*** программы, подключая к ***объектному модулю*** другие объектные модули, в том числе библиотеки, содержащие функции, обращение к которым осуществляется в программе. Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на

этапе компоновки. ***Исполняемый модуль*** имеет расширение `.exe` и он может быть запущен на выполнение.

Здесь следует подчеркнуть одну важную особенность. Хотя в ***заголовочных файлах*** содержится описание всех стандартных функций, в исполняемый код программы включаются только те функции, которые используются в программе. Выбор «нужных» функций осуществляет ***компоновщик*** на этапе «редактирования связей».

Таким образом, создаются ***исполняемые*** программы на C++. Конечно, это очень общее описание этого сложного процесса, но четко передает смысл всех этапов работы компилятора и компоновщика.

Средства управления препроцессорной обработкой программы

Как уже указывалось ранее каждая программа на языке C++ есть последовательность препроцессорных директив, описаний и определений глобальных объектов и функций. Препроцессорные директивы управляют созданием полного текста программы.

Задача препроцессора – преобразование текста программы до начала ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора.

Директивами препроцессора являются строки, начинающиеся с символа '#', за которым следует идентификатор, называемый именем директивы. Такие строки, воспринимаются препроцессором как команды, определяющие его действия по формированию и преобразованию текста программы. Разрешается использование пробелов перед и после символа '#'. Препроцессор на этапе обработки им программы «сканирует» исходный текст программы начиная с первого символа #.

Существует строгий набор директив и невозможно определить новые директивы.

Некоторые директивы требуют наличия аргументов, которыми является оставшаяся часть строки, отделенная от имени директивы одним или несколькими пробелами.

Обычно, директива препроцессора не может занимать более одной строки. Хотя, в некоторых случаях она может быть разбита на несколько строк с помощью последовательности `backslash-newline` (знак `'\'`). Если внутри директивы записаны комментарии, то они заменяются пробелами.

После окончания препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы. На этом этапе программа представляет собой набор описаний и определений, т.е. текст с полным

набором явных описаний библиотечных определений и выполненных макроподстановок.

Подключаемые файлы

Подключаемый файл — это файл, содержащий определения функций и переменных, а также макроопределения вместе с некоторыми исходными файлами.

Использование подключаемых файлов

Подключаемые файлы используются для двух целей:

- системные подключаемые файлы используются для определения интерфейсов к компонентам операционной системы (они подключаются для предоставления объявлений и определений, требуемых для работы с системными вызовами и библиотеками);
- подключаемые файлы разработчика содержат определения для интерфейсов между исходными файлами программы.

Включение подключаемого файла в программу дает такой же результат, как при копировании этого файла в каждый исходный файл этой программы.

Заголовочные файлы

Файлы `.cpp` не являются единственными файлами в проектах. Есть еще один тип файлов — ***заголовочные файлы*** (или «заголовки»), которые имеют расширение `.h`.

Термин «***заголовочный файл***» (`header file`) в применении к файлам, содержащим описания библиотечных функций, констант и операций, не случаен. Он предполагает включение этих файлов в начале программы, до использования в тексте программы библиотечных определений.

Хотя ***заголовочный файл*** может быть включен и не вначале программы, а непосредственно перед использованием к необходимому библиотечному определению, такое подключение не рекомендуется.

Целью ***заголовочных файлов*** является удобное хранение набора объявлений объектов для их последующего использования в других программах.

Далее приведен список системных ***заголовочных файлов***, используемых в языке C++:

- cassert - эта библиотека содержит только макрос assert, применяемый для проверки диагностических утверждений (будет рассмотрен позже);
- cctype - большинство функций этой библиотеки позволяет распознавать буквы, цифры и т.д.; остальные функции преобразуют строчные буквы в прописные, и наоборот; функции, предназначенные для распознавания, возвращают значение true, если символ ch принадлежит указанной группе; в противном случае они возвращают значение false;
- cfloat - в этой библиотеке определены именованные константы, указывающие диапазон изменения значений с плавающей точкой;
- climits - в этой библиотеке определены именованные константы, указывающие диапазон изменения целочисленных значений;
- cmath - функции, содержащиеся в этой библиотеке, предназначены для стандартных математических вычислений. Эти функции являются перегруженными и выполняют вычисления с числами, имеющими тип float, double и long double; если не указано иное, каждая функция имеет один аргумент, а возвращаемое значение и аргумент имеют одинаковый тип (float, double и long double);
- cstdlib - содержит в себе функции, занимающиеся выделением памяти, контролем процесса выполнения программы, преобразованием типов и другие;
- cstring - библиотека содержит функции, позволяющие манипулировать строками базового типа (одномерные массивы символов), завершающимися нулевым символом '\0'. Если не указано иное, функции возвращают указатель на результирующую строку, модифицируя один из аргументов;
- fstream - в данной библиотеке объявлены классы, поддерживающие ввод и вывод;
- iomanip - манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода;
- iostream - манипуляторы, содержащиеся в этой библиотеке, влияют на формат ввода и вывода (обратите внимание, что в библиотеке iomanip есть дополнительные манипуляторы);
- string - эта библиотека позволяет манипулировать со строками языка C++; к строкам можно применять операторы =, +, ==, !=, <, < =, >, >=, << и >>.

Директива #include

Для использования в программе подключаемых файлов применяется директива препроцессора `#include`.

Простейший пример работы директивы `#include`

Директива `#include` указывает препроцессору вставить указанный файл перед компиляцией. Пусть, дан следующий подключаемый файл `header.h`, содержит один прототип.

Пример.

main.cpp | header.h

```
1 char* test();
```

Кроме того, дана основная программа с именем `main.cpp` использующая этот файл.

Пример.

main.cpp | header.h

```
1 #include "header.h"
2
3 int main () {
4     printf (test ());
5     return 0;
6 }
```

После обработки директив препроцессора программа будет выглядеть следующим образом:

Пример.

main.cpp | header.h

```
1 char* test();
2
3 int main () {
4     printf (test ());
5     return 0;
6 }
```

Для подключаемых файлов нет ограничений на объявления и макроопределения. Любой фрагмент программы может быть включен в другой файл. Подключаемый файл может даже содержать начало выражения, заканчивающееся в исходном файле или окончание выражения, начало которого находится в исходном файле.

Однако комментарии и строковые константы ***не*** могут начинаться в подключаемом файле и продолжаться в исходном файле. Не завершенный комментарий, строковая или символьная константа в подключаемом файле приводят к возникновению ошибки в конце файла.

Важно понимать, что употребление в программе препроцессорной директивы `#include<имяЗаголовочногоФайла>` не подключает к программе коды функций, описанных в заголовочном файле, она позволяет только дополнить текст программы текстом определения функций из соответствующего ***заголовочного файла*** (дополняет прототипами). Подключение к программе кодов ***только используемых*** библиотечных функций осуществляется позже после компиляции, на этапе редактирования связей (компоновки, Рисунок 18).

Форматы использования директивы `#include`

Как файлы разработчика, так и системные файлы включаются в программу с использованием директивы препроцессора `#include`. Она имеет два формата:

```
#include <имяПодключаемогоФайла>
#include "путьИИмяПодключаемогоФайла"
```

где **путьИИмяПодключаемогоФайла** — это имя файла, которому при необходимости может предшествовать спецификация каталога. Имя файла должно указывать на существующий файл. Синтаксис инструкции **путьИИмяПодключаемогоФайла** зависит от операционной системы, в которой компилируется программа.

Обе синтаксические формы приводят к замене директивы `#include` всем содержимым указанного файла. Различие между двумя формами — это порядок путей, которые препроцессор ищет при неполном указании пути. В приведенной ниже таблице (Таблица 38) показывается различие между этими формами синтаксиса.

Таблица 38 – Различие, обусловленные синтаксисом директивы `include`

Форма синтаксиса	Действие
Форма с угловыми скобками	Эта модификация используется для подключения системных файлов. При ее выполнении производится поиск файла с именем **`имяПодключаемогоФайла`** в стандартном списке системных каталогов.
Форма в кавычках	Эта модификация применяется для подключаемых файлов разработчика. Сначала файл **`путьИИмяПодключаемогоФайла`** просматривается в текущем каталоге, а затем в каталогах для системных подключаемых файлов. Текущим каталогом является каталог хранения текущего обрабатываемого файла.

Символы `backslash` (`'\'`) интерпретируются как отдельные символы, а не начало `Esc`-последовательности. Таким образом, директива:

```
#include "x\n\\y"
```

указывает имя файла, содержащего три символа `backslash`.

Рассмотрим еще один вариант записи директивы:

```
#include ANYTHING_ELSE
```

Эта модификация называется «вычисляемой директивой `#include`». Любая директива `#include`, не соответствующая ни одной из модификаций, рассмотренных выше, является вычисляемой директивой. Строка `ANYTHING_ELSE` проверяется на наличие соответствующего макроса, значение которого затем заменяет его название. Полученная в результате строка должна уже в точности соответствовать одной из рассмотренных выше модификаций (то есть имя подключаемого файла должно быть заключено в кавычки или угловые скобки).

Эта возможность позволяет определять макросы, что дает возможность изменять имена подключаемых файлов. Она, например, может использоваться при переносе программ с одной операционной системы на другие.

Почему любой из стандартных заголовочных файлов при подключении пишется без окончания «.h»

Когда C++ только создавался, все файлы библиотеки `Runtime` имели окончание `.h`. Оригинальные версии `cout` и `cin` объявлены в файле с именем `iostream`. При стандартизации языка C++ комитетом ANSI, решили перенести все функции из библиотеки `Runtime` в пространство имен `std`, чтобы предотвратить возможность возникновения конфликтов имен с идентификаторами разработчика. Тем не менее, возникла проблема: если все функции переместить в пространство имен `std`, то старые программы перестали бы работать.

Для обеспечения обратной совместимости ввели новый набор заголовочных файлов с теми же именами, но без расширения «`.h`». Весь их функционал находится в пространстве имен `std`. Таким образом, старые программы с `#include <iostream.h>` не нужно было переписывать, а новые программы уже могли использовать `#include <iostream>`.

Замечание.

Когда разработчик подключает заголовочный файл из стандартной библиотеки C++, то он должен убедиться, что использует версию без `.h` (если она существует). В противном случае, он будет использовать устаревшую версию заголовочного файла, который уже больше не поддерживается.

Кроме того, многие библиотеки, унаследованные от языка C, которые до сих пор используются в C++, также были продублированы с добавлением префикса «c» (например, `stdlib.h` стал `cstdlib`). Функционал этих библиотек также перенесли в пространство имен `std`, чтобы избежать возможность возникновения конфликтов имен с идентификаторами разработчика.

Дополнительные сведения о директиве #define

Директиву `#define` можно использовать для создания макросов. Макрос — это правило, которое определяет конвертацию идентификатора в указанные данные.

Есть два основных типа макросов:

- макрозамена литералов;
- макросы (или макрозамена) с параметрами.

Ранее уже был рассмотрен пример ***макрозамены*** (***макроподстановки***) литералов (см. параграф «Простейший вид макроподстановки»). В

настоящем пункте дополнительно рассмотрим макрозамену ***с параметрами***. Формат:

```
#define ИМЯ(список, параметров) замещаемоеВыражение
```

Пример.

```
#define MAX(a, b) a >= b ? a : b
```

Работает также как и ранее: препроцессор везде, где встречает конструкцию `ИМЯ(список, параметров)`, заменяет ее на `замещаемоеВыражение` (с подстановкой соответствующих переменных).

С помощью `#define` можно осуществить замену вызова функции, оператора или блока операторов.

Параметр макроса:

- можно превратить в строку, добавив перед ним знак `'#'`.
- с помощью '##' можно приклеить дополнительную секцию к общей части имен группы идентификаторов, чтобы получить возможность обращаться к выбранной группе исключительно по несовпадающей части.

Пример.

main.cpp

```
1  #include <iostream>
2  //макрозамена блока операторов перестановки
3  #define SWAP(type, a, b) {type tmp = a; a = b; b = tmp;}
4  //макрозамена функции printf(), где значение #x - строка-им
5  #define PRINT(x) printf("Значение %s = %f\n", #x, x)
6  //склеивание части имени перем. value_ с вариативной частью
7  #define PRINT_VALUE(number) printf("%d", value_##number);
8
9  int main() {
10     double x = 27.3, y = 1.55;
11     SWAP(double, x, y)
12     PRINT(x);
13
14     int value_one = 10, value_two = 20;
15     PRINT_VALUE(one)
16     return 0;
17 }
```

Результат работы программы:

```
Значение x = 1.550000
10
```

К вопросу об области видимости директивы #define

О макросах важно понимать, что область видимости у них такая же, как у функций, то есть если макрос определен в отдельном заголовочном файле, то он виден везде, где этот файл был подключен директивой `#include`.

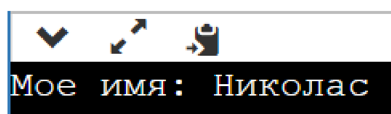
Директивы препроцессора выполняются перед компиляцией программы (проекта): сверху вниз, файл за файлом.

Пример.

main.cpp

```
1  #include <iostream>
2
3  void function() {
4      #define MY_NAME "Николас"
5  }
6
7  int main() {
8      std::cout << "Мое имя: " << MY_NAME;
9      return 0;
10 }
```

Результат выполнения программы:



Несмотря на то, что директива `#define MY_NAME "Николас"` определена внутри функции `function()`, препроцессор этого не заметит, так как он ***не*** обрабатывает функции и ***не*** разделяет текст на модули. Следовательно, выполнение этой программы будет идентично той, в которой бы `#define MY_NAME "Николас"` было определено ***до***, либо сразу ***после*** функции `function()`.

Замечание.

Если использование макрозамен обосновано требованиями разработки проекта, то для лучшей читаемости кода следует ***всегда*** *определять макрозамены с помощью* `#define` ***вне*** *функций.*

После того, как препроцессор завершит свое выполнение, все идентификаторы (определенные с помощью `#define`) из этого файла исключаются. Это означает, что директивы действительны только со строчки определения и до конца файла, в котором они определены и только в той последовательности, в которой определены.

Пример.

main.cpp

```
 1  #include <iostream>
 2
 3  void doSomething() {
 4      #ifdef PRINT
 5          std::cout << "Печатаем!";
 6      #endif
 7      #ifndef PRINT
 8          std::cout << "Не печатаем!";
 9      #endif
10  }
11
12  int main() {
13      #define PRINT
14      doSomething();
15      return 0;
16  }
```

Результат выполнения программы:

Несмотря на то, что `PRINT` объявлен в `main.cpp` (`#define PRINT`), это все равно не имеет никакого влияния на директивы условной компиляции, объявленные выше. Поэтому, при выполнении функции `doSomething()`, на экран выводится «`Не печатаем!`».

Директива #undef

Объявление любой макрозамены можно отменить. Формат отмены действия любого макроопределения:

`#undef` **ИДЕНТИФИКАТОР**

где ИДЕНТИФИКАТОР – это имя макрозамены литерала или имя макрозамены с параметрами.

Пример.

```
#undef PI
```

После этой строчки обращаться к `PI` будет уже нельзя.

Замечание.
Использование любых макрозамен (с параметрами или без них) считается плохой практикой в программировании на C++.

Директивы условной компиляции

Директивы условной компиляции позволяют определить, при каких условиях код будет компилироваться, а при каких — нет. Рассмотрим только следующие директивы условной компиляции:

`#ifdef` `#ifndef` `#endif`

Директива `#ifdef` (сокр. от англ. «if defined» = «если определено») позволяет препроцессору проверить, было ли значение ранее определено с помощью директивы `#define`. Если да, то код между `#ifdef` и `#endif` скомпилируется. Если нет, то код будет проигнорирован.

Пример.

main.cpp

```
1 #include <iostream>
2 #define PRINT_JOE
3
4 int main() {
5     #ifdef PRINT_JOE
6         std::cout << "Joe" << std::endl;
7     #endif
8
9     #ifdef PRINT_BOB
```

```
10          std::cout << "Bob" << std::endl;
11      #endif
12
13      return 0;
14  }
```

Результат работы программы:

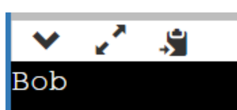
Поскольку макрос `PRINT_JOE` уже был определен, то строка `std::cout << "Joe" << std::endl;` скомпилируется и выполнится. Однако поскольку `PRINT_BOB` не был определен, то строка `std::cout << "Bob" << std::endl;` не будет откомпилирована и, следовательно, не выполнится.

Директива `#ifndef` (сокр. от «if not defined» = «если не определено») — это полная противоположность к `#ifdef`, которая позволяет проверить, не было ли значение ранее определено.

Пример.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      #ifndef PRINT_BOB
5          std::cout << "Bob" << std::endl;
6      #endif
7
8      return 0;
9  }
```

Результат работы программы:



Полученный результат объясняется следующим образом: так как `PRINT_BOB` ранее никогда не был определен, то на экран выводится

сообщение "Bob". Условная компиляция очень часто используется в качестве header guards.

Проблема дублирования объявлений

Как известно идентификатор может иметь только одно объявление. Таким образом, при компиляции программы с двумя объявлениями одной и той же переменной возникнет ошибка.

Поскольку заголовочный файл может кроме прототипов содержать объявления глобальных переменных, констант абстрактных типов (структур и классов), то довольно легко можно попасть в ситуацию, когда одни и те же заголовочные файлы будут подключаться больше одного раза в файл .cpp. Это обычно случается при подключении одного заголовочного файла другим заголовочным файлом с последующим подключением обоих заголовочных файлов в cpp-файл .

Пример.

```
//заголовочный файл first.h
```

```
main.cpp | first.h | second.h
1  const int a = 1;
```

```
//заголовочный файл second.h
```

```
main.cpp | first.h | second.h
1  #include "first.h"
```

```
// main.cpp точка входа в многофайловый проект
```

```
main.cpp | first.h | second.h
1  #include "first.h"
2  #include "second.h"
3
4  int main() {
5      return 0;
6  }
```

Эта, казалось бы, очевидная программа, не скомпилируется. Проблема кроется в определении константы в файле first.h:

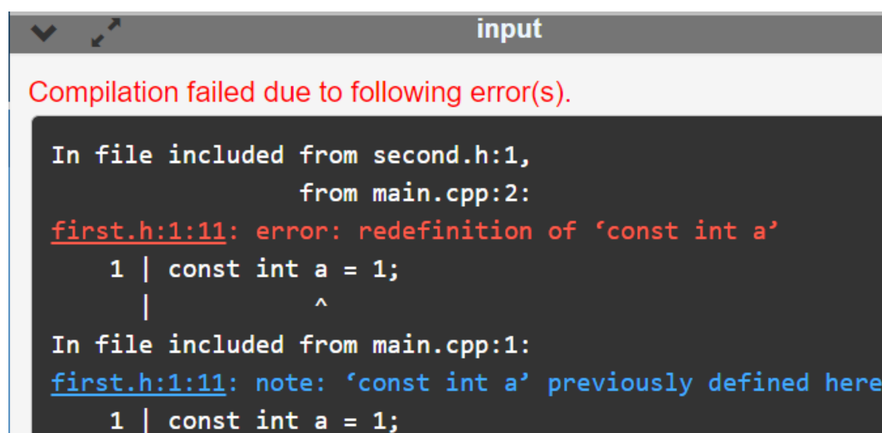
- сначала main.cpp подключает заголовочный файл first.h, вследствие чего определение константы a копируется в main.cpp;

- затем main.cpp подключает заголовочный файл second.h, который, в свою очередь, подключает first.h;
- в second.h находится копия определения константы, которая уже во второй раз копируется в main.cpp.

Таким образом, после выполнения всех директив #include, main.cpp будет выглядеть следующим образом:

```
const int a = 1;
const int a = 1;

int main() {
    return 0;
}
```

Таким образом, получается дублирование определений констант и выдача соответствующего сообщения компилятором:



```
input
Compilation failed due to following error(s).

In file included from second.h:1,
                 from main.cpp:2:
first.h:1:11: error: redefinition of ‘const int a’
    1 | const int a = 1;
      |           ^
In file included from main.cpp:1:
first.h:1:11: note: ‘const int a’ previously defined here
    1 | const int a = 1;
```

Header guards

Как видно из предыдущего примера следует избегать многократного подключения файлов. На самом деле существует простое решение — использование header guards (защиту подключения). Header guards — это директивы условной компиляции, которые состоят из следующего:

```
#ifndef ИМЯ_МАКРОСА
#define ИМЯ_МАКРОСА

    //Основная часть кода

#endif // ИМЯ_МАКРОСА
```

Имя макроса ИМЯ_МАКРОСА указывает на то, что файл уже однажды включался. Если подключить этот заголовочный файл, то первое, что он сделает — это проверит, был ли ранее определен идентификатор ИМЯ_МАКРОСА. Когда этот заголовок подключается впервые, то ИМЯ_МАКРОСА еще не был определен, будет компилироваться и выполняется основная часть заголовочного файла. Если же раньше этот заголовочный файл подключался, то макрос ИМЯ_МАКРОСА уже был определен. В таком случае, при подключении этого заголовочного файла во второй раз, его содержимое будет проигнорировано.

Замечание.

В подключаемых заголовочных файлах разработчика имена функций и макросов не должен начинаться с одинарного «_» или с двойного «__» символов подчеркивания во избежание возникновения конфликтов имен с именами функций и макросов подключаемых системных файлов.

Все заголовочные файлы должны иметь header guards. ИМЯ_МАКРОСА может быть любым идентификатором, но, как правило, в качестве идентификатора используется имя заголовочного файла с окончанием _H. Например, в файле first.h следует использовать идентификатор FIRST_H.

Возвращаясь, к примеру с двукратным определением константы a, исправим ситуацию с помощью применения header guards в заголовочном файле first.h.

Пример.

main.cpp | first.h | second.h

```
1 #ifndef FIRST_H
2 #define FIRST_H
3
4 const int a = 1;
5
6 #endif
```

Теперь, при подключении в main.cpp заголовочного файла first.h, препроцессор увидит, что макрос FIRST_H еще не был определен, следовательно, выполнится директива определения FIRST_H и содержимое first.h скопируется в main.cpp. Затем main.cpp подключает заголовочный файл second.h, который, в свою очередь, подключает first.h. Препроцессор видит, что FIRST_H уже был ранее определен и

часть содержимого second.h, касающаяся first.h уже не будет повторно вставлена в main.cpp.

Директива #pragma once

Существует также специальная директива, указывающая препроцессору, что файл должен быть включен не более одного раза. Эта директива называется #pragma once. Ранее она использовалась в дополнение к директиве #ifndef. Однако в настоящее время она ***устарела***.

В C++ существует модификация директивы #include, называемая #import, которая используется для включения файла не более одного раза. При использовании директивы #import вместо #include не требуется наличия условных оборотов для предотвращения многократной обработки файла.

Указатели и ссылки

Указатели

Указатель – это символическое представление адреса переменной, объекта или массива в памяти компьютера.

Для определения указателя надо указать тип объекта, адрес которого будет хранить указатель, символ звездочки * и имя указателя (свободно выбираемый идентификатор). Таким образом формат объявления указателя имеет вид:

```
~~модификатор~~ тип *имяУказателя;
```

где ~~модификатор~~ тип – модификатор и тип переменной, на которую будет указывать указатель (модификатор и тип могут быть любыми из базовых, а кроме того, тип также может быть классом созданным разработчиком), имяУказателя – свободно выбираемый идентификатор. Определим указатель pointer на значение типа int.

Пример.

```
int *pointer;
```

Пока указатель `pointer` не ссылается (или не указывает) ни на какую переменную. Далее присвоим указателю адрес переменной.

Пример.

```
int *pointer;    //объявляем указатель
int a = 10;      //определяем переменную
pointer = &a;    //указатель получ. адрес переменной
```

Для получения адреса переменной применяется операция **&** (операция взятия адреса). Важно подчеркнуть еще раз, что переменная `a` имеет тип `int`, и указатель, который указывает на ее адрес, тоже имеет тип `int`. Таким образом, между указателем и переменной должно быть соответствие по типу (с учетом модификаторов).

При выводе адреса переменной на консоль, можно увидеть, что он представляет шестнадцатеричное значение.

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      int *pointer;    // определяем указатель
5      int a = 10;      // определяем переменную
6      pointer = &a;    //указатель получает адрес a
7
8      std::cout << " pointer = " << pointer
9                << std::endl;
10     std::cout << " размер в байтах переменной a = "
11               << sizeof(int);
12     return 0;
13 }
```

Результат работы программы:

```
pointer = 0x7ffc9217e81c
размер в байтах переменной a = 4
```

Замечание.

В каждом при каждом запуске программы на исполнение в онлайн интегрированной среде адрес будет отличаться.

К примеру, в данном случае при использовании онлайн интегрированной среды `OnlineGDB beta` машинный адрес переменной `a` - это `0x7ffc5110c954`. То есть в памяти компьютера есть байт с адресом `0x7ffc5110c954`, по которому располагается переменная `a`. Переменная `a` представляет тип `int`, а, следовательно, на большинстве архитектур будет занимать последовательно выделенные `4` байта (Рисунок 19).

Поскольку, как уже известно, адресом ячейки памяти выделенной под хранение переменной `a` является адрес первого выделенного байта, то и указатель `pointer` имеет значение `0x7ffc5110c954`.

Таким образом, переменная `a` типа `int` последовательно займет ячейки памяти с адресами `0x7ffc5110c954`, `0x7ffc5110c955`, `0x7ffc5110c956`, `0x7ffc5110c957` (Рисунок 19).

0x7ffc5110c954	0x7ffc5110c955	0x7ffc5110c956	0x7ffc5110c957	Адреса байт памяти, выделенных под хранение значения переменной
первый байт	второй байт	третий байт	четвертый байт	
10				Значение переменной
a				Имя переменной

Рисунок 19 – Размещение в памяти переменной a

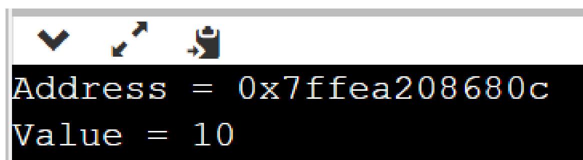
Кроме того, существует возможность получить хранящееся по этому адресу значение, то есть значение переменной `a`. Для этого применяется операция ***** (операция разыменования). Она применяется исключительно к указателям и результатом применения этой операции всегда является значение переменной, на которую указывает указатель.

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int a;
    int *pointer = &a; //инициализация указателя
            //адресом уже объявленной переменной
    a = 10; //присвоение значения после взятия адреса
    std::cout << "Address = " << pointer
              << std::endl;
    std::cout << "Value = " << *pointer
              << std::endl;
    return 0;
    return 0;
}
```

Результат работы программы:



Значение, которое получено в результате операции разыменования, можно присвоить другой переменной.

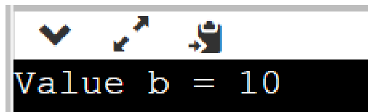
Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    int *pointer = &a;
    //инициализация новой переменной,
    //значением, расположенным по адресу
```

```
9       int b = *pointer;
10      cout << "Value b = " << b << endl;
11      return 0;
12  }
```

Результат работы программы:



```
Value b = 10
```

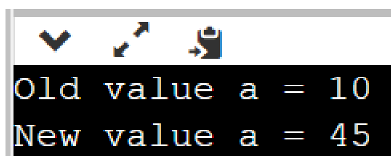
Также, используя указатель, можно менять значение переменной по адресу, который хранится в указателе.

Пример.

main.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       int a = 10;
6       cout << "Old value a = " << a << endl;
7       int *pointer = &a;
8       *pointer = 45;
9       cout << "New value a = " << a << endl;
10      return 0;
11  }
```

Результат работы программы:



```
Old value a = 10
New value a = 45
```

Объясним полученные результаты. Так как по адресу, на который указывает указатель `pointer`, располагается переменная `a`, то соответственно ее значение изменится при присваивании числа `45` разыменованному указателю `pointer` (выражение `*pointer`).

Таким образом, следует сделать следующий вывод если `pointer` это указатель определенного типа со значением адреса, то выражение `*pointer`

является обычной полноценной переменной, значение которой может участвовать, как арифметических операциях и выражениях, так и в вызовах функций.

Замечания:

- *рекомендуется имена переменных-указателей начинать с префикса, а именно строчной буквы* `p` *(сокращение от* `pointer`*), чтобы было ясно, что это указатель и требуется соответствующая обработка;*
- *этой рекомендации придерживаются достаточно редко, в основном в учебных пособиях.*

При объявлении нескольких указателей, звездочка должна находиться возле каждого идентификатора указателя.

Пример.

```
int *ptr3, ptr4; //ptr3 - это указатель на
                 //значение типа int, а ptr4 -
                 //это переменная типа int
```

Следующие выражения является ***не***допустимыми:

```
int *ptr = 7;
double *ptrD = 0x0012FF7C;
```

Это связано с тем, что указатели могут содержать только адреса. В обоих случаях компилятор воспринимает выражения, написанные после знака присваивания как целочисленные константы (хотя в втором случае и она является шестнадцатеричной) и сообщит, что он не может преобразовать целочисленное значение в указатель (т.е. приведения целочисленного значения к типу адреса невозможно).

Особый тип указателя void

Указатель типа `void` указывает на место в оперативной памяти и не содержит информации о типе объекта, т.е. он указывает не на сам объект, а на его возможное расположение. С указателями данного типа могут использоваться только операция простого присваивания и операции сравнения (см. ниже).

Нулевые указатели

Нулевой указатель (null pointer) — это указатель, который не указывает ни на какой объект. Он применятся в качестве присваиваемого указателю значения в том случае, если разработчик не желает, чтобы указатель указывал на какой-то реально существующий адрес.

Для присвоения переменной-указателю нулевого указателя можно применять различные синтаксические конструкции.

Формат использования:

```
модификатор тип *имяУказателя = nullptr;
модификатор тип *имяУказателя = 0;
модификатор тип *имяУказателя(0);
модификатор тип *имяУказателя{0};
```

Замечание.

Следует сразу инициализировать указатели нулевым значением, даже если в дальнейшем собираетесь присваивать им иные значения.

Макрос NULL

Язык C++ унаследовал от языка C специальный макрос препроцессора с именем NULL, который определен как значение 0. Подчеркнем, что для использования этого макроса никаких заголовочных файлов подключать не надо. Его использование достаточно распространено.

Формат использования:

```
модификатор тип *имяУказателя = NULL;
модификатор тип *имяУказателя(NULL);
модификатор тип *имяУказателя{NULL};
```

Замечание.

Поскольку NULL является макросом препроцессора и, технически, не является частью языка C++ (а относится к языку C), то его не рекомендуется использовать.

Разыменование нулевых указателей

При разыменовании нулевого указателя в большинстве случаев разработчик получит сбой в программе. Ведь разыменование указателя

означает, что нужно «перейти к адресу, на который указывает указатель, и достать из этого адреса значение». Нулевой указатель не имеет адреса, а тем более значения по адресу, поэтому это прогнозируемый результат.

Указатели на константы

Указатели могут указывать как на переменные, так и на константы. Для того чтобы определить указатель на константу, он тоже должен объявляться с ключевым словом **const**. Формат объявления:

```
const ~~модификатор~~ тип *имяУказателя;
```

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      const int a = 10;
5      const int *pa = &a;
6      std::cout << "address = " << pa
7                << "\tvalue = " << *pa
8                << std::endl;
9      return 0;
10 }
```

Результат работы программы:

```
address = 0x7ffca17c281c        value = 10
```

Здесь указатель `pa` указывает на константу `a`. Поэтому даже если разработчик захочет изменить значение по адресу, который хранится в указателе, он не сможет это сделать, а если попытается (например, `*pa = 34;`), то просто получит ошибку компиляции.

