

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

М.Л. РЫСИН, М.В. САРТАКОВ, О.В. МАКЕЕВА

**ОСНОВЫ ПРОГРАММИРОВАНИЯ
НА ЯЗЫКЕ C++**

УЧЕБНОЕ ПОСОБИЕ

Москва — 2022

УДК 004.43
ББК 32.973.26-018.1я73
Р 95

Рысин М.Л. Основы программирования на языке C++ [Электронный ресурс]: Учебное пособие / Рысин М.Л., Сартаков М.В., Макеева О.В. — М.: МИРЭА – Российский технологический университет, 2022. — 1 электрон. опт. диск (CD-ROM)

Учебное пособие посвящено начальным аспектам разработки компьютерных программ на языке высокого уровня C++. Показаны приёмы работы в инструментальной среде Visual Studio. Приводятся необходимые теоретические сведения и практические задания для самостоятельного освоения материала и организации практикума.

Материал пособия необходим при изучении дисциплин «Программирование на языке C++» и «Структуры и алгоритмы обработки данных».

Предназначено для студентов вузов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника», 09.03.03 «Прикладная информатика», 09.03.04 «Программная инженерия», 01.03.04 «Прикладная математика» и по другим направлениям IT-профиля. Пособие также может быть полезным для любых иных категорий читателей.

Учебное пособие издается в авторской редакции.

Авторский коллектив: Рысин Михаил Леонидович, Сартаков Михаил Валерьевич, Макеева Оксана Валерьевна.

Рецензенты:

Николаева Светлана Владимировна, д.т.н., профессор, профессор кафедры физики им. В.А. Фабриканта, ФГБОУ ВО «Национальный исследовательский университет «МЭИ»».

Ахмедова Хамида Гаджиалиевна, к.физ.-мат.н., доцент, доцент кафедры информатики и вычислительной техники, ФГБОУ ВО «Московский государственный университет пищевых производств».

Минимальные системные требования:

Наличие операционной системы Windows, поддерживаемой производителем.

Наличие свободного места в оперативной памяти не менее 128 Мб.

Наличие свободного места в памяти хранения (на жестком диске) не менее 30 Мб.

Наличие интерфейса ввода информации.

Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).

Подписано к использованию по решению Редакционно-издательского совета

МИРЭА – Российский технологический университет.

Объем 4,68 Мб

Тираж 10

ISBN 978-5-7339-1600-2

© Рысин М.Л., Сартаков М.В.,
Макеева О.В., 2022

© МИРЭА – Российский технологический университет,
2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	6
1. ЛИНЕЙНЫЕ ПРОГРАММЫ НА C++ В IDE VISUAL STUDIO	8
Понятие языка C++	8
Среда Visual Studio.....	8
Подготовка к практикуму.....	9
Пример программы на языке C++	10
Задание 1.1. Первый проект	11
Состав языка C++	14
Типы данных в языке C++.....	15
Переменные и константы	16
Задание 1.2. Программа с константой и переменной.....	18
Операции и выражения в C++. Ввод данных	18
Задание 1.3. Простейшие арифметические выражения	20
Математические вычисления в программах на C++	21
Задание 1.4. Пример использования математических функций	22
Индивидуальное задание №1	23
Вопросы для самоподготовки	24
2. ВЕТВЛЕНИЕ И ЦИКЛЫ В C++	25
Понятие ветвления. Условный оператор	25
Задание 2.1. Простые примеры бинарного ветвления.....	26
Множественное ветвление	27
Задание 2.2. Пример множественного ветвления	28
Индивидуальное задание №2.1	29
Понятие цикла	29
Цикл с предусловием while	30
Задание 2.3. Примеры использования цикла с предусловием. Отладка	30
Цикл с постусловием do-while	34
Задание 2.4. Примеры использования цикла с постусловием	34
Цикл с параметром for	35
Задание 2.5. Практика использования цикла с параметром	36
Индивидуальное задание №2.2	36
Вопросы для самоподготовки	37
3. УКАЗАТЕЛИ И МАССИВЫ В ЯЗЫКЕ C++	38
Понятие массива.....	38
Статические массивы.....	39
Задание 3.1. Приёмы работы со статическими линейными массивами	41

Память процесса.....	41
Указатели	43
Задание 3.2. Работа с указателями.....	46
Линейные массивы в «куче». Сортировка массива выбором.....	47
Задание 3.3. Приёмы работы с динамическими линейными массивами.....	49
Многомерные статические массивы в C++	49
Многомерные динамические массивы средствами C++	51
Задание 3.4. Приёмы работы с многомерными массивами в C++	53
Индивидуальное задание №3	53
Вопросы для самоподготовки	53
4. СИМВОЛЫ И СТРОКИ В ЯЗЫКЕ C++. СТРУКТУРЫ	55
Работа с символами в C++.....	55
Задание 4.1. Приёмы работы с символьными значениями	55
Строки в языке C++	56
Объявление и ввод-вывод C-строк в языке C++	56
Операции с C-строками	58
Задание 4.2. Ввод и поиск по строке	60
Структура	60
Задание 4.3. Приёмы работы со структурами в C++	62
Индивидуальное задание №4	62
Вопросы для самоподготовки	63
5. ПОДПРОГРАММЫ	64
Понятие подпрограммы.....	64
Функция main. Объявление, определение и вызов функций в C++.....	66
Задание 5.1. Пример создания и использования собственной функции	67
Параметры функций.....	67
Задание 5.2. Функция определения простого числа	69
Передача массивов и строк в функцию	69
Задание 5.3. Приёмы создания и использования функций в C++	70
Рекурсивные функции	71
Задание 5.4. Приёмы работы с рекурсивными функциями	71
Индивидуальное задание №5	72
Вопросы для самоподготовки	72
6. ФАЙЛЫ	73
Ввод-вывод: понятие.....	73
Работа с файлами в C++	74
Задание 6.1. Пример простого файлового ввода-вывода в C++.....	76

Битовое представление чисел и битовые операции.....	77
Задание 6.2. Приёмы использования битовых операций.	77
Задание 6.3. Задача сортировки числового файла с помощью битового массива	78
Индивидуальное задание № 6	79
Вопросы для самоподготовки	79
7. ОКОННЫЙ ИНТЕРФЕЙС	80
Создание нового проекта Windows Forms в среде Visual Studio.....	80
Этапы реализации Windows-приложения.....	82
Задание 7.1. Создание простого Windows-приложения	84
Элементы управления на форме: метка, кнопка, поле ввода	85
Задание 7.2. Игра «Угадай число!»	86
Элементы управления на форме: меню, флажок, переключатель, панель группировки, список	88
Задание 7.3. Практические приёмы использования управляющих элементов формы	89
Задание 7.4. Windows-приложение для работы с изображениями	92
Взаимодействие между формами	98
Индивидуальное задание №7	99
Вопросы для самоподготовки	100
8. ЭЛЕМЕНТЫ КОМПЬЮТЕРНОЙ ГРАФИКИ	101
Графическая поверхность.....	101
Задание 8.1. Простой пример прорисовки на графической поверхности	101
Карандаши и кисти.....	103
Задание 8.2. Приёмы использования карандашей и кистей	106
Графические примитивы	106
Задание 8.3. Приёмы использования графических примитивов	109
Задание 8.4. Пример с динамической графикой	109
Битовые образы (спрайты)	111
Задание 8.5. Анимация на основе битовых образов (масок)	114
Индивидуальное задание №8	114
СПИСОК ЛИТЕРАТУРЫ.....	117

ВВЕДЕНИЕ

Способность разрабатывать компьютерные программы необходима студенту IT-профиля и для освоения ряда дисциплин образовательной программы, и для формирования своего портфолио для будущих работодателей.

Учитывая это, курс основ программирования, во-первых, призван обеспечить актуализацию школьных знаний, умений и навыков, во-вторых, вносит свой вклад в формирование у студентов профессиональных компетенций на вузовском уровне, и, в-третьих, является базой для изучения других дисциплин профессионального цикла и самообразования.

Настоящее пособие представляет учебный материал таким образом, чтобы, с одной стороны, позволить новичку начать освоение программирования, с другой стороны, предоставить студенту, уже владеющему начальными навыками, возможность попрактиковаться на более сложных задачах.

Основной материал пособия разбит на восемь тем, в каждой имеется необходимая теоретическая часть, вопросы для самоконтроля, листинги с кодом для воспроизведения, изучения и доработки, а также задания для самостоятельной разработки. На освоение каждой темы предполагается в среднем 4 аудиторных академических часа и 4 часа самостоятельной работы.

Материал курса, представленный в данном пособии, изложен на примере языка высокого уровня (ЯВУ) C++. Это объясняется популярностью языка, востребованностью программистов C++ на рынке труда, а также тем, что этот язык впоследствии может быть использован во многих других дисциплинах образовательной программы IT-профиля – в системном, объектно-ориентированном (ООП), сетевом программировании, в разработке интеллектуальных систем.

В наибольшей степени содержание пособия посвящено структурному и модульному программированию. Элементы ООП используются в последних двух работах (Windows Forms), но без сколько-нибудь подробного изучения этой методологии. Проблематика ООП достойна рассмотрения в рамках отдельной дисциплины.

Задания предполагают написание текстов программ в популярной инструментальной среде Visual Studio Community, для личного использования она доступна для свободного скачивания на сайте производителя (компании Microsoft) [9]. Листинги и задачи можно реализовывать в любой иной среде или редакторе кода, поддерживающем язык программирования C++, например, Visual Studio Code.

Проверку выполнения заданий рекомендуется проводить в виде очной

защиты результата студентом.

Система заданий в настоящем пособии, вполне встраивается в балльно-рейтинговую систему контроля успеваемости студентов, используемую в настоящее время во многих вузах. Так, реализация студентом готовых листингов и умение их объяснить могут быть оценены на «удовлетворительно»; выполнение заданий на доработку (изменение) данного в листингах кода – «хорошо»; выполнение заданий на самостоятельную разработку программ – «отлично».

1. ЛИНЕЙНЫЕ ПРОГРАММЫ НА C++ В IDE VISUAL STUDIO

Цель: познакомиться с основами синтаксиса ЯВУ C++, а также получить начальные навыки работы в IDE Visual Studio на примере линейных программ.

Понятие языка C++

Язык C++ – это компилируемый *язык программирования* общего назначения, разработанный в начале 1980-х годов датчанином Бьёрном Страуструпом [6]. Язык C++ объединил в себе возможности быстрого, многофункционального и переносимого *языка C* (базового языка в операционной системе UNIX) с классами и объектами специализированного языка для моделирования сложных систем *Симула*. В результате C++ получился *универсальным* языком, поддерживающим различные *методологии* (парадигмы) программирования – процедурное, структурное, объектно-ориентированное, обобщённое.

Язык C++ стандартизован с 1998 г. Действующий на момент выхода этого пособия стандарт ISO/IEC 14882:2020 опубликован в декабре 2020 г., неофициально его обозначают C++20 [1].

Стандартная библиотека C++ регламентирована этим стандартом и включает в себя набор средств, которые должны быть доступны для любой реализации языка. Стандартная библиотека служит базисом для разработки широкого спектра прикладных приложений и специализированных библиотек.

Доступ к возможностям стандартной библиотеки C++ обеспечивается с помощью включения в программу (посредством директивы **#include**) соответствующих стандартных заголовочных файлов (см. листинг 1.1, строки 2-3).

Стандартная библиотека C++ включает в себя часть стандартной библиотеки языка C. Стандарт C++ содержит нормативную ссылку на стандарт C от 1990 г. и не переопределяет те функции в своей стандартной библиотеке, которые заимствуются из стандартной библиотеки языка C.

Примечание: C++ – самостоятельный язык программирования, а не надмножество C (как, например, *Objective C*). Большая часть кода C будет справедлива и для C++, но существует такой верный для C код, который неверен для C++.

Среда Visual Studio

Visual Studio – это интегрированная среда разработки (Integrated Development Environment, *IDE*) приложений для операционных систем Windows, Android и iOS, позволяющая осуществлять написание и редактирование кода, его компиляцию, отладку и запуск на языках *Visual C++* (это среда разработки C++ от Microsoft как компонент **Visual Studio**), VB.NET, Visual C#, Visual F#.

Среда **Visual Studio** – одна из самых популярных в списке IDE,

предназначенных для программирования на C++.

Visual Studio – открытая среда, т.е. она может быть дополнена сторонними компиляторами других языков (Ada, COBOL, FORTRAN, Lisp, Oberon и пр.). Поддерживаемые языки должны отвечать требованиям общезыковой спецификации CLS (Common Language Specification). С учетом компонентного подхода одно приложение в этой IDE может быть написано сразу на нескольких языках.

В **Visual Studio** можно создавать программы разных типов: консольные, оконные, браузерные (клиентские Web-приложения), Web-сервисы (серверные Web-приложения), облачные приложения, библиотеки.

Проект в Visual Studio – это объединение всех необходимых файлов, папок и других ресурсов одного приложения (*исходный код*, значки, изображения, параметры компилятора, конфигурационные данные для сторонних служб, с которыми взаимодействует программа и т.д.). В папку проекта помещается, например, *сборка* (assembly) – файл, создаваемый компилятором.

Связанные проекты объединены в *решения* (solution), как в контейнеры.

Подготовка к практикуму

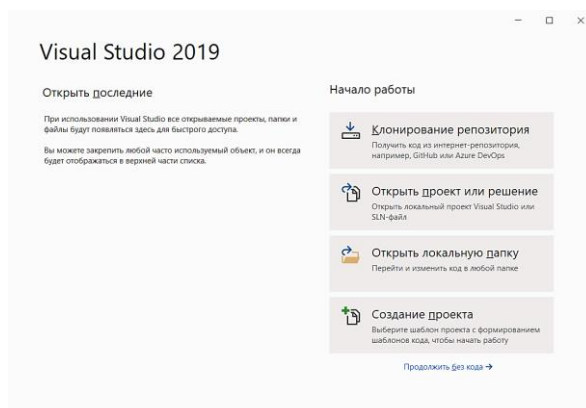
Обеспечение практикума:

- 1) ПК под управлением ОС Windows с выходом в сеть Интернет.
- 2) Интегрированная среда разработки **Visual Studio**.

Убедитесь¹, что на вашем компьютере установлена **IDE Visual Studio**. В случае отсутствия установите её (в версии **Community**) с сайта разработчика [9].

В ходе установки в списке рабочих нагрузок выберите «Разработка классических приложений на C++», а в перечне компонентов этой рабочей нагрузки укажите «Поддержка C++/CLI».

Откройте программу **Visual Studio** (рис. 1).



¹ Здесь и далее подчёркнутые слова требуют от студента практического исполнения.

Пример программы на языке C++

Проанализируем код листинга 1.1.

Листинг 1.1.

```
1 //Первый проект
2 #include <cstdlib> //заголовок для стандартной библиотеки C
3 #include <iostream> //заголовок для поддержки системы ввода-вывода в C++
4 #include <Windows.h> //заголовок для поддержки Windows API в C и C++
5
6 using namespace std; //подключение пространства имён стандартной библиотеки C++
7
8 int main() //начало программы
9 {
10     SetConsoleCP(1251); // установка кодовой страницы Windows-1251 в поток ввода
11     SetConsoleOutputCP(1251); // установка кодовой страницы Windows-1251 в поток вывода
12
13     cout << "Здравствуй, мир!\n";
14     system("PAUSE");
15
16     return 0; //признак нормального завершения работы программы
17 }
```

Здесь в строке 1 размещен *однострочный комментарий*, который всегда начинается с двойного следа (`//`). *Многострочные комментарии* можно размещать между символами `/*` и `*/`.

В строках 2-4 посредством директивы **#include** указываются *заголовки* (header), т.е. присоединяются заголовочные файлы, предоставляющие возможности имеющихся *библиотек* функций и классов, которые необходимы в основной программе (главным образом, стандартной библиотеки C++). В листинге 1.1 указаны: а) заголовочный файл **cstdlib** – обеспечивает доступ к возможностям стандартной библиотеки языка C; б) заголовок для работы с системой ввода-вывода (файл **iostream**) и в) заголовок для поддержки Windows API (**Windows.h**).

В строке 6 директива **using** указывает компилятору использовать *пространство имён* стандартной библиотеки C++ **std**. Это позволяет не указывать оператор *разрешения области видимости* `::` при каждом использовании идентификаторов из этого пространства имён (например, в строке 13 пишем не **std::cout**, а просто **cout**).

Примечание: с использованием директивы **using** повышается риск конфликта имён, определённых в пространстве подключенной библиотеки (которая может быть весьма обширной), и имён, определённых самим программистом.

Программа на языке C++ – это набор функций², а начинается её выполнение именно с функции **main** (строка 8). Т.о. **main** – основная функция или *точка входа* в программу.

² Подробнее функции будут рассмотрены в разделе 5.

Как и любая другая функция, **main** должна возвращать определенное значение. В нашем случае ключевое слово **int** (сокр. от integer - целое) в объявлении функции **main** указывает на то, что она должна вернуть целочисленное значение. Значение 0 возвращается в конце *тела функции* (строка 16) по команде **return**. Возврат нуля интерпретируется операционной системой как признак нормального завершения работы программы.

Для корректной работы с кириллицей в *консоли Windows* перед выводом строки в теле функции **main** реализованы дополнительные настройки (строки 10-11). Здесь для потоков *ввода* и *вывода* назначается кодовая таблица **Windows-1251** вместо старой **cp866**, которая осталась со времен операционной системы MS-DOS.

Примечание: строго говоря, ввода данных в этой программе нет, поэтому строку 9 можно было бы закомментировать (сделать из неё комментарий), вставив первыми символами два слэша **//**.

В строке 13 по команде **cout** (читаем «си-аут») с оператором вывода **<<** реализуется вывод на экран (в консоль) строки «Здравствуй, мир!». Здесь символы **\n** обозначают переход на новую строку, иначе последующий текст при выводе «приклеится» к предыдущему.

В строке 14 реализована пауза до нажатия на любую клавишу – без этого не успеть рассмотреть содержимое консоли (т.е. результат, вывод программы).

Задание 1.1. Первый проект

Создайте новый проект Visual C++ (рис. 2), выбрав кнопку **Создание проекта** → **Пустой проект**.

На следующем шаге (**Далее**) задайте проекту расположение и «говорящее» имя, чтобы по нему можно было понять, что в нём. Запомните для себя расположение папки проекта.

Примечание: в последующем для каждой новой задачи создавайте свой отдельный проект.

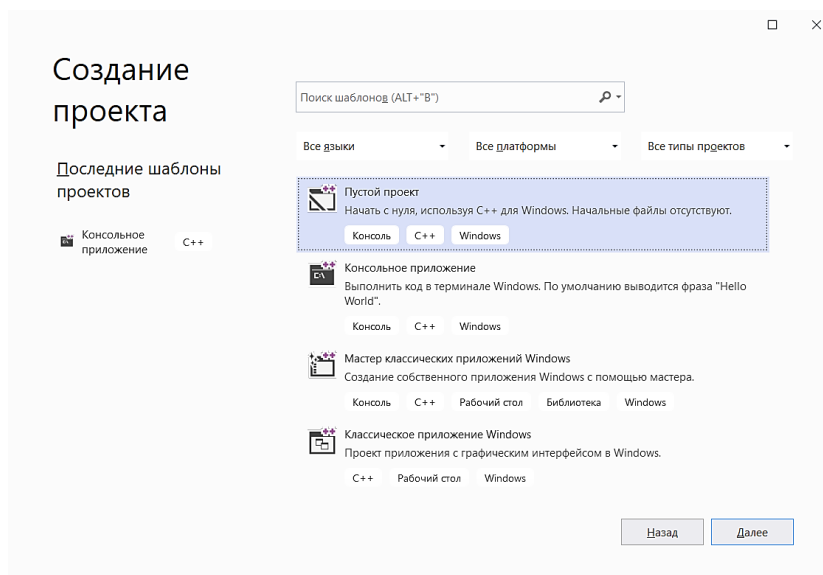


Рис. 2. Окно создания нового проекта.

Откроется окно нового проекта (рис. 3).

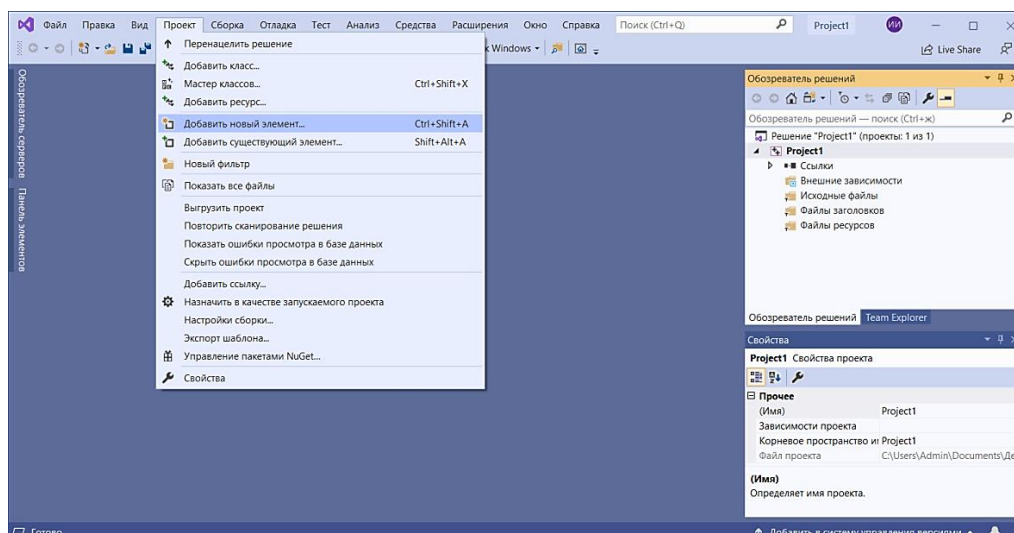


Рис. 3. Окно нового проекта.

В окне программы найдите главное меню и панели инструментов, окно с обозревателем решений (Solution Explorer), окно свойств текущего элемента проекта. Обратите внимание, что первым текущим элементом в окне свойств является сам проект.

Окно редактора пока не открыто, т.к. в проекте нет ни одного файла с исходным кодом.

Добавьте в проект новый **файл C++** с исходным кодом комбинацией клавиш **<Ctrl>+<Shift>+<A>**, назовите файл **first.cpp**

Примечание: добавить файл в проект можно также: а) из главного меню **Проект** → **Добавить новый элемент...** (см. рис. 3) или б) правым щелчком на имени проекта в обозревателе решений открыть контекстное меню, в котором выбрать **Добавить** → **Создать элемент...**

Включим нумерацию строк в окне редактора. Для этого откройте в меню **Средства** → **Параметры** → **Текстовый редактор** → **C/C++** → **Общие** и установите флажок **Номера строк**.

Реализуйте в добавленном файле код листинга 1.1.

Нажав клавишу <F5> (или по команде меню **Отладка (Debug)** → **Начать отладку (Start)**) проверьте работу созданного приложения (рис. 4).

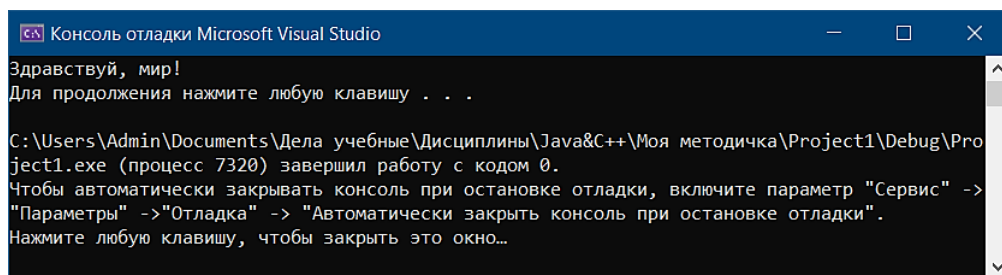


Рис. 4. Окно консоли с результатом работы программы.

В случае ошибки прочитайте соответствующее сообщение, найдите и исправьте обнаруженную в коде ошибку.

Примечание: в сообщении об ошибке при трансляции кода бывает не всегда точно указаны место и причина проблемы. Поэтому во время отладки просмотрите несколько строк кода, предшествующих выделенной строке.

В случае некорректного вывода кириллицы, в настройках окна консоли установите шрифт Licuda Console (целесообразно сделать это и в настройках окна по умолчанию).³

Проверьте вывод (перезапустите отладку), убрав из текста символы \n. Верните их обратно.

Измените в программе выводимый текст на свои фамилию, имя и отчество. Проверьте вывод.

Добавьте в программу к уже имеющемуся тексту вывод ещё 2-3 произвольных строк. Проверьте работу программы.

Найдите в папке проекта (в проводнике) созданный компилятором исполняемый файл с **нативным (машинным) кодом** (расширение **.exe**), определите его размер. Скопируйте его на Рабочий Стол и запустите. Запустится ли он на другом компьютере?

Выберите в меню **Средства** → **Параметры** → **Отладка**. Здесь установите флажок "Автоматически закрыть консоль при остановке отладки". Проверьте работу программы.

³ Другим возможным решением может стать настройка в тексте программы кодовой таблицы UTF-8 (идентификатор 65001), которая сейчас по умолчанию используется в консоли Windows.

Закройте текущий проект по команде меню **Файл** → **Заккрыть решение**.

Строки 1-11 и 14-17 листинга 1.1 будут необходимы во всех программах разделов 1-6 этого пособия, поэтому целесообразно сохранить этот код и использовать его как шаблон для будущих проектов.

Найдите файл с исходным кодом **first.cpp** в папке этого проекта, сделайте его копию вне папки проекта под именем, например, **shablon.cpp**.

С помощью любого текстового редактора (например, Блокнота или Notepad++) оставьте в этом файле только строки 1-11 и 14-17 и в других проектах используйте его в качестве шаблона.

Состав языка C++

Алфавит языка – это набор допустимых символов. Из символов составляется *лексема* (token) – минимальная единица языка, имеющая самостоятельный *смысл*. Из лексем составляются *операторы* и *выражения*.

Алфавит языка C++ включает [5, с. 17]:

- прописные и строчные буквы латинского алфавита;
- арабские цифры;
- специальные символы: + - / % . ? ! " > | \ ' _ & ~
- знаки пунктуации языка: [] () { } , ; : ... * = #
- пробельные символы: пробел, символ табуляции, символ перевода строки;
- прочие символы в тексте программы – только в комментариях.

Язык C++ различает большие и маленькие буквы, таким образом, name и Name – разные идентификаторы.

Выделяют пять типов лексем в C++:

- идентификаторы (имена),
- ключевые слова (табл. 1) – не могут использоваться в программе вне строго определенного смысла,
- знаки (символы) операций,
- литералы,
- разделители.

В состав языка входят также *директивы препроцессора*. *Препроцессор* – это программа для предварительного этапа трансляции – разбиения текста программы на лексемы (*лексический анализ*). Директивы – это особые указания препроцессору по обработке текста программы. В листинге 1.1 (строки 2-3), например, использовалась директива **#include**.

Таблица 1. Ключевые слова в языке C++

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinterpret_cast	return
short	signed	sizeof	static	static_cast
struct	switch	template	this	throw
typedef	true	try	typeid	typename
union	voidunion	using	virtual	void

Типы данных в языке C++

Компьютерная программа позволяет обрабатывать разного рода данные – числа, символы, строки и пр. В языке программирования для этого предусмотрены разнообразные **типы данных**. Тип данных задаёт:

- внутреннее представление данных в оперативной памяти ЭВМ:
 - 1) объем памяти для хранения значения (табл. 2),
 - 2) диапазон допустимых значений этого типа;
- набор допустимых действий со значением.

Таблица 2. Диапазоны значений простых типов данных в языке C++

Тип данных	Диапазон значений	Размер (байт)
bool	true, false (1, 0)	1
signed char	-128...127	1
unsigned char	0...255	1
signed short int	-32768...32767	2
unsigned short int	0...65535	2
signed long int	-2 147 483 648...2 147 483 647	4
unsigned long int	0...4 294 967 295	4
signed long long int	-9 223 372 036 854 775 808... 9 223 372 036 854 775 807	8
unsigned long long int	0...18 446 744 073 709 551 615	8
float	3.4e-38...3.4e+38	3
double	1.7e-308...1.7e+308	8
long double	3.4e-4932...3.4e+4932	10

Типы данных:

- по строению значения – основные (**простые**) и **составные** (структурированные);
- по созданию – **встроенные** (стандартные) и **пользовательские** (созданные программистом);
- по способу выделения памяти – **статические** и **динамические**;
- по способу хранения – **значимые** (типы-значения) и **ссылочные**.

Основные (простые, стандартные) типы данных часто называют *арифметическими*, поскольку всех их можно использовать в арифметических операциях. Для описания основных типов определены следующие ключевые слова:

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

Первые четыре типа называют *целочисленными* (целыми), последние два – типами с *плавающей точкой* (запятой). Код, который формирует компилятор для обработки целых величин, отличается от кода для величин с плавающей точкой.

Существует четыре *спецификатора* типа, уточняющих внутреннее представление и диапазон значений стандартных типов:

- **short** (короткий);
- **long** (длинный);
- **signed** (знаковый);
- **unsigned** (беззнаковый).

Размер значения типа **int** в памяти определяется не стандартом языка C++, а зависит от архитектуры компьютера и компилятора. Для 16-разрядного процессора под величины этого типа отводится 2 байта, для 32-разрядного – 4 байта.

Спецификатор **short** перед именем типа указывает компилятору, что под число требуется отвести 2 байта не зависимо от разрядности процессора. Спецификатор **long** означает, что целая величина будет занимать 4 байта. Таким образом, на 16-разрядном компьютере эквивалентны **int** и **short int**, а на 32-разрядном – **int** и **long int**.

Различные виды типов, различающиеся диапазоном и точностью представления данных, введены для того, чтобы дать программисту возможность наиболее эффективно использовать возможности конкретной аппаратуры, поскольку от выбора типа зависит скорость вычислений и объем памяти.

В языке C++ также существуют *перечислимый* тип **enum**, который является подмножеством целого типа, и тип **void**, который используется для объявления функций, возвращающих пустое значение (по сути, процедур – подпрограмм, которые в отличие от функций не возвращают значения).

Переменные и константы

Переменная – это именованная область в памяти ЭВМ для хранения значения определенного типа, и это значение может изменяться в ходе, или, как

говорят, на этапе выполнения программы. Примеры **объявления** и **инициализации** (т.е. присвоения начального значения) переменных в C++:

```
int i;  
float x;  
short int b = 105; //объявление с инициализацией (первый способ)  
float y (1.42); //объявление с инициализацией (второй способ)  
char ch, f = 'f'; //объявляются 2 переменные, но инициализируется только вторая
```

Константа – её значение не может изменяться по ходу выполнения программы. Пример объявления константы:

```
const int i = 5; // константа
```

Область действия переменной (т.е. где её можно использовать) – это часть программы, в которой её можно использовать для доступа к связанной с ней области памяти. В зависимости от области действия переменная может быть локальной или глобальной.

Если переменная определена внутри блока (блок ограничен фигурными скобками), она называется **локальной**, область ее действия – от точки описания до конца блока, включая все вложенные блоки. Если переменная определена вне любого блока, она называется **глобальной** и областью ее действия считается весь файл, в котором она определена от точки описания.