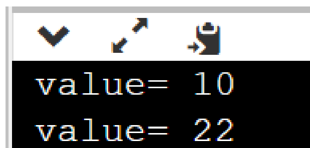
Возможна также ситуация, когда указатель на константу на самом деле указывает на переменную.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 10;
6      const int *pa = &a;
7      cout <<" value= " << *pa << endl; // старое зн. 10
8      a = 22;                  //изменяем значение переменной
9      cout <<" value= " << *pa << endl; // новое зн. 22
10     //*pa = 34;              //будет ошибка компиляции
11     return 0;
12 }
```

Результат работы программы:



Изменить значение переменной с помощью указателя и в этом случае ***не*** получиться.

Замечание.

Через указатель на константу нельзя изменять значение переменной/константы. Однако ему можно присвоить адрес любой другой константы или переменной.

Константный указатель

От указателей на константы надо отличать константные указатели. Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу. Формат объявления:

```
~~модификатор~~ тип *const имяУказателя;
```

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 10;
6      int *const pa = &a;
7      cout << " value= " << *pa << endl; // 10
8      // меняем значение с помощью указ.
9      *pa = 22;
10     cout << " value= " << *pa << endl; // 22
11     int b = 45;
12     // pa = &b; - будет ошибка компиляции
13     return 0;
14 }
```

Результат работы программы тот же, что и в предыдущем случае.

Константный указатель на константу

Объединением обоих предыдущих случаев является константный указатель на константу, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу. Формат объявления:

```
const ~~модификатор~~ тип *const имяУказателя;
```

Пример.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      int a = 10;
5      const int *const pa = &a;
6      //*pa = 22; - будет ошибка компиляции
7      int b = 45;
```

```
 8        //pa = &b;   - будет ошибка компиляции
 9        return 0;
10  }
```

Размер указателей

Размер указателя зависит от архитектуры, на которой скомпилирован исполняемый файл: 32-битный исполняемый файл использует 32-битные адреса памяти. Следовательно, указатель на 32-битном устройстве занимает 32 бита (4 байта).

С 64-битным исполняемым файлом указатель будет занимать 64 бита (8 байт). И это вне зависимости от того, на какой тип указывает указатель.

Как можно видеть, размер указателя всегда один и тот же. Это связано с тем, что указатель — это всего лишь адрес памяти, а количество бит, необходимое для доступа к адресу памяти на определенном устройстве, — всегда постоянное и зависит от используемой операционной системы.

Операции над указателями

Операция разыменования

Операция разыменования `*` (или операция доступа к содержимому объекта). Хотя эта операция уже неоднократно рассматривалась, но для полноты изложения ее еще раз необходимо упомянуть.

Формат операции:

```
*имяУказателя
```

Результатом операции является обращение к значению, расположенному по адресу, указанному указателем `имяУказателя`.

Операция взятия адреса переменной/объекта

Напомним, что для получения адреса объекта используется операция `&`. Формат:

```
&имяПеременной
```

Присваивание

Указателю можно присвоить либо адрес объекта того же типа, либо значение другого указателя. Формат операции:

```
имяУказателя1 = адресОбъекта
```

где `имяУказателя1` – указатель, `адресОбъекта` – адрес объекта либо другой указатель. Когда указателю присваивается другой указатель, то указатель, стоящий слева поле выполнения операции, будет указывать на тот же адрес, на который указывает указатель, стоящий справа.

Замечание.

При выполнении операции присваивания переменная `имяУказателя1` может иметь тип `void` независимо от типа объекта `адресОбъекта`.

Сложение указателя и целого числа, а также вычитание из указателя целого числа

Данные операции иногда называются увеличение и уменьшение указателя соответственно. Формат операции:

```
имяУказателя + i
имяУказателя - i
```

где `имяУказателя` – адрес области памяти, `i` – переменная целого типа. Тип результата обеих операций совпадает с типом переменной `имяУказателя`.

Операция используется для перемещения по участку памяти. Действие этой операции поясним на примере. Пусть имеется несколько переменных ***одного типа*** расположенные в памяти таким образом, чтобы за одним сразу же следовал другой элемент. Пусть указатель `pointer` указывает на одну из переменных в середине данного участка.

Пронумеруем переменные следующим образом: тот элемент, на который указывает `pointer` будет иметь номер нуль (и значение нуль), элементы в сторону увеличения адресов пронумеруем положительными целыми числами, а в сторону уменьшения отрицательными (Рисунок 20).

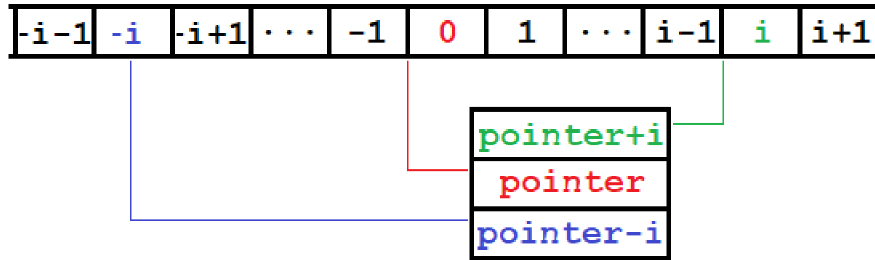


Рисунок 20 – Нумерация элементов в ленте относительно приятого за начало отсчета указателя pointer

Рассмотрим пример программы реализующий данный алгоритм. К сожалению, при использовании онлайн интегрированной среды ленту из подряд идущих переменных одного типа можно выделить, только в рамках объявления одномерного массива значений.

Воспользуемся этим и создадим массив из нечетного числа элементов, что будет гарантировать существование у массива срединного элемента, а также симметричность ленты памяти относительно этого элемента.

Значениям элементов массива присвоим их номера `n` относительно центрального элемента: в сторону возрастания - положительные, а в сторону убывания - отрицательные.

Пример.

main.cpp

```
1  #include <iostream>
2  #define SIZE 7
3
4  int main() {
5      int a[SIZE];
6      int *pointer = &a[SIZE / 2];//адрес срединного элем.
7      *pointer = 0;
8      for(int n = 1; n <= SIZE / 2; n++) {
9          *(pointer - n) = -n; //операция уменьшения
10         *(pointer + n) = n;  //операция увеличения
11     }
12
13     //вывод на экран заполненного массива
14     for(int i = 0; i < SIZE; i++) {
15         std::cout << " " << a[i];
16     }
17     return 0;
18 }
```

Результат работы программы:

Данная схема хождения по адресам относительно указанного (выбранного за нуль) элемента будут использоваться и далее по курсу, вне зависимости от того, каким конкретно образом выделялась лента памяти в программе. Хождение по адресам всегда можно реализовать, т.к. вся память компьютера по сути своей одномерный массив байт.

Замечания:

- *Следует подчеркнуть следующую особенность характерную для операций уменьшения/увеличения указателя. Например, если рассмотреть операцию* `pointer + 1`, *то это не значит, что адрес* `pointer` *увеличился на* `1` *байт. Адрес указателя увеличится на число байт, определяемое типом хранимого значения. Например, для типа* `int` *из программы выше увеличение произойдет на* `4` *байта.*
- *Соответственно при операции* `pointer + i` *адрес автоматически увеличится на* `4 * i` *байт. Именно поэтому тип и переменной, и указателя, содержащего адрес этой переменной, должны совпадать (для того, чтобы обрабатывать одинаковое число бит как при прямом обращении по имени, так и при косвенном обращении через указатель).*

Приведение типов

Иногда требуется присвоить указателю одного типа значение указателя другого типа. В этом случае следует выполнить операцию приведения типов с помощью выражения, имеющего формат:

```
(новыйТипУказателя*) имяПреобразУказателя
```

Для преобразования указателя к другому типу в скобках перед указателем ставится тип указателя (`новыйТипУказателя*`), к которому надо преобразовать. Причем если нельзя создать объект, например, переменную типа `void`, то для указателя на этот тип ограничений нет, т.е. можно создать указатель типа `void`.

Пример.

```
#include <iostream>

int main() {
    char c = 'N';
    char *pc = &c;
    int *pd = (int *)pc;   //первое к int*
    void *pv = (void*)pc; //второе к void*
    std::cout << " pv = " << pv
              << std::endl;
    std::cout << " pd = " << pd
              << std::endl;
    return 0;
}
```

Результат работы программы:

```
pv = 0x7ffde421a24f
pd = 0x7ffde421a24f
```

Кроме того, следует отметить, что при выводе значения указателя `pc` на тип `char` (`char *pc = &c`) с помощью обычной инструкции (`cout << "pc=" << pc << endl;`) система будет его интерпретировать как строку.

Поэтому если необходимо вывести на консоль адрес, который хранится в указателе `pc` типа `char`, то это указатель надо сперва преобразовать, например, к типу `void*`.

Замечание.

Указатель любого типа преобразуется к типу `void` *при присваивании без операции явного преобразования типов. Преобразование типов указателей следует использовать с осторожностью, т.к. можно внести ошибку в интерпретацию данных.*

Сложное присваивание

Для указателей существует только две операции сложного присваивания:

`имяУказателя += i`   или   `имяУказателя -= i`

где `имяУказателя` – идентификатор переменной типа указатель, `i` – целочисленная переменная.

Каждая из операций полностью аналогична соответствующим выражениям:

```
имяУказателя = имяУказателя + i
имяУказателя = имяУказателя - i
```

Индексирование

Формат операции:

```
имяУказателя[i]
```

где `имяУказателя` – идентификатор переменной типа указатель, `i` – целочисленная переменная.

Эта операция полностью эквивалентна выражению (Рисунок 20):

```
*(имяУказателя + i)
```

Операции инкремента и декремента

Формат операций в префиксной форме:

`++имяУказателя` или `--имяУказателя`

Формат операций в постфиксной форме:

`имяУказателя++` или `имяУказателя--`

где `имяУказателя` – идентификатор переменной типа указатель.

Выполнение этих операций аналогично соответствующим операциям над целочисленными типами, т.е. указатель будет смещаться вправо или влево на 1 элемент в ленте памяти (Рисунок 20).

После выполнения операций инкремента декремента значение самой переменной изменяется по тем же правилам, что указывалось раньше, т.е. адрес измениться на количество байт необходимых для хранения типа данного, указанного при объявлении или измененного при выполнении явного преобразования типов.

Вычитание указателей

Формат операции:

```
имяУказателя1 – имяУказателя2
```

где имяУказателя1, имяУказателя2 – имена указателей одного типа. Результатом операции является число типа int, равное количеству элементов использованного при объявлении указателей типа, которые можно расположить между ячейками памяти, на которые указывают переменные имяУказателя1 и имяУказателя2.

Операции сравнения (отношения) указателей

Формат операций:

```
имяУказателя1 == имяУказателя2
имяУказателя1 >= имяУказателя2
имяУказателя1 > имяУказателя2
имяУказателя1 < имяУказателя2
имяУказателя1 <= имяУказателя2
имяУказателя1 != имяУказателя2
```

где имяУказателя1, имяУказателя2 – имена указателей одного типа. Результатом операции является булевский тип, имеющий значения true или false, которые в зависимости от контекста могут быть представленных, как целочисленные 1 или 0.

Результат операции определяется находятся ли правее, или левее, или совпадают адреса в оперативной памяти, значение которых присвоено переменным имяУказателя1 и имяУказателя2.

Некоторые особенности операций с указателями

Очевидно, из изложенного выше следует, что при работе с указателями надо отличать операции с самим указателем и операции со значением, размещенным в памяти по адресу, на который указывает указатель.

Но в то же время есть особенности в использовании операций в сложных выражениях, в частности, с операциями инкремента и декремента. Дело в том, что операции *, ++ и -- имеют одинаковый приоритет и при размещении рядом без круглых скобок выполняются справа налево.

Из-за этого во многих случаях, когда разработчик подразумевал определенный результат, а программа выдавала совершенно другой, то виновницей могла стать неверная последовательность выполнения операций с указателями.

Замечание.
Следует обратить внимание еще раз, как и в случае с операциями над обычными переменными, что использование круглых скобок гарантирует разработчику необходимую последовательность операций.

Адрес указателя. Указатель на указатель

Указатель хранит адрес переменной. Но, очевидно, как и любая переменная, сам имеет адрес, по которому он располагается в памяти. Этот адрес можно получить также через операцию `&`. Но возникает вопрос: чему можно присвоить это значение (а именно адрес переменной-указателя)?

Для этого используются переменная ***указатель-на-указатель***. Формат объявления переменной-***указателя-на-указатель***:

~~`модификатор`~~ `тип **имяУказательУказатель;`

где `имяУказательУказатель` – произвольно выбираемый идентификатор.

Указатель-на-указатель работает подобно обычному указателю: его можно разыменовать для получения значения, на которое он указывает. Поскольку этим значением является другой адрес, то для получения значения исходной переменной потребуется выполнить разыменование еще раз (т.е. дважды).

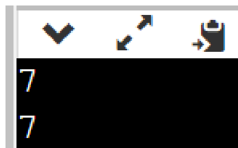
Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int value = 7;
    int *ptr = &value;
    //однократное разыменование указателя
    cout << *ptr << endl;
```

```
 9        //объявление указателя-на-указатель
10        int **ptrptr = &ptr;
11        //двукратное разыменование указ.-на-указ.
12        cout << **ptrptr << endl;
13        return 0;
14    }
```

Результат работы программы:



Следует обратить внимание на то, что нельзя инициализировать указатель-на-указатель напрямую адресом адреса значения.

Пример.

```
int value = 7;
int **ptrptr = &&value; // нельзя
```

Однако указателю-на-указатель можно задать значение `nullptr`:

```
int **ptrptr = nullptr; // можно использовать 0
```

Указатели и автоматические массивы

Взаимосвязь указателей и одномерных автоматических массивов

В C++ указатели и массивы тесно связаны. С помощью указателей можно манипулировать элементами массива, как и с помощью индексов.

Рассмотрим уже известное объявление автоматического одномерного массива с помощью инструкции:

~~модификатор~~ `тип имя[N];`

где N – целая константа.

Имя массива имя (без индекса и скобок) является указателем-константой (не изменяемой на протяжении всей работы программы), т.е. адресом первого элемента массива (&имя[0]).

Покажем это, используя последовательную запись в развернутом виде операции индексирования и операции взятия адреса.

Пусть указатель pointer того же типа что и одномерный массив проинициализирован адресом нулевого элемента массива:

~~модификатор~~ тип ***pointer** = &**имяМассива**[0];

тогда запишем равенство справа, расставив круглые скобки:

&**имяМассива**[0] это &(**имяМассива**[0]),

По определению операции индексирования последнее выражение равносильно записи:

&(*(**имяМассива** + 0)).

Очевидно, что указатель имя + 0 это просто значение указателя имя, учитывая это, уберем внутренние скобки, получаем запись:

&(***имяМассива**)

Уберем внешние скобки и, учитывая, то, что операция взятия адреса (&) и операция взятия содержимого по адресу (*), логически взаимно обратны (они просто уничтожают друг друга), то сравнивая начало и конец получаем, что:

~~модификатор~~ тип ***pointer** = **имяМассива;**

Продемонстрируем это с помощью очевидной программы. В качестве тестовых выберем два типа int и float (их можно менять на любые в том числе на типы с модификатором).

Пример.

main.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
```

```
 5     //объявляем автоматический целочисленный массив
 6     int arrayInt[3];
 7     //объявляем указатель типа int и присваиваем ему
 8     //адрес начала целочисленного массива
 9     int *pIntArray = &arrayInt[0];
10     //инициализируем автоматический массив чисел с плав.т.
11     float arrayFloat[] = {1., 21., 13., 64., 5.};
12     //объявляем указатель типа float и присваиваем ему
13     //адрес начала массива чисел с плав. точкой
14     float *pFloatArray = &arrayFloat[0];
15     //выводим для сравнения значения имен массивов и указат.
16     cout<< "arrayInt = "<< arrayInt
17         << "\tpIntArray = "<< pIntArray <<endl;
18     cout<< "arrayFloat = "<< arrayFloat
19         << "\tpFloatArray = "<< pFloatArray <<endl;
20     return 0;
21 }
```

Результат работы программы:

```
arrayInt = 0x7ffcef7a8a14        pIntArray = 0x7ffcef7a8a14
arrayFloat = 0x7ffcef7a8a20      pFloatArray = 0x7ffcef7a8a20
```

Таким образом, установлено, что имена массивов являются указателями и представляют собой символические адреса начала ленты памяти, выделенной под их (массивов) хранение. Это дает возможность использовать правила работы с указателями для обработки одномерными массивами.

Пусть инициализирован целочисленный массив с именем `a`:

```
int a = {9, 8, 7, 6, 5, 4};
```

Прибавляя к адресу нулевого элемента (имя массива `a`) некоторое целое число `i` (выражение вида `a + i`), можем получить адреса следующих элементов массива. Обратиться к их содержимому расположенному по адресу `(a + i)` можно с помощью операции разыменования (выражение вида `*(a + i)`), но это и есть по определению операция индексирования (т.е. выражение вида `a[i]`).

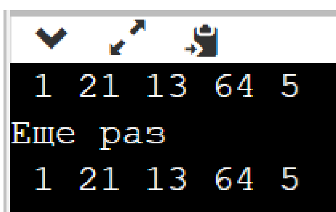
Организуем в цикле вывод на экран в строку значений одномерного массива с помощью указателей (Рисунок 21).

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    float a[] = {1., 21., 13., 64., 5.};
    int N = sizeof(a) / sizeof(float);
    //объявление константных указателей
    float *const begin = a;       //начало массива
    float *const end = &a[N - 1] + 1; //его конец
    //объявление переменной-указателя на элемент
    float *pointer;
    for(pointer = begin; pointer < end; pointer++) {
        cout<< " " << *pointer;
    }
    cout<< endl << "Еще раз" << endl;
    for(float *p = a; p < a + N; p++) {
        cout<< " " << *p;
    }
    return 0;
}
```

Результат работы программы:



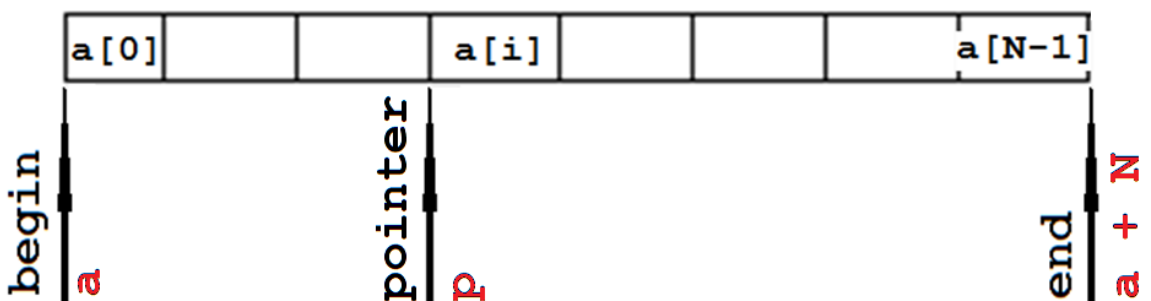


Рисунок 21 – Общая схема перемещения с помощью указателей по одномерному массиву

Одномерные автоматические массивы указателей

Формат объявления массива указателей:

~~модификатор~~ тип* имяУказателя**[количЭлементов]**;

где **количЭлементов** – некоторая целочисленная константа.

Массив указателей, может быть, проинициализировал, но для его инициализации необходимы адреса объектов. Например, инициализируем три целочисленных массива с именами a, b и c, а потом с помощью этих трех имен массивов, которые, как уже известно, являются указателями, проинициализируем уже массив указателей.

Пример.

```
#include <iostream>

int main() {
    int a[] = {1, 2, 3};
    int b[] = {4, 5, 6};
    int c[] = {7, 8, 9};

    int *ptrArray[] = {a, b, c};
    int **ptrptr = ptrArray;
    return 0;
}
```

Программа не имеет никакого результата выполнения кроме демонстрации того, что не только компилятор ее пропустит, но и она выполниться без каких-либо ошибок.

Кроме того, данная программа демонстрирует, что если имя одномерного массива является указателем, то имя массива, состоящего из указателей, является указателем-на-указатель.

Кроме тогда:

- ptrArray[0] – это обращение к адресу начала одномерного массива a;
- ptrArray[1] – это обращение к адресу начала одномерного массива b;
- ptrArray[2] – это обращение к адресу начала одномерного массива c.

Действуя далее в соответствии с логикой работы с указателями к элементу массива b[1] равного 5 можно обратиться с помощью выражения: *(ptrArray[1] + 1), что по определению это эквивалентно ptrArray[1][1].

Продолжая действовать так дальше, можно обосновать, что обращение к любому элементу из этих трех массивов может быть организовано с помощью выражения характерного для обращения к элементу двумерного массива:

ptrArray[i][j]

где:

- i = 0 – это начало массива a,
- i = 1 – это начало массива b,
- i = 2 начало массива c.

Далее:

- j = 0 – нулевой элемент каждого из массивов,
- j = 1 – это первый элемент каждого из массивов,
- j = 2 - второй элемент каждого из массивов.

Для демонстрации этого выведем двумерный массив ptrArray построчно на экран.

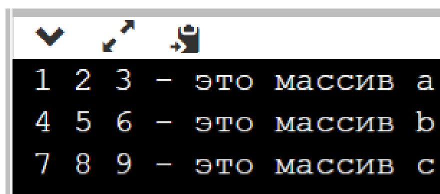
Пример.

main.cpp

```
1  #include <iostream>
2  #define N 3
3  using namespace std;
4
5  int main() {
6      int a[N] = {1, 2, 3};
7      int b[N] = {4, 5, 6};
8      int c[N] = {7, 8, 9};
9      //инициализация одномерного массива указателей
10     int *ptrArray[N] = {a, b, c};
11
12     for(int i = 0 ; i < N; i++) {
13         for(int j = 0 ; j < N; j++) {
14             cout <<' '<< ptrArray[i][j];
15         }
```

```
16         switch(i) {
17             case 0: cout<< " - это массив a"; break;
18             case 1: cout<< " - это массив b"; break;
19             case 2: cout<< " - это массив c"; break;
20         }
21         cout << endl;
22     }
23     return 0;
24 }
```

Результат выполнения программы:



```
1 2 3 - это массив a
4 5 6 - это массив b
7 8 9 - это массив c
```

Указатель-на-указатель и двумерные автоматические массивы

Рассмотрим, каким образом хранится двумерный массив в памяти компьютера. Аналогично предыдущему примеру, где рассматривался массив указателей, элементами которого являлись одномерные массивы, можно сказать, что двумерные массивы в С++ хранятся в памяти компьютера аналогичным образом (построчно, Рисунок 22). Т.е. существует строка (массив указателей, Рисунок 22), которая содержит адреса начала строк двумерного массива элементами которых уже являются данные.

Пример.

```
int arr[4][2];  //объявлен двумерны массив 4 строки
                //на 2 столбца
```

Таким образом, имя двумерного массива `arr`, как и в предыдущем примере `ptrArray` является ***указателем-на-указатель***.

Таким образом двумерный массив в языке С++ является одномерным массивом из одномерных массивов. Это видно из синтаксиса объявления двумерного массива (конструкции `имя[N][M]`), а также синтаксиса обращения к элементу двумерного массива `имя[i][j]`. Если бы в С++ существовали в чистом виде двумерные массивы, то синтаксис в обоих перечисленных случаях был бы `имя[N, M]` и `имя[i, j]`.

Рисунок 22 – Схема хранения в памяти двумерного массива

Куча. Средства работы с кучей

Куча — это участок оперативной памяти, который допускает выделение памяти в процессе выполнения программы, а не на этапе ее запуска. Она ***не*** работает по принципу ***стека***, т.к. является просто складом для переменных. По завершении приложения все выделенные участки памяти освобождаются. Размер ***кучи*** задается при запуске приложения, но, в отличие от ***стека***, он ограничен только размерами оперативной памяти компьютера, и это позволяет создавать, хранить и обрабатывать очень большие объекты.

Взаимодействие с ***кучей*** осуществляется посредством указателей. Центральный процессор не принимает участия в контроле над ***кучей*** и в языках без сборщика мусора (язык C++) разработчику нужно вручную освобождать участки памяти, которые больше не нужны. Если этого не делать, могут возникнуть утечки и фрагментация памяти, что существенно замедлит работу ***кучи***.

Размер ***стека*** намного меньше размера ***кучи***, и поскольку современное программирование связано с обработкой массивов, коллекций или объектов, занимающих значительные размеры оперативной памяти, то при работе с подобными объектами, следует ориентироваться на использование памяти именно в ***куче***, а не в ***стеке*** (точнее ***фрейме*** соответствующей функции ***стека вызовов***).

Замечание.

Использование инструментария языка C++ позволяет выделять память под хранение как единичных переменных, так и массивов во время выполнения программы, т.е. в случае когда на момент компиляции программы, например, неизвестно сколько элементов будет в обрабатываемом массиве.

Для выделения и освобождения памяти в ***куче*** используются операции `new` и `delete`:

- `new` — для выделения памяти;
- `delete` — для освобождения памяти.

Синтаксис этих операций для выделения памяти под скалярную переменную отличается от синтаксиса операции для выделения памяти под массив.

Работа со скалярными переменными в куче

Формат операции `new` для выделения памяти в ***куче*** для хранения скалярных переменных:

`new имяТипа;`   или   `new имяТипа(~~инициализатор~~);`

позволяет выделить и сделать доступным свободный участок памяти в ***куче***, размеры которого соответствуют типу данных, определяемому параметром `имяТипа` (возможно с модификатором).

В выделенный участок заносится значение, определяемое параметром ~~инициализатор~~ (в круглых скобках), который не является обязательным параметром. В случае успешного выделения памяти операция возвращает адрес начала выделенного участка памяти; если участок не может быть выделен, то возвращается `nullptr` и исполнение программы может быть прервано.

Пример.

```
…
//объявить указатель
int * pIndex;

//создать в куче переменную типа int и
//инициализировать ее знач. 10
pIndex = new int (10);
…
float *f;
//создать в куче перем. типа float
f = new float;
…
```

Перейдем к рассмотрению операции `delete`. Существует два варианта операции `delete`: один - для единичных объектов, другой – массивов. Для массивов будет рассматриваться далее. Однако формат операции для единичных объектов можно привести прямо сейчас:

`delete имяУказателя;`

Пример.

…
```
int * pIndex;
pIndex = new int (10);
```
…
```
delete pIndex; //удалить выделенную память для
               //динамической переменной по адресу pIndex
```
…

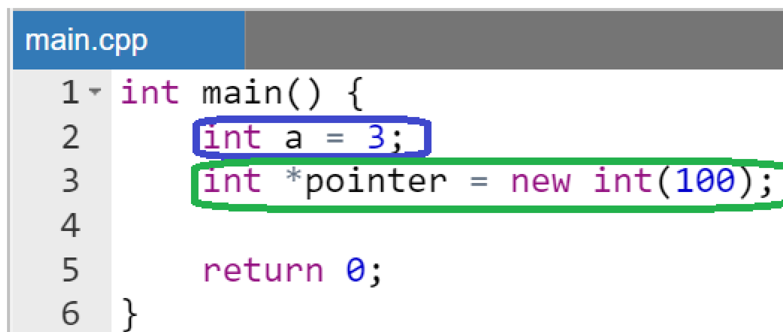
Замечание.

В отличие от обычных переменных, хранимых в ***стеке****, для переменной, хранимой в* ***куче****, программист сам определяет в области видимости переменной момент ее удаления из памяти с помощью операции* `delete`.

Взаимодействие стека и кучи при работе со скалярной переменной в куче

Рассмотрим пример элементарной программы.

Пример.



```
1 int main() {
2     int a = 3;
3     int *pointer = new int(100);
4
5     return 0;
6 }
```

Запуск программы на выполнение - это вызов функции `main()`. При запуске для `main()` создается ***фрейм***, который помещается в ***стек*** вызовов (Рисунок 23). Фрейм содержит все объявленные в `main()` переменные, в том числе и переменные-указатели (в примере выше `a` и `pointer`).

Приведем последовательность действий в соответствии с программой (Рисунок 23):

- в ***стеке*** создается переменная `a` и инициализируется значением `3`;

- команда `new int(100)` отводит в ***куче*** ячейку памяти для хранения целочисленного значения и инициализирует эту ячейку значением `100`;
- в ***стеке*** создается переменная-указатель `pointer`;
- адрес ячейки выделенной в ***куче*** операцией `new` присваивается переменной `pointer`, расположенной в ***стеке***.

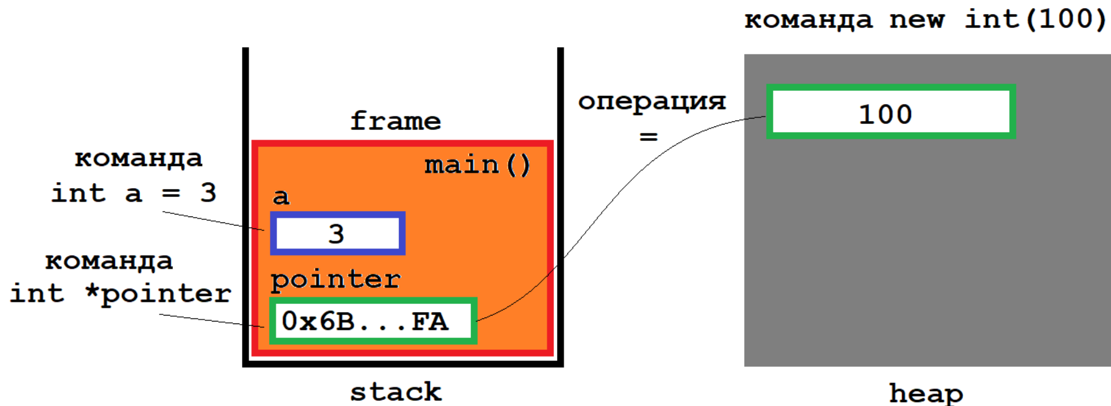


Рисунок 23 – Взаимодействие стека и кучи при работе с областью памяти в куче, выделенной для хранения скалярной переменной

Следует помнить, что из-за фрагментации памяти последовательные запросы о выделении памяти в ***куче*** не всегда приводят к выделению последовательных адресов памяти. При удалении выделенной в ***куче*** памяти с помощью `delete`, использовавшаяся память становиться снова открытой для новых выделений памяти для новых переменных и массивов и затем может быть переназначена (исходя из последующих запросов).

Работа с ***кучей*** имеет свои недостатки:

- выделение памяти в ***куче*** сравнительно медленное;
- память остается выделенной до тех пор, пока не будет освобождена операцией `delete` или пока не завершится программа;
- если во время работы программы выделение в ***куче*** памяти плохо контролировалось, то даже в ***куче*** может возникнуть переполнение;
- доступ к выделенной в куче памяти осуществляется только через указатель;
- разыменование указателя происходит медленнее, чем доступ к переменной напрямую.

Одномерные массивы в куче

Во многих старых учебных пособиях по языку C++ выделение памяти под массив в ***куче*** в процессе выполнения программы ассоциируется с понятием «динамического» массива. Однако в языке C++ нет средств изменения длины массива, выделенного в ***куче***. Поэтому использование словосочетания «динамический массив» в C++ не корректно и досталось по наследству от динамических массивов языка С.

Выделение памяти под хранение массива в куче

Операция `new` для выделения ленты памяти для хранения одномерного массива используется в следующем формате:

```
new ~~модификатор~~ ~~модификатор~~ тип[КоличЭлементов]
```

Последовательность действий при выделении памяти в ***куче*** под хранение одномерного массива:

1. объявить:
   a. ***указатель*** того типа, данные которого будут храниться в массиве,
   b. ***целочисленную переменную***, определяющую число элементов в массиве:

```
тип *имяМассиваИлиУказателя;
int n;
```

2. объявить и определить, например, с помощью средств ввода переменную (например, **n**) задающую размерность массива:

```
cout<<"Введите размерность";
cin>> n;
```

3. операцией `new` выделить ленту памяти, необходимую под хранение одномерного массива:

```
имяМассиваИлиУказателя = new тип[n];
```

Пример.

```
…
int *array, n;
…
cout<<"Введите размерность";
cin>> n;
array = new int [n];          //выделение ленты памяти
…
```

Освобождение памяти

Освобождение (удаление) *ленты памяти из* ***кучи***, выделенной для хранения одномерного массива, осуществляется с помощью операции `delete`. Формат оператора:

```
delete []имяМассиваИлиУказателя;
```

После того как массив стал не нужен, его также следует удалить, но если после оператора `delete` будет стоять только имя указателя (без квадратных скобок), то будет удалена только одна переменная из массива – переменная с нулевым индексом. Остальные останутся в памяти. Для того чтобы был удален весь массив, следует перед указателем поставить квадратные скобки.