Областью видимости идентификатора называется часть текста программы, из которой допустим обычный доступ к связанной с идентификатором области памяти. Чаще всего область видимости совпадает с областью действия. Исключением является ситуация, когда во вложенном блоке (между символами { }) описана переменная с таким же именем. В этом случае внешняя переменная во вложенном блоке невидима, хотя она входит в ее область действия. Тем не менее к этой переменной, если она глобальная, можно обратиться, используя **операцию доступа к области видимости ::** (два двоеточия). Пример:

```
int a; // объявление глобальной переменной a  
int main ()  
{ //начало нового блока  
    a=1; //присвоение значения глобальной переменной  
    int a; //объявление локальной переменной a  
    a=2; //присвоение значения локальной переменной  
    :: a=3; //переопределение значения глобальной переменной  
    return 0;  
} //конец блока
```

Пример **вывода** значения переменной в консоль представлен в лист.1.2 (строки 15 и 20).

```

1 //Пример константы и переменной
2 #include <cstdlib> //заголовок для стандартной библиотеки C
3 #include <iostream> //заголовок для поддержки системы ввода-вывода в C++
4 #include <Windows.h> //заголовок для поддержки Windows API в C и C++
5
6 using namespace std; //пространство имён стандартной библиотеки C++
7
8 int main() //начало программы
9 {
10     SetConsoleCP(1251); // установка кодовой страницы Windows-1251 в поток ввода
11     SetConsoleOutputCP(1251); // установка кодовой страницы Windows-1251 в поток вывода
12
13     const float pi = 3.1415926; //объявление с инициализацией константы Pi
14     cout << "Значение константы Pi="; //вывод отдельно подсказки пользователю
15     cout << pi; //и отдельно самого значения
16     cout << "\n";
17
18     int length(25); //объявление с инициализацией целой переменной
19     //вывод сразу и подсказки и значения:
20     cout << "Значение целочисленной переменной " << length << "\n";
21
22     system("PAUSE");
23     return 0; //признак нормального завершения работы программы
24 }

```

Задание 1.2. Программа с константой и переменной

Создайте новый проект, в нём – новый файл с исходным кодом C++.

Реализуйте код листинга 1.2. Проверьте его работу.

Отредактируйте код программы так, чтобы после чисел выводились точки.

Самостоятельно отредактируйте код, чтобы осуществлялся **форматированный вывод** вещественной константы **Pi** только с двумя знаками в дробной части (для поиска необходимого метода воспользуйтесь книжным или электронным справочником по C++, например, [5]).

Операции и выражения в C++. Ввод данных

Выражение (например, $-2 \cdot x/y$) – это правило вычисления значения; характеризуется собственно значением и его типом. Выражение состоит из **операндов** и **знаков операций**.

Операнды – литералы, переменные, константы и возвращаемые функциями значения.

Операции – унарные (с одним операндом), бинарные (с двумя операндами), тернарный. Порядок операций в выражении реализуется в соответствии с приоритетом.

Перечень операций в языке C++ см. в табл. 3.

Таблица 3. Операции в языке C++

Операция	Описание
Унарные операции	
++	инкремент (увеличение на 1)
--	декремент (уменьшение на 1)
sizeof	размер памяти
~	поразрядное отрицание
!	логическое отрицание
-	унарный минус
+	унарный плюс
&	взятие адреса
*	разадресация
new	выделение памяти
delete	освобождение памяти
(type)	преобразование типа
Бинарные операции	
*	умножение
/	деление
%	остаток от деления
+	сложение
-	вычитание
<<	сдвиг влево
>>	сдвиг вправо
<	меньше
<=	меньше или равно
>	больше
>=	больше или равно
==	равно
!=	не равно
&	поразрядная конъюнкция (И)
^	поразрядное исключающее ИЛИ
	поразрядная дизъюнкция (ИЛИ)
&&	логическое И
	логическое ИЛИ
=	присваивание
*=	умножение с присваиванием
/=	деление с присваиванием
%=	остаток от деления с присваиванием
+=	сложение с присваиванием
-=	вычитание с присваиванием
<<=	сдвиг влево с присваиванием
>>=	сдвиг вправо с присваиванием
&=	поразрядное И с присваиванием
=	поразрядное ИЛИ с присваиванием
^=	поразрядное исключающее ИЛИ с присваиванием
.	последовательное вычисление
Тернарный оператор	
?	действие в зависимости от проверки условия

В предыдущем примере (листинг 1.2) переменная `length` инициализируется литералом 25 (строка 18). Но часто возникает потребность реализовать ввод необходимого значения в переменную с клавиатуры.

Для реализации ввода в консоли Windows в наших программах уже есть поддержка: заголовок `iostream` и кодировка Windows-1251 (`SetConsoleCP`) для работы с кириллицей.

По аналогии с **выводом**, когда используется идентификатор `cout` вместе с оператором вывода `<<`, для **ввода** используется `cin` вместе с оператором ввода `>>`, например:

```
cin >> R;
```

Выполняя эту команду, программа выведет в окно консоли курсор и будет ожидать действий пользователя. Чтобы пользователь понимал, что от него требуется, ввод следует предварять выводом подсказки, например:

```
cout << "Введите целое число – величину радиуса>";  
cin >> R;
```

Возможен последовательный ввод сразу нескольких значений:

```
cin >> a >> b >> c;
```

При вводе нескольких значений на клавиатуре они должны разделяться так называемыми «пробельными» символами - пробелом, табуляцией (`\t`) или переводом строки (`\n`).

Задание 1.3. Простейшие арифметические выражения

Возьмем за основу предыдущую программу (листинг 1.2) и отредактируем её: оставим константу `Ri` и дополним код вычислением и выводом длины окружности (см. листинг 1.3).

1.3.а. Реализуйте код листинга 1.3 в новом проекте и проверьте результат работы программы.

В этой программе можно было бы обойтись без переменной `DlinaOkr`. Исправьте код листинга 1.3 так, чтобы программа выводила результат без неё.

Отредактируйте код листинга 1.3, реализовав возможность ввода значения радиуса окружности пользователем с клавиатуры.

1.3.б. Самостоятельно реализуйте в новом проекте программу, позволяющую поменять местами значения двух целочисленных переменных с помощью третьей переменной.

1.3.в. Эту же задачу решите без использования третьей переменной. Снабдите эту программу возможностью ввода с клавиатуры значений обеих переменных.

```

1 //Простой арифметический пример
2 #include <iostream> //заголовок для поддержки системы ввода-вывода в C++
3 #include <Windows.h> //заголовок для поддержки Windows API в C и C++
4
5 using namespace std; //пространство имён стандартной библиотеки C++
6
7 int main() //начало программы
8 {
9     SetConsoleCP(1251); // установка кодовой страницы Windows-1251 в поток ввода
10    SetConsoleOutputCP(1251); // установка кодовой страницы Windows-1251 в поток вывода
11
12    const float pi = 3.1415926; //константа Pi
13    float R = 10; //радиус окружности
14    float DlinaOkr;
15
16    DlinaOkr = 2 * pi * R;
17    //вывод результата:
18    cout << "Длина окружности радиуса " << R << " равна: " << DlinaOkr << ".\n";
19
20    system("PAUSE");
21    return 0; //признак нормального завершения работы программы
22 }

```

Математические вычисления в программах на C++

Математические функции в C++ доступны после присоединения заголовка **cmath** (листинг 1.4., строка 5). Примеры некоторых операций этой библиотеки представлены в табл. 4.

Таблица 4. Математические функции библиотеки cmath

Функция	Тип аргумента	Тип результата	Назначение
abs (x)	int	int	модуль целого числа
acos (x)	double	double	арккосинус в радианах
asin (x)	double	double	арксинус в радианах
atan (x)	double	double	арктангенс в радианах
ceil (x)	double	double	ближайшее целое справа
cos (x)	double	double	косинус (x в радианах)
exp (x)	double	double	экспонента
fabs (x)	double	double	модуль вещественного числа
floor (x)	double	double	ближайшее целое слева
fmod (x,y)	double, double	double	остаток от целочисленного деления
log (x)	double	double	натуральный логарифм
log10 (x)	double	double	десятичный логарифм
pow (x,y)	double, double	double	возведение x в степень y
sin (x)	double	double	синус (x в радианах)
sinh (x)	double	double	гиперболический синус
sqrt (x)	double	double	квадратный корень
tan (x)	double	double	тангенс (x в радианах)
tanh (x)	double	double	гиперболический тангенс

В математической библиотеке также predefinedены несколько математических констант (табл. 5).

Таблица 5. Математические константы в cmath

Константа	Выражение	Значение
M_E	e	2.71828182845904523536
M_LOG2E	$\log_2(e)$	1.44269504088896340736
M_LOG10E	$\log_{10}(e)$	0.434294481903251827651
M_LN2	$\ln(2)$	0.693147180559945309417
M_LN10	$\ln(10)$	2.30258509299404568402
M_PI	π	3.14159265358979323846
M_PI_2	$\pi/2$	1.57079632679489661923
M_PI_4	$\pi/4$	0.785398163397448309616
M_1_PI	$1/\pi$	0.318309886183790671538
M_2_PI	$2/\pi$	0.636619772367581343076
M_2_SQRTPI	$2/\sqrt{\pi}$	1.12837916709551257390
M_SQRT2	$\sqrt{2}$	1.41421356237309504880
M_SQRT1_2	$\sqrt{1/2}$	0.707106781186547524401

Чтобы использовать эти константы, необходимо перед подключением заголовка cmath определить макрос `_USE_MATH_DEFINES`:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

Задание 1.4. Пример использования математических функций

Для примера использования математических функций реализуем программу для вычисления расстояния между двумя точками A и B на плоскости (рис. 5).

Точки A и B заданы своими координатами – соответственно, (x_a, y_a) и (x_b, y_b) . Тогда формула для расчёта расстояния:

$$AB = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}.$$

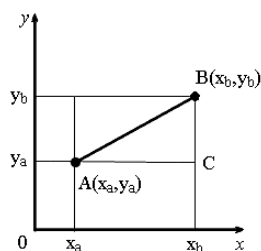


Рис. 5. Расстояние между двумя точками на плоскости.

В листинге 1.4 представлен код, реализующий вычисление значения по данной формуле.

Здесь использованы математические функции **pow** (возведение в степень) и **sqrt** (квадратный корень).

1.4.a. Создайте новый проект. Реализуйте код листинга 1.4, проверьте его работоспособность. Обратите внимание на использование формата для вывода конечного значения (строка 23).

```

1 //Расстояние между двумя точками
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 #include <cmath>
6
7 using namespace std;
8
9 int main()
10 {
11     SetConsoleCP(1251);
12     SetConsoleOutputCP(1251);
13
14     cout << "Программа для вычисления расстояния между двумя точками" << "\n";
15     cout << "-----" << "\n";
16     float xa, ya, xb, yb, AB;
17     cout << "Введите координату X точки A >"; cin >> xa;
18     cout << "Введите координату Y точки A >"; cin >> ya;
19     cout << "Введите координату X точки B >"; cin >> xb;
20     cout << "Введите координату Y точки B >"; cin >> yb;
21     AB = sqrt(pow((xb - xa), 2) + pow((yb - ya), 2));
22     cout << "-----" << "\n";
23     cout.precision(4);
24     cout << "Расстояние между точками равно: " << AB << "\n";
25
26     system("PAUSE");
27     return 0;
28 }

```

В окне Visual Studio в обозревателе решений раскройте: **Внешние зависимости** → **cmath** (здесь будут перечислены все доступные математические функции).

1.4.6. В программе вычисления длины окружности (задание 1.3.а, листинг 1.3) замените собственную константу **pi** на определённую в библиотеке **cmath** (см. табл. 5).

Индивидуальное задание №1

Реализуйте средствами C++ назначенный преподавателем⁴ вариант **индивидуального задания №1** на простую линейную программу. Снабдите вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата (с точностью до 3 знаков после запятой) по образцу:

Программа вычисления длины окружности

Введите радиус окружности>10

Ответ: 62.832

Протестируйте работу программы на нескольких наборах вводимых значений.

⁴ Здесь и в последующих индивидуальных заданиях (кроме №7) задачи могут браться, например, из сборника [3].

Вопросы для самоподготовки

- ✓ Что такое алгоритм? Что такое компьютерная программа?
- ✓ Что входит в состав языка программирования?
- ✓ Что называют языками высокого уровня?
- ✓ Что такое трансляция кода? Чем отличается компиляция от интерпретации?
- ✓ Чем отличаются интерфейсы консольного и оконного приложений?
- ✓ Запустится ли исполняемый exe-файл на другом компьютере без папки проекта?
- ✓ Что значит, что процесс завершил свою работу с кодом 0 (рис. 4)?
- ✓ Что такое переменная в компьютерной программе?
- ✓ Чем отличается объявление переменной от её инициализации?
- ✓ Что определяет тот или иной тип данных?
- ✓ Почему для хранения целых и дробных значений используются разные типы данных? Не проще ли создать один универсальный тип для всех числовых значений?
- ✓ Почему тип `char` считается целочисленным?
- ✓ Будет ли доступна переменная `length` за пределами тела функции `main` в листинге 1.2?
- ✓ Подойдёт ли для обмена значениями двух переменных `a` и `b` пара команд $a=b; b=a$?
- ✓ Как сбросить настройку количества знаков после запятой при выводе к исходному значению?

2. ВЕТВЛЕНИЕ И ЦИКЛЫ В C++

Цель: познакомиться с основами синтаксиса языка высокого уровня C++ на примере программ, реализующих алгоритмические конструкции ветвления и цикла.

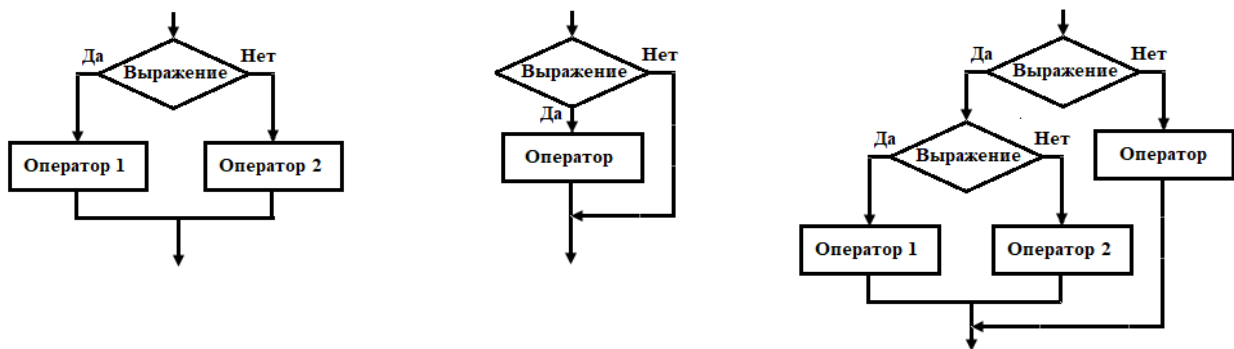
Понятие ветвления. Условный оператор

В предыдущем разделе рассматривались простейшие линейные программы, построенные в соответствии с алгоритмической конструкцией *простое следование*. Сама программа представляла собой одну цепочку последовательно выполняющихся команд (рис. 6).



Рис. 6. Блок-схема простого следования.

Более сложная конструкция – *ветвление* – предполагает разделение алгоритма на несколько веток, из которых выполнена будет только одна. Если алгоритм разделяется на *две ветки*, то это – *бинарное ветвление* (рис. 7, а). Выбор ветви определяется результатом проверки *условия* – *логического выражения*, значением которого может быть либо **true** (истинно), либо **false** (ложно).



а) бинарное ветвление б) вырожденное ветвление в) вложенное ветвление

Рис. 7. Блок-схемы вариантов бинарного ветвления.

Вырожденный вариант бинарного ветвления, когда оператор (блок операторов) есть только в одной из веток (true), вторая ветка (false) отсутствует (рис. 7, б).

Вложенное ветвление, когда в ветке в качестве одного из операторов используется другое ветвление (рис. 7, в). Вложенное бинарное ветвление является одним из способов организации *множественного ветвления*.

В языке C++ бинарное ветвление организуется с помощью *условного оператора*. Синтаксис (формат) условного оператора:

```
if (логическое_выражение) оператор_1; [else оператор_2;]
```

Оператором может выступать *блок операторов*, ограниченный фигурными скобками { }.

Примечание: квадратные скобки в описании синтаксиса в самом коде не пишутся и обозначают необязательность фрагмента (в этом случае имеем вырожденный вариант ветвления).

Пример реализации ветвления:

```
//1 – поиск максимума из двух чисел:
```

```
if (a > b) max = a;
```

```
else max = b;
```

Во втором примере демонстрируется вариант вложенных ветвлений:

```
//2 – поиск максимума из трёх чисел
```

```
if (a > b)
```

```
    if (a > c) max = a;
```

```
    else max = c;
```

```
else
```

```
    if (b > c) max = b;
```

```
    else max = c;
```

Примечание: во вложенных ветвлениях к какому **if** относится какой **else** наглядно показывают отступы в тексте программы – хорошим стилем программирования считается обязательность их использования.

В третьем примере представлено *составное условие* (оператор **&&** «логическое И»):

```
//3 – проверка принадлежности x диапазону (0,5]
```

```
if (x > 0 && x <= 5) a++;
```

Задание 2.1. Простые примеры бинарного ветвления

2.1.а. Реализуйте в новом проекте приведённый выше пример 2 на нахождение максимального значения из трёх чисел. Снабдите программу вводом значений **a**, **b** и **c** с клавиатуры, проверьте работу программы на нескольких примерах.

2.1.б. Самостоятельно напишите программу для определения чётности натурального числа.

2.1.в. В следующем примере сделаем программу, в которой по номеру года (по григорианскому календарю) определяется, *високосный* ли он. Год високосный, если его номер: а) делится нацело (без остатка) на 400, или б) делится нацело на 4 и при этом не делится на 100.

В виде блок-схемы этот алгоритм с вложенными ветвлениями изображён на рис. 8.

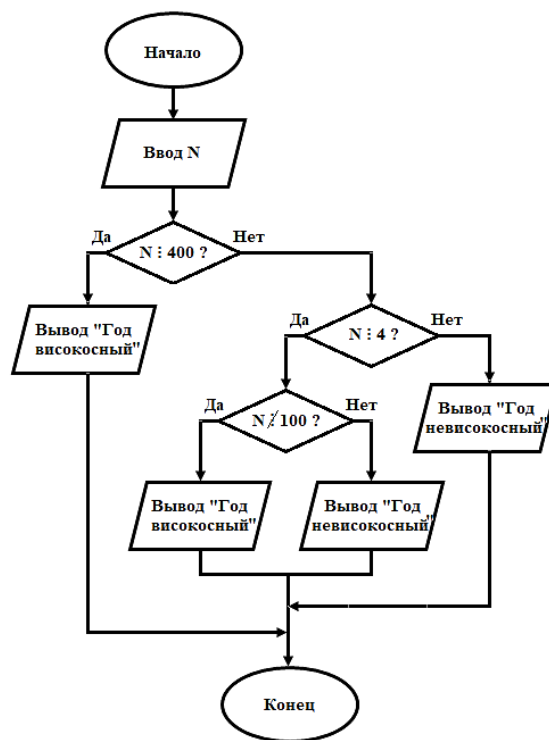


Рис. 8. Блок-схема алгоритма определения високосного года.

Средствами С++ самостоятельно реализуйте код алгоритма по его блок-схеме на рис. 8, проверьте работу программы на значениях для каждой из четырёх веток. Исправьте код программы так, чтобы с помощью операторов **&&** (логическое **И**) и **||** (логическое **ИЛИ**) получилось одно *составное условие*, заменяющее собой три отдельных вложенных. Составное условие позволит реализовать функциональность с помощью только одного ветвления. Проверьте результат.

Множественное ветвление

Конструкция множественного ветвления подразумевает, что, в зависимости от значения выражения, алгоритм может разделяться на *несколько* веток, из которых выполнена будет только одна (рис. 9).

В отличие от бинарного ветвления, здесь у выражения должен быть тип, предполагающий более, чем два возможных варианта значения.

Ветка **default** выполнится, если значение выражения не будет ни одним из явно заданных в других ветках.

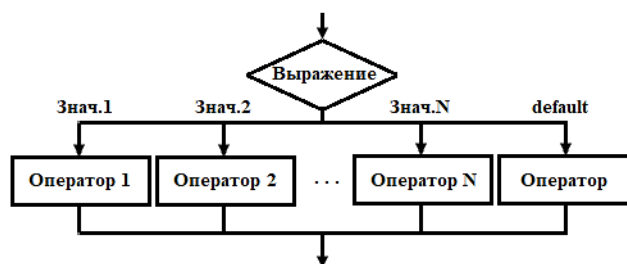


Рис. 9. Блок-схема множественного ветвления.

В языке С++ множественное ветвление реализуется с помощью оператора

switch. Синтаксис (формат) оператора **switch**:

```
switch (выражение) {  
    case значение_1: [оператор_1; break;  
    case значение_2: [оператор_2; break;  
    ...  
    case значение_n: [оператор_n; break;  
    [default: операторы]  
}
```

Команда **break** позволяет прервать выполнение и текущей ветки, и всего оператора **switch** с переходом к следующей команде в программе.

Примечание: если выход из переключателя **switch** явно не указан (командой **break**), последовательно выполняются все остальные ветви.

Как и в бинарном ветвлении, оператором в ветке может выступать *блок операторов*, ограниченный фигурными скобками { }.

Задание 2.2. Пример множественного ветвления

Простой пример использования множественного ветвления – вывод названия дня недели по его номеру (1 – понедельник, 2 – вторник и т.д.) представлена в листинге 2.1.

Здесь выражение состоит из одной целочисленной переменной **day**, в зависимости от значения которой программа выводит на экран разные сообщения.

Реализуйте код листинга 2.1, проверьте работу программы.

```

1  //День недели по номеру
2  #include <cstdlib>
3  #include <iostream>
4  #include <Windows.h>
5  using namespace std;
6
7  int main()
8  {
9      SetConsoleCP(1251);
10     SetConsoleOutputCP(1251);
11
12     cout << "Программа для вывода дня недели по его номеру" << "\n";
13     cout << "-----" << "\n";
14     int day;
15     cout << "Введите номер дня недели (целое число от 1 до 7) >"; cin >> day;
16     switch (day) {
17         case 1: cout << "Понедельник" << "\n"; break;
18         case 2: cout << "Вторник" << "\n"; break;
19         case 3: cout << "Среда" << "\n"; break;
20         case 4: cout << "Четверг" << "\n"; break;
21         case 5: cout << "Пятница" << "\n"; break;
22         case 6: cout << "Суббота" << "\n"; break;
23         case 7: cout << "Воскресенье" << "\n"; break;
24         default: cout << "Ошибка при вводе!" << "\n"; break;
25     }
26
27     system("PAUSE");
28     return 0;
29 }

```

Индивидуальное задание №2.1

Реализуйте средствами языка С++ назначенный преподавателем вариант **индивидуального задания №2.1** на ветвление. Снабдите вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата.

Протестируйте работу программы на нескольких наборах входных данных.

Понятие цикла

Цикл – это алгоритмическая конструкция для выполнения блока кода (*тела цикла*) *многократно* (подряд несколько раз). Один проход цикла – *итерация*. Будет ли осуществлена следующая итерация, зависит от значения логического выражения – *условия* продолжения (или выхода) из цикла.

Базовые циклы: с предусловием, с постусловием, с параметром (параметрический или со счётчиком). Первые два используются обычно, когда число итераций заранее не известно.

Язык С++ позволяет с помощью *операторов передачи управления* осуществлять принудительное завершение: а) текущей итерации (выполняется не всё тело цикла) – по команде **continue**; б) всего цикла (до проверки и выполнения условия выхода из цикла) – по **break**.

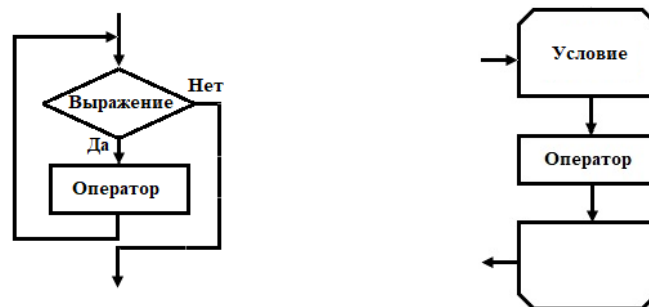
Но оператор передачи управления – это вариант *безусловного перехода*

(**goto**), с позиций структурного программирования его использование нельзя назвать приемлемым – в больших программах это усложняет чтение кода и может порождать ошибки (т.н. спагетти-код)⁵.

В определённых ситуациях цикл может продолжаться бесконечно (**бесконечный цикл**, заикливание), и программа не в состоянии перейти к следующему после цикла оператору. Так происходит, если в теле цикла не происходит ничего, что приближало бы ситуацию к изменению истинности условия и, соответственно, выходу из цикла.

Цикл с предусловием **while**

В цикле с **предусловием** сначала проверяется **условие** – логическое выражение, затем осуществляется очередная итерация (рис. 10). Этот вариант цикла используется, когда заранее неизвестно число итераций. Цикл с предусловием может не выполниться ни разу.



а) смысловое представление б) стандартными символами⁶

Рис. 10. Блок-схема цикла с предусловием.

В языке C++ этот цикл реализуется с помощью оператора **while**. Синтаксис (формат):

```
while (выражение) оператор;
```

Оператором в теле цикла может быть блок операторов в фигурных скобках { }.

Пример ошибки в коде, приводящей к бесконечному циклу:

```
while (true) { ... }
```

Здесь что бы ни происходило в теле цикла, условие никогда не станет ложным.

Задание 2.3. Примеры использования цикла с предусловием. Отладка

2.3.а. Изучите код программы для вывода в консоль целых чисел в диапазоне от -25 до +25 с шагом +5 (листинг 2.2).

⁵ В профессиональной среде к этой проблеме неоднозначное отношение, многие программисты в своих проектах используют операторы передачи управления, например, для быстрого выхода из вложенных циклов. В учебных примерах, всё же, мы рекомендуем воздерживаться от использования подобных приёмов.

⁶ См. стандарт [2]

```

1  //Простой пример цикла while
2  #include <cstdlib>
3  #include <iostream>
4  #include <Windows.h>
5  using namespace std;
6  int main()
7  {
8      SetConsoleCP(1251);
9      SetConsoleOutputCP(1251);
10     cout << "Программа демонстрации работы цикла while" << "\n";
11     cout << "-----" << "\n";
12     short int d = -25;
13     while (d <= 25) //заголовок цикла
14     { //тело цикла - блок операторов
15         cout << d << "\n";
16         d = d + 5;
17     } //конец цикла
18     system("PAUSE");
19     return 0;
20 }

```

Реализуйте код листинга 2.2, проверьте его работу.

В среде **Visual Studio** нажатие клавиши <F5> означает запуск приложения в режиме *отладки*. При этом разработчику становятся доступны многие инструменты для наблюдения за исполнением кода. Одним из них является *точка останова* – возможность «поставить на паузу» исполнение программы в заданной программистом точке кода.

Сделайте левый щелчок мышью на поле слева от строки 13 программы из листинга 2.2 (слева от номера строки) – появится красная точка – точка останова (рис. 11).

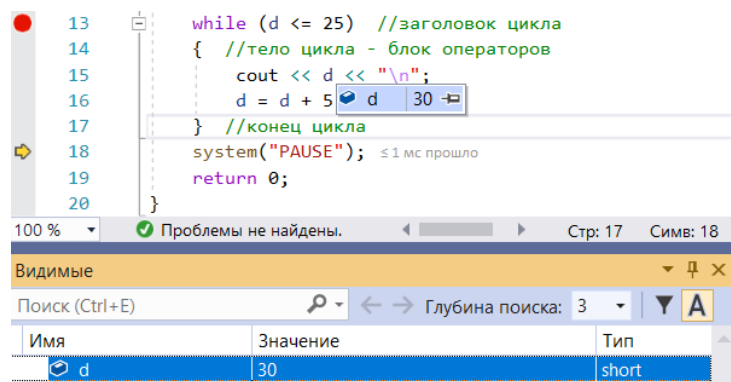


Рис. 11. Код программы из листинга 2.2 на шаге после выхода из цикла.

Нажмите <F5> – как обычно запустится программа в режиме отладки, но исполнение быстро остановится. Вас вернут в редактор кода и стрелка на точке останова покажет строку, на которой исполнение остановилось (рис. 11).

Расположите окна Visual Studio и консоли приложения так, чтобы они были видны сразу оба.

Нажмите <F11> (или **Отладка** → **Шаг с заходом**), метка текущей строки

переместится на следующий оператор. Этот приём позволит пошагово исполнить код программы.

Нажмите <F11> еще несколько раз и при этом следите за состоянием консоли приложения. Добейтесь пошаговым исполнением вывода в консоль 3-4 значений.

Обратите внимание на окно **Видимые** в окне Visual Studio (рис. 11). В нём отображаются переменные со своими значениями и типом текущей и трёх предыдущих строк кода.

Определите значение переменной **d** на текущем шаге исполнения программы. В какие моменты оно изменяется?

По ярлыку внизу откройте окно **Локальные**. В нём отображаются переменные текущей *области действия*.

По ярлыку внизу откройте окно **Контрольные значения**. В нём отображается набор контролируемых переменных, сформированный самим программистом (правый щелчок на имени переменной → **Добавить контрольное значение**).

Текущее значение интересующей вас переменной на этапе отладки можно определить и просто наведя курсор мыши на неё в коде, при этом всплывёт небольшое окно (рис. 11).

Вызовите всплывающее окно для переменной **d**, щёлкните на значок кнопки в нём. Это позволит закрепить это окно на всё время отладки. Переместите мышью окно со значением переменной в любое удобное место в окне редактора.

Пошаговым исполнением проследите значение переменной **d** до выхода из цикла, после чего нажмите <F5> – возврата к точке останова уже не будет, программа продолжит своё обычное исполнение. Завершите работу приложения.

Щелчком на точке останова удалите её.

Исправьте код программы, чтобы числа выводились в обратном порядке и в одну строку.

2.3.6. Следующий пример (листинг 2.3) более точно отразит специфику использования цикла с предусловием, когда *заранее не известно количество итераций*.

Вспомним, что натуральное число является *простым*, если оно делится нацело только на 1 и на само себя (2, 3, 5, 7, 11, 13, 17, ...). Если у числа более двух целочисленных делителей, то такое число называется *составным*.


```

1  //Программа определения простого числа
2  #include <cstdlib>
3  #include <iostream>
4  #include <Windows.h>
5  #include <cmath>
6  using namespace std;
7
8  int main()
9  {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12     cout << "Программа определения простого числа" << "\n";
13     cout << "-----" << "\n";
14     int n, s = 0, i = 2;
15     cout << "Введите натуральное число (>=2) >"; cin >> n;
16     while (i <= sqrt(n)) {
17         if ((n % i) == 0) s++;
18         i++;
19     }
20     if (s == 0)
21         cout << "Число простое!" << "\n";
22     else
23         cout << "Число составное!" << "\n";
24
25     system("PAUSE");
26     return 0;
27 }

```

В листинге 2.3 в переменную **n** вводится проверяемое число (строка 15); в переменной **s** в цикле **while** подсчитывается количество целочисленных делителей введённого числа (строка 17); в **i** – проверяемые делители числа **n** в диапазоне от 2 до квадратного корня из **n** с шагом +1.

После цикла по получившемуся **s** определяется – простое число **n** или составное (строки 20-23).

Реализуйте код листинга 2.3. Проверьте работу программы. При необходимости используйте описанные выше средства отладчика.