Пример.

```
…
delete []array;
…
```

Взаимодействие стека и кучи в случае работы с одномерными массивами

Схема взаимодействия ***стека*** и ***кучи*** при работе с одномерными массивами, выделение памяти для которых происходило с помощью операции `new` остается такой же, как и в случае скалярных переменных. Рассмотрим простейшую программу. При ее запуске для `main()` создается фрейм, который помещается в ***стек*** вызовов (Рисунок 24).

Пример.

```
main.cpp
1 int main() {
2     int a = 3;
3     int *array = new int[5];
4
5     return 0;
6 }
```

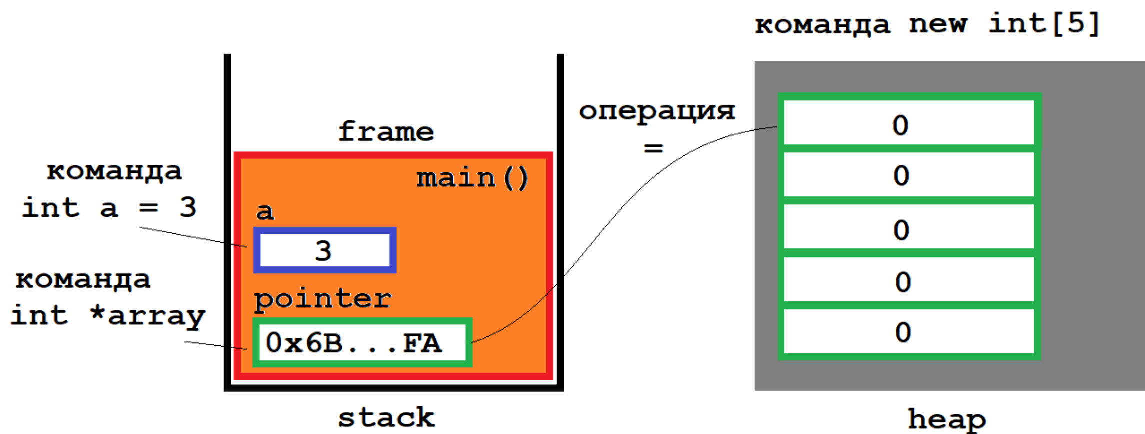


Рисунок 24 – Стек и куча при работе с массивами в куче

Приведем последовательность действий в соответствии с программой (Рисунок 24):

- в ***стеке*** создается переменная `a` и инициализируется значением `3`;
- команда `new int[5]` отводит в ***куче*** непрерывную ленту памяти для хранения пяти целочисленных значений и инициализирует все ячейки ленты значением `0`;
- в ***стеке*** создается переменная-указатель `array`;
- адрес ленты памяти, выделенной операцией `new` в ***куче***, присваивается переменной `array`, расположенной в ***стеке***.

Пример программы с использованием одномерного массива, хранящегося в куче

Работа с элементами массива выполняется на основе операции индексирования, т.е. если одномерный массив с именем `array` создан в ***куче***, то и обращение к значению `i`-го элемента массива осуществляется с помощью уже известной синтаксической конструкции `array[i]`.

Рассмотрим программу, выделяющую память в ***куче*** под хранение одномерного массива, заполняющую его случайными числами и в конце удаляющую ленту массива из памяти.

Пример.

main.cpp

```cpp
#include <iostream>
#include <cstdlib>  //для функций rand() и srand()
#include <ctime>    //для функции time
#define MAX_RAND_VALUE 100
using namespace std;

int main() {
    int *array, n;

    while(true) {
        cout<<"Введите число элементов массива"
        << endl;
        cin>> n;
        if(0 < n) break;
    }
    //выделение ленты памяти
    array = new int [n];
    //заполнение массива псевдослуч. числами
    srand(time(0));
    for(int i = 0; i < n; i++) {
        array[i] = rand() % MAX_RAND_VALUE;
    }
    cout<<"Рез. зап. дин. массива псевдослуч. числ."
        << endl;
    for(int i = 0; i < n; i++) {
        cout<<' '<< array[i];
    }
    //Далее можно вставлять в эту программу любые
    //алгоритмы для обработки одномерного массива и
    //вывода на экран результата

    //удаление ленты памяти
    delete []array;
    return 0;
}
```

Результат выполнения программы:

```
Введите число элементов массива
7
Рез. зап. дин. массива псевдослуч. числ.
 65 87 84 31 94 76 37
```

Язык С. Особенности работы с одномерным массивом в куче

В C++ также доступны средства языка С для работы с ***кучей***. В этом случае работать с ***кучей*** можно при помощи функций распределения памяти (`malloc()`, `calloc()`,`free()`, Таблица 39), для чего необходимо подключить заголовочный файл `malloc.h` или `cstdlib` (заголовочный файл C++). Следует отметить, что существует еще функция `realloc()` (Таблица 39), позволяющая менять в ходе программы количество элементов в уже выделенном массиве, но в данном курсе она не будет рассматриваться.

Таблица 39 – Функции языка С для работы с памятью в куче

Функция	Прототип и краткое описание
`malloc()`	`void *malloc(unsigned s);` Возвращает указатель на начало области памяти в ***куче*** длиной `s` байт, при неудачном завершении возвращает `NULL`
`calloc()`	`void *calloc(unsigned n, unsigned m);` Возвращает указатель на начало области памяти в ***куче*** для размещения `n` элементов по `m` байт каждый, при неудачном завершении возвращает `NULL`.
`realloc()`	`void *realloc(void *p, unsigned s);` Изменяет размер блока ранее выделенной памяти в ***куче*** до размера `s` байт. Величина `s` задается в байтах и может быть больше или меньше оригинала. Параметр `p` – адрес начала исходного массива. Возвращается указатель на ленту памяти, поскольку может возникнуть необходимость переместить ленту при возрастании ее размера. В таком случае содержимое старого массива копируется в новый массив без потери информации.
`free()`	`void *free(void *p);` Освобождает ранее выделенный в ***куче*** участок памяти, где `p` – адрес первого байта удаляемого участка.

Рассмотрим более детально, как работаю функции `malloc()`, `calloc()` и `free()`. Формат использования `malloc()` в паре с `free()` следующий (для сокращения записи модификатор типа не указывается, хотя, очевидно, он может присутствовать):

```
…
тип *имяМассива;          //объявление указателя
int n;                    //количество элементов
…
//выделение ленты памяти в куче под массив
имяМассива = (тип*)malloc(n * sizeof(тип));
…
//удаление ленты памяти
free(имяМассива);
…
```

Формат использования `calloc()` с `free()` практически аналогичен предыдущему случаю:

```
…
тип *имяМассива;            //объявление указ.
int n;                      //количество элем.
…
//выделение ленты памяти в куче под массив
имяМассива = (тип*)calloc(n, sizeof(тип));
…
//удаление ленты памяти
free(имяМассива);
…
```

Изменим предыдущий пример заполнения одномерного массива в ***куче*** случайными числами для демонстрации работы с функциями из С.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>   //для функций rand() и srand() и
3                       //функций malloc() и calloc()
4  #include <ctime>     //для функции time()
```

```
 5  #define MAX_RAND_VALUE 100
 6  using namespace std;
 7
 8  int main() {
 9      int *array, n;
10      while(true) {
11          cout<<"Введите число элементов массива"
12          << endl;
13          cin>> n;
14          if(0 < n) break;
15      }
16      //выделение ленты памяти
17      array = (int *)calloc(n, sizeof(int));
18      //или array = (int *)malloc(n * sizeof(int));
19
20      //заполнение массива псевдослуч. числами
21      srand(time(0));
22      for(int i = 0; i < n; i++) {
23          array[i] = rand() % MAX_RAND_VALUE;
24      }
25      cout<<"Рез. зап. дин. массива псевдослуч. числ."
26          << endl;
27      for(int i = 0; i < n; i++) {
28          cout<<' '<< array[i];
29      }
30      //Далее можно вставлять в эту программу любые
31      //алгоритмы для обработки одномерного массива и
32      //вывода на экран результата
33
34      //конец программы - удаление ленты памяти
35      free(array);
36      return 0;
37  }
```

Результат выполнения программы совершенно аналогичен предыдущему случаю.

Двумерные массивы в куче

Порядок действий при выделении памяти

При выделении памяти в ***куче*** под двумерный массив, как и в одномерном массиве необходимо соблюдать определенную последовательность действий:

1. объявить:
   a. ***указатель-на-указатель*** того типа, данные которого будут храниться в двумерном массиве,
   b. две ***целочисленные переменные*** (**n** и **m**), определяющие количество строк и столбцов или одну переменную (**n**), если массив квадратный:

```
тип **имяМассиваИлиУказНаУказ;
int n, m;
```

2. определить с помощью средств ввода значения целочисленных переменных (**n** и **m**) или одну переменную (**n**), если массив квадратный):

```
cout<< "Введите параметры n, m ";
cin>> n>> m;
```

3. операцией `new` выделить одномерный массив размерностью равной количеству строк в двумерном массиве, предназначенный для хранения ***адресов*** строк (массив указателей):

```
имяМассиваИлиУказНаУказ = new тип*[n];
```

4. в цикле с числом повторений равном количеству строк выделить одномерные массивы для непосредственного хранения данных в построчном виде (количество элементов в строках двумерного массива должно совпадать с количеством столбцов):

```
for(int i =0; i < n; i++) {
    имяМассиваИлиУказНаУказ[i] = new тип[m];
}
```

Замечание.

Для сокращения записи модификатор типа не указывался, хотя, очевидно, он может присутствовать.

Пример.

```
…
int **array, n, m;                //первое действие
…
cout<< "Введите параметры n, m ";
cin>> n>> m;                      //второе действие

array = new int* [n];             //третье действие
for(int i =0; i < n; i++) {       //четвертое действие
    array[i] = new int[m];
}
…
```

Освобождение памяти от двумерного массива

Последовательность действий при освобождении двумерного массива также несколько сложнее чем в одномерном случае:

1. в цикле с известным числом повторений с помощью операции `delete` удаляются строки хранящие значения элементов двумерного массива:

```
for(int i =0; i < n; i++) {
    //построчное удал. элементов массива
    delete []имяМассиваИлиУказНаУказ[i];
}
```

2. с помощью операции `delete` удаляется массив адресов строк двумерного массива:

```
delete []имяМассиваИлиУказНаУказ;
```

Пример.

```
…
for(int i = 0; i < n; i++) { //первое действие
    delete []array[i];
}
delete []array;              //второе действие
…
```

Заполнение псевдослучайными значениями двумерного массива, хранящегося в куче

Рассмотрим уже известный по работе с массивами автоматической памяти пример. Внесем в него некоторые изменения для того, чтобы продемонстрировать каким образом проходит работа с двумерными массивами, хранящимися в ***куче***.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>  //для функций rand() и srand()
3  #include <ctime>        //для функции time
4  #define MAX_RAND_VALUE 10
5
6  int main() {
7      double **a;
8      int n, m; // количество строк и столбцов
9      //проверка корректности введеных параметров
10     while(true) {
11         std::cout<<"Введите число строк\n";
12         std::cin>> n;
13         std::cout<<"Введите число столбцов\n";
14         std::cin>> m;
15         if(0 < n && 0 < m) break;
16     }
17     //выделение памяти под двумерный массив
18     a = new double* [n];
19     for(int i =0; i < n; i++) {
20         a[i] = new double[m];
21     }
22     //заполнение псевдосл. числами
23     srand(time(0));
24     for(int i = 0; i < n; i++) {
25         for(int j = 0; j < m; j++) {
26             a[i][j] = rand() % MAX_RAND_VALUE;
27         }
28     }
```

```
29      std::cout<<"Вывод созданного массива"
30              << std::endl;
31      for(int i = 0; i < n; i++) {
32          for(int j = 0; j < m; j++) {
33              std::cout<<' '<< a[i][j];
34          }
35          std::cout<<' '<<std::endl;
36      }
37      //Далее можно вставлять в эту программу любые
38      //алгоритмы для обработки двумерного массива и
39      //вывода на экран результата
40
41      //удаление двумерного массива
42      for(int i = 0; i < n; i++) {
43          delete []a[i];
44      }
45      delete []a;
46      return 0;
47  }
```

Результат работы программы:

```
Введите число строк
2
Введите число столбцов
3
Вывод созданного массива
 6 7 1
 5 7 6
```

Язык С. Дополнительный пример по работе с кучей в случае двумерного массива

В данном примере будем использовать функции `malloc()` и `free()` (очевидно, что применение `calloc()` не должно вызвать никаких проблем). Для выделения памяти в ***куче*** под хранение двумерного массива формат и порядок использования `malloc()` следующий:

```
…
//первое действие
тип **имяМассива;
int n, m;
…
//второе действие
cout<< "Введите параметры n и m ";
cin>> n>> m;
…
//третье действие
имяМассива = (тип**)malloc(n * sizeof(тип*));;

//четвертое действие
for(int i =0; i < n; i++) {
    имяМассива[i] =
            (тип*)malloc((тип*)malloc(m* sizeof(тип));
}
…
```

В случае освобождения памяти порядок аналогичен операции `delete`:

```
…
//первое действие
for(int i =0; i < n; i++) {
    //построчное удал. элементов массива со значен.
    free(имяМассива[i]);
}

//второе действие - удал. массива указат. на строки
free(имяМассива);
…
```

Изменим предыдущий пример заполнения двумерного массива, хранящегося в ***куче***, случайными числами для демонстрации работы с функциями из C.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>  //для функций rand() и srand()
3                      //функций malloc() и calloc()
4  #include <ctime>    //для функции time
5  #define MAX_RAND_VALUE 10
6
7  int main() {
8      double **a;
9      int n, m; // количество строк и столбцов
10     //проверка корректности введеных параметров
11     while(true) {
12         std::cout<<"Введите число строк\n";
13         std::cin>> n;
14         std::cout<<"Введите число столбцов\n";
15         std::cin>> m;
16         if(0 < n && 0 < m) break;
17     }
18     //выделение памяти под двумерный массив
19     a = (double**)malloc(n * sizeof(double*));;
20     for(int i =0; i < n; i++) {
21         a[i] = (double*)
22                 (double*)malloc(m * sizeof(double));
23     }
24     //заполнение псевдосл. числами
25     srand(time(0));
26     for(int i = 0; i < n; i++) {
27         for(int j = 0; j < m; j++) {
28             a[i][j] = rand() % MAX_RAND_VALUE;
29         }
30     }
31     std::cout<<"Вывод заданного массива"
32               << std::endl;
33     for(int i = 0; i < n; i++) {
34         for(int j = 0; j < m; j++) {
35             std::cout<<' '<< a[i][j];
36         }
37         std::cout<<' '<<std::endl;
38     }
39     //Далее можно вставлять в эту программу любые
```

```
40      //алгоритмы для обработки двумерного массива и
41      //вывода на экран результата
42
43      //удаление двумерного массива
44      for(int i =0; i < n; i++) {
45          free(a[i]);
46      }
47      free(a);
48      return 0;
49  }
```

Результат работы программы полностью аналогичен предыдущему случаю.

Ссылки

Основные сведения о ссылке

До этого момента уже рассмотрены два основных типа переменных:

- обычные переменные, которые хранят значения напрямую;
- указатели, которые хранят адрес другого значения, для доступа к которому выполняется операция разыменования указателя (**`*`**).

Ссылки — это третий основной вид переменных в языке C++, который работает как псевдоним другой переменной или объекта. Язык C++ по аналогии с указателями поддерживает следующие типы ссылок:

- ссылки на ***не***константные значения (обычно их называют просто «ссылки»);
- ссылки на константные значения;
- константные ссылки.

<u>Замечание.</u>

*Аналогия использования квалификатора **`const`** между ссылками и указателями **не** полная. Например, в языке C++ отсутствуют константные ссылки на константные значения.*

Ссылки на неконстантные значения

Ссылка объявляется с использованием знака амперсанд (**`&`**) между типом данных и именем ссылки. Формат объявления ссылки:

~~модификатор~~ тип **&**имяСсылки = **имяПеременной**;

где имяСсылки – произвольно выбираемый идентификатор, **имяПеременной** – имя уже существующей к моменту объявления ссылки переменной в программе. В этом контексте амперсанд не означает «оператор адреса», он означает «ссылка на».

Пример.

```
…
int value = 7; // обычная переменная
int &ref = value; // ссылка на переменную value
…
```

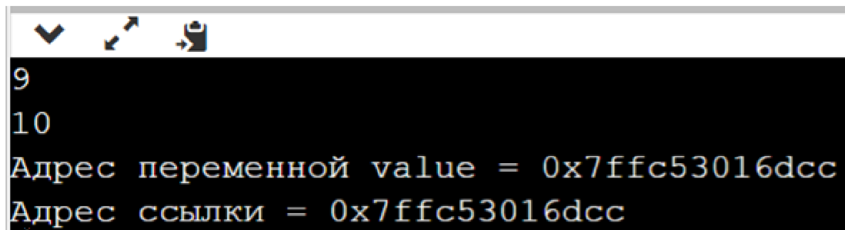
Ссылки обычно ведут себя идентично переменным, на которые они ссылаются. В этом смысле ссылка работает как ее псевдоним.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      // обычная переменная
6      int value = 7;
7      // ссылка на переменную value
8      int &ref = value;
9      //изменение значения по ссылке
10     ref = 9;
11     //проверка значения исходной переменной
12     cout << value << endl;
13     //еще раз изменяется значение по ссылке
14     ++ref;
15     //еще раз проверяется исходная переменная
16     cout << value << endl;
17     //сравнение адресов переменной и ссылки
18     cout << "Адрес переменной value = " << &value
19             << endl;
20     cout << "Адрес ссылки = " << &ref;
21     return 0;
22 }
```

Результат выполнения программы:

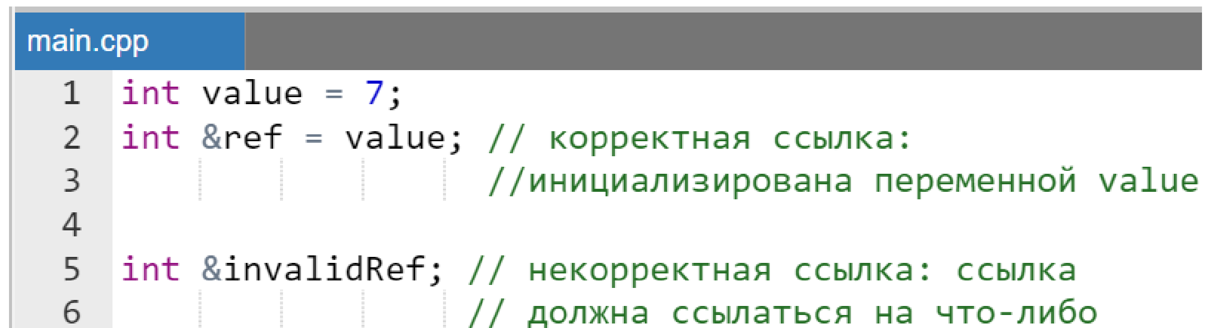

```
9
10
Адрес переменной value = 0x7ffc53016dcc
Адрес ссылки = 0x7ffc53016dcc
```

В примере, приведенном выше использование взятия адреса от переменной и от ссылки на нее приведет к возврату одного и того же значения. Это дополнительно подтверждает уже высказанную мысль о том, что ссылки есть псевдонимы уже существующих переменных.

Замечание.

Ссылки всегда должны быть проинициализированы при создании. Ссылки нулевыми быть ***не*** *могут в отличие от указателей, которые могут содержать нулевое значение.*

Пример.


main.cpp
```
int value = 7;
int &ref = value; // корректная ссылка:
                  //инициализирована переменной value

int &invalidRef; // некорректная ссылка: ссылка
                 // должна ссылаться на что-либо
```

После инициализации изменить объект, на который указывает ссылка нельзя. Даже если присвоить ссылке другую переменную (`ref = a`), то это будет означать, что по ссылке `ref` переменной `b` присвоено значение, хранящееся в другой переменной `a`, но не имя новой переменной и тем более не ее адрес.

Ссылки vs указатели

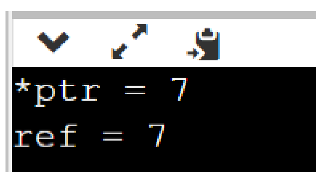
Ссылка — это тот же указатель, который неявно разыменовывается при доступе к значению, на которое он указывает («под капотом» ссылки реализованы с помощью указателей).

Пример.

main.cpp

```cpp
#include <iostream>
using namespace std;

int main() {
    int value = 7;
    int *ptr = &value;
    int &ref = value;
    cout << "*ptr = " << *ptr << endl
         <<"ref = " << *ptr << endl;
    return 0;
}
```

Результат работы программы:



Таким образом, приведенном примере `*ptr` и `ref` обрабатываются одинаково. Поэтому и результат одинаков при выводе на экран.

Поскольку ссылки должны быть инициализированы конкретными объектами (они не могут быть нулевыми) и не могут быть изменены позже, то они, как правило, безопаснее указателей, т.к. риск разыменования нулевого указателя отпадает. Однако они несколько ограничены в функциональности по сравнению с указателями.

Если определенное задание может быть решено с как помощью ссылок, так и указателей, то лучше использовать ссылки. Указатели следует использовать только в тех ситуациях, когда ссылки являются недостаточно эффективными (например, при выделении памяти в ***куче***).

Ссылки на константные значения

Если при инициализации ссылки используется именованная константа, то компилятор выдаст ошибку. В противном случае, была бы возможность изменить значение константного объекта через ссылку, что уже является нарушением понятия «константа».

Пример.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      const int b = 8;
5      int &ref = b; // ошибка компиляции
6      return 0;
7  }
```

Объявить ссылку на константное значение можно путем добавления ключевого слова **const** перед типом данных. Ссылки на константные значения часто называют просто «ссылки на константы». Формат:

Формат объявления ссылки на константное значение:

```
const ~~модификатор~~ тип &имяСсылки = имяПеременной;
```

где `имяСсылки` – произвольно выбираемый идентификатор, `имяПеременной` – имя уже существующей к моменту объявления ссылки переменной в программе.

Пример.

```
const int value = 7;
//далее объявляется ссылка на константу
const int &ref = value;
```

В отличие от ссылок на неконстантные значения, которые могут быть инициализированы только неконстантными величинами, ссылки на константы могут быть инициализированы, как неконстантными, так и константными именованными и неименованными величинами.

Пример.

```
main.cpp
1  #include <iostream>
2
3  int main() {
4      //иниц. константной ссылки с помощью переменной
5      int a = 7;
```

```
6       const int &ref1 = a;
7       //иниц. константной ссылки с пом. именов. конст.
8       const int b = 9;
9       const int &ref2 = b;
10      //иниц. константной ссылки с пом. неимен. конст.
11      const int &ref3 = 5;
12      return 0;
13  }
```

При доступе к значению через ссылку на константное значение, это значение автоматически считается **const**, даже если исходная переменная таковой не является:

Пример.

```
int value = 7;
//создаем ссылку на константу
const int &ref = value;

value = 8; // можно: value - это не константа
ref = 9;   // нельзя: ref - это константа
```

Обычно память, хранящая значение выражений, уничтожается в конце, после того как вычислен его результат. Однако, когда константная ссылка инициализируется значением выражения, то время жизни области памяти, содержащей результат выражения, продлевается до конца времени жизни ссылки.

Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 7;
6      const int &ref = a + 4;
7      cout << ref;
8      return 0;
9  }
```

Результат работы программы:

Константные ссылки

Также существуют константные ссылки. Формат объявления константной ссылки:

~~модификатор~~ тип **const** **&**имяСсылки = **имяПеременной**;

где `имяСсылки` – произвольно выбираемый идентификатор, `имяПеременной` – имя уже существующей в программе к моменту объявления ссылки переменной или константы.

Замечание.

По характеру своего действия ссылка на константное значение и константная ссылка ничем не различаются. В обоих случаях они блокируют косвенное изменение через аппарат ссылок значения переменной, на которую ссылаются.

Ссылки на указатели

Так как ссылка не является объектом, то нельзя определить указатель на ссылку, однако можно определить ссылку на указатель. Через подобную ссылку можно изменять значение, на которое указывает указатель или изменять адрес самого указателя.

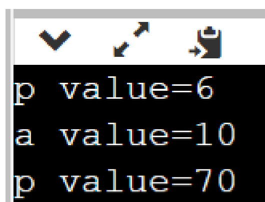
Пример.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int a = 10;
6      int b = 6;
7
```

```
 8      int *p = &a;      //указатель
 9      int *&pRef = p; //ссылка на указатель
10      // компилятор допускает int *(&pRef) = p;
11      pRef = &b;       //через ссылку указателю p
12                       //присваивается адрес переменной a
13      cout << "p value=" << *p << endl;    // 10
14      *pRef = 70;      //изменяем значение по адресу, на
15                       //который указывает указатель
16      cout << "a value=" << a << endl;     // 70
17
18      pRef = &b;      //изменяем адрес, на который
19                      //указывает указатель
20      cout << "p value=" << *pRef << endl;    // 6
21      return 0;
22  }
```

Результат работы программы:



Указатели и ссылки в параметрах функций

Ссылки и указатели на переменные в параметрах функций

Указатели в параметрах функции

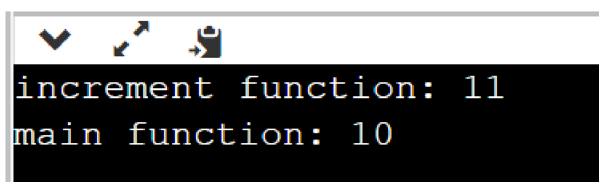
Параметры функции в C++ могут быть указателями. Указатели передаются в функцию по значению, то есть функция получает копию указателя, который будет в качестве значения иметь тот же адрес, что оригинальный указатель. Поэтому используя в качестве параметров указатели, можно получить доступ к значению аргумента и изменить его.

Например, рассмотрим простейшую функцию, которая увеличивает число на единицу.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  //функция не возвращает значение
5  void increment(int n) {
6      n++;
7      cout << "increment function: " << n << endl;
8      //нет return
9  }
10
11 int main() {
12     int a = 10;
13     increment(a);
14     cout << "main function: " << a;
15     return 0;
16 }
```

Результат выполнения программы:



```
increment function: 11
main function: 10
```

В примере выше переменная `a` передается в качестве аргумента для формального параметра `n`, в заголовке функции `increment()`. Передача происходит по значению, поэтому любое изменение локального параметра `n` в функции `increment()` никак не скажется на значении переменной `a` в `main()`.

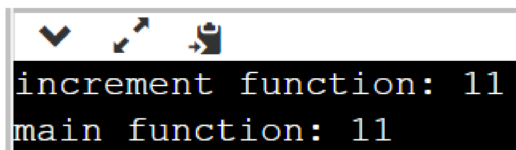
Теперь изменим функцию `increment()`, использовав в качестве параметра указатель:

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
```

```
 4 void increment(int *pn) {
 5     (*pn)++; //изменение значения по адресу
 6     cout << "increment function: " << *pn << endl;
 7 }
 8
 9 int main() {
10     int a = 10;
11     increment(&a); // передача адреса перем. в функ.
12     cout << "main function: " << a;
13     return 0;
14 }
```

Результат работы программы:


```
increment function: 11
main function: 11
```

Поскольку в текущем примере функция `increment()` в качестве параметра принимает указатель, то при ее вызове необходимо передать адрес переменной: `&a`.

В функции `increment()` для изменения значения параметра по адресу, переданному через указатель `pn`, применяется операция разыменования с последующим инкрементом: `(*pn)++`. Это изменяет значение, которое находится по адресу, хранимому в указателе `x`. В итоге изменение внутри функции также повлияет на переменную `a` из `main()`.

Замечание.

Использование указателей в параметрах функций позволяет не только получать, но возвращать без использования оператора `return` *результаты работы функций.*

Ссылки в качестве параметров функций

Ссылки чаще всего используются в качестве параметров в функциях. В этом контексте ссылка-параметр работает как псевдоним аргумента, а сам аргумент не копируется при его передаче. Это в свою очередь улучшает производительность, если аргумент слишком большой или затратный для копирования.

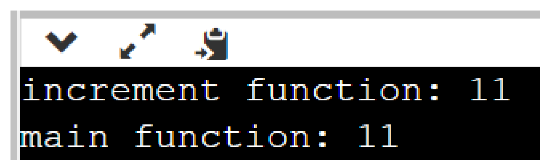
Поскольку ссылка-параметр - это псевдоним аргумента (напомним, что ссылка — это тоже адрес, но с автоматическим разыменованием), то функция,

использующая ссылку-параметр, может изменять аргумент, расположенный непосредственно во фрейме `main()`, как и при использовании указателя (адреса).

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  //параметр функции - ссылка
5  void increment(int &n) {
6      n++;
7      cout << "increment function: " << n << endl;
8  }
9
10 int main() {
11     int a = 10;
12     increment(a);
13     cout << "main function: " << a;
14     return 0;
15 }
```

Результат выполнения программы:



Когда аргумент `a` передается в функцию, то параметр `n` функции `increment()` становится псевдонимом аргумента `a` и ссылается на ту же область памяти, что и `a`. Это позволяет функции изменять значение локальной для `main()` переменной `a` с помощью внешней функции.

Замечание.

Как и в случае с указателями использование ссылок в параметрах функций позволяет не только получать, но и без использования оператора `return` *возвращать результаты работы функций.*

Ссылка и указатель в качестве возвращаемого функцией значения

Ссылки и указатели могут использоваться в качестве возвращаемого функцией значения. Так, для того чтобы:

- вернуть указатель следует записать имя типа (если необходимо с модификаторами) и знак `*` в прототипе и заголовке функции:

```
тип *имяФункции(~~список, форм., парам.~~);
```

- вернуть ссылку следует записать имя типа (если необходимо с модификаторами) и знак `&` в прототипе и заголовке функции:

```
тип &имяФункции(~~список, форм., парам.~~);
```

Замечание.
Нельзя вернуть ссылку на литерал или выражение.

Рассмотрим работу с обычными локальными переменными во ***фрейме*** вызываемой функции. Пусть функция возвращает по ссылке или указателю значение локальной переменной, расположенной во ***фрейме*** стека. В конце работы функции все локальные переменные из ***фрейма*** функции будут удалены (как и сам ***фрейм***), и в точке вызова вызывающая функция получит ссылку или указатель на мусор. При этом компилятор может выдать только предупреждение.

Однако если рассмотреть работу с указателями более детально, то можно обратить внимание, что, если функция будет возвращать указатель на область памяти, выделенную во время ее работы в ***куче***, то результат будет вполне осмысленным, т.е. функция вернет адрес памяти, которая будет продолжать существовать и после удаления ***фрейма***.

Замечание.
Поскольку ссылки в С++ не работают с кучей, то подобное осмысленное возвращение результата по ссылке возможно, только если используется ссылка на указатель.

Ссылки и указатели на константы в параметрах функций

Указатели на константы в параметрах функций

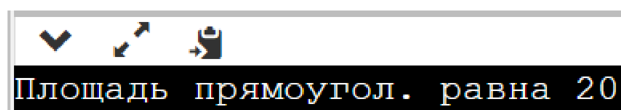
Не сложно переделать предыдущую программу заменив константные ссылки на константные указатели. Просто выполним замены знаков операций и изменим имена формальных параметров функции на имена с префиксом p для указания, что это будут указатели.

Пример.

main.cpp

```
#include <iostream>
using namespace std;

int square(const int *pa, const int *pb) {
    // (*pa) = (*pa) * (*pa); - так нельзя сделать
    // (*pb) = (*pb) * (*pb); - так нельзя сделать
    return (*pa) * (*pb);
}

int main() {
    const int a = 4;
    const int b = 5;
    std::cout<<"Площадь прямоуг. равна "
              << square(&a, &b);
    return 0;
}
```

Результат работы программы:



Отметим, что компилятор допускает еще один вариант использования квалификатора **const** в прототипе функции `square()` - это константные указатели.