Дополните код, чтобы при вводе 1 сообщалось, что это и не простое и не составное число.

Дополните программу выводом числа итераций. Нужна ли для этого отдельная переменная?

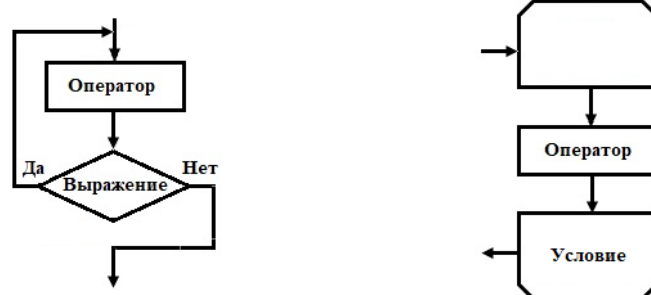
2.3.в. Если внимательно изучить представленный в листинге 2.3 алгоритм, то можно обнаружить, что он не вполне оптимален. Действительно, подсчёт целочисленных делителей числа **n** всегда будет осуществляться до заданного граничного значения **i** – корня из **n**. Но, если будет найден хотя бы один целочисленный делитель – это уже признак того, что введённое число составное, далее подсчёт количества делителей не нужен.

Отредактируйте код листинга 2.3 так, чтобы цикл прекращался не только по достижению **i** значения корня из **n**, но и ранее – если обнаружится хотя бы один

целочисленный делитель (понадобится составное условие).

Цикл с постусловием **do-while**

В цикле с *постусловием* сначала осуществляется выполнение очередной итерации, затем проверяется условие (рис. 12). Т.о. цикл с постусловием всегда выполнится хотя бы один раз. Этот вариант цикла обычно используется, когда заранее неизвестно число итераций.



а) смысловое представление б) стандартными символами

Рис. 12. Блок-схема цикла с постусловием.

В языке C++ реализуется с помощью **операторов do-while**. Синтаксис (формат):
do оператор while (выражение);

В качестве оператора может выступать блок операторов в фигурных скобках.

Для примера рассмотрим программу для угадывания натурального числа в диапазоне от 1 до 10 (листинг 2.4). Хотя бы одну попытку пользователь всегда сделает, поэтому здесь и уместен цикл с постусловием.

Функция из стандартной библиотеки C **rand()** в строке 19 возвращает *псевдослучайное* целое число в диапазоне от 0 до значения **RAND_MAX** — константы, определённой в той же библиотеке. Инициализация («встряхи́вание») датчика псевдослучайных чисел в строке 16 необходима, чтобы при каждом запуске генерировалась разная числовая последовательность.

Задание 2.4. Примеры использования цикла с постусловием

2.4.а. Реализуйте код листинга 2.4. Проверьте работу программы.

Уточните значение константы **RAND_MAX**.

2.4.б. Доработайте программу из листинга 2.3, добавив в неё возможность многократного ввода проверяемого числа с помощью цикла с постусловием (до ввода, например, числа 0). Не забудьте дополнить интерфейс пользователя информацией об условии выхода из программы. Составьте блок-схему получившегося алгоритма с *вложенным циклом*.

```

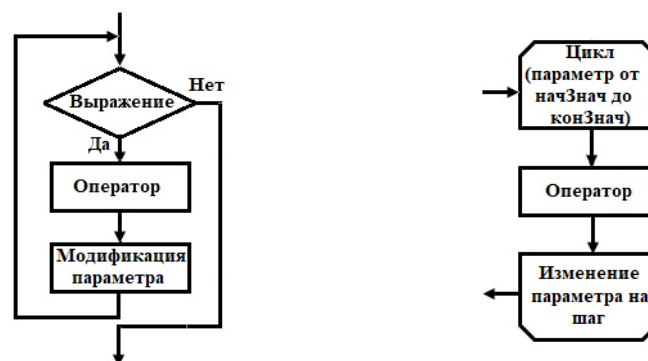
1 //Программа "Угадай число"
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 #include <ctime>
6
7 using namespace std;
8
9 int main()
10 {
11     SetConsoleCP(1251);
12     SetConsoleOutputCP(1251);
13     cout << "Угадай число!" << "\n";
14     cout << "-----" << "\n";
15     int n, rn; //n - попытка игрока, rn - загаданное программой число
16     srand(time(NULL)); //рандомизация - инициализация ДСЧ
17     do
18     {
19         rn = rand() % 10 + 1; //генерация случ.числа в диап. от 1 до 10
20         cout << "Введите натуральное число от 1 до 10 (0 - выход) >";
21         cin >> n;
22         if (n == rn)
23             cout << "Ура! Вы угадали" << "\n";
24         else
25             if (n != 0) cout << "Упс... Попробуйте еще раз!" << "\n";
26     } while (n != 0);
27     return 0;
28 }

```

Цикл с параметром for

Здесь в предусловии проверяется выражение с *параметром*, изменяющимся с каждой следующей итерацией на фиксированное значение – *шаг* (рис. 13). При достижении параметром порогового значения цикл завершается.

Параметрический цикл является частным случаем цикла с предусловием (и легко им заменяем). Но в отличие от него, цикл с параметром используется, когда число итераций заранее известно (по начальному и граничному значениям параметра и шагу).



а) смысловое представление б) стандартными символами

Рис. 13. Блок-схема параметрического цикла.

В языке C++ цикл с параметром реализуется с помощью *оператора for*. Синтаксис (формат) оператора **for**:

for (инициализация_параметра; выражение; модификация_параметра) оператор;

В качестве оператора может выступать блок операторов в фигурных скобках.

В листинге 2.5 приведён пример простой программы на вычисление факториала неотрицательного целого числа. Накопление произведения в переменной **factorial** удобно осуществлять в цикле **for**, в котором его параметр выступает в ходе итерации очередным множителем.

Примечание: модификация параметра цикла **for** допускается только в его заголовке. Попытки менять значения параметра помимо этого ещё и в теле цикла могут привести к возникновению заикливания или иным ошибкам. С позиций хорошего стиля программирования такие приёмы недопустимы.

Листинг 2.5.

```
1 //Программа вычисления факториала
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 using namespace std;
6
7 int main()
8 {
9     SetConsoleCP(1251);
10    SetConsoleOutputCP(1251);
11    cout << "Программа вычисления факториала целого неотрицательного числа" << "\n";
12    cout << "-----" << "\n";
13    int n; //Вводимое число
14    double factorial = 1; //Результат
15    cout << "Введите целое неотрицательное число >";
16    cin >> n;
17    for (int i = 2; i <= n; i++) //заголовок цикла, i - параметр
18        factorial = factorial * i; //тело цикла
19    cout << "Ответ: " << n << "! = " << factorial << "\n"; //вывод результата
20    system("PAUSE");
21    return 0;
22 }
```

Задание 2.5. Практика использования цикла с параметром

2.5.a. Реализуйте код листинга 2.5. Проверьте работу программы.

Получите значение 10! Объясните вывод программы.

2.5.б. Измените код программы из листинга 2.2, заменив в нём цикл с предусловием на цикл с параметром.

Индивидуальное задание №2.2

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №2.2** на циклическую конструкцию. Снабдите вывод программы (интерфейс) описанием задачи, вводимых данных и результата.

Протестируйте работу программы на нескольких наборах входных данных.

Составьте блок-схему алгоритма программы.

Вопросы для самоподготовки

- ✓ Что такое алгоритмическая конструкция?
- ✓ Какие алгоритмические конструкции называют базовыми и почему?
- ✓ Кто в истории информатики впервые назвал три базовые алгоритмические конструкции?
- ✓ Чем отличаются операторы **&&** и **||** от, соответственно, **&** и **|** (двойные от одиночных)?
- ✓ В примере 2 на стр. 24 что нужно изменить, чтобы определялся не максимум, а минимум из трёх чисел?
- ✓ Можно ли определить максимум (или минимум) из двух чисел без конструкции ветвления (линейной программой)?
- ✓ В программе из листинга 2.1 для каких значений переменной **day** сработает ветка **default**?
- ✓ Может ли тело цикла **while** не выполниться ни разу? Цикла **do-while**?
- ✓ В чём в общем случае заключается ошибка, приводящая к заикливанию?
- ✓ Почему в примере с простым числом (листинг 2.3) проверка осуществляется не до самого числа **n**, а до квадратного корня из него?
- ✓ Объясните формулу для масштабирования при генерировании случайного значения, приведённую в листинге 2.4 (строка 19).
- ✓ Приведите общую формулу, по которой можно было бы в C++ выводить случайные целые числа в произвольном диапазоне.

3. УКАЗАТЕЛИ И МАССИВЫ В ЯЗЫКЕ C++

Цель: познакомиться с возможностями и приёмами работы с указателями, статическими и динамическими массивами в языке C++.

Понятие массива

В предыдущих программах каждому значению какого-либо из простых типов соответствовало своё имя (идентификатор). Это может быть неудобным, если необходимо обработать длинную последовательность однотипных значений. В таком случае эффективнее присвоить одно имя для всей этой последовательности, а к самим значениям обращаться по их номерам (**индексам**) в этой последовательности. Такой тип данных называется **массивом** (рис. 14).

Объявление массива включает в себя имя массива, тип входящих в массив элементов и его **размер** (количество элементов в нём). Размер массива вместе с типом его элементов определяет необходимый для его хранения объём оперативной памяти.

Статические массивы – с фиксированной (константной) размерностью, тогда как у **динамических массивов** размерность в ходе работы программы может быть изменена.

Константная размерность статического массива единожды определяется **на этапе компиляции** и фиксируется в машинном коде (не может изменяться в ходе выполнения программы). Размерность динамического массива может определяться (и переопределяться) в ходе работы (**на этапе исполнения**) программы.

Статические массивы

Объявление статического массива в языке C++ включает в себя указание типа входящих в массив элементов, имени массива и его размера, например:

```
int a[25]; //массив из 25 целых чисел
float b[7]; //массив из 7 чисел с плавающей точкой
```

Примечание: здесь квадратные скобки являются элементом синтаксиса языка, а не признаком необязательности.

При описании массива используются те же модификаторы (например, **const** или **static**), что и при описании простых переменных.

Возможен вариант объявления массива одновременно с его **инициализацией** (т.е. присвоением начальных значений его элементам):

```
int m[7] = { 0, 2, 0, 4, 77 }; //использование инициализатора
```

Примечание: если инициализирующих значений меньше, чем элементов в массиве, то остаток массива обнуляется; если больше – лишние значения не используются.

Элементы массива **нумеруются** с 0, поэтому максимальный индекс (номер) элемента всегда на 1 меньше размера. В примере с массивом **m** номера его элементов от 0 до 6. Номер вызываемого элемента массива задаётся после его имени в квадратных скобках: **m[0]**, **m[5]**.

Примечание: автоматический контроль выхода индекса за границу массива в C++ не выполняется, программист должен следить за этим самостоятельно.

В C++ размер статического массива может быть задан единожды и только целой положительной константой (или константным выражением), не переменной.

Если при объявлении массива не указан размер, должен присутствовать **инициализатор**, в этом случае компилятор выделит память по количеству инициализирующих значений.

В следующем примере создаётся массив с инициализацией значений его элементов, после чего эти значения выводятся в консоль (листинг 3.1).

```

1 //Простой статический массив
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5
6 using namespace std;
7
8 int main()
9 {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12     const int n = 11; //размер массива
13     //объявление массива и инициализация его элементов:
14     int a[n] = { -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 };
15     //вывод значений массива в консоль:
16     cout << "Целочисленный массив a:\n";
17     for (int i = 0; i < n; i++) {
18         cout << "a[" << i << "]=" << a[i] << "\n";
19     }
20     system("PAUSE");
21     return 0;
22 }

```

В следующем примере (листинг 3.2) создаётся массив из 7 случайных положительных целых чисел и определяется максимальное значение в этом массиве.

```

1 //Статический массив случайных чисел
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 #include <ctime>
6
7 using namespace std;
8
9 int main()
10 {
11     SetConsoleCP(1251);
12     SetConsoleOutputCP(1251);
13
14     const int n = 7; //размер массива
15     int a[n]; //объявление массива
16
17     //заполнение массива и вывод значений в консоль
18     srand(time(NULL)); //рандомизация - инициализация ДСЧ
19     cout << "Целочисленный массив из " << n << " случайных чисел:\n";
20     for (int i = 0; i < n; i++) {
21         a[i] = rand() % 100;
22         cout << "a[" << i << "]=" << a[i] << ", ";
23     }
24     cout << "\n";
25 }

```



```

26 //поиск максимума
27 int maxA = 0;
28 for (int i = 0; i < n; i++) {
29     if (a[i] > maxA)
30         maxA = a[i];
31 }
32 cout << "Максимальное значение в массиве = " << maxA << "\n";
33
34 system("PAUSE");
35 return 0;
36 }

```

Задание 3.1. Приёмы работы со статическими линейными массивами

3.1.а. Реализуйте код листинга 3.1, проверьте его работу.

Исправьте код так, чтобы элементы массива можно было вводить с клавиатуры.

3.1.б. Реализуйте код листинга 3.2, проверьте его работу.

Исправьте код так, чтобы в массиве могли быть не только положительные, но и отрицательные случайные числа.

Исправьте код так, чтобы максимум определялся в одном цикле с заполнением массива.

Память процесса

При запуске прикладной программы операционная система (ОС) создаёт программный *процесс*. Для процесса создаётся учётная запись – *дескриптор* и выделяются необходимые для его исполнения *ресурсы* – в первую очередь процессор и оперативная память (чаще в форме аппаратной абстракции – виртуальной памяти). Собственная память процесса включает в себя (рис. 15):

- область для хранения машинного кода – **Text**;
- области для хранения *инициализированных* и *неинициализированных* глобальных или статических переменных – соответственно, **Data** и **BSS** (Block Started by Symbol);
- область для хранения аргументов, локальных переменных и метаинформации каждой из вызываемых функций – *стек вызовов* (или просто *стек*);
- *динамически распределяемая* область – **Heap** («куча»).

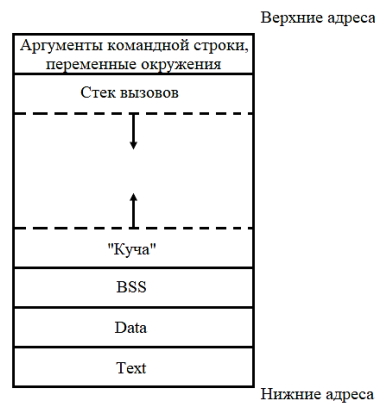


Рис. 15. Схема памяти процесса.

Представленная на рис. 15 схема справедлива для ОС UNIX/Linux. В ОС семейства Windows NT структура памяти программного процесса сложнее. Память процессу выделяется регионами по 64 Кб, и реальное расположение блоков определяется диспетчером памяти динамически⁷.

Т.о. в схеме памяти процесса можно выделить память, распределяемую:

- 1) **статически** компилятором (области Data и BSS), т.е. определяемая ещё до исполнения программы – **на этапе компиляции** кода;
- 2) **автоматически** для вызываемых функций (стек вызовов) – заполняется/очищается автоматически при вызове/завершении каждой функции, в т.ч. **main**;
- 3) **динамически** по мере необходимости («куча») – её заполнение и очистка определяются программистом произвольно и реализуются **на этапе исполнения** программы.

К значениям, размещённым в статической и автоматической памяти, можно обратиться как по адресам соответствующих ячеек в памяти, так и по идентификаторам (именам переменных и констант); к значениям в динамической памяти – только по адресам.

Статические массивы размещаются в статических областях памяти: Data, BSS и в стеке (в зависимости от того, каким объявлен массив – инициализированным, неинициализированным или локальным соответственно); динамические массивы (и массивы переменной длины) – в «куче».