Пример.

```
int square(int const *pa, int const *pb);
```

Замечание.

Однако ***константные*** *указатели на* ***константу*** *использовать уже нельзя. Следующее использование квалификатора* **const** *вызовет ошибку компиляции:*

```
int square(const int *const pa, const int *const pb);
```

Ссылки на константы в параметрах функций

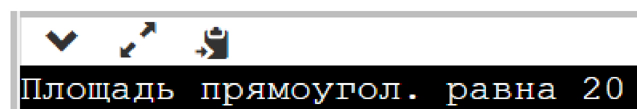
Для того чтобы передать в функцию константу через параметры по ссылке, то эти параметры функции (через которые передается константа) тоже должны представлять собой ссылку на константу.

Пример.

main.cpp

```
1  #include <iostream>
2  
3  int square(const int &a, const int &b) {
4      // a = a * a; - так нельзя сделать
5      // b = b * b; - так нельзя сделать
6      return a * b;
7  }
8  
9  int main() {
10     //объявление двух констант a и b
11     const int a = 4, b = 5;
12     std::cout<<"Площадь прямоугол. равна "
13              << square(a, b);
14     return 0;
15 }
```

Результат работы программы:



Если в функцию необходимо передать большие объекты, которые не должны изменяться, то определение параметров именно как константных ссылок больше всего подходит для данной задачи.

Отметим еще возможный вариант оформления ссылок в параметрах функций с использованием квалификатора **`const`** - это константные ссылки.

Пример.

```
int square(int const &a, int const &b);
```

Обработка в функциях массивов с помощью указателей

Функции и одномерные массивы

Рассмотрим наиболее универсальный метод обработки одномерных массивов с помощью указателей. Его универсальность кроется в том, что в независимости каким образом организован массив (в ***стеке*** или ***куче***) его имя всегда представляет собой указатель. Поэтому продемонстрированные в данном разделе примеры и подходы, используемые для обработки одномерных массивов практически универсальны для всех типов памяти (***стек*** или ***куча***), в которой создаются массивы для хранения значений практически всех типов данных.

Оговорка «практически» насчет типов данных касается строк, заимствованных из языка С. Напомним, что у строк существует маркер конца строки – специальный символ `'\0'`, что может существенно упрощать работу с этим типом массивов.

Универсальность организации обработки одномерных массивов заключается в выполнении двух пунктов:

- имя одномерного массива передается в функцию через параметр типа указатель;
- количество обрабатываемых элементов в массиве (кроме массива символов) через дополнительный целочисленный параметр.

Методика создания разработчиком функции, выводящей на экран инициализированный в `main()` автоматического массива

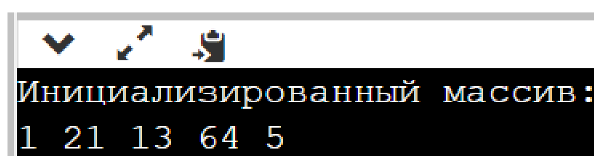
Начнем с программы, которую к настоящему времени должен уметь написать каждый учащийся, а именно с уже разобранного ранее примера вывода на экран инициализированного в `main()` автоматического массива, размещенного в ***стеке***.

Напомним этот пример и далее пошагово перейдем от реализации алгоритма в `main()` к реализации алгоритма в функции.

Пример.

main.cpp
```
1  #include <iostream>
2
3  int main() {
4      float array[] = {1., 21., 13., 64., 5.};
5      int n = sizeof(array) / sizeof(float);
6      std::cout << "Инициализированный массив:"
7                << std::endl;
8      //блок вывода инициализиров. массива на экран
9      for(int i = 0; i < n; i++) {
10         std::cout << array[i] << " ";
11     }
12     return 0;
13 }
```

Результат работы программы:



Первым действием перенесем инициализацию массива и вычисление его размерности `n` перед `main()`. Это действие сделает глобальными массив и переменную `n`, отвечающую за размерность. Подобная перестановка никак не скажется на работоспособности программы:

Пример.

main.cpp

```cpp
#include <iostream>

//глобальные массив и переменная
float array[] = {1., 21., 13., 64., 5.};
int n = sizeof(array) / sizeof(float);

int main() {
    std::cout << "Инициализированный массив:"
              << std::endl;
    //блок вывода инициализиров. массива на экран
    for(int i = 0; i < n; i++) {
        std::cout << array[i] << " ";
    }
    return 0;
}
```

Результат работы программы не изменился.

Переименуем функцию `main()`, например, в `printArray()`) и ниже ее создадим новый `main()`, уже вызывающий эту функцию, созданную разработчиком. После проведенных манипуляций функциональность программы полностью сохраниться.

Пример.

main.cpp

```cpp
#include <iostream>

//глобальные массив и переменная
float array[] = {1., 21., 13., 64., 5.};
int n = sizeof(array) / sizeof(float);

//функция вывода
int printArray() {
    std::cout << "Инициализированный массив:"
              << std::endl;
    //блок вывода инициализиров. массива на экран
```

```
12      for(int i = 0; i < n; i++) {
13          std::cout << array[i] << " ";
14      }
15      return 0;
16  }
17  //новый main()
18  int main() {
19      printArray();    //вызов функции вывода
20      return 0;
21  }
```

Результат работы программы не изменился.

Далее, перенесем вспомогательный `cout` из `printArray()` в `main()`, а также в связи с отсутствием необходимости возвращать значение функцией `printArray()` изменим тип возвращаемого значения на `void`, удалив при этом в ней оператор `return`.

Пример.

main.cpp

```
1   #include <iostream>
2
3   //глобальные массив и переменная
4   float array[] = {1., 21., 13., 64., 5.};
5   int n = sizeof(array) / sizeof(float);
6
7   //функция вывода
8   void printArray() {
9       //блок вывода инициализиров. массива на экран
10      for(int i = 0; i < n; i++) {
11          std::cout << array[i] << " ";
12      }
13  }
14  //новый main()
15  int main() {
16      std::cout << "Инициализированный массив:"
17                << std::endl;
18      printArray();    //вызов функции вывода
19      return 0;
20  }
```

Результат работы программы также не изменился.

Поскольку использование глобальных переменных допускается только в крайних случаях, то перенесем инициализацию массива и переменной, имеющей значение размерности обратно в `main()`. Но для сохранения работоспособности программы уже с локальными для `main()` массивом и переменной, определяющей его размерность, необходимо в заголовок функции `printArray()` добавить два параметра:

- для передачи имени массива – указатель (в данном случае на типа `float`);
- для передачи размерности массива – целое число (тип `int`).

<u>*Пример.*</u>

main.cpp

```
1  #include <iostream>
2  
3  //функция вывода
4  void printArray(float * array, int n) {
5      //блок вывода инициализиров. массива на экран
6      for(int i = 0; i < n; i++) {
7          std::cout << array[i] << " ";
8      }
9  }
10 
11 int main() {
12     //локальные данные: массив и его размерность
13     float array[] = {1., 21., 13., 64., 5.};
14     int n = sizeof(array) / sizeof(float);
15     std::cout << "Инициализированный массив:"
16               << std::endl;
17     printArray(array, n);    //вызов функции вывода
18     return 0;
19 }
```

Результат работы программы останется прежним.

В заключение следует:

- перенести функцию `printArray()`, расположив после `main()`,
- скопировать заголовок, вставить его перед `main()` и оформить прототип.

Таким образом, финальный вариант программы может выглядеть следующим образом:

Пример.

main.cpp

```cpp
#include <iostream>

//прототип
void printArray(float *, int);

int main() {
    //локальные данные: массив и его размерность
    float array[] = {1., 21., 13., 64., 5.};
    int n = sizeof(array) / sizeof(float);
    std::cout << "Инициализированный массив:"
              << std::endl;
    printArray(array, n);   //вызов функции вывода
    return 0;
}
//функция вывода
void printArray(float * array, int n) {
    //блок вывода инициализиров. массива на экран
    for(int i = 0; i < n; i++) {
        std::cout << array[i] << " ";
    }
}
```

Функциональность программы не изменится.

Работа с массивом, размещенным в куче

Пусть у учащегося имеется программа, выполняющая в `main()` следующие действия:

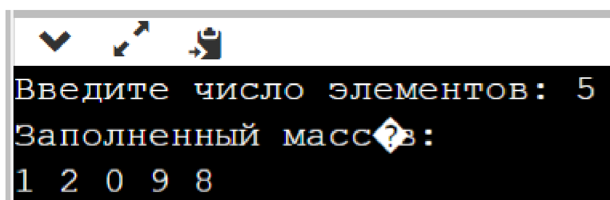
- ввод с клавиатуры размерности массива (целое число);
- выделение памяти в ***куче*** под массив чисел плавающей точкой;
- заполнение массива случайными числами;
- вывод массива на экран.

Пример.

main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    int size; //размерность массива
    std::cout << "Введите число элементов: ";
    std::cin >> size;
    //выделение памяти в куче
    float * array = new float [size];
    //блок заполнения массива псевдослучайными числ.
    srand(time(0));
    for(int i = 0; i < size; i++) {
        array[i] = rand() % 10;
    }
    std::cout<<"Заполненный массив:" << std::endl;
    //блок вывода на экран заполненного массива
    for(int i = 0; i < size; i++) {
        std::cout<< array[i]<<" ";
    }
    //удаление массива в куче
    delete []array;
    return 0;
}
```

Результат работы программы:



Перед учащимся стоит задача выделить в отдельные функции два последних действия:

- заполнение массива случайными числами;
- вывод массива на экран.

Как и в предыдущем параграфе, не нарушив работоспособности программы, сделаем глобальными указатель и переменную, имеющую значение размерности.

Пример.

main.cpp

```
 2  #include <cstdlib>
 3  #include <ctime>
 4  
 5  //глобальные переменные
 6  int size;           //размерность массива
 7  float * array = 0;  //указатель
 8  
 9  int main() {
10      std::cout << "Введите число элементов: ";
11      std::cin >> size;
12      //выделение памяти в куче
13      array = new float [size];
14      //блок заполнения массива псевдослучайными числ.
15      srand(time(0));
16      for(int i = 0; i < size; i++) {
17          array[i] = rand() % 10;
18      }
19      std::cout<<"Заполненный массив:" << std::endl;
20      //блок вывода на экран заполненного массива
21      for(int i = 0; i < size; i++) {
22          std::cout<< array[i]<<" ";
23      }
24      //удаление массива в куче
25      delete []array;
26      return 0;
27  }
```

Результат работы программы не измениться.

Замечание.

В данном примере под неизменностью результата будет предполагаться неизменность формата вывода массива. Однако, очевидно сами значения массива будут меняться случайны образом при каждом новом запуске программы после ее редактирования.

Переименуем функцию `main()` в функцию `userFunction()` и ниже ее создадим новый `main()`, уже вызывающий созданную разработчиком функцию.

Пример.

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

//глобальные переменные
int size;           //размерность массива
float * array = 0;  //указатель

int userFunction() {
    std::cout << "Введите число элементов: ";
    std::cin >> size;
    //выделение памяти в куче
    array = new float [size];
    //блок заполнения массива псевдослучайными числ.
    srand(time(0));
    for(int i = 0; i < size; i++) {
        array[i] = rand() % 10;
    }
    std::cout<<"Заполненный массив:" << std::endl;
    //блок вывода на экран заполненного массива
    for(int i = 0; i < size; i++) {
        std::cout<< array[i]<<" ";
    }
    //удаление массива в куче
    delete []array;
    return 0;
}

int main() {
    userFunction();
    return 0;
}
```

Результат работы программы не измениться.

Перенесем строки с `10` по `13`, а также `24` и `25` и из `userFunction()` в `main()`, а также в связи с отсутствием необходимости возвращать значение функцией `userFunction()` изменим тип возвращаемого значения на `void` с одновременным удалением в ней оператора `return`.

Пример.

main.cpp

```
#include <iostream>
#include <cstdlib>
#include <ctime>

//глобальные переменные
int size;              //размерность массива
float * array = 0;     //указатель

void userFunction() {
    //блок заполнения массива псевдослучайными числ.
    srand(time(0));
    for(int i = 0; i < size; i++) {
        array[i] = rand() % 10;
    }
    std::cout<<"Заполненный массив:" << std::endl;
    //блок вывода на экран заполненного массива
    for(int i = 0; i < size; i++) {
        std::cout<< array[i]<<" ";
    }
}

int main() {
    std::cout << "Введите число элементов: ";
    std::cin >> size;
    //выделение памяти в куче
    array = new float [size];
    userFunction();
    //удаление массива в куче
    delete []array;
    return 0;
}
```

Результат работы программы не измениться.

В связи с тем, что необходимо для повышения универсализации и удобства использования чтобы функция выполняла только одно действие, то очевидно, что функцию `userFunction()` следует формально разделить на две:

- `defArray()` – функцию заполняющую одномерный массив псевдослучайными числами;

- `printArray()` – функцию, выводящую значения массива на экран.

При этом в `main()` необходимо заменить формальный вызов одной функции `userFunction()` последовательным вызовом новых функций `defArray()` и `printArray()`.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  //глобальные переменные
6  int size;             //размерность массива
7  float * array = 0;    //указатель
8
9  void defArray() {
10     //блок заполнения массива псевдослучайными числ.
11     srand(time(0));
12     for(int i = 0; i < size; i++) {
13         array[i] = rand() % 10;
14     }
15 }
16
17 void printArray() {
18     std::cout<<"Заполненный массив:" << std::endl;
19     //блок вывода на экран заполненного массива
20     for(int i = 0; i < size; i++) {
21         std::cout<< array[i]<<" ";
22     }
23 }
24
25 int main() {
26     std::cout << "Введите число элементов: ";
27     std::cin >> size;
28     //выделение памяти в куче
29     array = new float [size];
30     defArray();
31     printArray();
```

```
32     //удаление массива в куче
33     delete []array;
34     return 0;
35 }
```

Результат работы программы не измениться.

Перенесем объявление глобальных переменных (строки 5 – 7) в main(). Это потребует передачи в каждую функцию адреса начала ленты памяти, выделенной под массив, через указатель и размерности массива через целочисленный параметр (в заголовках). Перенесем также вспомогательный cout (строка 18) из функции printArray() в main():

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  void defArray(float * array, int size) {
6      //блок заполнения массива псевдослучайными числ
7      srand(time(0));
8      for(int i = 0; i < size; i++) {
9          array[i] = rand() % 10;
10     }
11 }
12
13 void printArray(float * array, int size) {
14     //блок вывода на экран заполненного массива
15     for(int i = 0; i < size; i++) {
16         std::cout<< array[i]<<" ";
17     }
18 }
19
20 int main() {
21     //локальные переменные
22     int size;              //размерность массива
23     float * array = 0;     //указатель
24     std::cout << "Введите число элементов: ";
25     std::cin >> size;
26     //выделение памяти в куче
```

```
27      array = new float [size];
28      defArray(array, size);
29      std::cout<<"Заполненный массив:" << std::endl;
30      printArray(array, size);
31      //удаление массива в куче
32      delete []array;
33      return 0;
34  }
```

Результат работы программы не измениться.

Перенося определение функций после `main()`, а также оформив их прототипы до `main()` с точностью до обозначений формальных параметров функций окончательно программу можно привести к следующему виду.

Пример.

main.cpp

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <ctime>
4
5   //прототипы
6   void defArray(float *, int);
7   void printArray(float *, int);
8
9   int main() {
10      //локальные переменные
11      int size;               //размерность массива
12      float * array = 0;      //указатель
13      std::cout << "Введите число элементов: ";
14      std::cin >> size;
15      //выделение памяти в куче
16      array = new float [size];
17      defArray(array, size);
18      std::cout<<"Заполненный массив:" << std::endl;
19      printArray(array, size);
20      //удаление массива в куче
21      delete []array;
22      return 0;
23  }
24
```

```
25 void defArray(float * a, int n) {
26     //блок заполнения массива псевдослучайными числ.
27     srand(time(0));
28     for(int i = 0; i < n; i++) {
29         a[i] = rand() % 10;
30     }
31 }
32
33 void printArray(float * a, int n) {
34     //блок вывода на экран заполненного массива
35     for(int i = 0; i < n; i++) {
36         std::cout<< a[i]<<" ";
37     }
38 }
```

Результат работы программы не измениться.

Сравнивая результаты преобразования программ в предыдущем и текущем параграфах, можно заметить, что функция `printArray()` имеет одно и тоже тело. Это и имелось ввиду под универсальностью: массивы организованы по-разному (один – в ***стеке***, а второй – в ***куче***), а их обработка проводится совершенно одинаково.

Заполнение одномерного массива, размещенного в куче с клавиатуры

Больше не будем тратить время на демонстрацию методики «разрезания» одной большой программы на вызовы функций. Далее будем предполагать, что учащийся уже в достаточной степени освоил основы процедурного программирования.

В данном параграфе рассмотрим задачу аналогичную предыдущей (рассмотренной в предыдущем параграфе). Однако в качестве отличия будем предполагать, что массив необходимо заполнить не случайными числами, а используя консольные средства ввода языка С++.

Пример.

main.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 void newDefArray(float *, int);
5 void printArray(float *, int);
```

```
6
7  int main() {
8      int size;
9      cout << "Введите количество элементов"
10         << endl;
11     //ввод размерности
12     cin >> size;
13     //выделение памяти в куче
14     float * array = new float [size];
15     //вызов функции, осуществляющей
16     //заполнение массива значениями
17     //с клавиатуры
18     newDefArray(array, size);
19     cout << "Введенный массив:" << endl;
20     printArray(array, size);
21     delete []array;
22     return 0;
23 }
24
25 void newDefArray(float *a, int n) {
26     for(int i = 0; i < n; i++) {
27         cout << " a["<< i <<"]= ";
28         cin >> a[i];
29     }
30 }
31
32 void printArray(float *a, int n) {
33     for(int i = 0; i < n; i++) {
34         cout << a[i] << " ";
35     }
36 }
```

Результат работы программы:

```
Введите количество элементов
3
 a[0]= 1
 a[1]= 2
 a[2]= 3
Введенный массив:
1 2 3
```

Выделение в отдельной функции памяти в куче под одномерный массив

До настоящего времени выделение памяти в ***куче*** под хранение одномерного массива осуществлялось в `main()`, но эту операцию можно локализовать и в функции, которая будет вызываться из `main()` и возвращать в точку вызова начальный адрес выделенной в ***куче*** памяти. Именно его можно будет присвоить переменной-указателю, далее с которой можно будет работать как с именем массива.

Вначале напишем необходимую функцию. Будем считать, что обрабатываем массив чисел с плавающей точкой.

Пример.

```
float * memoryDef(int n) {
    return new float [n]; //выдел. памяти в куче
}
```

Приведенная функция удовлетворяет всем перечисленным выше требованиям и далее требуется включить ее в программу. Для этого встроим ее в уже известную программу, использующую функцию заполнения одномерного массива псевдослучайными числами.

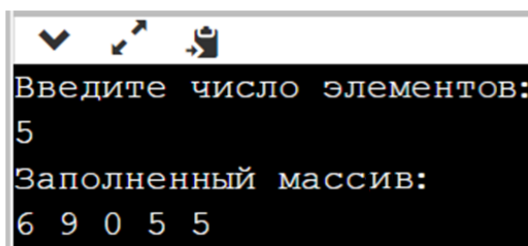
Пример.

main.cpp

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  float * memoryDef(int);
6  void defArray(float *, int);
7  void printArray(float *, int);
8
9  int main() {
10     int size;
11     std::cout << "Введите число элементов"
12               << std::endl;
13     std::cin >> size;
```

```
14      //инициализация указателя адресом ленты
15      //памяти, выделенной с помощью функции
16      //memoryDef()
17      float * array = memoryDef(size);
18      //заполнение массива случайными значениями
19      defArray(array, size);
20      std::cout<< "Заполненный массив:"
21                << std::endl;
22      printArray(array, size);
23      delete []array;
24      return 0;
25  }
26
27  float * memoryDef(int n) {
28      return new float [n]; //выдел. памяти в куче
29  }
30
31  void defArray(float *a, int n) {
32      srand(time(0));
33      for(int i = 0; i < n; i++) {
34          a[i] = rand() % 10;
35      }
36  }
37
38  void printArray(float *a, int n) {
39      for(int i = 0; i < n; i++) {
40          std::cout << a[i]<<" ";
41      }
42  }
```

Результат работы программы:


```
Введите число элементов:
5
Заполненный массив:
6 9 0 5 5
```

Функция выбора минимума в одномерном массиве

Как показали предыдущие три примера достаточно воспользоваться одним из уже написанных каркасных решений, и дополнять его произвольными дополнительными функциями для обработки одномерного массива. Рассмотрим, например, функцию выбора минимума из одномерного массива.

Пример.

```
float minArray(float *a, int n) {
    float min = a[0];
    for(int i = 1; i < n; i++) {
        if (min > a[i]) min = a[i];
    }
    return min;
}
```

Читателю остается расположить прототип функции `minArray()`:

```
float minArray(float *, int);
```

перед `main()`, тело функции скопировать после `main()` и вызвать функцию в `main()`, там, где сочтет нужным разработчик.

Вероятно, читатель уже заметил, что тела функций обработки одномерного массива сильно напоминают алгоритмические блоки из рассмотренных ранее программ, целиком написанных в `main()`. Это так и есть.

Замечание.

Многие еще не созрели для программирования с помощью функций. Это нормально. На первом шаге перехода к программированию с помощью функций можно, сначала написать весь код алгоритма задачи в `main()`. Потом эту огромную программу разрезать на фрагменты кода вставляя их в тела соответствующих функции, а в `main()` заменяя вырезанные фрагменты вызовами этих функций.

Использование указателей на константу в параметрах функций для обработки одномерных массивов

Поскольку при передаче массива передается фактически указатель на первый элемент, то используя этот указатель, можно изменить элементы массива (как, например, в случае с функциями заполнения `defArray()`, `newDefArray()`).

Если же нет необходимости в изменении массива (как, например, с функциями `printArray()`, `minArray()`), то лучше параметр функций, передающий массив (его адрес) выполнить с помощью с помощью указателя на константу.

Пример.

```
void printArray(const float *A, int n) {
    for(int i = 0; i < n; i++) {
        std::cout << A[i] << " ";
    }
}
```

Однако в этом случае не следует забывать и про соответствующие изменения в прототипе:

```
void printArray(const float *A, int n);
```

Замечание.

Если в заголовке функции для указателя использован квалификатор **`const`**, *то в случае попытки изменения элемента массива в теле функции уже на этапе компиляции будет выдано сообщение, что память, хранящая массив, открыта только для чтения.*

Особенность работы с памятью в функциях, получающих массив через параметр типа указатель

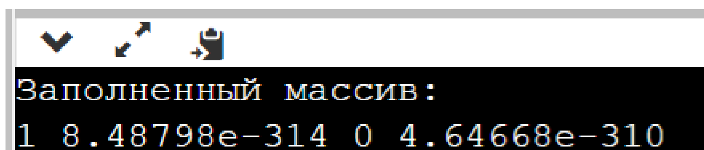
Для демонстрации особенностей обработки массивов с помощью указателей вернемся к уже рассмотренному ранее простейшему примеру вывода на экран с помощью функции массива, созданного в ***стеке***.

Однако внесем в этот пример исправление в функции `printArray()`, которое не повлияет на компилируемость программы - поставим знак минус перед значением индекса в операции разыменования (выделено цветом).

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  void printArray(double *, int);
5
6  int main() {
7      double array[] = {1., 2., 3., 4.};
8      int size = sizeof(array)/sizeof(double);
9
10     cout<<"Заполненный массив:" << endl;
11     printArray(array, size);
12
13     return 0;
14 }
15
16 void printArray(double *a, int n) {
17     for(int i = 0; i < n; i++) {
18         cout<< a[-i]<<" ";
19     }
20 }
```

Результат работы программы:



Программа сохранит свою работоспособность. Но следует разобрать, результат, хотя он очевиден исходя из схемы расположения элементов созданного массива в ***стеке*** и фактического обращения к другим ячейкам памяти с помощью операции индексирования с отрицательным параметром `[-i]` (Рисунок 25).

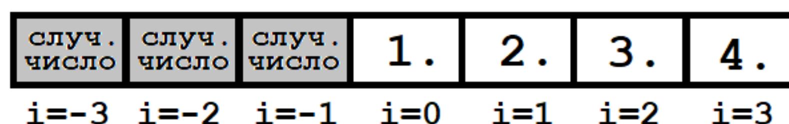


Рисунок 25 – Лента, выделенная под хранение массива (белые ячейки) и ячейки, к которым в действительности происходит обращение (серые ячейки)

Это происходит потому, что для указателя `a`, используемого в параметрах функции `printArray()` разрешены операции обращения с помощью операции индексирования как к памяти справа от указателя имеющего адрес начала массива (формат `[i]`), так и к памяти слева от него (формат `[-i]`). Именно это подчеркивает опасность использования указателей в C++.

Замечание.

Использование квалификатора ***const*** *в параметрах функции* ***не*** *оказывает никакого влияния на возможность обращения через указатель к любой ячейке памяти, расположенной на любом расстоянии вправо или лево относительно начала массива с помощью операции индексирования.*

Дополнительные примеры обработки массивов функциями, не возвращающими результата

Как следует из изложения одно из удобств структурного (процедурного) программирования – это возможность рассмотрения отдельных функций, без привязки к `main()`, а именно их вызовам.

Это уже было продемонстрировано на примере функции выбора минимума и константном указателе. Дополним теперь набор примеров функций для обработки массивов, которые можно использовать при расширении функциональности уже существующих программ, перечисленных выше.

Одним из базовых примеров является функция, обеспечивающая реверс одномерного массива. Она не возвращает результат с помощью оператора `return`. Однако, как и в случае с функцией заполнения массива, множественный результат остается в том же массиве, который приходит на ее «вход». Будем рассматривать массив из ***чисел с плавающей точкой***.

Пример.

```
void reverseArray(float *a, int n) {
    for(int i = 0; i < n / 2; i++) {
        float buf = a[i];
            a[i] = a[n - 1 - i];
            a[n - 1 - i] = buf;
    }
}
```

Вторым базовым примером является пример циклического сдвига ***влево*** одномерного массива на один элемент. Будем рассматривать массив из ***целых чисел***.

Пример.

```
void leftShiftArray(int *a, int n) {
    float temp = a[0];
    for(int i = 0; i < n - 1; i++) {
        a[i] = a[i + 1];
    }
    a[n - 1] = temp;
}
```

Рассмотрим функцию циклического ***сдвига вправо*** одномерного массива ***чисел с плавающей точкой удвоенной точности***.

Пример.

```
void rightShiftArray(double *a, int n) {
    float temp = a[n - 1];
    for(int i = n - 1; i > 0; i--) {
        a[i] = a[i - 1];
    }
    a[0] = temp;
}
```

Пусть перед разработчиком стоит задача написать функцию, сортирующую одномерный массив ***целых чисел*** методом выбора.

Пример.

```
void selectionSort(int *a, int n) {
    for(int i = 0; i < n - 1; i++) {
        //блок выбора минимума по индексу
        int min = i;
        for(int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        //блок перестановки значений
        float temp = a[i];
            a[i] = a[min];
            a[min] = temp;
    }
}
```

Использование указателей на начало и конец одномерного массивов любого типа

Еще один подход к организации обработки одномерных массивов заключается в передаче указателя не только на начало, но и на конец массива. Для этого следует использовать встроенные библиотечные функции `begin()` и `end()`. Эти имена определены в стандартном пространстве имен (`std`).

Рассмотрим уже известный пример, касающийся вывода на экран инициализированного в `main()` одномерного массива чисел с плавающей точкой и внесем необходимые изменения в программу.

Пример.

main.cpp

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  //прототип функции с двумя указателями
 5  void printArray(float*, float*);
 6
 7  int main() {
 8      float array[] = {1., 21., 13., 64., 5.};
 9      float * beginArray = begin(array);
10      float * endArray = end(array);
11      cout << "Инициализированный массив:"
12           << endl;
13      printArray(beginArray, endArray);
14      return 0;
15  }
16
17  void printArray(float *begin, float *end) {
18      for(float *ptr = begin; ptr < end; ptr++) {
19          cout << *ptr << " ";
20      }
21  }
```

Результат работы программы в точности совпадает с уже обсуждавшимся ранее результатом, полученным с помощью функции,

обрабатывающей одномерный массив с помощью указателя и целочисленного параметра, указывающего количество элементов.

Замечание.

Функции `begin()` *и* `end()` *можно использовать только для автоматических массивов, память для которых выделяется в* ***стеке****.*

Отметим, что синтаксис предыдущего примера можно сделать чуть более профессиональным (в частности, без использования инструкции `using`).

Пример.

main.cpp

```cpp
#include <iostream>

//прототип функции с двумя указателями
void printArray(float*, float*);

int main() {
    float array[] = {1., 21., 13., 64., 5.};
    std::cout << "Инициализированный массив:"
              << std::endl;
    printArray(std::begin(array), std::end(array));
    return 0;
}

void printArray(float *begin, float *end) {
    for(float *ptr = begin; ptr < end; ptr++) {
        std::cout << *ptr << " ";
    }
}
```

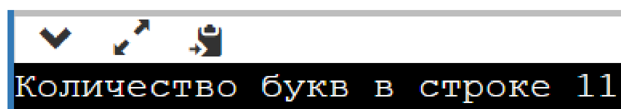
Определение длины строки встроенного типа с помощью арифметики указателей

Дополнительно к уже обсужденному ранее алгоритму определения длины строки, основанному на изменении индекса символа в строке, представляемой в виде массива символов, приведем еще один возможный вариант реализации этого алгоритма с помощью арифметики указателей.

Пример.

main.cpp
```cpp
#include <iostream>

int strLenPtr(char*);

int main() {
    char str[] = "Hello World";
    std::cout << "Количество букв в строке "
              << strLenPtr(str);
    return 0;
}

int strLenPtr(char* str) {
    char* ptr = str;
    for(; (*ptr) !='\0'; ptr++);
    return ptr - str;
}
```

Результат работы программы:



Перевод целого числа в массив символов

Представляется интересным в учебных целях продемонстрировать каким образом можно реализовать алгоритм перевода числа в строку. Данный алгоритм иногда обсуждается на форумах, но далеко не всегда приводимые там решения имеют законченный работоспособный вид.

Особенностью предлагаемого решения является то, что алгоритм реализуется с помощью цепочки вызовов функций, т.е. после ***начального*** перевода числа в массив символов с помощью функции `intToArrayChar()` для получения ***финального*** результата созданный ***начальный*** массив следует зеркально переставить (выполнить реверс) с помощью функции `reversArrayChar()`.

Замечание.

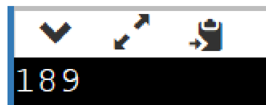
Из-за того, что в примере используются две функции с достаточно развитым кодом, то для повышения читаемости программы код функции `main()` *будет выделен цветом.*

Пример.

main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <cstring>

//предполагаемое наибольшее число
//позиций в целом числе
const int N_ID = 6;

void intToArrayChar(char*, int);
void reversArrayChar(char*);

int main() {
    char number[N_ID + 1];
    intToArrayChar(number, 189);

    std::cout << number;
    return 0;
}

void intToArrayChar(char* str, int m) {
    int n = 0;
    while ( (m <= (int) pow(10, N_ID)) && m > 0) {
        str[n] = m % 10 + '0';
        m = m / 10;
        n++;
    }
    str[n] = '\0';
    reversArrayChar(str);
}

void reversArrayChar(char* str) {
    int n = strlen(str);
    for(int i = 0; i < n / 2 ; i++) {
        char temp = str[i];
        str[i] = str[n - 1 - i];
        str[n - 1 - i] = temp;
    }
}
```

Результат работы программы:



Использование метода `getline()` потока `cin` для ввода строки

Метод `getline()` предназначен для ввода данных из потока, например, для ввода данных из консольного окна. Если формально описывать его функционал, то он извлекает данные из входного потока ***до строкового разделителя***, который не записывается в получившийся массив данных.

В итоге, получается извлечение одной строки и присвоение ее значения указателю на символ. Сама конструкция вызова `getline()` как метода `cin` выглядит так:

```
cin.getline(строка, количество, разделитель);
```

где `строка` – переменная типа `char*`, в которую запишется строка, `количество` – максимально количество символов, которое может быть записано в строку, и ~~`разделитель`~~ – строковый разделитель, показывающий на конец строки. Последний параметр функции можно опустить, тогда будет задан сепаратор по умолчанию - `'\n'`.

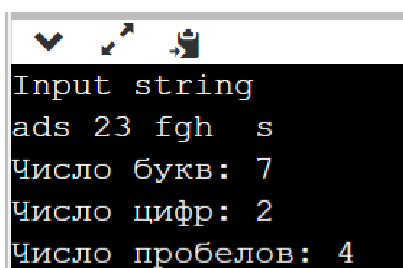
Рассмотрим программу, которая с ***помощью функций*** вводит строку с клавиатуры, считает в ней количество букв, цифр, и пробелов. Результаты подсчетов выводит на экран.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int LeterNumber(char *);
5  int DigitNumber(char *);
6  int SpaceNumber(char *);
7
8  int main() {
9      const int STRLEN = 25;
10     char *str = new char[STRLEN];
11     cout << "Input string" << endl;
```

```
12      cin.getline(str, STRLEN);
13      cout<<"Число букв: "
14          << LeterNumber(str)<<endl
15          <<"Число цифр: "
16          << DigitNumber (str)<<endl
17          <<"Число пробелов: "
18          << SpaceNumber(str)<<endl;
19      delete []str;
20      return 0;
21  }
22
23  int LeterNumber(char *string) {
24      int Number = 0;
25      for(int i = 0; string[i]!='\0'; i++) {
26          if((string[i]>= 'A' && string[i]<= 'Z') ||
27              (string[i]>= 'a' && string[i]<= 'z')) Number++;
28      }
29      return Number;
30  }
31
32  int DigitNumber(char * string) {
33      int Number = 0;
34      for(int i = 0; string[i]!='\0'; i++) {
35          if(string[i]>= '0' && string[i]<= '9') Number++;
36      }
37      return Number;
38  }
39
40  int SpaceNumber(char * string) {
41      int Number = 0;
42      for(int i = 0; string[i]!='\0'; i++) {
43          if(string[i]==' ') Number++;
44      }
45      return Number;
46  }
```

Результат работы программы:



Особенности попеременного ввода числовых и строковых данных с помощью `cin`

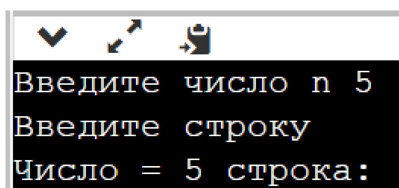
Рассмотрим задачу: пусть при выполнении программы необходимо ввести сперва некоторое число, а затем строку. Теоретически данную задачу должен решить следующий простейший код:

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      const int STRLEN = 25;
5      char * str = new char[STRLEN];
6      int n;
7      //ввод числа
8      std::cout << "Введите число n ";
9      std::cin >> n;
10     //ввод строки
11     std::cout << "Введите строку" << std::endl;
12     std::cin.getline(str, STRLEN);
13     //вывод результатов ввода
14     std::cout << "Число = " << n
15               << " строка: " << str;
16     return 0;
17 }
```

Результат работы программы:



После запуска программы произойдет следующее:

- на экране появиться сообщение `"Введите число n "`;
- после ввода любого числового значения с помощью `cin` будет прочитано число, но в буфере ввода останется символ перевода каретки на новую строку;

- после вывода на экран второго сообщения `"Введите строку"` застрявший символ в буфере `cin` будет прочитан с помощью метода `getline()` (строка, выделенная красным);
- управление программой будет сразу передано на заключительный `cout` и пользователь не сможет ввести следующее строковое значение.

Для исправления ситуации можно использовать метода `ignore()` либо без параметров (`cin.ignore();`), либо с параметрами (в контексте рассматриваемого примера `cin.ignore(STRLEN, '\n');`).

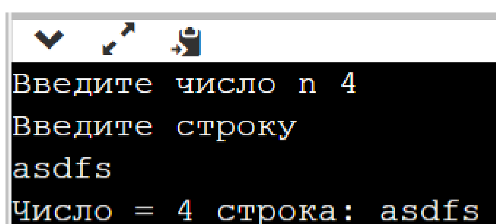
Исправление в коде выделено цветом и позволяет проигнорировать не только «застрявший» после ввода числа в буфере символ, но во втором случае и целую строку, завершающуюся управляющим символом `'\n'`.

Пример.

main.cpp

```
1  #include <iostream>
2
3  int main() {
4      const int STRLEN = 25;
5      char * str = new char[STRLEN];
6      int n;
7      //ввод числа
8      std::cout << "Введите число n ";
9      std::cin >> n;
10     std::cin.ignore(STRLEN, '\n');
11     //ввод строки
12     std::cout << "Введите строку" << std::endl;
13     std::cin.getline(str, STRLEN);
14     //вывод результатов ввода
15     std::cout << "Число = " << n
16               << " строка: " << str;
17     return 0;
18 }
```

Результат работы программы с исправленным вводом:



Перевод строки-числа в числовой формат

На самом деле пример в предыдущем разделе, демонстрирующий как бороться с погрешностями работы `cin` при вводе разных типов данных является своеобразным «костылем», который все равно может допускать погрешности ввода, хотя и менее значительные. Так, например, в некоторых случаях использование метода `ignore()` полностью исправляет ситуацию, но в некоторых случаях появляется такая ошибка ввода, которая выражается в «глотании» первой буквы строки после применения `ignore()`.

С этим можно столкнуться на лабораторных занятиях по работе с абстрактными типами, такими как структуры и классы. Поскольку структуры и объекты абстрактных типов в соответствии с постановкой задачи будут содержать разные базовые типы данных, то возникнет необходимость бороться с переключением `cin` с ввода чисел на ввод строк.

Для того чтобы повысить надежность ввода данных различных типов следует с помощью `cin` осуществлять ввод только строк, а затем строки-числа переводить из строкового в числовой формат. В этом случае программист защитит себя ото всех возможных погрешностей работы `cin` и его методов.

Программа определяющая может ли введенная строка быть числом

Сначала рассмотрим пример однофайловой программы, определяющей, может ли введенная строка быть преобразована в число. В данном случае очевидно, что достаточно с помощью цепочки вызовов функций разработчика проверить входит ли в состав строки хотя бы один символ-цифра.

Задачу решим с помощью двух функций:

- `firstIndexCh()` – функция определяющая, какой индекс в полученной строке имеет переданный символ;
- `isNumber()` – функции делающей заключение по результатам работы `firstIndexCh()` может ли являться числом полученная строка.

Пример.

main.cpp

```
1 #include <iostream>
2 #include <string.h>          //для strlen()
3 using namespace std;
```

```
 4
 5  //функция определенния индекса первого
 6  //вхождения заданного символа по порядку слева
 7  int firstIndexCh(char* str, char ch) {
 8      int i, n = strlen(str);
 9      for(i = 0; (str[i] != ch) && i < n; i++);
10      return i;
11  }
12
13  //функция, определяющая, что строка может
14  //быть числом
15  bool isNumber(char* str) {
16      char validChar[] = "0123456789";
17      int n = strlen(validChar);
18      for(int i = 0; i < n; i++) {
19          if (firstIndexCh(validChar, str[i]) < n) {
20              return 1;
21          }
22      }
23      return 0;
24  }
25
26  int main() {
27      const int STRLEN = 25; //размер вводимой строки
28      char* str = new char[STRLEN];
29      while(true) {
30          //ввод строки
31          cout<<"Введите строку, содержащую цифры: ";
32          cin.getline(str, STRLEN);
33          if (isNumber(str)) break;
34          cout <<"\nВвед. строка не может быть числом\n";
35          cout <<"\nПопробуйте еще раз!!!\n";
36      }
37      cout<< "Строка может быть числом";
38      return 0;
39  }
```

Результат работы программы:

```
Введите строку, содержащую цифры: 123dfg354
Строка, может быть числом
```

Далее из приведенной выше программы впервые будет создан многофайловый проект «по умолчанию» (или безымянный проект) в среде `OnlineGBD beta` (https://www.onlinegdb.com/online_c++_compiler).

Однако для этого следует преобразовать к стандартному виду с объявлением прототипов перед `main()` и описанием самих функций после `main()`.

Пример.

main.cpp

```
1   #include <iostream>
2   #include <string.h>          //для strlen()
3   using namespace std;
4
5   //функция определенния индекса первого
6   //вхождения заданного символа по порядку слева
7   int firstIndexCh(char* , char);
8
9   //функция, определяющая, что строка может
10  //быть числом
11  bool isNumber(char*);
12
13  int main() {
14      const int STRLEN = 25; //размер вводимой строки
15      char* str = new char[STRLEN];
16      while(true) {
17          //ввод строки
18          cout<<"Введите строку, содержащую цифры: ";
19          cin.getline(str, STRLEN);
20          if (isNumber(str)) break;
21          cout <<"\nВвед. строка не может быть числом\n";
22          cout <<"\nПопробуйте еще раз!!!\n";
23      }
24      cout<< "Строка может быть числом";
25      return 0;
26  }
27
28  //функция определенния индекса первого
29  //вхождения заданного символа по порядку слева
30  int firstIndexCh(char* str, char ch) {
31      int i, n = strlen(str);
32      for(i = 0; (str[i] != ch) && i < n; i++);
33      return i;
34  }
```

```
35
36 //функция, определяющая, что строка может
37 //быть числом
38 bool isNumber(char* str) {
39     char validChar[] = "0123456789";
40     int n = strlen(validChar);
41     for(int i = 0; i < n; i++) {
42         if (firstIndexCh(validChar, str[i]) < n) {
43             return 1;
44         }
45     }
46     return 0;
47 }
```

Результат работы программы остается таким же, как и ранее.

Подчеркнем, что

- красным выделена часть кода, содержащая прототипы функций;
- голубым – подключение заголовочного файла `string.h` (для возможности использования функции `strlen()`) и описание (определение) всех функций разработчика, соответствующих списку прототипов.

Таким образом программа разделена на три фрагмента, которые в дальнейшем с небольшими поправками будут преобразованы в отдельные файлы многофайлового проекта.

Общие сведения о многофайловых проектах

В многофайловых проектах императивного программирования в языке С++ используется определенная иерархия файлов:

- все прототипы функций, макроопределения и глобальные константы выносятся в заголовочные файлы (расширение `.h`);
- все описания функций выносятся в файлы с расширением `.cpp`;
- следует (но не обязательно) использовать одинаковые имена заголовочных файлов и соответствующих им `cpp`-файлов;
- программа (функция `main()`) в единственном числе остается в файле `main.cpp`;
- перед `main()` пишется набор необходимых директив препроцессора (в частности, с помощью `#include` подключаются необходимые заголовочные файлы).

При компиляции многофайлового проекта на первом шаге исполняются директивы препроцессора во всех `cpp`-файлах и формируется объединенный объектный код программы.

Использование многофайловой структуры позволяет несопоставимо облегчить работу коллектива программистов, каждый из которых разрабатывает и тестирует свой набор функций (в рамках собственного cpp-модуля), а также пишет для своего cpp-файла (cpp-модуля) еще и заголовочный файл (.h).

В этом случае сборка проекта осуществляется простым копированием отдельных файлов (модулей) в общий проект, дописыванием директив #include в файл, содержащий функцию main() а также последовательных вызовов новых функций в main().

Использование многофайловых проектов дает возможность:

- доработки каждого из файлов в отдельности;
- подключения дополнительных файлов (расширения функциональности).

При этом сохраняется персональная ответственность каждого из разработчиков за качество работы функций, собранных в персонально разрабатываемых модулях.

Добавление файла в безымянный проект онлайн IDE OnlineGDB beta

Файл в проект «по умолчанию» (или безымянный проект) онлайн среды OnlineGDB beta добавляется нажатием на кнопку New File (Ctrl+M) (Рисунок 26).

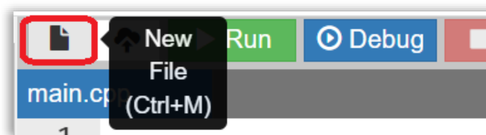


Рисунок 26 – Кнопка New File (Ctrl+M)

Далее появляется окно New File (Рисунок 27), в поле которого (выделено красным) следует написать имя файла и его расширение. Напомним, что в проекте могут использоваться файлы с расширением .h и .cpp.

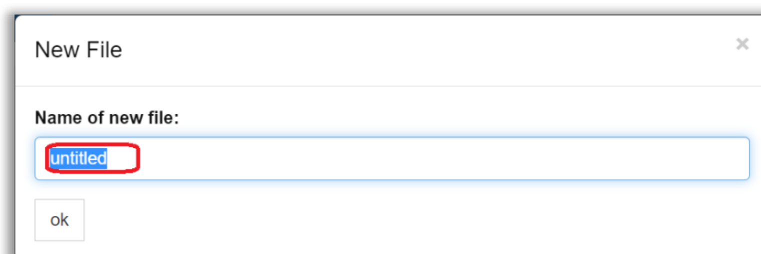


Рисунок 27 – Окно New File

По завершению ввода имени файла и расширения следует нажать на кнопку `ok` (или клавишу `Enter` на клавиатуре). В проекте появиться новая закладка, соответствующая имени созданного файла. Например, если в поле ввода окна `New File` (Рисунок 27) ввести название `example.cpp`, то результат операции будет иметь вид, указанный на рисунке ниже (Рисунок 28).

Рисунок 28 – Созданная закладка в безымянном проекте с введенным именем файла

Базовый многофайловый проект

Используя последний пример программы и сведения, касающиеся создания многофайловых проектов в среде `OnlineGDB beta`, создадим базовый проект.

Замечание.

Слово «базовый» означает, что в следующем параграфе функциональность этого проекта будет расширена.

По сути дела, базовый проект создается из выделенных цветом фрагментов кода предыдущей программы. Структура базового проекта имеет вид (Рисунок 29):

- файл `main.cpp` будет содержать необходимые директивы препроцессора, а также функцию `main()`, в которой с помощью вызова функции `isNumber()` выводится на экран сообщение можно ли введенную строку интерпретировать как число;
- файл `info.h` содержит прототипы двух функций:
  - `int firstIndexCh(char*, char);`
  - `bool isNumber(char*);`
- файл `info.cpp` содержит описание двух функций:
  - `int firstIndexCh(char*, char);`
  - `bool isNumber(char*);`

Рисунок 29 – Структура базового проекта

Пример.

//файл проекта main.cpp

main.cpp | info.h | info.cpp

```
#include <iostream>
#include "info.h"
using namespace std;

int main() {
    const int STRLEN = 25; //размер вводимой строки
    char* str = new char[STRLEN];
    while(true) {
        //ввод строки
        cout<<"Введите строку, содержащую цифры: ";
        cin.getline(str, STRLEN);
        if (isNumber(str)) break;
        cout <<"\nВвед. строка не может быть числом\n";
        cout <<"\nПопробуйте еще раз!!!\n";
    }
    cout << "Строка, может быть числом ";

    return 0;
}
```

//файл проекта info.h

main.cpp | info.h | info.cpp

```
//функция определенния индекса первого
//вхождения заданного символа по порядку слева
int firstIndexCh(char* , char);

//функция, определяющая, что строка может
//быть числом
bool isNumber(char*);
```

//файл проекта info.cpp

main.cpp | info.h | info.cpp

```
#include <string.h>

//функция определенния индекса первого
//вхождения заданного символа по порядку слева
int firstIndexCh(char* str, char ch) {
```

```
 6      int i, n = strlen(str);
 7      for(i = 0; (str[i] != ch) && i < n; i++);
 8      return i;
 9  }
10
11  //функция, определяющая, что строка может
12  //быть числом
13  bool isNumber(char* str) {
14      char validChar[] = "0123456789";
15      int n = strlen(validChar);
16      for(int i = 0; i < n; i++) {
17          if (firstIndexCh(validChar, str[i]) < n) {
18              return 1;
19          }
20      }
21      return 0;
22  }
```

Результат работы программы остается прежним.

Расширение функциональности проекта

Расширим функциональность базового проекта за счет разработки дополнительного модуля, содержащего функцию подготовки строки-числа к представлению в виде целого числа.

В данном случае в результате работы новой функции будет сохраняться строковый формат хранения числа, однако удаляются все символы, не являющиеся цифрами.

Знак минус будет сохраняться только если он имеет нулевой индекс в строке. В любых других случаях он будет удаляться.

Таким образом проект будет дополнен тремя файлами (Рисунок 30):

- `integerStr.h` – заголовочный файл, содержащий два прототипа функций: `deleteCh()`, `intDelInvalid()`;
- `integerStr.cpp` – файл, содержащий описание двух функций: `deleteCh()`, `intDelInvalid()`;
- `validChar.h` – заголовочный файл, в который будет вынесено общее определение одноименного массива цифр.

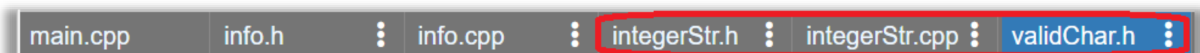


Рисунок 30 – Структура дополненного проекта

Замечание.

Хотя файл `info.h` *расширяемого проекта останется без изменений из базового проекта, но для полноты изложения, чтобы не запутать читателя он также будет продублирован в примере.*

Пример.

//файл проекта main.cpp - изменения в коде базового //проекта, выделены красным

main.cpp | info.h | info.cpp | integerStr.h | integerStr.cpp | vali

```
1  #include <iostream>
2  #include <string.h>          //для функции atoi()
3  #include "info.h"
4  #include "integerStr.h"
5  using namespace std;
6
7  int main() {
8      const int STRLEN = 25; //размер вводимой строки
9      char* str = new char[STRLEN];
10     while(true) {
11         //ввод строки
12         cout<<"Введите строку, содержащую цифры: ";
13         cin.getline(str, STRLEN);
14         if (isNumber(str)) break;
15         cout <<"\nВвед. строка не может быть числом\n";
16         cout <<"\nПопробуйте еще раз!!!\n";
17     }
18     //удаление лишних символов
19     intDelInvalid(str);
20     cout << "Целое число в виде строки: "
21          << str << endl
22          << "Значение типа int : " << atoi(str);
23     return 0;
24 }
```

//файл проекта info.h

main.cpp | info.h | info.cpp | integerStr.h | integer

```
1  //функция определенния индекса первого
2  //вхождения заданного символа по порядку слева
3  int firstIndexCh(char* , char);
4
5  //функция, определяющая, что строка может
```

```
6  //быть числом
7  bool isNumber(char*);
```

//файл проекта info.cpp - изменения в коде базового
//проекта, выделены красным

main.cpp | info.h | info.cpp | integerStr.h | integerStr.cpp

```
1  #include <string.h>
2
3  //функция определенния индекса первого
4  //вхождения заданного символа по порядку слева
5  int firstIndexCh(char* str, char ch) {
6      int i, n = strlen(str);
7      for(i = 0; (str[i] != ch) && i < n; i++);
8      return i;
9  }
10
11 //функция, определяющая, что строка может
12 //быть числом
13 bool isNumber(char* str) {
14     #include "validChar.h"
15     int n = strlen(validChar);
16     for(int i = 0; i < n; i++) {
17         if (firstIndexCh(validChar, str[i]) < n) {
18             return 1;
19         }
20     }
21     return 0;
22 }
```

//файл проекта integerStr.h

main.cpp | info.h | info.cpp | integerStr.h | i

```
1  //функция удаляющая i-й символ во введеной
2  //строке
3  void deleteCh(char*, int);
4
5  //удаление цифр не являющихся цифрами
6  void intDelInvalid(char*);
```

//файл проекта integerStr.cpp

```
main.cpp | info.h | info.cpp | integerStr.h | integerStr.cpp
```

```
1   #include <string.h>
2   #include "info.h" //для функции firstIndexCh()
3
4   //функция удаляющая i-й символ во введеной
5   //строке
6   void deleteCh(char* str, int j) {
7       int n = strlen(str);
8       for(int i = j; i < n; i++){
9           str[i] = str[i + 1];
10      }
11  }
12
13  //удаление цифр не являющихся цифрами
14  void intDelInvalid(char* str) {
15      #include "validChar.h"
16      int n = strlen(validChar);
17      //инициализация индекса символа начала введенной
18      //строки в зависимости от знака числа
19      int i = (str[0] == '-'? 1: 0);
20      for(; i < strlen(str); ){
21          if(firstIndexCh(validChar, str[i]) < n) i++;
22          else deleteCh(str, i);
23      }
24  }
```

//файл проекта validChar.h

```
o.cpp | integerStr.h | integerStr.cpp | validChar.h
```

```
1   char validChar[] = "0123456789";
```

Результат работы программы:

```
Введите строку, содержащую цифры: --123wer34
Целое число в виде строки: -12334
Значение типа int : -12334
```

<u>Замечание.</u>

Проект по переводу строки, стожащей символы-цифры, в число доступен по ссылке URL: https://www.onlinegdb.com/edit/UydJ5EbHG (дата доступа 02.02.2023)

Обработка двумерных массивов в функциях

Язык С++ допускает различные варианты передачи двумерного массива через указатели в функцию, но в данном разделе будем использовать традиционную методику:

- двумерный массив создается в `main()` в виде автоматического, созданного в ***стеке***, либо память под массив выделяется в ***куче***;
- имя двумерного массива передается в вызываемую функцию через параметр «указатель-на-указатель»,
- размерность двумерного массива передается через:
  - один целочисленный параметр (если двумерный массив квадратный);
  - два целочисленных параметра (если массив прямоугольный).

Пример обработки двумерного массива с помощью функций

Будем предполагать, что поставлена задача в `main()`:

- ввести с клавиатуры целое число - размерность квадратного двумерного массива;
- выделить в ***куче*** память под двумерный массив целых чисел;

далее с помощью функций разработчика:

- инициализировать двумерный массив псевдослучайными целыми числами;
- вывести исходный двумерный массив на экран;
- выполнить транспонирование квадратного двумерного массива;
- вывести полученный двумерный массив на экран;
- очистить память в ***куче*** от двумерного массива.

Многофайловый проект будет состоять из следующих файлов (Рисунок 31):

- `main.cpp` – содержит подключение необходимых файлов, инструкцию `using`, а также собственно программу (функцию `main()` или так называемую «точку входа»);
- `userMatrix.h` – содержит прототипы используемых функций:
  - `void initMatrix (int**, unsigned);` – прототип функции инициализирующая двумерный массив псевдослучайными числами;
  - `void displayMatrix (int**, unsigned);` – прототип функции выводящей двумерный массив на экран;

- o `void transpouseMatrix (int**, unsigned);` - прототип функции транспонирующей квадратный двумерный массив;
  - o `void freeMemory (int**, unsigned);` – прототип функции освобождающей память в ***куче*** от двумерного массива.
- `userMatrix.cpp` – содержит необходимые директивы `#include`, а также описания всех функций, прототипы которых перечислены в заголовочном файле

Рисунок 31 – Структура многофайлового проекта для транспонирования двумерного массива

Пример.

//файл main.cpp

main.cpp | userMatrix.h | userMatrix.cpp

```
1  #include <iostream>
2  #include "userMatrix.h"
3  using namespace std;
4
5  int main(void) {
6      unsigned size;
7      cout<<"Введите количество строк/столбцов: ";
8      cin>> size;
9
10     //выделение в куче памяти в main()
11     int** matrix = new int* [size];
12     for (int i = 0; i < size; i++ ) {
13         matrix[i] = new int [size];
14     }
15     //инициализация массива псевдосл. числами
16     initMatrix (matrix, size);
17
18     //вывод на экран исходной матрицы
19     cout<<"\tИсходная матрица\n";
20     displayMatrix(matrix, size);
21
22     //транспонирование матрицы
23     transpouseMatrix(matrix, size);
24
```

```
25      //вывод на экран результатов транспонирования
26      cout<<"\tТранспонированная матрица\n";
27      displayMatrix(matrix, size);
28
29      //освобождение памяти в куче
30      freeMemory (matrix, size);
31      return 0;
32  }
```

//файл userMatrix.h

main.cpp | userMatrix.h | userMatrix.cpp

```
1  void initMatrix (int** , unsigned);
2  void displayMatrix (int** , unsigned);
3  void transpouseMatrix (int** , unsigned);
4  void freeMemory (int** , unsigned);
```

//файл userMatrix.cpp

main.cpp | userMatrix.h | userMatrix.cpp

```
1   #include <iostream>
2   #include <cstdlib>
3   #include <ctime>
4
5   void initMatrix (int **a, unsigned n) {
6       srand(time(0));
7       for (int i = 0; i < n; i++ ) {
8           for (int j = 0; j < n; j++ ) {
9               a[i][j] = rand() % 10;
10          }
11      }
12  }
13
14  void displayMatrix (int **a, unsigned n) {
15      for (int i = 0; i < n; i++ ) {
16          for (int j = 0; j < n; j++ ) {
17              std::cout<<a[i][j]<<" ";
18          }
19          std::cout<< std::endl;
20      }
21  }
22
```

```
23  void transpouseMatrix (int **a, unsigned n) {
24      for (int i = 0; i < n; i++) {
25          for (int temporary, j = i + 1; j < n; j++) {
26              temporary = a[i][j];
27              a[i][j] = a[j][i];
28              a[j][i] = temporary;
29          }
30      }
31  }
32
33  void freeMemory (int **a, unsigned n) {
34      for (int i = 0; i < n; i++ ) {
35          delete []a[i];
36      }
37      delete []a;
38  }
```

Результат работы программы:

```
Введите количество строк/столбцов: 4
        Исходная матрица
5  3  2  1
5  2  9  4
5  4  2  5
3  8  4  2
        Транспонированная матрица
5  5  5  3
3  2  4  8
2  9  2  4
1  4  5  2
```

Замечание.

Проект по заполнению двумерного массива псевдослучайными числами и его транспонирования доступен по ссылке URL: https://www.onlinegdb.com/edit/khMN7KW07 (дата доступа 02.02.2023)

Выделение в куче с помощью отдельной функции памяти под двумерный массив

В предыдущем примере выделение в ***куче*** памяти под хранение одномерного массива осуществлялось соответствующими операциями и

операторами непосредственно в `main()`. Но иногда эти действия следует локализовать в функции, которая будет возвращать начальный адрес выделенной в ней памяти в `main()` и его можно будет присвоить переменной ***указателю-на-указатель***. Далее с ней можно будет работать как с именем двумерного массива. Будем считать, что обрабатываем двумерный массив целых чисел.

Пример.

```
int** memoryDefMatrix(int n) {
    int** matrix = new int* [n];
    for (int i = 0; i < n; i++ ) {
        matrix[i] = new int [n];
    }
    return matrix;
}
```

Приведенная функция удовлетворяет всем перечисленным выше требованиям и далее требуется включить ее в программу. Но для этого необходимо не забыть про прототип:

```
int** memoryDefMatrix(int);
```

Детально рассматривать процесс использования данной функции в предыдущей программе не будем, т.к. он совершенно аналогичен случаю, рассмотренному для одномерного массива. Вкратце в коде примера укажем только изменения, которые необходимо буде сделать в файлах предыдущего проекта (доступного по ссылке URL: https://www.onlinegdb.com/edit/khMN7KW07, дата доступа 02.02.2023).

Пример.

```
//изменения в файле main.cpp указанного проекта
main.cpp | userMatrix.h | userMatrix.cpp
#include <iostream>
#include "userMatrix.h"
using namespace std;

int main(void) {
    unsigned size;
    cout<<"Введите количество строк/столбцов: ";
```

```
    cin>> size;

    //выделение в куче памяти в main()
    int** matrix = memoryDefMatrix(size);

    //инициализация массива псевдосл. числами
    initMatrix (matrix, size);

    ... //далее без изменений

    return 0;
}
```

```
//изменения в файле userMatrix.h указанного проекта
```

```
int** memoryDefMatrix(int);
void initMatrix (int** , unsigned);
...//далее без изменений
```

```
//изменения в файле файл userMatrix.cpp указанного
//проекта
```

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int** memoryDefMatrix(int n) {
    int** matrix = new int* [n];
    for (int i = 0; i < n; i++ ) {
        matrix[i] = new int [n];
    }
    return matrix;
}
...//далее без изменений
```

Результат работы данного проекта не отличается от результатов работы исходного исправляемого варианта кода.

Передача в функцию автоматических массивов по ссылке

В случае если программист использовал в программе автоматический одномерный массив, то напомним, что память для его хранения выделяется в ***стеке***.

Обработать этот массив в функции можно двумя способами:

- передать его начало как указатель на ленту памяти, выделенную для его хранения используя имя массива (в этом случае это указатель-константа);
- либо передать этот массив функцию по ссылке.

Первый вариант уже был рассмотрен ранее. Поэтому представляет интерес собственно каким образом можно передать одномерный массив, расположенный в ***стеке*** для обработки в функции.

Рассмотрим пример формального синтаксиса реализации второго способа организации вычислений.

Использование ссылки на одномерный автоматический массив

Будем предполагать, что поставлена задача в `main()`:

- создать автоматический массив с наибольшей возможной размерностью (константа `N`);
- ввести с клавиатуры целое число, определяющую «активную часть» автоматического одномерного массива;

далее с помощью функций разработчика:

- инициализировать «активную часть» одномерного автоматического массива псевдослучайными целыми числами;
- вывести проинициализированную часть одномерного массива на экран;
- найти минимум по значению в созданном массиве.

Многофайловый проект будет состоять из следующих файлов (Рисунок 32):

- `main.cpp` – содержит подключение необходимых файлов, инструкцию `using`, а также собственно программу (функцию `main()` или так называемую «точку входа»);
- `userVector.h` – содержит прототипы используемых функций:
  - `void initArray(int (&)[N], int);` - прототип функции инициализирующей псевдослучайными числами активный участок одномерного массив;

- o `void printArray(int (&)[N], int);` - прототип функции выводящей активный участок одномерного массива на экран;
  - o `int minArray(int (&)[N], int);` - прототип функции, определяющей минимальное значение в активной части одномерного массива;
- `userVector.cpp` – содержит необходимые директивы `#include`, а также описания всех функций, прототипы которых пересилены в заголовочном файле.
- `globalCONST.h` – содержит определение целочисленной константы `N`, задающей максимально возможный размер массива, выделяемого в ***стеке***.

Рисунок 32 – Структура многофайлового проекта

Пример.