Примечание: выделение памяти в «куче», хоть и медленнее, чем в стеке, но позволяет программисту по мере необходимости создавать и уничтожать большие структуры данных в оперативной памяти без риска переполнения стека (фатальной ошибки) и, как следствие, аварийного завершения программы.

⁷ Изучить карту памяти программного процесса позволит, например, утилита VMMap из пакета Windows Sysinternals.

Указатели

При объявлении переменной компилятор выделяет область в памяти размером, соответствующим типу значения (см. табл. 2). При последующих действиях с переменной (присвоение значения, использование в выражениях и пр.) компилятор подменяет **имя** (идентификатор) переменной на **адрес** соответствующей области памяти.

В языке C++ имеется механизм для работы с адресами областей памяти – **указатели** трёх видов: на объект, на функцию и на **void**.

Указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа (простого или составного). Синтаксис объявления указателя на объект:

```
тип * имя_указателя;
```

Символ «звёздочка» относится к имени, поэтому для того, чтобы объявить несколько указателей, требуется ставить её перед именем каждого из них. Так, в примере:

```
int *a, b, *c;
```

описываются два указателя на целочисленные переменные **a** и **c**, и переменная **b**. Размер указателя зависит от модели памяти.

Примечание: тип может быть любым, кроме ссылки и битового поля, причём составной тип может быть к этому моменту объявлен, но ещё не определён.

Указатель сам по себе может быть константой или переменной, а также может и указывать на константу или переменную:

```
int *pi; // указатель на целую переменную
```

```
const int *pci; // указатель на целую константу
```

```
//...
```

```
int i; // целая переменная
```

```
int *const cp = &i; // указатель-константа на целую переменную i
```

```
//...
```

```
const int ci = 1; // целая константа
```

```
const int *const cps = &ci; // указатель-константа на целую константу ci
```

Как видно из примеров, модификатор **const**, находящийся между именем указателя и звездочкой, относится к самому указателю и запрещает его изменение, а **const** слева от звездочки задаёт постоянство значения, на которое он указывает.

Для инициализации указателей в последних двух примерах использована операция получения адреса **&** (операция **разыменовывания**).

Величины типа указатель подчиняются общим правилам определения области действия, видимости и времени жизни.

Указатель содержит адрес некоего значения в памяти, но хранимый адрес – это тоже числовое значение, которое расположено в определённом участке памяти, поэтому можно определить *указатель на указатель* (т.е. адрес адреса некоего значения в памяти):

```
int **j;
```

Указатели в языке C++ используют обычно для работы с динамически распределяемой памятью – «кучей», которая выделяется ОС работающей программе и которую можно выделять и освобождать в соответствии с потребностями (см. рис. 15).

Доступ к выделенным участкам «кучи» (*динамическим значениям* или *структурам*) в программах на C++ производится только через указатели. Время жизни динамических значений (структур) – от точки создания до конца программы или до *явного* освобождения памяти.

Способы инициализации указателей:

1. Присваивание указателю адреса уже существующего значения (структуры):

- с помощью операции получения адреса (*разыменовывание*):

```
int a = 5; // целая переменная
int *p = &a; // в указатель p записывается адрес переменной a
int *p (&a); // то же самое другим способом
```

- присвоением значения другого инициализированного указателя:

```
int *r = p; //указатель r ранее был уже определён
```

- с помощью *имени массива* или *функции*, которые трактуются как адрес:

```
int b[10]; // массив
int *t = b; // присваивание указателю t адреса начала массива b
...
f(int a) { /* ... */ } // определение функции8
int (*pf)(int); // объявление указателя на целочисленную функцию f с одним параметром
pf = f; // присваивание адреса функции f указателю pf
```

2. Присваивание указателю адреса области памяти в явном виде (*литералом*):

```
char *xx = (char *) 0xB8000055;
```

Здесь 0xB8000055 – шестнадцатеричное числовое значение, (char *) – *операция приведения типа*: число преобразуется к типу «указатель на char».

3. Присваивание *пустого значения*:

```
int* pNull = NULL; // то же самое, что =0
```

Здесь используется константа NULL, определенная как указатель, равный нулю. Поскольку гарантируется, что объектов с нулевым адресом нет, пустой

⁸ Подробнее о функциях см. раздел 5.

указатель в программе можно использовать для проверки, ссылается указатель на конкретный объект или нет.

4. Выделение участка динамической памяти с присвоением её адреса указателю:

- с помощью C++ оператора **new**:

```
int* n = new int; //1  
int* m = new int (77); // 2  
int* q = new int [25]; // 3
```

- с помощью C-функции **malloc**:

```
int* u = (int *) malloc ( sizeof(int) ); // 4
```

Здесь в примере #1 операция **new** выполняет создание целочисленного значения с выделением достаточного для размещения величины типа **int** участка в «куче», а также возвращает адрес начала этого участка в переменную **n**.

В примере #2, кроме описанных выше действий (в примере #1), производится дополнительно запись в выделенную динамическую память значения 77.

В примере #3 операция **new** выполняет выделение участка в «куче» под последовательность из 25 величин типа **int** (по сути, **динамического массива** из 25 элементов) и возвращает адрес начала этого участка (начального значения массива) в переменную **q**, которая может трактоваться как имя массива.

Примечание: если память выделить не удалось, то должно породиться исключение **bad_alloc**.

В примере #4 делается то же, что и в #1, но с помощью функции выделения памяти **malloc** из стандартной библиотеки C. В функцию **malloc** передаётся один параметр – количество выделяемой памяти в байтах. Оператор **(int*)** используется для приведения типа указателя, возвращаемого C-функцией **malloc**, к требуемому типу. Если память выделить не удалось, функция возвращает 0.

Примечание: в C++ предпочтительнее использовать оператор **new**, чем C-функцию **malloc**, особенно при работе с объектами.

Освобождение памяти, выделенной с помощью операции **new**, должно выполняться с помощью **деструктора delete**, а памяти, выделенной C-функцией **malloc** – посредством C-функции **free**. При этом переменная-указатель сохраняется и может инициализироваться повторно. Невыполнение очистки динамической памяти приводит к эффекту «**утечки памяти**».

Созданные выше динамические переменные (примеры ##1-4) уничтожаются следующим образом соответственно:

```
delete n; delete m; // удаление динамических значений 1 и 2  
delete [] q; // удаление массива 3  
free (u); // очистка кучи от массива 4
```

Если память выделялась с помощью **new []** (пример #3), для освобождения памяти необходимо применять **delete []** без указания размера. Если квадратных скобок в **delete** не написать, то сообщения об ошибке не появится, но помечен как свободный будет только первый элемент последовательности (массива), а остальные окажутся недоступны для дальнейших операций. Такие ячейки памяти становятся «мусором».

С указателями можно выполнять следующие операции: *разадресация* или косвенное обращение к объекту (*), присваивание, сложение с константой, вычитание, инкремент (++), декремент (--), сравнение, приведение типов. При работе с указателями часто используется операция получения адреса существующего в памяти значения – *разыменовывание* (&).

В следующем примере (листинг 3.3) приведены некоторые действия с указателями на значения в различных областях собственной памяти программного процесса.

Задание 3.2. Работа с указателями

Реализуйте код листинга 3.3, проверьте его работу.

Добавьте в программу объявление указателя-константы.

Добавьте в программу указатель на любой из имеющихся указателей и его вывод в консоль с соответствующим комментарием.

Листинг 3.3.

```
1 //Указатели
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 using namespace std;
6
7 int k; //глобальная переменная (без инициализации)
8 const float pi = 3.1415926; //глобальная константа
9
10 int main()
11 {
12     SetConsoleCP(1251);
13     SetConsoleOutputCP(1251);
14
15     int* u_k = &(k); //указатель на неинициализированную глобальную переменную
16     cout << "Адрес глобальной неинициализированной переменной (BSS): " << u_k << "\n";
17     cout << "-----\n";
18
19     const float* u_pi = &(pi); //указатель на глобальную константу
20     cout << "Адрес глобальной константы (Data): " << u_pi << "\n";
21     cout << "-----\n";
22
23     int a = 25; //локальная переменная
24     int* u_a = &(a); //указатель на локальную переменную
25     cout << "Адрес локальной переменной (Stack): " << u_a << "\n";
26     cout << "-----\n";
27 }
```

```

28     float arr_s[1000]; //статический неинициализированный массив в стеке
29     cout << "Адрес локального статического массива (Stack): " << arr_s << "\n";
30
31     //работа с динамической памятью (кучей)
32     float* arr_d = new float[1000]; //массив из 1000 вещественных чисел в куче
33     cout << "Адрес в памяти массива (Heap): " << arr_d << "\n";
34     cout << "-----\n";
35     delete[] arr_d; //массив больше не нужен - освобождаем память в куче
36
37     int (*u_main)(); //объявление указателя на функцию
38     u_main = main; //присвоение значения указателю на функцию
39     cout << "Адрес в памяти кода функции main (Text): " << u_main << "\n";
40     cout << "-----\n\n";
41
42     system("PAUSE");
43     return 0;
44 }

```

Линейные массивы в «куче». Сортировка массива выбором

Статические массивы, рассмотренные выше, размещаются чаще всего в весьма ограниченной области выделяемой программе памяти – в *стеке вызовов*. Переполнение стека (что весьма вероятно в случае больших массивов) приведет к аварийному завершению работы программы. Поэтому массивы целесообразно размещать в динамической (динамически распределяемой) памяти – «*куче*».

Размещение массива в «куче» позволит также уйти от константной определенности размера массива. Размер массива в «куче» можно задать с помощью переменной или выражения.

Рассмотрим два типа массивов, размещаемых в «куче»:

- а) массивы переменной длины (*new/delete*);
- б) *malloc*-массивы, унаследованные из языка C (*malloc/free*).

С помощью команд ***new/delete*** можно соответственно создавать/уничтожать *массивы переменной длины* в динамически распределяемой памяти:

```

int n; int* q = new int [n];
//...
delete [] q;

```

Квадратные скобки здесь – элемент синтаксиса, а не признак необязательности.

Размер уже созданного массива переменной длины изменить не удастся, но его можно многократно пересоздавать заново (с потерей хранящихся в нём значений), каждый раз задавая новый размер (например, посредством ввода с клавиатуры).

Примечание: массивы переменной длины в языке C++ часто называют *динамическими массивами*, что, с одной стороны, отражает место расположения массива – в динамической памяти («куче»), но, с другой стороны, вводит в заблуждение – ведь изменить размер уже созданного массива без его удаления нельзя.

В следующем примере (листинг 3.4) массив переменной длины: создается (размер вводится с клавиатуры при выполнении программы) – строки 13-15; заполняется значениями (с клавиатуры) – строки 16-19; *сортируется* по возрастанию (точнее, по неубыванию) *методом выбора* – строки 20-33; выводится в консоль – строка 35; удаляется из памяти – строка 36.

В С++ размер массива, созданного командой **new []**, без его удаления не изменить.

Листинг 3.4.

```

1  //сортировка массива переменной длины
2  #include <cstdlib>
3  #include <iostream>
4  #include <Windows.h>
5
6  using namespace std;
7
8  int main()
9  {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12     int n, i, imin;
13     cout << "Задайте размер линейного массива = ";
14     cin >> n;
15     int* arr = new int[n]; //при объявлении массива размер уже должен быть определён
16     for (i = 0; i < n; i++) {
17         cout << "arr[" << i << "]=";
18         cin >> arr[i]; //заполняем массив с клавиатуры
19     }
20     //сортировка методом выбора:
21     for (i = 0; i < n - 1; i++) { //n-1 раз ищем наименьший элемент
22         imin = i; // принимаем за наименьший первый из рассматриваемых элементов
23         // поиск номера минимального элемента из неупорядоченных (справа от i-го):
24         for (int j = i + 1; j < n; j++)
25             // если нашли меньший элемент, запоминаем его номер:
26             if (arr[j] < arr[imin]) imin = j;
27         if (imin != i) {
28             // обмен элементов с номерами i и imin:
29             arr[i] += arr[imin];
30             arr[imin] = arr[i] - arr[imin];
31             arr[i] = arr[i] - arr[imin];
32         }
33     }
34     // вывод упорядоченного массива:
35     for (i = 0; i < n; i++) cout << arr[i] << " ";
36     delete[] arr; //высвобождение динамической памяти
37     system("PAUSE");
38     return 0;
39 }

```

В полном смысле динамический массив (размер которого при необходимости может изменяться на этапе выполнения программы) в С++ реализуют с помощью функции **malloc** из стандартной библиотеки языка С:

```

int *u = (int *) malloc(5*sizeof(int)); //массив 5 целых чисел
//...
free(u); //освобождение памяти

```

Здесь функция **free** освобождает «кучу» после окончания работы с

массивом. Динамически изменить размер такого массива позволит функция **realloc**:

```
u = (int *) realloc (u, 25 * sizeof(int));
```

При этом размер массива **u** увеличится на 25 ячеек для хранения целых чисел.

Задание 3.3. Приёмы работы с динамическими линейными массивами

3.3.а. Реализуйте код листинга 3.4, проверьте его работу.

В режиме отладки (см. стр. 29) проследите за содержимым массива по ходу выполнения программы.

Объясните на простом примере суть алгоритма сортировки выбором.

3.3.б. Самостоятельно реализуйте программу, в которой введённое пользователем неотрицательное целое число записывается в **malloc**-массив его цифр, причём изначально массив создаётся размером в одну ячейку и динамически расширяется на нужное количество ячеек (по количеству цифр во исходном числе). Дополнительное уточнение – разрядность вводимого числа заранее не известна, в программе допускается использовать лишь один цикл.

Многомерные статические массивы в C++

Многомерные массивы (см. рис. 16, б и в) в языке C++ объявляются с указанием каждого *измерения* в квадратных скобках, например, оператор

```
int a[6][8];
```

задает описание двумерного *статического* массива из 6 строк и 8 столбцов.

В памяти такой массив располагается в последовательных ячейках построчно (т.е. двумерный массив можно рассматривать как линейный массив линейных массивов). Строки в памяти ничем не разделены, т.е. прямоугольной матрицей этот массив является только в нашем воображении.

Для доступа к элементу многомерного массива последовательно указываются все его индексы: **a[i][j]**. Можно обратиться к элементу и другими способами: ***(a[i]+j)** или ***(*(a+i)+j)**.

Рассмотрим подробнее последний способ, т.к. в нём в явном виде записаны действия, реализуемые компилятором при обращении к элементу массива.

Пусть необходимо обратиться к элементу, расположенному на пересечении второй строки и третьего столбца – **a[2][3]**⁹. Как и для одномерных массивов, имя массива **a** является константным указателем на начало массива.

Сначала требуется обратиться ко второй строке массива, то есть одномерному массиву **a[2]**. Для этого надо прибавить к адресу начала массива (нулевой

⁹ Напомним, что индексы отсчитываются от 0.

строке) *смещение*, равное количеству строк, и выполнить *разадресацию*: $*(a + 2)$. При сложении указателя с константой компилятором учитывается длина адресуемого элемента, поэтому на самом деле число 2 автоматически умножается на длину хранимого в памяти элемента, в нашем случае $2 \cdot (8 \cdot \text{sizeof}(\text{int}))$, поскольку элементом является строка, состоящая из 8 элементов типа **int**.

Далее требуется обратиться к 3-му элементу массива $a[2]$. Для получения его адреса опять применяется сложение указателя с константой 3, т.е. прибавляется $3 \cdot \text{sizeof}(\text{int})$, а затем повторно применяется разадресация для получения значения элемента: $((*a + 2) + 3)$.

В C++ элементы статического массива задаются *инициализатором* в порядке их расположения в памяти. Например, команда:

```
int b[2][5] = { 0, 2, 1, 7, 8, 2, 3, 4, 5, 7 };
```

определяет матрицу со следующими значениями элементов:

0	2	1	7	8
2	3	4	5	7

Если количество значений в фигурных скобках превышает количество элементов в массиве, возникает ошибка компиляции. Если значений меньше, оставшиеся элементы массива инициализируются значением по умолчанию (для основных типов это 0).

При инициализации многомерного массива он представляется либо как *массив из массивов*, при этом каждый массив заключается в свои фигурные скобки (тогда левую размерность при описании можно не указывать), либо задается общий список элементов в порядке расположения элементов в памяти:

```
int mass2[][2] = {{1,1},{0,2},{1,0}};  
int mass2[3][2] = {1,1,0,2,1,0};
```

В следующем примере программа определяет в *двумерном статическом массиве* (в целочисленной прямоугольной матрице) номер строки, которая содержит наибольшее количество элементов, равных нулю (листинг 3.5).

Здесь номер искомой строки хранится в переменной **str0**, количество нулевых элементов в текущей (i-й) строке матрицы – в **s0**, текущий максимум нулевых элементов – в **Max0**.

Массив просматривается по строкам, в каждой подсчитывается количество нулевых элементов (**s0** обнуляется перед просмотром каждой следующей строки). Если в текущей строке нулей больше, чем в предыдущих, то наибольшее количество нулей (**Max0**) и номер соответствующей строки (**str0**) запоминаются.

```

1 //Двумерный статический массив
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 #include <ctime>
6 using namespace std;
7
8 int main()
9 {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12     const int nstr = 4, nstb = 5; //размеры матрицы
13     int b[nstr][nstb]; //объявление массива
14     int Max0 = 0, str0 = -1; //максимальное количество нулей и строка, где они
15     int s0 = 0; //количество нулей в текущей строке матрицы
16     int i, j; //индексы - строки и столбцы
17     srand(time(NULL)); //инициализация ДСЧ
18
19     //заполнение и вывод массива:
20     cout << "Исходный массив:\n";
21     for (i = 0; i < nstr; i++) {
22         for (j = 0; j < nstb; j++) {
23             b[i][j] = rand() % 2;
24             cout << b[i][j] << " ";
25         }
26         cout << "\n";
27     }
28
29     //перебираем строки
30     for (i = 0; i < nstr; i++) {
31         s0 = 0;
32         //подсчитываем нули в текущей строке:
33         for (j = 0; j < nstb; j++)
34             if (b[i][j] == 0) s0++;
35         if (s0 > Max0) { str0 = i; Max0 = s0; } //запоминаем, если нулей больше
36     }
37
38     if (str0 == -1) cout << "Нулевых элементов нет!\n";
39     else
40         cout << "Номер строки с наибольшим кол-вом 0: " << str0 << "(начиная с 0-й)\n";
41
42     system("PAUSE");
43     return 0;
44 }

```

Многомерные динамические массивы средствами C++

N-мерный массив в динамически распределяемой памяти в языке C++ представляется как **линейный массив указателей** на (N-1)-мерные массивы (например, двумерный массив – это линейный массив указателей на линейные массивы-строки – см. рис. 16).

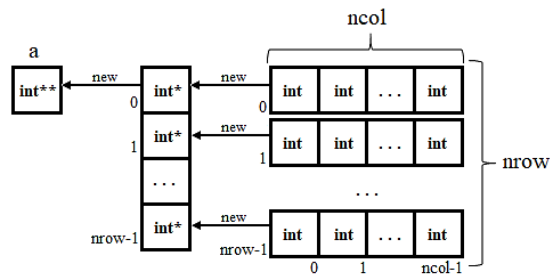


Рис. 16. Выделение памяти под двумерный целочисленный массив переменной длины.

Пример создания в «куче» **двухмерной прямоугольной матрицы**:

```
int nrow, ncol; //две размерности – строки и столбцы
cout << "Введите количество строк и столбцов: "; cin >> nrow >> ncol;
int **a = new int *[nrow]; //1
for (int i=0; i < nrow; i++) //2
    a[i] = new int[ncol];    //3
```

Оператор **new** делает две вещи: а) выделяет область в «куче» под заданное количество элементов определённого типа; б) возвращает адрес начала массива (нулевого элемента) в выделенной области. Этот адрес присваивается указателю соответствующего типа.

Тогда в строке #1:

- слева от оператора присваивания объявляется переменная **a** типа «указатель на указатель на значение типа **int**»;

- **new** выделяет память в «куче» под пустой линейный массив из **nrow** указателей (по количеству строк в матрице) на начальные элементы **nrow** целочисленных строк;

- переменной **a** присваивается возвращённый оператором **new** адрес начала этого служебного массива указателей на строки (**a** будет именем всей матрицы).

2: организуется цикл для выделения памяти под каждую из **nrow** строк матрицы.

3: в цикле в каждый элемент линейного массива **a** указателей на строки возвращается адрес начала участка памяти, выделенного под соответствующую строку. Каждая строка состоит из **ncol** элементов типа **int** (рис. 16).

Обращение к элементам динамических массивов в программе производится так же, как к элементам статических – с помощью конструкции вида **a[i][j]**.

Освобождение памяти в «куче» из-под многомерного массива выполняется с помощью операций **delete []** в порядке, обратном порядку создания этого массива. Так, массив **a** из предыдущего примера будет удалён следующим образом:

```
for (int i=0; i < nrow; i++) // в цикле по количеству строк
    delete a[i]; // удаляем из кучи строки
delete []a; // удаляем служебный массив указателей
```

Если удалить только массив указателей, то массивы-строки с данными

продолжат занимать «кучу», что, в свою очередь, приведёт к эффекту «утечки» памяти.

Задание 3.4. Приёмы работы с многомерными массивами в C++

3.4.a. Реализуйте код листинга 3.5, проверьте его работу.

Объясните выражение в строке 23. Измените его так, чтобы массив заполнялся целыми числами в диапазоне $[-1;1]$.

Отредактируйте код листинга 3.5, чтобы вывод массива осуществлялся ровными столбцами (например, с помощью форматного вывода).

3.4.б. Самостоятельно перепишите код листинга 3.5 под работу с массивом переменной длины, чтобы размерность массива (и количество строк, и количество столбцов) задавалось пользователем с клавиатуры.

Примечание: не забудьте после использования массива удалить его из динамически распределяемой памяти («кучи»).

Индивидуальное задание №3

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №3** на массивы. Снабдите вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата.

Примечание: используйте только массивы переменной длины (**new/delete**) или динамические (**malloc/free**). Массив может заполняться как псевдослучайными значениями, так и вводиться с клавиатуры в зависимости от условия задачи.

Протестируйте работу программы на нескольких наборах входных данных.

Вопросы для самоподготовки

- ✓ Что такое массив? Что такое многомерный массив?
- ✓ Где в собственной памяти программного процесса и что содержится в случае объявленного в функции **main**, но неинициализированного статического массива? Как он хранится в исполняемом файле?
- ✓ Если запустить программу из листинга 3.3 несколько раз, будет ли расположение блоков памяти неизменным?
- ✓ Что такое глобальная переменная? Что такое статическая переменная?
- ✓ В чём преимущество расположения массива в «куче»?
- ✓ Когда память в стеке освободится от массива **arr_s** в программе из листинга 3.3?
- ✓ Зачем может понадобиться сортировка массива?
- ✓ Относится ли метод сортировки массива выбором к быстрым сортировкам?
- ✓ Назовите примеры алгоритмов быстрой сортировки.

- ✓ В чём отличие сортировки по возрастанию и по убыванию?
- ✓ Как сделать копию расположенного в «куче» массива?
- ✓ В чём отличие динамического массива (**malloc/free**) от массива переменной длины (**new/delete**)?
- ✓ Как в программах на C++ располагается в памяти многомерный массив?

4. СИМВОЛЫ И СТРОКИ В ЯЗЫКЕ C++. СТРУКТУРЫ

Цель: познакомиться с возможностями и приёмами работы с символами и строками, а также структурным типом в языке C++.

Работа с символами в C++

Для хранения отдельных символов используются переменные типа **char**. В C/C++ **char** – это целочисленный тип, имеющий ширину 8 бит (кодировка [ASCII](#)).

Ввод-вывод символов также может выполняться с помощью операций помещения в поток << и извлечения из потока >>, а также методов **get()** и **getchar()**.

Пример применения операций ввода-вывода:

```
char c, d, e;
cin >> c; cin >> d >> e;
cout << c << d << e << endl;
```

Вводимые символы могут разделяться или не разделяться пробельными символами, поэтому таким способом ввести символ пробела нельзя. Для ввода любого символа, включая пробельные, можно воспользоваться методом **get()**:

```
char c, d, e;
c = cin.get(); cin.get(d); cin.get(e);
cout << c << d << e << endl;
```

Метод **get()** возвращает код извлеченного из потока символа или EOF.

Задание 4.1. Приёмы работы с символьными значениями

В следующем примере (листинг 4.1) приведена программа преобразования введенного с клавиатуры символа (в кодировке Windows-1251) из заглавного в строчный и наоборот на основе анализа кода этого символа.

Листинг 4.1.

```
1 //Смена регистра
2 #include <iostream>
3 #include <cstdlib>
4 #include <cstring>
5 #include <Windows.h>
6
7 using namespace std;
8
9 int main()
10 {
11     SetConsoleCP(1251);
12     SetConsoleOutputCP(1251);
13
14     unsigned char ch;
15     cin >> ch;
16     if (((int)ch < 91) || (((int)ch > 191) && (((int)ch < 224))))
17         cout << (char)(ch + 32);
18     else if (((int)ch > 96) && (((int)ch < 123))) || (((int)ch > 223) && (((int)ch <= 255))))
19         cout << (char)(ch - 32);
20
21     system("PAUSE");
22     return 0;
23 }
```

4.1. Реализуйте код листинга 4.1, **проверьте** его работу на латинских и русских заглавных и прописных буквах. **Сформируйте** интерфейсную часть – вывод в консоль описания программы и подсказок пользователю по вводу данных.

Самостоятельно **дополните** код возможностью ввода символов в цикле, пока не будет введена, например, цифра 0, или (более сложный вариант) нажата клавиша <Esc>.

Проверьте работу программы на символах «Ё» и «ё». **Отредактируйте** код, чтобы он корректно обрабатывал и эти символы.

Примечание: посредством заголовочного файла **cctype** имеется доступ к целому ряду функций, проверяющих принадлежность символа какому-либо множеству, например, множеству букв (**isalpha**), разделителей (**isspace**), знаков пунктуации (**ispunct**), цифр (**isdigit**) и т.д.

Строки в языке C++

В C++ поддерживается два типа строк: С-строка и объект.

Строку можно рассматривать как **массив элементов символьного типа**, который завершается (ограничивается) специальным нулевым символом '\0' (т.е. символом с кодом 0). Таким образом, **строка** состоит из символов с нуль-символом на конце.

Такие строки унаследованы из языка С и в настоящее время широко используются в программировании. Поэтому, используя термин «строка», C++-программист обычно имеет в виду именно строку с завершающим нулем (С-строку).

Средства C++ позволяют обращаться к отдельным символам С-строки как к элементам массива – по их индексам.

В языке C++ существует ещё один способ представления строк – **класс string** (т.е. строка – это объект, экземпляр класса **string**), что подразумевает объектно-ориентированный подход к обработке строк.

Объявление и ввод-вывод С-строк в языке C++

Объявляя символьную последовательность, предназначенную для хранения строки с завершающим нулем, необходимо учитывать признак её завершения и задавать длину для строки на единицу больше длины самой большой цепочки символов из тех, которые предполагается хранить в этой строке. Например, при объявлении строки `str`, в которую предполагается поместить 80-символов, следует объявить:

```
char str[81];
```

Здесь размер 81 позволяет зарезервировать место и для нулевого символа в конце строки.

Размерность статической строки может быть задана только константным выражением. Часто размер такой строки задают отдельной константой:

```
const int len_str = 81;  
char str[len_str];
```

Это удобно, поскольку при возможном изменении длины строки потребуется изменить программу только в одном месте.

При описании строки её можно сразу инициализировать строковой константой (0-символ в позиции, следующей за последним символом, сформируется автоматически):

```
char a[100] = "Никогда не сдавайся!";
```

Если строка инициализируется, ее размерность можно не указывать – компилятор сам выделит память, достаточную для размещения строки и завершающего её нуля:

```
char a[] = "Никогда не сдавайся!"; // 21 символ
```

Для размещения строки в динамической памяти следует описать указатель на char, а затем выделить память с помощью **new** или **malloc** (аналогично рассмотренным в предыдущей работе динамическим массивам):

```
char *p = new char [m]; // строка переменной длины  
char *q = (char *) malloc(m * sizeof(char)); // динамическая строка
```

В этом случае длина строки может быть переменной и задаваться на этапе выполнения программы.

Примечание: динамические строки, как и другие динамические массивы, нельзя инициализировать при создании.

Для ввода-вывода строк используются объекты cin и cout:

```
#include <iostream>  
using namespace std;  
int main() {  
    const int n = 81;  
    char s[n];  
    cin >> s; cout << s << "\n";  
}
```

Строка вводится так же, как и переменные других типов. Ввод выполняется до первого пробельного символа (пробела, знака табуляции или символа перевода строки): если ввести строку из нескольких слов, будет воспринято только первое слово.

При этом *отсутствует граничная проверка*: если во вводимой строке больше символов, чем может вместить выделенная для ее хранения область, поведение программы не определено (она может завершиться аварийно).

Если требуется ввести строку, состоящую из нескольких слов, в одну

строковую переменную, используются методы (функции) **getline** или **get** класса **istream**, объектом которого является **cin**. Синтаксис вызова метода:

```
const int n = 81;
char s[n];
cin.getline(s, n); cout << s << endl;
cin.get(s, n); cout << s << endl;
```

Здесь метод **getline** считывает из входного потока ($n - 1$) символов или менее (если символ перевода строки встретится раньше) и записывает их в строковую переменную *s*. Символ перевода строки `"\n"` (появляется во входном потоке при нажатии клавиши <Enter>) считывается (и удаляется) из входного потока, но не записывается в строковую переменную, вместо него размещается завершающий 0-символ.

Примечание: если в исходных данных более $n - 1$ символов, следующий ввод будет выполняться из той же строки, начиная с первого нечитанного символа.

Метод **get** работает аналогично, но оставляет в потоке символ перевода строки. В строковую переменную добавляется завершающий 0.

Оставляемый в потоке методом **get** символ `"\n"` не позволит ввести этим же методом следующую строку (в переменную запишется пустая строка, а символ `"\n"` вновь останется в потоке). Удалить символ `"\n"` из потока можно вызвав метод **get** без параметров: **cin.get()**.

Более простое решение – использовать метод **getline**, который после прочтения строки не оставляет во входном потоке символ `"\n"`. Если требуется ввести строку несколько раз, метод **getline** удобно использовать в заголовке цикла, например:

```
const int n = 81;
char s[n];
while (cin.getline(s, n)) {
    cout << s << endl;
    ... // обработка строки
}
```

Операции с С-строками

Язык C++ поддерживает множество функций обработки С-строк¹⁰. Для вызова этих функций в программу необходимо включить заголовок `<cstring>` (он же заголовочный файл `string.h`). Подробное описание и синтаксис этих функций можно найти в справочной литературе.

В C++ для С-строк не определена операция присваивания, поскольку они не являются основным типом данных. Присваивание делается с помощью функций

¹⁰ Символы и строки в C++: [<http://cppstudio.com/post/437/>]

библиотеки или посимвольно в цикле. Например, чтобы присвоить строке **p** строку **a**, можно воспользоваться функцией **strcpy**:

```
char a[100] = "Никогда не сдавайся!";  
char *p = new char [m];  
strcpy(p, a);
```

Программист должен сам заботиться о том, чтобы в строке-приемнике хватило места для строки-источника (в данном случае при выделении памяти значение переменной **m** должно быть больше или равно 100), и о том, чтобы строка всегда имела завершающий нуль-символ.

Примечание: выход за границы строки и отсутствие нуль-символа являются распространенными причинами ошибок в программах обработки строк.