```
//файл main.cpp
```

main.cpp | userVector.h | userVector.cpp | globalCONST.h

```
1  #include <iostream>
2  #include "userVector.h"
3  #include "globalCONST.h"
4
5  using namespace std;
6
7  int main() {
8      int n;
9      int array[N];
10
11     cout<<"Введите количество элементов < "<< N
12         << endl;
13     cin>> n; //ввод активного количества элмент.
14
15     initArray(array, n);
16
17     cout<<"Заданный массив:"<<endl;
18     printArray(array, n);
19
```

```
20      cout<< endl<< "Минимальное значение "
21          << minArray(array, n);
22      return 0;
23  }
```

//файл userVector.h

main.cpp | userVector.h | userVector.cpp | globalCONST.h

```
1  #include "globalCONST.h"
2
3  //прототипы функций без имен параметров
4  void initArray(int (&)[N], int);
5  void printArray(int (&)[N], int);
6  int minArray(int (&)[N], int);
```

//файл userVector.cpp

main.cpp | userVector.h | userVector.cpp | globalCONST.h

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  #include "globalCONST.h"
5
6  void initArray(int (&a)[N], int n) {
7      srand(time(0));
8      for(int i = 0; i < n; i++) {
9          a[i] = rand() % 10;
10     }
11 }
12
13 void printArray(int (&a)[N], int n) {
14     for(int i = 0; i < n; i++) {
15         std::cout<< a[i] <<" ";
16     }
17 }
18
19 int minArray(int (&a)[N], int n) {
20     int min = a[0];
21     for(int i = 1; i < n; i++) {
22         if (min > a[i]) min = a[i];
```

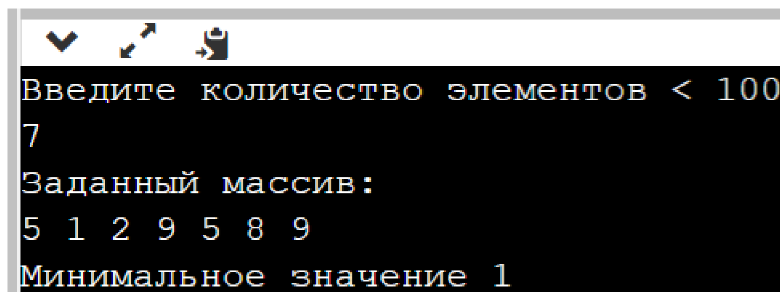
```
23        }
24        return min;
25    }
```

//файл globalCONST.h

main.cpp | userVector.h | userVector.cpp | globalCONST.h

```
1   #ifndef GLOBAL_CONST_h
2   #define GLOBAL_CONST_h
3
4   const int N = 100;
5
6   #endif //GLOBAL_CONST_h
```

Результат работы программы:



```
Введите количество элементов < 100
7
Заданный массив:
5 1 2 9 5 8 9
Минимальное значение 1
```

Замечание:

Проект по выбору минимального значения из автоматического массива с помощью функций, принимающих ссылку на массив доступен по ссылке URL: https://www.onlinegdb.com/edit/4YCg7h4oC# (дата доступа 02.02.2023).

Ссылка на двумерный автоматический массив

Совершенно аналогично можно передать по ссылке в функцию для обработки и двумерный массив. Рассмотрим пример проекта выполняющего транспонирование квадратного двумерного массива с помощью функций с использованием аппарата ссылок.

Для этого немного изменим структуру уже рассмотренного проекта (Рисунок 31), в котором использовались указатели. Это изменение касается создания дополнительного заголовочного файла `globalCONST.h` (Рисунок 33), содержащего определение константы, задающей максимально возможный размер автоматического квадратного двумерного массива.

Далее с помощью директивы `#include` подключить заголовочный файл `globalCONST.h` в файлах `main.cpp`, `userMatrix.h` и `userMatrix.cpp` .

Рисунок 33 – Структура проекта, основанного на использовании в прототипах и заголовках функций ссылок на двумерный массив

Кроме того, необходимо внести в набор формальных параметров прототипов и заголовков всех функций следующее изменения:

- в перечисление типов параметров в прототипах функций (заголовочный файл `userMatrix.h`) должна использоваться конструкция

```
(int (&)[N][N], unsigned)
```

вместо

```
(int** , unsigned);
```

- в перечисление формальных параметров в заголовке описания функций (файл `userMatrix.cpp`) должна использоваться конструкция уже с именами параметров

```
(int (&a)[N][N], unsigned n)
```

вместо

```
(int **a, unsigned n).
```

Далее следует удалить из проекта прототип и описание функции `freeMemory()`, т.к. память в ***стеке*** освобождается автоматически после завершения работы `main()`.

Таким образом, кроме редакционных изменений в прототипах и заголовках функций в файлах `userMatrix.h` и `userMatrix.cpp` никаких изменений нет.

Пример.

//файл main.cpp

main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h

```
1  #include <iostream>
2  #include "userMatrix.h"
3  #include "globalCONST.h"
4
5  using namespace std;
6
```

```
 7 int main(void) {
 8     unsigned size;
 9     cout<<"Введите количество строк/столбцов: ";
10     cin>> size;
11     //выделение в стеке памяти под массив
12     int matrix[N][N];
13 
14     //инициализация массива псевдосл. числами
15     initMatrix (matrix, size);
16 
17     //вывод на экран исходной матрицы
18     cout<<"\tИсходная матрица\n";
19     displayMatrix(matrix, size);
20 
21     //транспонирование матрицы
22     transpouseMatrix(matrix, size);
23 
24     //вывод на экран результатов транспонирования
25     cout<<"\tТранспонированная матрица\n";
26     displayMatrix(matrix, size);
27     return 0;
28 }
```

```
//файл userMatrix.h
```

```
main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h
1 #include "globalCONST.h"
2 
3 //прототипы функций без имен параметров
4 void initMatrix (int (&)[N][N], unsigned);
5 void displayMatrix (int (&)[N][N], unsigned);
6 void transpouseMatrix (int (&)[N][N], unsigned);
```

```
//файл userMatrix.cpp
```

```
main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 #include "globalCONST.h"
5 
```

```
 6 void initMatrix (int (&a)[N][N], unsigned n) {
 7     srand(time(0));
 8     for (int i = 0; i < n; i++ ) {
 9         for (int j = 0; j < n; j++ ) {
10             a[i][j] = rand() % 10;
11         }
12     }
13 }
14
15 void displayMatrix (int (&a)[N][N], unsigned n) {
16     for (int i = 0; i < n; i++ ) {
17         for (int j = 0; j < n; j++ ) {
18             std::cout<<a[i][j]<<" ";
19         }
20         std::cout<< std::endl;
21     }
22 }
23
24 void transpouseMatrix (int (&a)[N][N], unsigned n) {
25     for (int i = 0; i < n; i++) {
26         for (int temporary, j = i + 1; j < n; j++) {
27             temporary = a[i][j];
28             a[i][j] = a[j][i];
29             a[j][i] = temporary;
30         }
31     }
32 }
```

//файл globalCONST.h

```
main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h
1 #ifndef GLOBAL_CONST_h
2 #define GLOBAL_CONST_h
3
4 const int N = 100;
5
6 #endif //GLOBAL_CONST_h
```

Результат работы программы такой же, как и в предыдущем проекте (Рисунок 31).

Замечание.

Проект по работе с двумерным массивом с помощью аппарата ссылок доступен по ссылке URL: https://www.onlinegdb.com/edit/Qbd4djyJJ (дата доступа 02.02.2023).

Использование константных ссылок на массив

В случае проекта, посвященного обработке ***одномерного*** массива (URL: https://www.onlinegdb.com/edit/4YCg7h4oC, дата доступа 02.02.2023) в прототипе и заголовке функции `printArray()` следует использовать квалификатор **`const`** перед указанием типа элементов массива.

Пример.

```
void printArray(const int (&)[N], int);
```

Для проекта, обрабатывающего ***двумерный*** массива (URL: https://www.onlinegdb.com/edit/Qbd4djyJJ, дата доступа 02.02.2023) в прототипе и заголовке функции `displayMatrix()` квалификатор **`const`** также следует указать перед типом элементов массива:

Пример.

```
void displayMatrix (const int (&)[N][N], unsigned);
```

Замечание.

*Использование квалификатора **`const`** в параметрах функции перед ссылкой на одномерный или двумерный автоматический массив, приводит к тому, что память, используемая для хранения массива, будет доступна в вызываемой функции только для чтения (т.е. внести изменения в массив в теле вызываемой функции нельзя).*

Использование в параметрах функций ссылки на указатель

В предыдущем разделе рассматривался вариант передачи по ссылке в функцию автоматических одномерных и двумерных массивов. Однако как уже отмечалось в настоящее время загружать ***стек*** лишними переменными, а тем более массивами считается не профессиональным.

Как уже отмечалось для хранения и обработки больших объемов информации предназначена ***куча***. Встает естественный вопрос можно ли в функцию передать ***одномерный*** массив из ***кучи*** по ***ссылке***.

Ответ однозначен – это можно сделать, но для этого используется ***ссылка на указатель***. В качестве примера приведем изменения, которые необходимо внести в проект, доступный по ссылке: https://www.onlinegdb.com/edit/4YCg7h4oC (дата доступа 02.02.2023), для того чтобы использовать в программе массив из ***кучи***.

Пример.

```
//изменения в файле main.cpp указанного проекта
```

main.cpp | userVector.h | userVector.cpp | globalCONST.h

```
#include <iostream>
#include "userVector.h"
#include "globalCONST.h"

using namespace std;

int main() {
    int n;
    int *array = new int[N]; //выделяем массив в куче

    cout<<"Введите количество элементов < "<< N
        << endl;

    ... //остальной код без изменений

    return 0;
}
```

```
//изменения в файле userVector.h указанного проекта
```

pp | userVector.h | userVector.cpp | globalCONS

```
#include "globalCONST.h"

//прототипы функций без имен параметров
void initArray(int *&, int);
void printArray(int *&, int);
int minArray(int *&, int);
```

```
//изменения в файле userVector.cpp указанного проекта
```

o | userVector.h | userVector.cpp | globalCONST.h

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "globalCONST.h"

void initArray(int *&a, int n) {
    ... // код тела функции без изменений
}

void printArray(int *&a, int n) {
    ... // код тела функции без изменений
}

int minArray(int *&a, int n) {
    ... // код тела функции без изменений
    return min;
}

//файл globalCONST.h указанного проекта без изменений
```

Результат работы данного проекта не отличается от результатов работы исходного исправляемого варианта кода.

<u>*Замечание.*</u>

- *поскольку работа с указателями является в C++ более универсальной, то можно передать одномерный массив, сформированный в* ***стеке*** *в функцию с помощью ссылки на указатель, однако, в этом случае следует использовать константную ссылку и прототип функции, например,* `minArray()` *из рассматриваемого проекта приобретет вид:*

```
int minArray(int* const &a, int n);
```

- *в этом случае размерность автоматического массива, передаваемого в функцию,* ***не указывается****.*

Перейдем к обсуждению возможности передачи в функцию ссылок на более сложные типы указателей, например ***ссылки*** на ***указатель-на-***

указатель. Очевидно, это даст возможность передавать в функцию массивы произвольной размерности, созданные в ***куче***.

В данном случае можно также в качестве базового использовать уже созданный проект, который доступен по ссылке URL: https://www.onlinegdb.com/edit/Qbd4djyJJ (дата доступа 02.02.2023).

Пример.

```
//изменения в файле main.cpp указанного проекта
main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h
#include <iostream>
#include "userMatrix.h"
#include "globalCONST.h"

using namespace std;

int main(void) {
    unsigned size;
    cout<<"Введите количество строк/столбцов: ";
    cin>> size;
    //выделение в куче памяти под массив
    int **matrix = new int* [size];
    for( int i = 0; i < size; i++) {
        matrix[i] = new int[size];
    }

    //инициализация массива псевдосл. числами
    initMatrix (matrix, size);

    ...//остальной код без изменений

    return 0;
}
```

```
//изменения в файле userMatrix.h указанного проекта
main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h
1  #include "globalCONST.h"
2
3  //прототипы функций без имен параметров
4  void initMatrix (int **&, unsigned);
5  void displayMatrix (int **&, unsigned);
6  void transpouseMatrix (int **&, unsigned);
```

```
//изменения в файле userMatrix.cpp указанного проекта
```

main.cpp | userMatrix.h | userMatrix.cpp | globalCONST.h

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  #include "globalCONST.h"
5
6  void initMatrix (int **&a, unsigned n) {
7       ...//код тела функции без изменений
8  }
9
10 void displayMatrix (int **&a, unsigned n) {
11      ...//код тела функции без изменений
12 }
13
14 void transpouseMatrix (int **&a, unsigned n) {
15      ...//код тела функции без изменений
16 }
```

```
//файл globalCONST.h указанного проекта без изменений
```

Результат работы данного проекта не отличается от результатов работы исходного исправляемого варианта кода.

Замечания:

- *очевидно, можно использовать ссылку на указатель соответствующего типа (указатель-на-указатель-на-указатель и далее) для передачи многомерного массива произвольной размерности, созданного в* ***куче*** *в функцию для обработки;*
- *к сожалению, выполнить, как в одномерном случае передачу автоматического двумерного массива из* ***кучи*** *с помощью* ***константной*** *ссылки на указатель-на-указатель* ***нельзя****.*

Указатели и ссылки на функции

Подобно переменным, функции также имеют свой адрес в памяти. Когда функция вызывается (с помощью операции «()»), точка выполнения переходит к адресу вызываемой функции. Покажем это на примере.

Пример.

```
main.cpp
1  #include <iostream>
2  using namespace std;
3
4  //Пользовательская функция без тела
5  void userFunction() {}
6
7  int main() {
8      //вывод адреса пользовательской функции
9      cout << (void*) userFunction;
10     return 0;
11 }
```

Результат работы программы:

Замечание.

Преобразование типов (конструкция `(void*)`*) необходимо для преобразования указателя неизвестно какого типа (указатель на функцию не имеет типа) к указателю на базовый тип* `void`*. Иначе в* `cout` *произойдет автоматическое преобразование и* ***не****типизированное ненулевое значение адреса функции, например, в онлайн IDE onlineGDB будет интерпретировано как* `true` *и автоматически преобразовано к целочисленному значению* `1`*. В других средах могут быть другие целочисленные значения, если будет отсутствовать промежуточное преобразование к булевскому типу.*

Таким образом, имя функции (как и в случае с массивами) это адрес ее расположения в оперативной области памяти при выполнении программы.

Так же, как можно объявить переменную-указатель на обычную переменную, можно объявить и переменную-указатель на функцию.

Синтаксис создания указателя на функцию

Формат объявления указателя на функцию, которая возвращает значения типа `типВозврЗнач`, а получает определенный набор параметров,

указанных в ~~списке, типов~~, который может отсутствовать, если параметров нет:

`типВозврЗнач (*`**`имяУказФункц`**`) (`~~`список, типов`~~`);`

Скобки вокруг `*`**`имяУказФункц`** необходимы для соблюдения приоритета операций, в противном случае указанная выше запись будет интерпретироваться как предварительное объявление функции `имяУказФункц`, которая возвращает указатель на тип `типВозврЗнач`.

Для создания константного указателя на функцию следует использовать **`const`** после звездочки:

`типВозврЗнач (*`**`const имяУказФункц`**`) (`~~`список, типов`~~`);`

Обычный (неконстантный) или константный указатель на функцию могут быть инициализированы именем функции. Формат:

`типВозврЗнач (*`**`имяУказФункц`**`) (`~~`список, типов`~~`) =` **`имяФункции`**`;`

или

`типВозврЗнач (*`**`const имяУказФункц`**`) (`~~`список, типов`~~`) =` **`имяФункции`**`;`

Однако, очевидно, константный указатель после инициализации уже не сможет изменить своего значения, а не константному можно присвоить имя любой другой соответствующей по прототипу функции в любом месте кода.

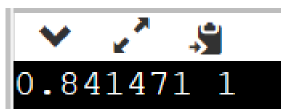
Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  double user(double x) {
6      return x;
7  }
8
9  int main() {
10     double (*ptrFunction) (double) = user;
```

```
11      ptrFunction = sin;
12      double (*const ptrF) (double) = cos;
13
14      cout << ptrFunction(1) << " "
15           << ptrF(0);
16      return 0;
17  }
```

Результат работы программы:



В примере вызовы функций `sin()` и `cos()` выполнены через неявное разыменование `ptrFunction(1)` и `ptrF(0)` (простое обращение к функциям `sin()` и `cos()` через имена указателей `ptrFunction` и `ptrF` как через имена функций), а можно пойти сложнее и обратиться через явное разыменование (`(*ptrFunction)(1)` и `(*ptrF)(0)`).

Замечания:

- *при работе с указателями на функции, необходимо помнить, что им присваиваются только имена функций (без скобок);*
- *в указателе на функцию и самой функции должны совпадать тип возвращаемого значения, а также количество и порядок типов формальных параметров;*
- *инициализация параметров функции «по умолчанию» будет игнорироваться при обращении через указатель на функции.*

Выбор функции, решающей задачу

Пусть даны две функции: одна обрабатывает попадание точки в круг, а вторая в квадрат (радиус круга и длина стороны квадрата определяются ***случайным*** образом). Предполагается, что и центр круга, и центр квадрата совпадают с началом координат в координатной плоскости, кроме того, стороны квадрата ориентированы вдоль осей декартовой системы координат.

Проект имеет структуру, приведенную на рисунке (Рисунок 34).

main.cpp | areaFunction.h | areaFunction.cpp

Рисунок 34 – Структура проекта позволяющего выбирать рабочую функцию пользователю

Описание файлов, составляющих проект:

- файл `main.cpp` – содержит программу (функцию `main()`), при запуске которой пользователь после ввода двух координат (с именами `x` и `y`) точки на плоскости, может выбрать попадание в какую именно из случайно сгенерированных областей (круг или квадрат) он хочет проверить;
- файл `areaFunction.h` – заголовочный файл, который содержит прототипы двух функций, определяющих попадание в круг или квадрат (`circle()` и `square()` соответственно).
- файл `areaFunction.cpp` – содержит описание функций `circle()` и `square()`.

Пример.

//файл main.cpp

main.cpp | areaFunction.h | areaFunction.cpp

```
 1  #include <iostream>
 2  #include "areaFunction.h"
 3
 4  using namespace std;
 5
 6  int main() {
 7      bool (*area) (double, double);
 8      double x, y;
 9
10      cout << "Координаты точки x и y\n";
11      cin >> x >> y;
12
13      int i;
14      cout<< "Доступны две области со "
15          << "случ. размер.\n"
16          << "   1 - Квадрат \t 2 - Круг\n"
17          << "Введите целое число (1 или 2):\n";
18      cin >> i;
19
```

```
20      switch(i) {
21          case 1: area = square; break;
22          case 2: area = circle; break;
23          default:
24              cout << "Область выбрана неверно";
25      }
26      if (area(x, y)) cout<< "Попала";
27      else cout<< "Не попала";
28  }
```

//файл areaFunction.h

main.cpp | areaFunction.h | areaFunction.cpp

```
1  bool circle(double, double);
2  bool square(double, double);
```

//файл areaFunction.cpp

main.cpp | areaFunction.h | areaFunction.cpp

```
1   #include <cstdlib>
2   #include <ctime>
3
4   bool circle(double x, double y) {
5       srand(time(0));
6       double r = (rand() % 1000 + 1) / 100.;
7       return x * x + y * y <= r * r;
8   }
9
10  bool square(double x, double y) {
11      srand(time(0));
12      //2*a   длина стороны квадрата
13      double a = (rand() % 1000 + 1) / 100.;
14      return -a <= x && x <= a && -a <= y && y <= a;
15  }
```

Результат работы программы:

```
Координаты точки x и y
1 2
Доступны две области со случ. размер.
   1 - Квадрат    2 - Круг
Введите целое число (1 или 2):
1
Попала
```

Замечание.

Проект по работе с указателем на функцию доступен по ссылке URL: https://www.onlinegdb.com/edit/NebYe8j7_ (дата доступа 02.02.2023)

Передача указателя на функцию другой функции через параметры. Выбор направления сортировки

Представляет интерес не только использование указателей на функцию в пределах `main()`, но и прежде всего возможность передачи указателей на функции через систему параметров другим функциям.

Если разработчику дать возможность выбирать операцию сравнения, то появиться возможность универсализации алгоритма выбора максимума/минимума в одномерном массиве, а также его сортировки по возрастанию/убыванию в рамках одного и того же кода.

В начале напишем функции-компараторы, устанавливающие правила сравнения (больше или меньше) для величин целочисленного типа. Поскольку собираемся сортировать массив целых чисел, то функции будут иметь вид:

```
bool moreThan(int a, int b) {
    return a > b;
}

bool lessThan(int a, int b) {
    return a < b;
}
```

После написания этих двух функций становится очевидным, что указатель на функцию должен иметь следующий формат с точностью до выбора имени указателя:

```
bool (*имяУказФункц) (int, int).
```

Перейдем к выбору максимума/минимума с помощью одной функции в зависимости от переданной через параметр функции-компаратора, замещающей операцию сравнения. Проект имеет структуру, приведенную на рисунке (Рисунок 35).

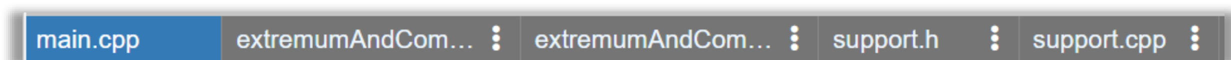


Рисунок 35 – Структура проекта позволяющего выбирать экстремумы в одном

Описание файлов, составляющих проект:

- файл `main.cpp` – содержит программу (функцию `main()`), в которой с помощью одной и той же функции выбирается и минимум, и максимум из одномерного массива;
- файл `extremumAndComparators.h` – заголовочный файл, содержащий прототипы функций:
    - `int functionMinMax(int *, int, bool (*) (int, int));`
    - `bool moreThan(int, int);`
    - `bool lessThan(int, int);`
- файл `extremumAndComparators.cpp` содержит описание функций, прототипы которых перечислены в заголовочном файле `extremumAndComparators.h`;
- файл `support.h` – заголовочный файл, содержащий прототип сервисной функции `printArray()`, выводящей на экран проинициализированный одномерный массив;
- файл `support.cpp` содержит описание функции `printArray()`.

Пример.

//файл main.cpp

main.cpp | extremumAndCom… | extremumAndCom… | support.h

```
1  #include <iostream>
2  #include "extremumAndComparators.h"
3  #include "support.h"
4  using namespace std;
5
6  int main() {
7      int array[] = {1, 21, 13, 64, 5};
8      int n = sizeof(array) / sizeof(float);
9
10     cout<<"Инициализированный массив:"<<endl;
11     printArray(array, n);
12
13     cout<<"Результат выбора минимума: "
14         // передача в функцию компаратора moreThan()
15         << functionMinMax(array, n, moreThan)
16         << endl;
17
18     cout<<"Результат выбора максимума:"
19         // передача в функцию компаратора lessThan()
```

```
20         << functionMinMax(array, n, lessThan)
21         << endl;
22     return 0;
23 }
```

//файл extremumAndComparators.h

main.cpp | extremumAndCom... | extremumAndCom... | support.h | su

```
1  //функция определяющая экстремум
2  int functionMinMax(int *,
3                     int,
4                     bool (*) (int, int));
5
6  //компараторы
7  bool moreThan(int, int);
8  bool lessThan(int, int);
```

//файл extremumAndComparators.cpp

main.cpp | extremumAndCom... | extremumAndCom... | support.h

```
1  //функция определяющая экстремум
2  int functionMinMax(int *a,
3                     int n,
4                     bool (*compare)(int, int)) {
5      int valueMinMax = a[0];
6      for(int i = 1; i < n; i++) {
7          if (compare(valueMinMax, a[i])) {
8              valueMinMax = a[i];
9          }
10     }
11     return valueMinMax;
12 }
13
14 //компараторы
15 bool moreThan(int a, int b) {
16     return a > b;
17 }
18
19 bool lessThan(int a, int b) {
20     return a < b;
21 }
```

//файл support.h

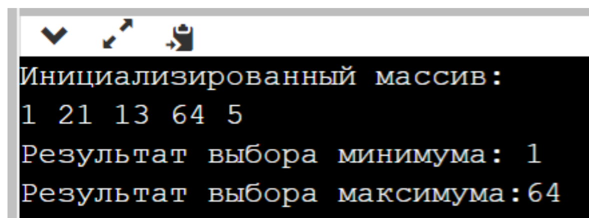
main.cpp | extremumAndCom... | extremumAndCom... | support.h | sup

```
1  void printArray(int*, int);
```

//файл support.cpp

```
... ⋮ extremumAndCom... ⋮ support.h ⋮ support.cpp ⋮
1 #include <iostream>
2 using namespace std;
3
4 void printArray(int *a, int n) {
5     for(int i = 0; i < n; i++) {
6         cout<< a[i]<<" ";
7     }
8     cout << endl;
9 }
```

Результат работы программы:



Замечание.

Проект по выбору экстремумов в одномерном массиве доступен по ссылке URL: https://www.onlinegdb.com/edit/va2zTgRL3 (дата доступа 02.02.2023)

Несколько более сложным примером является пример сортировки в произвольном направлении (по возрастанию или убыванию) в зависимости от переданной через указатель функции сравнения (компаратора).

Программа будет иметь вид многофайлового проекта, структура и состав которого аналогичны предыдущему случаю выбора экстремума. Изменение заключается в том, что вместо функции `functionMinMax()`, выбирающей экстремум в проекте, будет присутствовать прототип и описание функции `bubbleSort()`, сортирующей одномерный массив в произвольном направлении в зависимости от переданной функции-компаратора. Прототипы и функции-компараторы из предыдущего примера останутся неизменными.

В составе нового проекта, также изменяться названия одного `h`-файла и `cpp`-файла (Рисунок 36):

- вместо `extremumAndComparators.h` будет использоваться `sortAndComparators.h;`
- вместо `extremumAndComparators.cpp` будет использоваться `sortAndComparators.cpp;`

main.cpp | sortAndComparato… | sortAndComparato… | support.h | support.cpp

Рисунок 36 – Проект сортировки одномерного массива с использованием компараторов

Пример.

//файл main.cpp

main.cpp | sortAndComparato… | sortAndComparato… | support.h

```
1  #include <iostream>
2  #include "sortAndComparators.h"
3  #include "support.h"
4  using namespace std;
5
6  int main() {
7      int array[] = {1, 21, 13, 64, 5};
8      int n = sizeof(array) / sizeof(float);
9
10     cout<<"Инициализированный массив:"<<endl;
11     printArray(array, n);
12
13     cout<<"По возростанию:"<<endl;
14     bubbleSort(array, n, moreThan);
15     printArray(array, n);
16
17     cout<<"По убыванию:"<<endl;
18     bubbleSort(array, n, lessThan);
19     printArray(array, n);
20     return 0;
21 }
```

//файл sortAndComparators.h

main.cpp | sortAndComparato… | sortAndComparato… | support.h

```
1  //функция сортировки
2  void bubbleSort(int *,
3                  int,
4                  bool (*) (int, int));
5
6  //компараторы
7  bool moreThan(int, int);
8  bool lessThan(int, int);
```

//файл sortAndComparators.cpp

main.cpp | sortAndComparato… | sortAndComparato… | support

```
//функция сортировки
void bubbleSort(int *a,
                int n,
                 bool (*compare)(int, int)) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n - i - 1; j++) {
            if (compare(a[j], a[j + 1])) {
                float temp = a[j];
                      a[j] = a[j + 1];
                      a[j + 1] = temp;
            }
        }
    }
}

//компараторы
bool moreThan(int a, int b) {
    return a > b;
}

bool lessThan(int a, int b) {
    return a < b;
}
```

//файл support.h

main.cpp | extremumAndCom… | extremumAndCom… | support.h | sup

```
void printArray(int*, int);
```

//файл support.cpp

… | extremumAndCom… | support.h | support.cpp

```
#include <iostream>
using namespace std;

void printArray(int *a, int n) {
    for(int i = 0; i < n; i++) {
        cout<< a[i]<<" ";
    }
    cout << endl;
}
```

Результат работы программы:

```
Инициализированный массив:
1 21 13 64 5
По возростанию:
1 5 13 21 64
По убыванию:
64 21 13 5 1
```

<u>Замечание.</u>

Проект по сортировке одномерных массивов в зависимости от выбранного компаратора доступен по ссылке URL: https://www.onlinegdb.com/edit/wiUrtHJRv (дата доступа 02.02.2023)

Массив указателей на функцию

Кроме одиночных указателей на функции можно определять их массивы. Для этого используется следующий формальный синтаксис:

```
тип (*имяМассива[количЭлементов]) (параметры)
```

где **количЭлементов** – количество элементов в массиве указателей на функцию. Массив указателей на функцию можно инициализировать стандартным образом присваивая список имен функций, имеющих схожие прототипы. В этом случае параметр **количЭлементов** лучше не указывать:

```
тип (*имяМассива[]) (параметры) = {Имя1, …, ИмяN};
```

В качестве примера, что, собственно, дает применение массива указателей на функцию рассмотрим уже известный пример попадания в круг или квадрат случайного размера с центром в начале координатной плоскости (параграф «Выбор функции, решающей задачу», Рисунок 34).

Поскольку единственным изменением в проекте будет оформление функции `main()` с учетом возможности использования ***массива указателей на функцию***, то приведем только измененный файл `main.cpp`, указанного проекта (Рисунок 34). Предполагая, что читатель сам сможет внести указанные изменения в уже существующий проект.

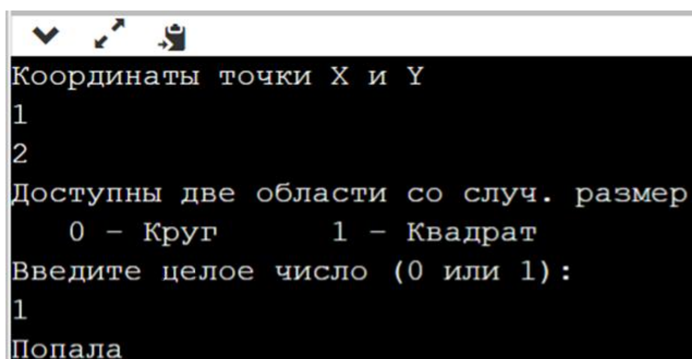
Пример.

//файл main.cpp

main.cpp | areaFunction.h | areaFunction.cpp

```
1  #include <iostream>
2  #include "areaFunction.h"
3
4  using namespace std;
5
6  int main() {
7      bool (*area[]) (double, double)={circle, square};
8      double x, y;
9
10     cout << "Координаты точки X и Y\n";
11     cin >> x >> y;
12
13     int i;
14     cout<< "Доступны две области со случ. размер.\n"
15         << "    0 - Круг \t 1 - Квадрат \n"
16         << "Введите целое число (0 или 1):\n";
17     cin >> i;
18
19     if (area[i](x, y)) cout<< "Попала";
20     else cout<< "Не попала";
21     return 0;
22 }
```

Результаты выполнения программы:



Замечание.

- *как видно из текста программа стала короче на оператор* `switch();`
- *полный код многофайлового проекта, использующего массив указателей на функцию доступен по ссылке URL: https://www.onlinegdb.com/edit/k954cTwN_ (дата доступа 02.02.2023).*

Создание псевдонимов типов указателей на функцию

Язык C++ позволяет определять имена новых типов данных с помощью инструкции using или ключевого слова typedef. На самом деле эти конструкции не создают новый тип данных, а определяют новое имя для уже существующего типа. Иногда это может облегчить читаемость кода. Он также может помочь документировать код, позволяя назначать содержательные имена стандартным типам данных.

Из-за значительной синтаксической сложности описания указателей на функцию особенно при передаче имени функции в качестве параметра в другую функцию использование ключевых слов using или typedef для определения краткого синонима типа приобретает особенно важное значение.

Ключевое слово typedef

Формат стандартного вида оператора typedef:

```
typedef существующийТип новоеИмя;
```

где существующийТип — это любой существующий тип данных, а новоеИмя - это новое имя для данного типа. Новое имя определяется в дополнение к существующему имени типа, а не замещает его.

Замечание.

Как и всегда если оператор typedef расположен за пределами определения любой функции, то это глобальный псевдоним типа. Если внутри функции, то это локально определенный псевдоним.

Применительно к указателям на функцию и их использования в программе сортировки можно создать с помощью typedef, например, новый тип typeCompare.

Пример.

```
typedef bool (*typeCompare)(int, int);
```

и использовать для сокращении записей типов параметров прототипа и самой функции, например, bubbleSort(), а, следовательно, для повышения читаемости кода.

Вернемся к рассмотрению проекта сортировки одномерного массива с использованием компараторов (Рисунок 36).

Добавим в него еще один заголовочный файл `typeDef.h` . В нем напишем указанное выше определение нового типа `typeCompare` для определения указателя на функцию.

Пример.

```
Comparato… | support.h | support.cpp | typeDef.h
1  typedef bool (*typeCompare)(int, int);
```

Подключим этот файл с помощью `#include` в файлах проекта `sortAndComparators.h` и `sortAndComparators.cpp`, а после этого выполним замену типов в прототипе и описании функции `bubbleSort()`.

Пример.

```
//файл sortAndComparators.h
```

```
main.cpp | sortAndComparato… | sortAndC
1  #include "typeDef.h"
2
3  //функция сортировки
4  void bubbleSort(int *,
5                  int,
6                  typeCompare);
7
8  //компараторы
9  bool moreThan(int, int);
10 bool lessThan(int, int);
```

```
//файл sortAndComparators.cpp
```

```
main.cpp | sortAndComparato… | sortAndComparato… | suppo
1  #include "typeDef.h"
2
3  //функция сортировки
4  void bubbleSort(int *a,
5                  int n,
6                  typeCompare compare) {
7      for(int i = 0; i < n; i++) {
8          for(int j = 0; j < n - i - 1; j++) {
9              if (compare(a[j], a[j + 1])) {
10                 float temp = a[j];
```

```
11                      a[j] = a[j + 1];
12                      a[j + 1] = temp;
13              }
14          }
15      }
16  }
17
18  //компараторы
19  bool moreThan(int a, int b) {
20      return a > b;
21  }
22
23  bool lessThan(int a, int b) {
24      return a < b;
25  }
```

Остальные файлы старого проекта останутся без изменений.

Замечание.

Полный код многофайлового проекта использующего определение псевдонима (синонима) типа указателя на функцию с помощью ключевого слова `typedef` *доступен по ссылке URL: https://www.onlinegdb.com/edit/smpU4tuaN (дата доступа 02.02.2023)*

Инструкция using

Формат инструкции `using` при определении синонима типа:

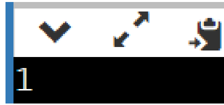
```
using новоеИмя = существующийТип;
```

Пример.

main.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   using ptrFunction = double (*)(double);
5
6   int main() {
7       ptrFunction f = cos;
8       std::cout << f(0);
9       return 0;
10  }
```

Результат работы программы:



Ссылки на функцию

Формат объявления ссылки на функцию, которая возвращает значения типа `типВозврЗнач`, а получает определенный набор параметров, указанных в ~~списке, типов~~, который может отсутствовать, если параметров нет:

```
типВозврЗнач (&имяСсылкиНаФункц) (список, типов);
```

Скобки вокруг **&имяСсылкиНаФункц** необходимы для соблюдения приоритета операций, в противном случае указанная выше запись будет интерпретироваться как предварительное объявление функции `имяУказФункц`, которая возвращает ссылку на тип `типВозврЗнач`.

Формат инициализации ссылки на функцию:

```
типВозврЗнач (&имяСсылкиНаФункц) (список, типов) =
                                          имяФункции;
```

Замечание.

- *синтаксис ссылки на функцию не допускает использования квалификатора* **`const`***;*
- *использование ссылки на функцию практически полностью аналогично использованию указателя на функцию, однако, очевидно, в отличие от указателя на функцию к ссылке невозможно применить явную операцию разыменования.*

Макросы проверки ошибок

Инструкция assert()

Инструкция `assert()` (или «оператор проверочного утверждения») в языке C++ — это макрос, который обрабатывает условное выражение во время выполнения. Если условное выражение истинно, то инструкция `assert()` ничего не делает. Если же оно ложное, то выводится сообщение

об ошибке, и программа завершается. Это сообщение об ошибке содержит ложное условное выражение, а также имя файла с кодом и номером строки с `assert()`. Таким образом, можно легко найти и идентифицировать проблему, что очень помогает при отладке программ.

Сам `assert()` реализован в заголовочном файле `cassert` и часто используется для:

- проверки корректности переданных параметров функции;
- проверка значений внутри алгоритма, реализуемого функцией;
- проверки возвращаемого функцией значения.

Пример.

main.cpp

```
1  #include<iostream>
2  #include<cassert>
3
4  int getArrayValue(int* array, int index) {
5      //блок проверок входных параметров
6      assert(array != nullptr);
7      assert(0 <= index);
8      //реализация алгоритма
9      return array[index];
10 }
11
12 int main() {
13     int a[] = {1,2,3,4};
14     std::cout << "Значение элемента массива = "
15               << getArrayValue(a, -2);
16     return 0;
17 }
```

Результат работы программы (выдача сообщения об ошибке):

```
input
a.out: main.cpp:6: int getArrayValue(int*, int): Assertion `0 <= index' failed.
```

Обычно утверждения `assert()` бывают недостаточно содержательными. Однако есть способ улучшения ситуации с информативностью: следует просто добавить сообщение в качестве строки `C-style` вместе с логическим оператором И («`&&`»):

Пример.

main.cpp

```
1   #include<iostream>
2   #include<cassert>
3
4   int getArrayValue(int* array, int index) {
5       //блок проверок входных параметров
6       assert(array != nullptr && "Указатель нулевой");
7       assert(0 <= index && "Индекс отрицательный");
8       //реализация алгоритма
9       return array[index];
10  }
11
12  int main() {
13      int a[] = {1,2,3,4};
14      std::cout << "Значение элемента массива = "
15                << getArrayValue(nullptr, -2);
16      return 0;
17  }
```

Результат работы программы:

```
input
a.out: main.cpp:6: int getArrayValue(int*, int): Assertion
`array != nullptr && "Указатель нулевой"' failed.
```

Строка C-style всегда принимает значение `true`. Поэтому, если `found` примет значение `false`, то `false && true = false`. Если же `found` примет значение `true`, то `true && true = true`. Таким образом, строка C-`style` вообще не влияет на обработку утверждения.

Однако, если `assert()` сработает, то строка C-`style` будет включена в сообщение `assert()` и это даст дополнительное объяснение того, что пошло не так.

Приведем пример использования `assert()` внутри алгоритма, реализуемого функцией.

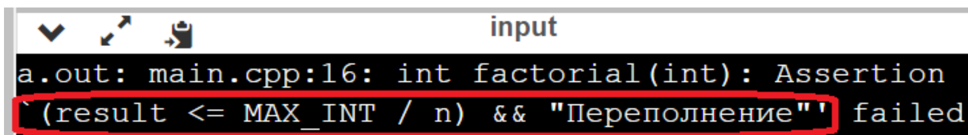
Пример.

main.cpp

```
1   #include<iostream>
```

```
 2  #include<cmath>
 3  #include<cassert>
 4
 5  const int NUMBER_BITS = sizeof(int) * 8;
 6  const int MAX_INT = (int) pow(2, NUMBER_BITS - 1);
 7
 8  int factorial(int n) {
 9      //проверка корректности полученого значения
10      assert(0 <= n && "Получено отрицательное значение");
11      //алгоритм
12      if (n == 0) return 1;
13      int result = 1;
14      for(int i = 1; i <= n; i++) {
15          //проверка корректности значений внутри алгоритма
16          assert((result <= MAX_INT / n) && "Переполнение");
17          result = result * i;
18      }
19      return result;
20  }
21
22  int main() {
23      std::cout << "Факториал = " << factorial(100);
24      return 0;
25  }
```

Результат работы программы:


```
a.out: main.cpp:16: int factorial(int): Assertion
`(result <= MAX_INT / n) && "Переполнение"' failed
```

Совершенно аналогично можно проверить возвращаемое функцией значение. В случае того же алгоритма вычисления факториала если не проверять переполнение в теле цикла, то критерием переполнения будет отрицательное или нулевое значение переменной `result` после цикла `for()`. Поэтому значение `result` следует проверить перед инструкцией `return result;`.

Пример.

```
main.cpp
 1  #include<iostream>
 2  #include<cmath>
 3  #include<cassert>
```

```
 4
 5  const int NUMBER_BITS = sizeof(int) * 8;
 6  const int MAX_INT = (int) pow(2, NUMBER_BITS - 1);
 7
 8  int factorial(int n) {
 9      //проверка корректности полученого значения
10      assert(0 <= n && "Получено отрицательное значение");
11      //алгоритм
12      if (n == 0) return 1;
13      int result = 1;
14      for(int i = 1; i <= n; i++) {
15          result = result * i;
16      }
17      //проверка корректности возвращаемого результата
18      assert(result > 0 && "Переполнение");
19      return result;
20  }
21
22  int main() {
23      std::cout << "Факториал = " << factorial(100);
24      return 0;
25  }
```

Результат работы программы:

```
input
a.out: main.cpp:18: int factorial(int): Assertion `result > 0 && "Переполнение"' failed.
```

Особенности использования инструкций `assert()` при тестировании и отладке:

- они существенно упрощают локализацию ошибок в коде;
- большое количество инструкций `assert()` ***не*** ухудшит ясность кода и ***не*** существенно замедлит выполнение вашей программы;
- инструкции `assert()` визуально выделяются из общего кода и несут важную информацию о предположениях, на основе которых работает данный код;
- правильно расставленные инструкции `assert()` и широкое использование в них дополнительных пояснительных строк способны заменить большинство комментариев в коде;

Однако:

- использование инструкций `assert()` после каждой строчки кода не сильно улучшит эффективность отлова ошибок;

- не существует единого мнения насчет оптимального количества инструкций `assert()` в коде, также как и насчет оптимального количество комментариев в программе;
- инструкции `assert()` ***не*** могут заменить обработку ***исключительных ситуаций***, которые не являются ошибками программирования (например, ошибок работы с файлами, базами данных и пр.), потому что, когда срабатывает инструкция `assert`() работа программы немедленно прекращается, без возможности, например, закрыть файл или базу данных;
- обычно инструкции `assert()` оставляют «включенными» во время разработки и тестирования программ, но отключают в релиз-версиях программ

Замечание.

*Отключение инструкций `assert()` **не** должно менять поведение программы, например, не рекомендуется вызвать в `assert()` функцию, возвращающую значение и изменяющую состояние программы либо внешнего окружения программы. В этом случае следует вначале присвоить некоторой переменной возвращаемое функцией значение, а затем обработать это значение в `assert()`.*

С приобретением опыта использования данного инструмента каждый разработчик должен сформулировать для себя ряд правил, определяющих, когда и где следует использовать данную инструкцию. Например, можно существенно уменьшить количество инструкций `assert()` без существенного ухудшения эффективности отлова ошибок путем размещения этих инструкций лишь для тех значений формальных параметров, которые непосредственно используются в теле функций, а не просто передаются «транзитом» по цепочке вызовов.

Наилучшим использованием инструкции `assert()` считается останов программы при возникновении ***исключительной ситуации*** (будут рассмотрены позже) ***уровня операционной системы***, например, выход индекса за пределы коллекции. Однако даже в этом случае если разработчик уверен, что ***исключительная ситуация*** (например, ошибка работы с файлом или базой данных) не приведет к «фатальному» повреждению работоспособности программы, то ***вместо*** использования инструкции `assert()` и останова программы ***следует*** обработать это ***исключение*** и позволить программе продолжать функционировать.

Макрос NDEBUG

Функция assert() тратит мало ресурсов на проверку условия. Кроме того, инструкции assert() (в идеале) никогда не должны срабатывать в релизном коде (потому что код к этому моменту сдачи уже должен быть тщательно протестирован).

В языке C++ есть возможность отключить все инструкции assert() в окончательном коде, используя директиву, определяющую макрос NDEBUG:

```
#define NDEBUG
```

При использовании этого макроопределения все инструкции assert() будут проигнорированы вплоть до самого конца этого файла.

Некоторые IDE устанавливают NDEBUG по умолчанию, как часть параметров проекта в конфигурации Release.

Инструкция static_assert()

В C++11 добавили еще один тип assert-а — static_assert. В отличие от assert, который срабатывает во время выполнения программы, static_assert срабатывает во время компиляции, вызывая ошибку компилятора, если условие не является истинным. Если условие ложное, то выводится диагностическое сообщение.

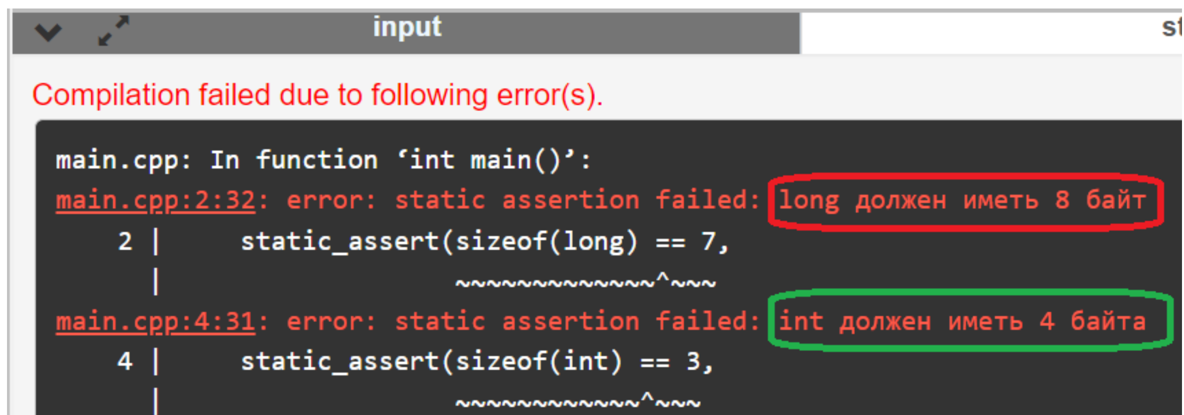
Рассмотрим пример использования static_assert для проверки размеров определенных типов данных.

Пример.

main.cpp

```
1 int main() {
2     static_assert(sizeof(long) == 7,
3                       "long должен иметь 8 байт");
4     static_assert(sizeof(int) == 3,
5                       "int должен иметь 4 байта");
6     return 0;
7 }
```

Результат работы программы:


Compilation failed due to following error(s).

```
main.cpp: In function 'int main()':
main.cpp:2:32: error: static assertion failed: long должен иметь 8 байт
    2 |     static_assert(sizeof(long) == 7,
      |                   ~~~~~~~~~~~~~^~~~
main.cpp:4:31: error: static assertion failed: int должен иметь 4 байта
    4 |     static_assert(sizeof(int) == 3,
      |                   ~~~~~~~~~~~~^~~~
```

Поскольку `static_assert` обрабатывается компилятором, то условная часть `static_assert` также должна обрабатываться во время компиляции.

Поскольку `static_assert` не обрабатывается во время выполнения программы, то инструкции `static_assert` могут быть размещены в любом месте кода (даже в глобальном пространстве).

Замечание.

В C++11 диагностическое сообщение должно быть обязательно предоставлено в качестве второго параметра. Начиная с C++17 предоставление диагностического сообщения является необязательным.

Простейшие численные методы

Численные методы — это раздел математики, содержащий методы решения математических, физических, инженерных и экономических задач в численном виде. В основе численных методов лежат алгоритмические схемы обработки информации с целью нахождения приближенного решения рассматриваемой задачи в числовой форме. В настоящее время в связи с бурным развитием и миниатюризацией вычислительных устройств численные методы стали основным инструментом решения современных прикладных задач, т.к. аналитическое решение задачи можно найти далеко не всегда в силу сложного, во многих случаях, нелинейного, вида систем уравнений, описывающих задачу.

Правило округления чисел

Чтобы округлить число до `n` значащих цифр, отбрасывают все цифры, стоящие справа от `n`-й значащей цифры. При этом:

- если первая отброшенная цифра меньше 5, то оставшиеся десятичные знаки сохраняют без изменения;
- если первая отброшенная цифра больше либо равна 5, то к последней оставшейся цифре прибавляют единицу;
- если первая отброшенная цифра равна 5 и среди остальных отброшенных цифр есть ненулевые, то к последней оставшейся цифре прибавляют единицу;
- если первая из отброшенных цифр равна 5 и все отброшенные цифры являются нулями, то последняя оставшаяся цифра оставляется неизменной, если она четная, и увеличивается на единицу, если нет (правило четной цифры).

Это правило гарантирует, что сохраненные значащие цифры числа являются верными в узком смысле, т. е. погрешность округления не превосходит половины разряда, соответствующего последней оставленной значащей цифре. Правило четной цифры должно обеспечить компенсацию знаков ошибок.

Замечание.

При округлении целого числа отброшенные знаки следует заменять на соответствующие степени 10 (символическое представление порядка).

Элементы теории погрешностей

Значащие цифры числа называются все цифры в его записи, начиная с первой ненулевой цифры слева.

Пример

- $x = 2.396029$ – все цифры и нуль значащие;
- $x = 0.00267$ – значащие только цифры 2, 6, 7, т.к. можно записать $x = 2.67 \cdot 10^{-3}$;
- $x = 2\ 270\ 000$ – если предполагается, что число точное, то все цифры значащие.

Точность числа определяется количеством значащих цифр, если число $x = 3200$ получено с точностью до двух значащих цифр, то для него следует использовать запись $x = 3.2 \cdot 10^3$, т.к. два нуля в исходной записи числа представляют собой незначащие цифры.

Значащая цифра в записи числа называется ***верной***, если абсолютная погрешность вычисляемого действительного числа не превосходит 0.5 единицы разряда, соответствующего этой цифре.

Замечание.

Если число 0.0213 *было вычислено с заданной точностью* 0.0005*, то верными цифрами следует считать* 2 *и* 1*, т.к. это число можно представить* $2.1 \cdot 10^{-3}$.

Если число имеет лишь верные цифры, то его округленное значения также имеет лишь верные цифры.

Те цифры, которые превосходят количество верных цифр справа в записи числа называются ***запасными***. Их относят к разряду ***сомнительных*** (это те, которые не попали под определение верных).

Виды погрешностей

Пусть x – точное значение величины, а $\tilde{x}$ – ее приближенное значение. Различают два вида погрешностей – ***абсолютную*** и ***относительную***:

- ***абсолютная погрешность*** $\Delta\tilde{x}$ некоторого числа равна разности между его точным значением x и приближенным значением $\tilde{x}$, полученным в результате вычисления или измерения:

$$\Delta\tilde{x} = |x - \tilde{x}|.$$

- ***относительная погрешность*** $\delta\tilde{x}$ – это отношение абсолютной погрешности к приближенному значению числа:

$$\delta\tilde{x} = \frac{\Delta x}{\tilde{x}} = \frac{|x - \tilde{x}|}{\tilde{x}}.$$

Погрешность арифметических действий над приближенными числами

При выполнении операций над приближенными числами можно оценить предельную погрешность результата в зависимости от выполняемой операции. При сложении или вычитании чисел их ***абсолютные погрешности*** складываются:

$$\Delta(\tilde{x} \pm \tilde{y}) = \Delta\tilde{x} + \Delta\tilde{y}.$$

Относительная погрешность суммы положительных слагаемых вычисляется как:

$$\delta(\tilde{x}+\tilde{y}) = \frac{\Delta(\tilde{x}+\tilde{y})}{\tilde{x}+\tilde{y}} = \frac{\Delta\tilde{x}}{\tilde{x}}\frac{\tilde{x}}{\tilde{x}+\tilde{y}} + \frac{\Delta\tilde{y}}{\tilde{y}}\frac{\tilde{y}}{\tilde{x}+\tilde{y}} = \frac{\tilde{x}\cdot\delta\tilde{x}+\tilde{y}\cdot\delta\tilde{y}}{\tilde{x}+\tilde{y}}.$$

Поскольку очевидно, что $min(\delta\tilde{x},\delta\tilde{y}) \le \delta\tilde{x} \le max(\delta\tilde{x},\delta\tilde{y})$, а также $min(\delta\tilde{x},\delta\tilde{y}) \le \delta\tilde{y} \le max(\delta\tilde{x},\delta\tilde{y})$, то из верхнего равенства отсюда следует, финальная оценка:

$$0 \le \delta(\tilde{x}+\tilde{y}) \le \frac{\tilde{x}\cdot max(\delta\tilde{x},\delta\tilde{y}) + \tilde{y}\cdot max(\delta\tilde{x},\delta\tilde{y})}{\tilde{x}+\tilde{y}} \le max(\delta\tilde{x},\delta\tilde{y}).$$

Замечание.

На практике именно последнее неравенство используется для оценки относительной погрешности при сложении чисел.

При ***умножении или делении*** чисел друг на друга их ***относительные*** погрешности складываются:

$$\delta(\tilde{x}\cdot\tilde{y}) = \delta\tilde{x} + \delta\tilde{y}.$$

При ***возведении в степень*** приближенного числа его ***относительная*** погрешность умножается на показатель степени:

$$\delta(\tilde{x}^k) = k\cdot\delta\tilde{x}.$$

При умножении приближенного числа $\tilde{x}$ на точный множитель μ абсолютная погрешность возрастает в μ раз ($\Delta(\mu\cdot\tilde{x}) = \mu\cdot\Delta\tilde{x}$), а относительная не изменяется ($\delta(\mu\cdot\tilde{x}) = \delta(\tilde{x})$).

Элементарные методы решения нелинейных уравнений

Отделение корней уравнения

Пусть задана непрерывная функция $f(x)$. Требуется определить корни уравнения $f(x) = 0$. Такая задача встречается в различных областях научных исследований, в том числе и при управлении строительным производством. Методы решения уравнений делятся на прямые и итерационные. Прямые методы позволяют записать корни в виде некоторого конечного соотношения. Если не удается решить уравнения прямыми методами, то для их решения используются итерационные методы, т.е. методы последовательных приближений.

Алгоритм нахождения корня уравнения с помощью итерационного метода состоит из двух этапов:

- отделения корня - отыскания приближенного значения одного из корней или отрезка, содержащего единственный корень;
- уточнения значения интересующего корня до некоторой заданной точности в смысле абсолютной погрешности.

Приближенное значение единственного корня (начальное приближение) может быть найдено различными способами из физических соображений, из решения аналогичной задачи при других исходных данных, с помощью графических методов (непосредственно из построенного графика).

Метод деления отрезка пополам (метод дихотомии)

Описание метода

Деление отрезка пополам (метод дихотомии) — это численный метод нахождения (одного) решения x (с заданной точностью ε) нелинейного уравнения вида:

$$f(x) = 0.$$

Допустим, что каким-либо образом найден отрезок $[a; b]$, в котором расположено искомое значение ***единственного*** корня рассматриваемого уравнения (Рисунок 37).

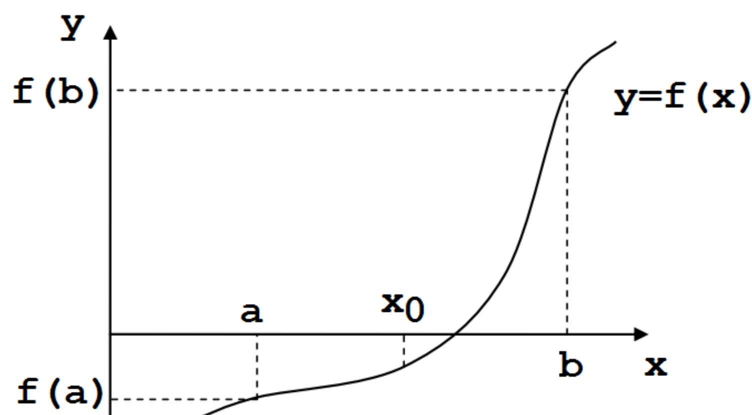


Рисунок 37 – Графическая интерпретация исходных данных для метода деления отрезка пополам

Пусть для определенности $f(a)<0$, $f(b)>0$ (Рисунок 37). В качестве начального приближения корня $\tilde{x}$ принимается середина этого отрезка, т.е. $x_0 = (a + b)/\ 2$. Далее исследуем значение функции $f(x)$ на концах отрезков

$[a\,; x_0\,]$ и $[\,x_0; b\,]$. Тот из них, на концах которого $f(x)$ принимает значения разных знаков, содержит искомый корень. Поэтому его принимаем в качестве нового отрезка, а вторую половину исходного отрезка $[a;\ b]$ отбрасываем.

В качестве первой итерации нахождения корня принимаем середину нового отрезка и т. д. Таким образом, после каждой итерации отрезок, на котором расположен корень, уменьшается вдвое, т.е. после n итераций он сокращается в 2^n раз. Если длина полученного отрезка становится меньше наперед заданной погрешности, т.е. $|\,b-a\,|<\varepsilon$, счет прекращается.

Пример реализации метода деления отрезка пополам

Напишем метод деления отрезка пополам сразу в наиболее обобщенном виде, но без использования указателей на функции. Для удобства и взаимно-однозначного понимания теории и практики функцию, которая задает нелинейное уравнение, обозначим через $f(x) = x^2 - 1$. Таким образом будем решать уравнение:

$$x^2 - 1 = 0.$$

Оно имеет два корня:

- $x_1 = -1$, локализованном на отрезке $[-2; 0]$;
- $x_2 = 1$, локализованном на отрезке $[0; 2]$.

Указывая любой из этих отрезков при начале работы программы, пользователь неявно определяет какой именно корень он будет искать с заданной точностью.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  double f(double);
6  double halfDivision(double, double, double);
7
8  int main() {
9      double a, b, eps;
10     cout<< "Введите отрезок на котором локализ. "
```

```
11            << "единственный корень уравнения [a, b]:"
12            << endl;
13        cin >> a >> b;
14        cout<< "Введите точность:" << endl;
15        cin >> eps;
16
17        cout<<"Корень равен = "<<halfDivision(a, b, eps);
18        return 0;
19    }
20
21    double f(double x) {
22        return x * x - 1;
23    }
24
25    double halfDivision(double a, double b, double eps) {
26        double x = (a + b) / 2;
27        while( f(x) != 0 && fabs(b - a) >= eps) {
28            if (f(a) * f(x) < 0) b = x;
29            else a = x;
30        }
31        return (a + b) / 2;
32    }
```

Результаты работы программы:

```
input
Введите отрезок на котором локализ. единственный корень уравнения [a, b]:
0 2
Введите точность:
0.005
Корень равен = 1
```

Метод простых итераций

Описание метода

Для использования этого метода исходное нелинейное уравнение

$$f(x) = 0$$

необходимо привести к виду

$$x = \varphi(x).$$

В качестве $\varphi(x)$ можно принять функцию

$$\varphi(x) = x - \frac{f(x)}{M},$$

где M - неизвестная постоянная величина, которая определяется из условия сходимости метода простой итерации $0 < |\varphi'(x)| < 1$.

Если известно начальное приближение корня $x = x_0$, подставляя это значение в правую часть уравнения $x = \varphi(x)$, получаем новое приближение $x_1 = \varphi(x_0)$. Далее подставляя каждый раз новое значение корня в уравнение $x = \varphi(x)$, получаем последовательность значений:

$$x_2 = \varphi(x_1), \qquad x_3 = \varphi(x_2) \quad ,\ldots, \qquad x_n = \varphi(x_{n-1}).$$

Итерационный процесс прекращается, если результаты двух последовательных итераций близки, т.е. $|x_{k+1} - x_k| < \varepsilon$.

Пример реализации метода простой итерации

Будем искать единственный корень уравнения:

$$x = \frac{x^2}{5}.$$

В качестве нулевого приближения выберем точку $a = 1$.

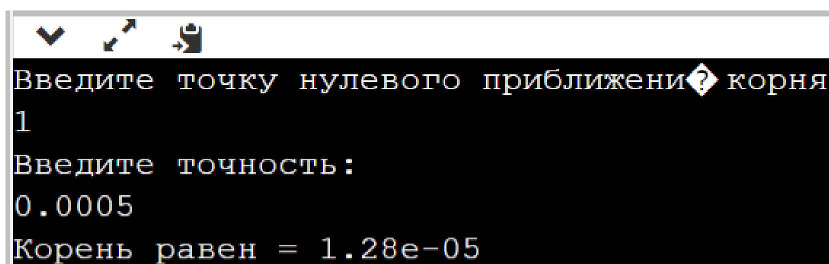
Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  double f(double);
6  double simpleIteration(double, double);
7
8  int main() {
9      double a, eps;
10     cout<< "Введите точку нулевого приближения корня"
11         << endl;
12     cin >> a;
```

```
13        cout<< "Введите точность:" << endl;
14        cin >> eps;
15
16        cout<<"Корень равен = "<<simpleIteration(a, eps);
17        return 0;
18    }
19
20    double f(double x) {
21        return x * x / 5;
22    }
23
24    double simpleIteration(double a, double eps) {
25        double currentX = a;
26        double nextX = f(a);
27        while(fabs(currentX - nextX) >= eps) {
28            currentX = nextX;
29            nextX = f(currentX);
30        }
31        return currentX;
32    }
```

Результат работы программы:


```
Введите точку нулевого приближени� корня
1
Введите точность:
0.0005
Корень равен = 1.28e-05
```

Метод Ньютона (метод касательных)

Базовые сведения о производной

Пусть функция $y = f(x)$ определена в точках x и $x + \Delta x$, где Δx – приращение аргумента. Разность $f(x) - f(x + \Delta x)$ называют приращением функции и обозначают Δf (Рисунок 38).

Определение.

Производной функции f называется предел отношения приращения значения функции Δf к приращению аргумента, когда приращение аргумента Δx стремится к нулю и обозначается f':

$$f' = \frac{\Delta f}{\Delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \text{ при } \Delta x \to 0.$$

Используя определение предела (или хотя бы интуитивное понятие о нем) словесное определение производной можно переписать в формальном виде:

$$f' = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Если функция $f(x)$ имеет производную в точке x, то эта функция называется дифференцируемой в этой точке. Если функция $f(x)$ имеет производную в каждой точке некоторого промежутка, то эта функция дифференцируема на этом промежутке. Операция нахождения производной называется дифференцированием.

Геометрический смысл производной

Отношение $\frac{f(x+\Delta x)-f(x)}{\Delta x}$ соответствует тангенсу угла наклона хорды AB (секущей линии) (Рисунок 38). Когда $\Delta x \to 0$, точка B стремится к точке A, при этом хорда превращается в касательную к графику функции, проходящую через точку $(x, f(x))$ на координатной плоскости (Рисунок 38). Соответственно, предел отношения превращается в тангенс угла наклона касательной в точке x. Итак, $f'(x) = tg\alpha$ - значение производной равно тангенсу угла наклона касательной к графику функции, проходящей через точку с координатами $(x, f(x))$ (Рисунок 38).

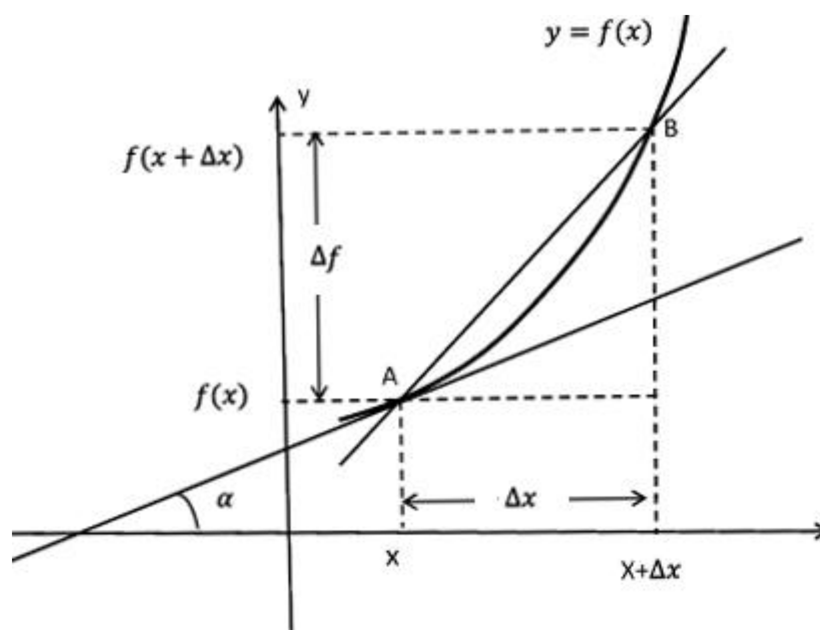


Рисунок 38 – Геометрическая интерпретация производной

Можно приращение аргумента Δx обозначать через dx, а приращение значения функции f через df. Тогда производная получит еще одно обозначение, называемым дифференциалом функции f, обозначаемым $\frac{df}{dx}$:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = f'.$$

Основные соотношения

Рассмотрим применение метода Ньютона для решения нелинейного уравнения вида:

$$f(x) = 0.$$

где $f(x)$ – непрерывно-дифференцируемая функция.

Если уже определен отрезок $[a;\ b]$ на котором существует единственный интересующий исследователя корень рассматриваемого уравнения, то за начальное приближение x_0, необходимое для запуска итерационного процесса необходимо принять один из концов отрезка.

Иногда бывает довольно сложно выбрать начальное приближение. В этом случае проще экспериментально выбрать сначала один конец отрезка и запустить метод для решения, а в случае зацикливания просто назначить второй конец начальным приближением.

Возможно иногда, придется дополнительно сузить интервал единственности корня. Все зависит от опыта исследователя при применении данного метода.

Замечание.

В некоторых учебных пособиях авторы «предлагают» какие-то интуитивно-экспериментальные неравенства, выполнение которых якобы обеспечивает сходимость метода, например:

$$f(x_0) \cdot f''(x_0) > 0,$$

но это не универсальные теоретически обоснованные рекомендации, и их выполнение в некоторых случаях может приводить к желаемому результату, а в некоторых нет.

Для получения уравнения для итерационного определения корня методом Ньютона следует предположить, что существует некоторая бесконечная последовательность точек $\{x_n\}_{n=0}^{\infty}$ координатной оси $0x$

сходящаяся к корню a. Для любой пары соседних точек x_n и x_{n+1} перепишем асимптотическое определение производной в точке x_n:

$$f'(x_n) \approx \frac{f(x_{n+1}) - f(x_n)}{x_{n+1} - x_n}.$$

Предполагается, что последовательность $\{x_n\}_{n=0}^{\infty}$ такова, что $\{f(x_n)\}_{n=0}^{\infty}$ ***монотонно*** стремиться к нулю (т.к. решается уравнение $f(x) = 0$), следовательно $|f(x_n)| > |f(x_{n+1})|$. Тогда выполняется:

$$1 > \frac{|f(x_{n+1})|}{|f(x_n)|}.$$

Отсюда можно сделать вывод, что $|f(x_n)/f'(x_n)|$ мало по сравнению с единицей. В правой части исходного асимптотического уравнения, определяющего производную в точке x_n вынесем отношение $f(x_n)/f'(x_n)$ за скобки. Получим:

$$f'(x_n) \approx \frac{(f(x_{n+1})/f(x_n) - 1) \cdot f(x_n)}{x_{n+1} - x_n}$$

Исходя из вышесказанного при решении нелинейного уравнения $f(x) = 0$ отношением $f(x_{n+1})/f(x_n)$ можно пренебречь, т.е.:

$$f'(x_n) \approx -\frac{f(x_n)}{x_{n+1} - x_n}.$$

Проведем последнее очевидное преобразование:

$$x_{n+1} - x_n \approx -\frac{f(x_n)}{f'(x_n)}.$$

Таким образом в окончательном варианте можно записать:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, n = 0,1, \dots .$$

Предполагается, что $f'(x_n) \neq 0$ ($\forall n = 0,1, \dots \infty$) во избежание деления на нуль. Из всех рассмотренных в данном разделе методов метод Ньютона является самым быстросходящимся, для достижения результата с помощью этого метода требуется наименьшее число итераций.

Пример реализации метода Ньютона

Напишем программу реализующую метод Ньютона без использования указателей на функции. Для удобства и взаимно-однозначного понимания теории и практики функцию, которая задает нелинейное уравнение, обозначим через $f(x) = x^4 - 16$. Таким образом будем решать нелинейное уравнение:

$$x^4 - 16 = 0.$$

Оно имеет два кратных корня:

- $x_1 = -2$, локализованном на отрезке $[-2.5; -0.5]$;
- $x_2 = 2$, локализованном на отрезке $[0.5; 2.5]$.

Указывая любой из этих отрезков при начале работы программы, пользователь неявно определяет какой именно корень будем искать с заданной точностью.

Пример.

main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  double f(double);
6  double df(double x);
7  double NewtonMethod(double, double);
8
9  int main() {
10     double a, eps;
11     cout<< "Введите точку нулевого приближения корня"
12         << endl;
13     cin >> a;
14     cout<< "Введите точность:" << endl;
15     cin >> eps;
16     cout<<"Корень равен = "<< NewtonMethod(a, eps);
17     return 0;
18 }
19
20 double f(double x) {
21     return pow(x, 4) - 16;
22 }
```

```
double df(double x) {
    return 4 * pow(x, 3);
}

double NewtonMethod(double a, double eps) {
    double xCurrent = a;
    double xNext  = a - f(a)/df(a);
    while (fabs(xNext - xCurrent)>eps) {
        xCurrent = xNext ;
        xNext = xCurrent - f(xCurrent)/df(xCurrent);
    }
    return xCurrent;
}
```

Результат работы программы:

```
Введите точку нулевого приближения корня
1
Введите точность:
0.005
Корень равен = 2
```

Вычисление определенных интегралов

Рассмотрим методы численного интегрирования. Под определенным интегралом Римана функции $f(x)$ на отрезке $[a, b]$ понимается площадь, отсекаемая криволинейной трапецией, образованной упомянутым отрезком и соответствующим участком функции $f(x)$ (Рисунок 39).

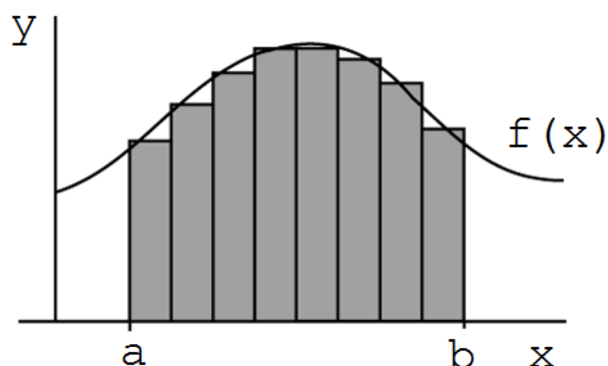


Рисунок 39 – Геометрическая интерпретация определенного интеграла

Символическая запись определенного интеграла функции $f(x)$ на отрезке $[a, b]$ имеет вид:

$$\int_a^b f(x)dx.$$

В литературе часто используют обозначение $I = \int_a^b f(x)dx$, либо иногда $I_a^b = \int_a^b f(x)dx$.

Поскольку фактически определенный интеграл – это сумма площадей прямоугольников, покрывающих криволинейную трапецию (Рисунок 39), то используется такое понятие как «интегральная сумма», обозначаемая через S_n, где n – количество прямоугольников, покрывающих трапецию.

Интуитивно понятно, что при увеличении количества прямоугольников $(n \to \infty)$ то и $S_n \to I_a^b$, т.е. I_a^b – это предельное значение, к которому сходятся интегральные суммы с увеличивающимся количеством интервалов в разбиении отрезка интегрирования $[a, b]$:

$$\lim_{n\to\infty} S_n = \int_a^b f(x)dx.$$

Поскольку формально интегральную сумму S_n можно записать несколькими способами, то в методах численного интегрирования выделяются несколько простейших квадратурных формул, которые именуются способами вычисления интегральных сумм S_n.

Выделяют следующие квадратурные формулы:

- левых прямоугольников;
- правых прямоугольников;
- средних прямоугольников;
- формула трапеций.

Следует отметить, что обычно при изучении темы квадратурных формул в рамках дисциплины «Численные методы» обычно рассматривается еще и формула Симпсона, но в рамках данного курса она рассматриваться не будет.

Кроме того, в литературе по математическому анализу можно найти еще две квадратурные формулы, использующиеся исключительно для доказательства некоторых теорем в рамках математического анализа и не упоминаемых в численных методах:

- нижняя сумма Дарбу;
- верхняя сумма Дарбу.

Квадратурные формулы левых, правых и средних прямоугольников

Разобьем отрезок интегрирования $[a; b]$ на n равных частей: $x_0 = a$, $x_1 = x_0 + h$, $x_2 = x_1 + h$, … , $x_n = b$, где $h = (b - a)/n$. Площадь всей криволинейной трапеции заменим суммой площадей криволинейных трапеций, образованных при проведении вертикальных прямых $x = x_i$.

Заменим при вычислении площади ***каждую*** «маленькую» криволинейную трапецию на интервалах $[x_i; x_{i+1}]$ соответствующим прямоугольником. Фактически заменим кривую $f(x)$ ступенчатым приближением.

При этом прямоугольники на основании отрезков $[x_i, x_{i+1}]$ можно построить тремя разными способами взяв за одну из сторон прямоугольника значение (Рисунок 40):

- $f(x_i)$ – квадратурная формула левых прямоугольников;
- $f(x_{i+1})$ – квадратурная формула правых прямоугольников;
- $f\left(\frac{x_i+x_{i+1}}{2}\right)$ – квадратурная формула средних прямоугольников.

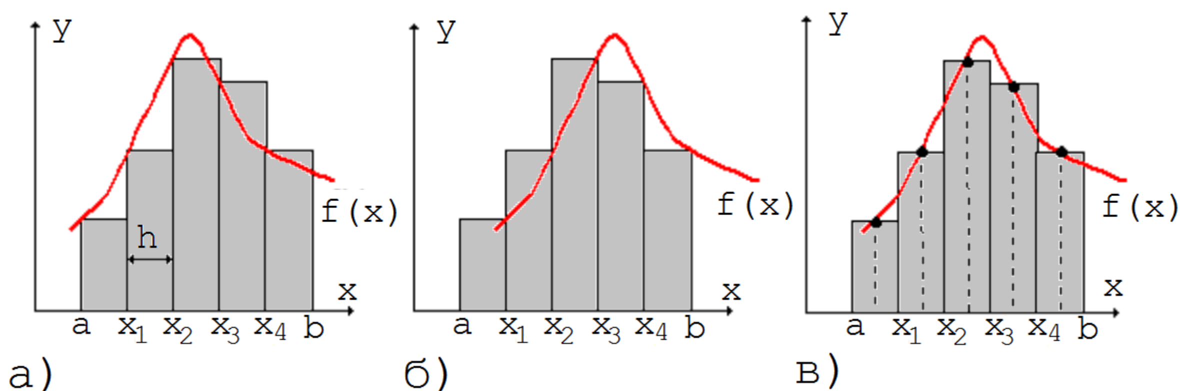


Рисунок 40 – Различия в формировании интегральных сумм с помощью разных квадратурных формул: а) левые прямоугольники; б) правые прямоугольники; в) средние прямоугольники

Исходя из геометрической интерпретации формирования интегральных сумм можно записать в явном виде следующие квадратурные формулы:

- левые прямоугольники (Рисунок 40, а):

$$I_a^b \approx (f(x_0) \cdot h + f(x_1) \cdot h + \dots + f(x_{n-1}) \cdot h) = h \cdot \sum_{i=0}^{n-1} f(x_i);$$

- правые прямоугольники (Рисунок 40, б):

$$I_a^b \approx (f(x_1) \cdot h + f(x_2) \cdot h + \dots + f(x_n) \cdot h) = h \cdot \sum_{i=0}^{n-1} f(x_{i+1});$$

- средние прямоугольники (Рисунок 40, в):

$$I_a^b \approx \left(f\left(\frac{x_0 + x_1}{2}\right) \cdot h + f\left(\frac{x_1 + x_2}{2}\right) \cdot h + \cdots + f\left(\frac{x_{n-1} + x_n}{2}\right) \cdot h\right) = h \cdot \sum_{i=0}^{n-1} f\left(\frac{x_i + x_{i+1}}{2}\right);$$

Формула трапеций

В этом методе на каждом отрезке $[x_{i-1}, x_i]$ строиться не аппроксимирующий прямоугольник, а аппроксимирующая трапеция (Рисунок 41).

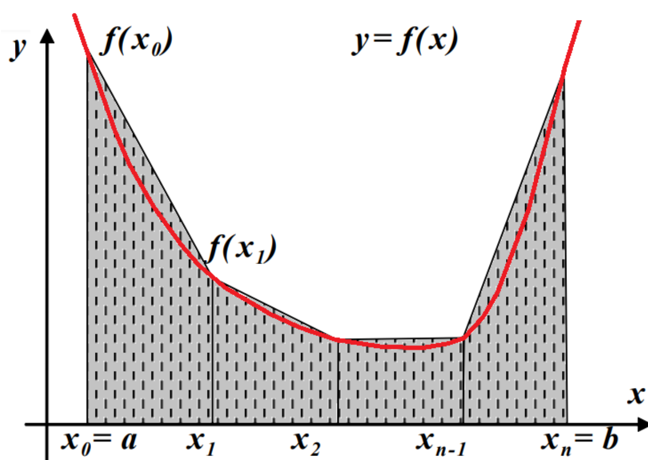


Рисунок 41 – Геометрическая интерпретация формирования квадратурной формулы трапеций

Запишем в явном виде формулу трапеций

$$I_a^b \approx \left(\frac{f(x_0) + f(x_1)}{2} \cdot h + \frac{f(x_1) + f(x_2)}{2} \cdot h + \cdots + \frac{f(x_{n-2}) + f(x_{n-1})}{2} \cdot h + \frac{f(x_{n-1}) + f(x_n)}{2} \cdot h\right) = h \cdot \left(\frac{f(x_0) + f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i)\right)$$

Нижняя и верхняя суммы Дарбу

В рассматриваемых суммах на каждом отрезке $[x_{i-1}, x_i]$ для верхней суммы Дарбу (I_{max}) выбирается max из соседних значений $\{f(x_{i-1}), f(x_i)\}$ подъинтегральной функции, а для нижней суммы Дарбу (I_{min}) выбирается min из этих же значений (Рисунок 42).

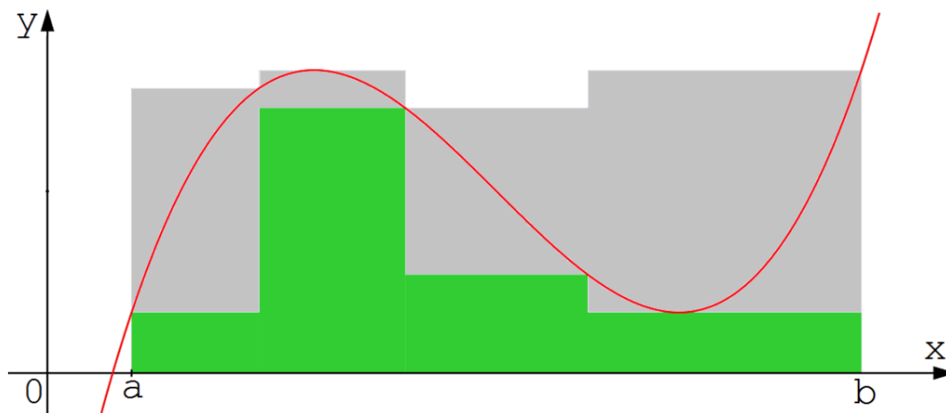


Рисунок 42 – Геометрическая интерпретация сумм Дарбу (зеленый цвет нижняя, а серый цвет - верхняя)

Таким образом, формально запись для I_{max} и I_{min} выглядит следующим образом:

$$I_{max} \approx (max\{f(x_0), f(x_1)\} \cdot h + \cdots + max\{f(x_{n-1}), f(x_n)\} \cdot h) = \\ = h \cdot \sum_{i=1}^{n} max\{f(x_{i-1}), f(x_i)\};$$

$$I_{min} \approx (min\{f(x_0), f(x_1)\} \cdot h + \cdots + min\{f(x_{n-1}), f(x_n)\} \cdot h) = \\ = h \cdot \sum_{i=1}^{n} min\{f(x_{i-1}), f(x_i)\};$$

Пример вычисления нижней суммы Дарбу

Рассмотрим функцию, реализующую вычисление определенного интеграла от функции $f(x)$ на отрезке $[a, b]$ с помощью нижней суммы Дарбу. При этом передаваться подынтегральная функция будет через указатель на функцию (параметр `double (*f)(double)`).

Пример.

```
double minDarboux(double a, //начало отрезка интегр.
                  double b, //конец отрезка интегр.
                  int n,//количество отрезков в разб.
                  double (*f)(double))//подинтег. ф.
{
    //использование fabs() позволяет пользователю
    //не следить за порядком концов интервала интегрир.
    double h = fabs(b - a) / n;
    double x = a;
    double integralSum = 0;
    for(int i = 1 ; i < n; i++) {
        double minValue =
                    f(x) < f(x + h) ? f(x) : f(x + h);
        integralSum = integralSum + minValue;
        x = x + h;
    }
    return integralSum * h;
}
```

Правило Рунге (правило двойного пересчета)

Правило Рунге — правило оценки погрешности численных методов, использующих равноотстоящие узлы разбиения на заданном ограниченном интервале численного решения какой-либо задачи. Оно было предложено К. Рунге в начале 20 века. Основная идея состоит в вычислении приближения выбранным методом на заданном отрезке с разбиением его на n интервалов, а затем на $2 \cdot n$ интервалов, и дальнейшем вычислении разности результатов этих двух приближений.

Увеличивая количество интервалов на каждой итерации в два раза, можно ожидать, что через некоторое время будет достигнута необходимая точность численного решения.

Правило носит экспериментально-практический характер, теоретического обоснования не имеет, но закрепилось в вычислительной математике как традиционное при численном решении многих задач.

В случае организации итерационного вычисления определенных интегралов правило Рунге, определяющее остановку вычислений, имеет вид:

$$|S_n - S_{2 \cdot n}| < \varepsilon.$$

где ε – заданная точность вычислений интеграла.

Оно указывает, что увеличение числа интервалов разбиения отрезка интегрирования при итеративной процедуре вычислений происходит каждый раз в два раза и завершается, когда приведенное неравенство будет выполнено. При этом за значение интеграла принимается величина S_n (но не $S_{2 \cdot n}$).

Рассмотрим функцию `integralIterator()`, которая будет организовывать итерирование по правилу Рунге вычислений определенного интеграла с помощью функции, определяющей интегральную сумму с использованием какой-либо квадратурной формулы `sum` (тип указателя на функцию `quadratureFormula`).

Пример.

```
typedef double (*quadratureFormula)(double,
                                    double,
                                    int,
                                    double (*)(double));

double integralIterator(double a,
                        double b,
                        double eps,
                        quadratureFormula sum,
                        double (*f)(double)) {
    int n = 50;
    double currentSum = sum(a, b, n, f);
    double nextSum = sum(a, b, 2*n, f);
    while( fabs(currentSum - nextSum) >= eps) {
        n = 2 * n;
        currentSum = nextSum;
        nextSum = sum(a, b, 2*n, f);
    }
    return currentSum;
}
```

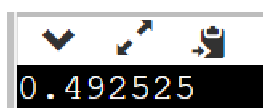
Пример программной реализации вычисления определенного интеграла с заданной точностью

main.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

double function(double x);

double minDarboux(double,
                  double,
                  int,
                  double (*)(double));

typedef double (*quadratureFormula)(double,
                                    double,
                                    int,
                                    double (*)(double));

double integralIterator(double,
                        double,
                        double,
                        quadratureFormula sum,
                        double (*)(double));

int main() {
    cout<< integralIterator(0, 1, 0.005,
                            minDarboux, function);
    return 0;
}

double function(double x) {
    return x;
}

double minDarboux(double a,
                  double b, int n,
                  double (*f)(double)) {
    double h = fabs(b - a) / n;
    double x = a;
```

```
38     double integralSum = 0;
39     for(int i = 1 ; i < n; i++) {
40         double minValue =
41                     f(x) < f(x + h) ? f(x) : f(x + h);
42         integralSum = integralSum + minValue;
43         x = x + h;
44     }
45     return integralSum * h;
46 }
47 
48 double integralIterator(double a,
49                         double b,
50                         double eps,
51                         quadratureFormula sum,
52                         double (*f)(double)) {
53     int n = 50;
54     double currentSum = sum(a, b, n, f);
55     double nextSum = sum(a, b, 2 * n, f);
56     while(fabs(currentSum - nextSum) >= eps) {
57         n = 2 * n;
58         currentSum = nextSum;
59         nextSum = sum(a, b, 2 * n, f);
60     }
61     return currentSum;
62 }
```

Результат работы программы:



Особенности вычисления определенного интеграла с точностью с помощью сумм Дарбу

Необходимо отметить, что вычисление с точностью по правилу Рунге отдельно верхней или нижней сумм Дарбу – это неверный шаг с точки зрения теории. Хотя с практической точки зрения все выглядит обосновано. Пример приведенный выше написан, только для демонстрации того, как реализуется правило Рунге в программировании.

С другой стороны, следует помнить исходя из геометрического смысла (Рисунок 42), что разность верхней и нижней сумм Дарбу, собственно, и

оценивает точность вычисления интеграла ***на одном и том же разбиении*** (в отличие от правила Рунге).

Таким образом, условием остановки вычислений в случае использования интегральных сумм Дарбу является неравенство:

$$\left|S_n^{max} - S_n^{min}\right| < \varepsilon,$$

где S_n^{max} – верхняя сумма Дарбу, S_n^{min} – нижняя сумма Дарбу, ε – заданная точность вычислений интеграла.

Поэтому в случае использования сумм Дарбу вычисление значения определенного интеграла можно несколько упростить, убрав один в данном случае ненужный указатель на функцию определяющую квадратурную формулу.

С другой стороны, следует также отметить, что и верхняя и нижняя суммы Дарбу алгоритмически полностью совпадают до знака сравнения (больше / меньше) в тернарной операции. Поэтому для описания обоих сумм Дарбу можно использовать код только одной функции, в которую следует передавать указатель на функцию, выполняющую сравнение (так называемый компаратор). Как это ранее выполнялось в функции выбора экстремального значения (максимума или минимума) в одномерном массиве.

Пример.

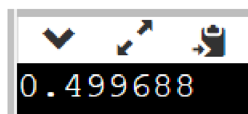
main.cpp

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  //Подинтегральная функция
6  double function(double x);
7
8  //Функции компараторы
9  bool moreThan(double, double);
10 bool lessThan(double, double);
11
12 //Универсальная формула суммы Дарбу
13 double Darboux(double,
14              double,
15              int,
16              double (*)(double),       //подинт. функ.
17              bool (*) (double, double)); //компаратор
18
```

```
19 double DarbouxIterator(double,
20                         double,
21                         double,
22                         double (*)(double));
23
24 int main() {
25     cout<< DarbouxIterator(0, 1, 0.0005, function);
26     return 0;
27 }
28
29 double function(double x) {
30     return x;
31 }
32
33 bool moreThan(double a, double b) {
34     return a > b;
35 }
36
37 bool lessThan(double a, double b) {
38     return a < b;
39 }
40
41 double Darboux(double a,
42                 double b, int n,
43                 double (*f)(double),
44                 bool (*than) (double, double)) {
45     double h = fabs(b - a) / n;
46     double x = a;
47     double integralSum = 0;
48     for(int i = 1 ; i < n; i++) {
49         double minValue =
50               than(f(x), f(x + h)) ? f(x) : f(x + h);
51         integralSum = integralSum + minValue;
52         x = x + h;
53     }
54     return integralSum * h;
55 }
56
57 double DarbouxIterator(double a,
58                         double b,
59                         double eps,
60                         double (*f)(double)) {
```

```
61        int n = 50;
62        double minInteg = Darboux(a, b, n, f, lessThan);
63        double maxInteg = Darboux(a, b, n, f, moreThan);
64        while(maxInteg - minInteg >= eps) {
65            n = 2 * n;
66            minInteg = Darboux(a, b, n, f, lessThan);
67            maxInteg = Darboux(a, b, n, f, moreThan);
68        }
69        return (minInteg + maxInteg) / 2;
70    }
```

Результат работы программы:



Численное решение обыкновенного дифференциального уравнения первого порядка

Будем рассматривать дифференциальное уравнение вида:

$$y' = f(t, y),$$

где $f(t, y)$ — заданная непрерывная функция в двумерной области D ($D \subset R \times R = R^2$).

Пусть требуется найти решение задачи Коши (1)-(2) на отрезке $t \in [a, b]$. Это решение должно удовлетворять начальному условию:

$$y(t_0) = y_0,$$

где введено обозначение $t_0 = a$, т.е. начальной точке интервала решения дифференциального уравнения.

<u>Замечание.</u>
Существует сленговое различие в терминологии:

- *если t – переменная по времени, то говорят о начальном условии;*
- *если в место времени t используется координата (например x), то говорят о краевом условии.*

Описание метода Эйлера

Разобьем отрезок $[a, b]$ на n равных частей точками:

$$t_i = a + i \cdot h,$$

где $i = 0, \dots, n$, $h = \frac{b-a}{n}$.

На основе системы равноотстоящих узлов $\{t_i\}_{i=0}^{n}$ гипотетически построим систему дискретных значений функции y (предполагая, что она уже известна). Обозначим эту систему значений следующим образом $\{y_i = y(t_i)\}_{i=0}^{n}$.

Очевидно, что в результате численного решения задачи Коши для дифференциального уравнения первого порядка будут известны только эти две системы дискретных значений исходная (система равноотстоящих узлов $\{t_i\}_{i=0}^{n}$) и построенная с помощью численного метода (система значений функции $\{y_i\}_{i=0}^{n}$).

Преобразуем определение производной $y' = \frac{\Delta y}{\Delta t}$ в приближенное равенство с использованием этих двух систем дискретных значений:

$$y'_i \approx \frac{y_i - y_{i-1}}{h}.$$

Замечание.
Эта замена называется заменой производной ее разностным аналогом.

Тогда исходное дифференциальное уравнение можно переписать в виде:

$$\frac{y_i - y_{i-1}}{h} = f(t_{i-1}, y_{i-1}).$$

После несложных математических операций можно получить окончательное рекуррентное соотношение метода Эйлера:

$$y_i = y_{i-1} + h \cdot f(t_{i-1}, y_{i-1}).$$

Смысл метода Эйлера заключается в том, что каждое последующее значение y_i вычисляется на основании знания предыдущего значения y_{i-1}, а также значения предыдущего узла t_{i-1}. Начало же вычислений задается предопределенным начальным условием.

Пример программной реализации метода Эйлера

Для того, чтобы написать функцию, решающую методом Эйлера дифференциальное уравнение первого порядка необходимо предположить, что уже определена одноименная функция $f(x,y)$.

Кроме того, будем предполагать, что требуется сохранить в ***куче*** в виде массива с именем `result` и размерностью `n` вычисленных значений $\{y_i\}_{i=0}^{n}$.

Для решения данной задачи разработаем многофайловый проект, имеющий структуру, представленную на рисунке (Рисунок 43).

Рисунок 43 – Структура проекта для численного решения дифференциального уравнения первого порядка методом Эйлера

Файлы проекта:

- `main.cpp` – точка входа, содержит вывод вспомогательных сообщений и ввод необходимых значений для запуска численной процедуры решения дифференциального уравнения, а также вызов функций:
  - решающей поставленную задачу и выводящей;
  - выводящую результаты решения на экран;
- `function.h` – заголовочный файл, содержащий прототипы функций, необходимых для реализации данного метода;
- `function.cpp` – текстовый файл, содержащий описание функций, перечисленных в одноименном заголовочном файле.

Пример.

//файл main.cpp

main.cpp function.h function.cpp

```
1  #include <iostream>
2  #include <cmath>
3  #include "function.h"
4
5  int main() {
6      double a, b, y0, h;
7      //исходные данные
8      std::cout<< "Введите отрезок [a, b] для метода Эйлера\n";
9      std::cin >> a >> b;
```

```
10      std::cout<< "Введите начальное значение функции f(a)\n";
11      std::cin >> y0;
12      int n = 100; //количество отрезков разбиения
13
14      //построение дискретного решения
15      double* result = Euler(a, b, y0, n);
16
17      //вывод результата на экран
18      std::cout<< "Результат применения метода Эйлера\n";
19      printEuler(result, a, b, n);
20
21      delete []result;
22      return 0;
23  }
```

//файл function.h

main.cpp | function.h | function.cpp

```
1  double f(double, double);
2  double* Euler(double, double, double, int);
3  void printEuler(double*, double, double, int);
```

//файл function.cpp

main.cpp | function.h | function.cpp

```
1   #include <iostream>
2   #include <cmath>
3
4   double f(double t, double y) {
5       return t;
6   }
7
8   double* Euler(double a, double b, double y0, int n) {
9       double h = fabs(b - a)/n;
10      double* y = new double[n];
11      int i = 0;
12      y[i] = y0;
13      for(double t = a; t <= b; t += h) {
14          i++;
15          y[i] = y[i - 1] + h * f(t, y[i - 1]);
16      }
17      return y;
18  }
19
20  void printEuler(double* result, double a, double b, int n) {
21      double h = fabs(b - a)/n;
22      int i = 0;
```

```
23      std::cout << a <<" "<< result[i]<< std::endl;
24      for(double t = a + h; t < b; t += h) {
25          std::cout << t <<" "<< result[++i]<< std::endl;
26      }
27  }
```

Результаты работы программы:

```
Введите отрезок [a, b] для метода Эйлера
0 1
Введите начальное значение функции f(a)
0
Результат применения метода Эйлера
0 0
0.01 0
0.02 0.0001
0.03 0.0003
0.04 0.0006
0.05 0.001
0.06 0.0015
0.07 0.0021
0.08 0.0028
0.09 0.0036
0.1 0.0045
```

```
0.9 0.4005
0.91 0.4095
0.92 0.4186
0.93 0.4278
0.94 0.4371
0.95 0.4465
0.96 0.456
0.97 0.4656
0.98 0.4753
0.99 0.4851
```

Замечание.

Полный код многофайлового проекта численно решающего дифференциальное уравнение первого порядка с помощью метода Эйлера доступен по ссылке URL: https://www.onlinegdb.com/edit/2cpZmu7xPd (дата доступа 02.02.2023)

Литература

1. Павловская, Т.А. С/С++. Программирование на языке высокого уровня / Т.А. Павловская. - Санкт-Петербург [и др.] : Питер, 2017. 460 с.
2. Демидович, Е.М. Основы алгоритмизации и программирования. Язык Си / Е.М. Демидович – Санкт-Петербург: BHV, 2008. – 440 с.
3. Подбельский, В.В. Программирование на языке Си / В.В. Подбельский, С.С. Фомин – М.: Финансы и статистика, 2001. – 600 с.
4. Монастырный, П.И. Системы нелинейных численных уравнений / П.И. Монастырный, М.В. Игнатенко, Н.П.Феденко и др. – Мн.: БГУ, 2003. – 30 с.
5. Бронштейн, И.Н. Справочник по математике для инженеров и учащихся втузов / И.Н. Бронштейн, К.А. Семендяев – М.: Наука, 1986. – 544 с.
6. Ахмадиев, Ф.Г. Численные методы. Примеры и задачи / Ф.Г. Ахмадиев, Ф.Г. Габбасов, Л.Б. Ермолаева, И.В. Маланичев. - Казань: КГАСУ, 2017. – 107 с.
7. Смалюк, А.Ф. Объектно-ориентированное программирование (язык С++) / А.Ф. Смалюк, Д.В. Макарчук, Д.С. Карпович. – Институт повышения квалификации и переподготовки кадров по новым направлениям развития техники, технологии и экономики БНТУ. – Мн.: БНТУ, 2006. – 72 с.
8. Аленский, Н.А. Практическое руководство по языку С++ / Н.А. Аленский, ГУО «Акад. Последипл. Образования». – Минск: АПО, 2007. – 276 с.
9. Пацей, Н.В. Основы алгоритмизации и программирования / Н.В. Пацей – Минск : БГТУ, 2010. – 289 с.
10. Кравчук, А.И. Сборник лабораторных работ и примеров решения задач по алгоритмизации и программированию на языке Си / А.И. Кравчук, А.С. Кравчук - Минск: Технопринт, 2002. - 116 с.
11. Аксенкин, М.А. Язык С / М.А. Аксенкин, О.Н. Целобенок - Мінск: Універсітэцкае, 1995. - 302 с.
12. Саркисян Г.Ф. Введение в программирование. Язык «С» / Г.Ф. Саркисян – Мн.: ЗАО «БелХард Групп», 2005 – 136 с.
13. Многофайловый проект. Изучение С++ для начинающих - [Электронный ресурс], URL: https://www.youtube.com/watch?v=pAxEfF2yVlM (дата обращения: 20.01.23)