Функция **strlen()** возвращает фактическую длину строки, не включая нуль-символ.

Для преобразования строки в целое число используется функция **atoi()**. Она преобразует строку, содержащую символьное представление целого числа, в соответствующее целое число. Признаком конца числа служит первый символ, который не может быть интерпретирован как принадлежащий числу. Если преобразование не удалось, возвращает 0.

Аналогичные функции преобразования строки в длинное целое число (**long**) и в вещественное число с двойной точностью (**double**) называются **atol** и **atof** соответственно. Пример применения функций преобразования:

```
char a[] = "10) Рост - 162 см, вес - 63.5 кг";  
int num = atoi(a);  
long height = atol(&a[11]);  
double weight = atof(&a[25]);  
cout << num << " " << height << " " << weight;
```

Библиотека предоставляет также функции для сравнения строк и подстрок, объединения строк, поиска символа и подстроки, выделения из строки лексем.

В следующем примере (листинг 4.2) реализована программа, которая определяет, встречается ли в данном тексте заданная последовательность символов. Длина строки текста не превышает 80 символов, текст не содержит переносов слов, последовательность не содержит пробельных символов.

```

1 //Поиск подстроки
2 #include <iostream>
3 #include <stdlib.h>
4 #include <cstring>
5 #include <Windows.h>
6
7 using namespace std;
8
9 int main()
10 {
11     SetConsoleCP(1251);
12     SetConsoleOutputCP(1251);
13
14     const int len = 81;
15     char word[len], line[len];
16     cout << "Введите текст: "; cin >> line;
17     cout << "Введите слово для поиска: "; cin >> word;
18     if (strstr(line, word))
19         cout << "Присутствует!" << endl;
20     else
21         cout << "Отсутствует!" << endl;
22
23     system("PAUSE");
24     return 0;
25 }

```

Здесь для анализа строки применяется функция **strstr**. Она выполняет поиск подстроки **word** в строке **line**. Обе строки должны завершаться нуль-символами. В случае успешного поиска функция возвращает указатель на найденную подстроку, в случае неудачи – NULL (пустой указатель). Если искомая подстрока пуста, функция возвращает указатель на начало строки **line**.

Задание 4.2. Ввод и поиск по строке

4.2.а. Реализуйте код листинга 4.2, проверьте его работу.

4.2.б. Самостоятельно реализуйте в коде возможность поиска подстроки в тексте, содержащем пробельные символы.

Структура

В языке C++ помимо массивов (и, как следствие, C-строк) к типам данных, определяемых пользователем, относятся *структурный тип (структуры)*.

Массив позволяет объединить несколько однотипных значений в единое целое. В отличие от массива, структура может объединить в себе (в одном экземпляре) значения *разных типов*. Другое отличие заключается в том, что в массиве значения пронумерованы, а в структуре – *поименованы*, т.е. к значению в структуре обращаются по символьному имени (*ключу*).

Структурный тип определяется перечнем *полей* с указанием их имён и типов – любых, даже других структур (кроме самой этой структуры, но допустимы указатели на неё):

```
struct [имя_типа] {
```

```

    тип_1 имя_поля_1;
    тип_2 имя_поля_2;
...
    тип_n имя_поля_n;
} [список_описателей] ;

```

Здесь квадратные скобки – это признак необязательности. Если отсутствует **имя типа**, то должен быть **список описателей** – переменных, указателей или массивов этого типа.

Описание структуры всегда оканчивается символом ; (точка с запятой).

Пример:

```

struct Employee {
    short id;
    char name[30];
    int age;
    double salary;
};

```

Здесь определена структура с именем **Employee**, которая содержит 4 поля:

- 1) **id** типа **short**;
- 2) символьный массив (строка) **name[30]**;
- 3) **age** типа **int**;
- 4) **salary** типа **double**.

После этого в программе объявляют новую переменную (или несколько переменных) типа этой структуры (один или несколько **экземпляров** структуры этого типа):

```

Employee michael;    // создаем экземпляр структуры Employee - michael
Employee john;        // создаем ещё один экземпляр структуры Employee - john

```

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее), если компилятору не требуется знать **размер структуры**. Этот приём позволяет создавать **связные списки** структур.

Обращение к полям экземпляра структуры (например, для инициализации):

```

john.id = 8; // присваиваем значение полю id структуры john
john.name = "Павлов"; // присваиваем значение полю lastname структуры john
john.age = 27; // присваиваем значение полю age структуры john
john.salary = 32.17; // присваиваем значение полю salary структуры john

```

Другой (более простой) способ – объявление с инициализацией экземпляра структуры:

```

Employee james = {9, "Саблин", 30, 28.35};

```

Примечание: размер структуры будет не меньше суммы размеров всех её полей (но может быть и больше). По соображениям производительности компилятор

может добавлять «пробелы/промежутки» в структуры (т.е. выравнивать данные). Размер структуры можно узнать с помощью функции **sizeof()**.

Важным преимуществом структуры является возможность передать с её помощью на обработку в функцию сразу несколько разнотипных значений, равно как и вернуть их из функции в точку вызова.

Освоить структуры важно, поскольку они – шаг к пониманию ООП. Действительно, любая **предметная область** может рассматриваться как множество связанных **сущностей** (классов и их экземпляров – **объектов**). Сущность же описывается как набор **атрибутов** (полей), представляемый нами в виде структуры. Отметим только, что помимо набора полей **активная** сущность характеризуется ещё и **поведением**, но это уже выходит за рамки предмета нашего рассмотрения.

Задание 4.3. Приёмы работы со структурами в C++

4.3. Самостоятельно реализуйте в новой программе:

- Описание структуры с именем **Airport**, содержащей следующие поля:
 - номер рейса;
 - название пункта вылета;
 - название пункта назначения;
 - тип самолета;
 - число посадочных мест.
- Создание динамического массива (программно), состоящего из 5-7 различных элементов – экземпляров типа **Airport** (данные полей любые).
- Текстовый интерфейс, позволяющий пользователю:
 - ввести с клавиатуры новые элементы этого массива;
 - вывести на экран в алфавитном порядке пункты назначения и номера рейсов, обслуживаемых самолетом, тип которого задан с клавиатуры.

Проверьте работу получившейся программы.

Индивидуальное задание №4

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №4** на строки. Снабдите вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата.

Можно использовать в решении любое представление строки (и как массив и как объект).

Примечание: формулировка в задачнике [3] «дано число n , последовательность символов s_1, \dots, s_n » означает, что просто дана произвольная строка. Длина строки (n) определяется по результату ввода строки пользователем, а **не задаётся явно**.

Примечание: если задача непосредственно решается имеющейся встроенной функцией или методом, то необходимо представить решение как с использованием этой функции (метода), так и обязательно – **без неё**.

Протестируйте работу программы на нескольких наборах входных данных.

Вопросы для самоподготовки

- ✓ К каким типам в языке C++ относится тип `char`?
- ✓ В чём отличие C-строки от представления строки в виде объекта?
- ✓ Что такое конкатенация строк?
- ✓ Какова максимально допустимая длина строки в языках C++?
- ✓ Сформулируйте определение структуры.
- ✓ В чём отличия структуры от массива?
- ✓ Что такое вложенная структура?

5. ПОДПРОГРАММЫ

Цель: познакомиться с понятием подпрограммы и приёмами работы с ними в языке C++ в рамках парадигмы структурного программирования.

Понятие подпрограммы

В концепции [структурного программирования](#) одним из важных понятий является *подпрограмма*. Подпрограмма – это именованный (идентифицируемый) фрагмент программы, реализующий законченное действие, и который можно вызвать из других точек программы многократно. Пример – подпрограмма, вычисляющая значение синуса угла.

Подпрограмма может быть несколько раз вызвана по своему имени при выполнении программы. Например, если требуется вычислить значение по формуле:

$$\sin A * \cos B + \cos A * \sin B,$$

то потребуется вычислять и синус, и косинус дважды, т.е. дважды вызвать соответствующие подпрограммы с разными *аргументами*.

Подпрограммы упрощают структуру программы, их используют в целях повышения компактности кода, его читаемости (так, в предыдущем примере – нет необходимости повторно писать один и тот же код для вычислений синуса или косинуса) и надёжности (повторно используется уже протестированный надёжный код).

Написанная однажды и всесторонне проверенная подпрограмма может быть включена в *библиотеку* и использоваться в других программах. Существует множество полезных библиотек подпрограмм, которыми могут пользоваться все программисты.

Подпрограммы реализуются по типу *процедур* – определенного набора действий, не требующего возврата какого-либо значения, и *функций* – подпрограмм с возвратом результата (значения определенного типа).

В языке C++ все подпрограммы (и функции, и процедуры) принято называть функциями. Процедуру здесь можно рассматривать как частный случай функции, не возвращающей значения.

Примечание: в последующем все подпрограммы будем называть функциями.

Входящие в состав класса (объекта) функции называются *методами* этого класса (объекта). Подключаемые *библиотеки классов* являются поставщиками в том числе и необходимых программистам функций.

Любая функция должна быть: а) *объявлена* и б) *определена*. Объявление

функции должно находиться в тексте программы раньше её **вызова** для того, чтобы компилятор мог осуществить проверку правильности вызова.

Объявление функции (метода) – это задание **прототипа** (заголовка, сигнатуры), включающего имя, тип возвращаемого значения и (при наличии) список передаваемых **параметров**:

[модификаторы] тип имя ([список параметров]) [throw (исключения)]

Составные части определения функции (метода) в С-подобных языках:

- **Модификаторы** (*спецификаторы*). Модификатор – это ключевое слово, которое может изменить (уточнить) смысл некоторого определения (переменной, функции, класса, метода)
- **Тип** – тип возвращаемого функцией значения может быть любым, кроме массива и функции (но может быть указателем на массив или функцию). Если функция не должна возвращать значение (т.е. является по сути процедурой), указывается ключевое слово **void**.
- **Список параметров** (формальных) – определяет величины, которые требуется передать в функцию при ее вызове. Элементы списка параметров разделяются запятыми. Для каждого параметра, передаваемого в функцию, указывается его тип и имя. В определении, в объявлении и при вызове одной и той же функции типы и порядок следования параметров должны совпадать.

Примечание: на имена параметров ограничений по соответствию не накладывается, поскольку функцию можно вызывать с различными аргументами, а в прототипах имена компилятором игнорируются (они служат только для улучшения читаемости программы).

Определение функции помимо заголовка включает в себя **тело функции**. В программе может содержаться произвольное количество объявлений одной и той же функции и только одно определение.

Для **вызова функции** в программе нужно указать её имя, за которым в круглых скобках через запятую перечисляются имена передаваемых аргументов.

Вызов функции может находиться в любом месте программы, где по синтаксису допустимо значение того типа, который возвращает функция.

Если тип возвращаемого функцией значения не **void**, она может входить в состав выражений, в т.ч. располагаться в правой части оператора присваивания **=**.

При вызове функции выделяется область памяти в стеке вызовов (стековый кадр) для хранения адреса возврата (адреса команды, следующей за вызовом функции), переданных функции аргументов, а также **локальных** констант и

переменных.

После вызова управление программой переходит к функции. Когда функция завершается, управление возвращается команде, следующей за вызовом этой функции.

Функция **main**. Объявление, определение и вызов функций в C++.

Как уже говорилось, в C++ вся программа – это набор функций. Основная функция в программе (по умолчанию) – **main**:

```
int main () // прототип (заголовок, сигнатура) функции main
{ // тело функции main
// ...
return 0; //возвращаемое функцией main значение
}
```

В прототипе (заголовке, сигнатуре) функции **main** по умолчанию отсутствует список передаваемых параметров, сама функция возвращает целочисленное значение (тип **int**).

В определении функции обязательно присутствует команда возврата значения **return**. Функция **main** как основная не требует явного вызова (вызывается автоматически).

Примечание: оператор **return** можно опускать: а) для функции типа **void**, если возврат из неё происходит перед закрывающей фигурной скобкой, и б) для функции **main**.

Пример определения в C++ функции, вычисляющей площадь круга по его радиусу:

```
double pIKR(double R) // прототип (заголовок) функции
{
    const double PI = 3.14159265;
    return PI*R*R; //возвращаемое функцией значение
}
```

Здесь **PI** – *локальная* (описанная внутри функции) константа. В отличие от локальных, *глобальные* величины (константы и переменные) видны во всех функциях, где не описаны локальные величины с теми же именами.

В дальнейшем можно несколько раз вызвать эту функцию в основной программе с различными значениями аргумента.

В программировании на C++ часто разделяют объявление функции, т.е. задание её заголовка (прототипа, сигнатуры) и определение. Так, в листинге 5.1 функция **pIKR** объявлена до функции **main**, а определена – после неё. Объявить функцию необходимо обязательно до её вызова.

Обычно заголовки всех функций размещают в самом начале файла (перед

основной функцией **main**) или в отдельном *заголовочном файле*.

Как уже было сказано, в заголовке (прототипе, сигнатуре) функции могут использоваться необязательные параметры – модификаторы (спецификаторы). Например, модификатор **inline** – рекомендует компилятору C++ вместо обращения к функции помещать её код непосредственно в точку вызова.

Листинг 5.1.

```
double p1KR(double R);    // прототип (заголовок) функции

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    cout << "ВОПРОС:\n";
    cout << "Когда пиццы будет больше, если:\n";
    cout << "1) взять две радиусом 20 см или\n";
    cout << "2) одну радиусом 30?\n";
    cout << "Введите 1 или 2: ";
    int v;
    cin >> v;
    cout << "Площадь двух пицц радиусом 20 см равна " << 2*p1KR(20) << "см^2\n";
    cout << "Площадь одной пиццы радиусом 30 см равна " << p1KR(30) << "см^2\n";
    cout << "Поэтому Вы ";
    if (v == 2)
        cout << "правы!\n";
    else
        cout << "не правы!\n";

    system("PAUSE");
    return 0;
}

// определение функции
double p1KR(double R)
{
    const double PI = 3.14159265;
    return PI*R*R;    //возвращаемое функцией значение
}
```

Примечание: модификатор **inline** применяется для коротких функций, чтобы снизить накладные расходы на вызов (сохранение и восстановление регистров, передача управления). Использование inline-функций может увеличить объем исполняемой программы.

Полный список модификаторов (спецификаторов) можно узнать в справочниках по языку C++ (например, [5]).

Задание 5.1. Пример создания и использования собственной функции

Реализуйте код листинга 5.1 в новом проекте, проверьте его работу. В заголовок (прототип, сигнатуру) функции добавьте модификатор **inline**.

Параметры функций

Механизм параметров является основным способом обмена информацией между вызываемой и вызывающей функциями. Параметры, перечисленные в

сигнатуре функции, называются **формальными параметрами**, или просто параметрами. Параметры, указанные при вызове функции – **фактические параметры**, или **аргументы**.

При вызове функции в первую очередь вычисляются выражения, стоящие на месте аргументов; затем в стеке выделяется память под формальные параметры функции в соответствии с их типом, и каждому из них присваивается значение соответствующего аргумента. При этом проверяется соответствие типов и при необходимости выполняются их неявные преобразования. При несоответствии типов выдается диагностическое сообщение.

Существует два способа передачи данных в функцию: **по значению** и **по адресу**.

При передаче **по значению** в стек заносятся копии значений аргументов, и операторы в теле функции работают с этими копиями. Доступа к оригинальным значениям параметров у функции нет, а, следовательно, нет и возможности их изменить.

Передача в функцию **plKR** радиуса по значению использовалась в примере с пиццами (листинг 5.1):

```
int main () {  
    //...  
    cout << plKr(30);  
    //...  
}
```

При передаче **по адресу** в стек заносятся адреса значений аргументов (не сами значения), тогда функция осуществляет доступ к ячейкам памяти по этим адресам и может изменить исходные значения аргументов.

Пример – вызов функции **f** с тремя параметрами, второй и третий передаются по адресу:

```
void f(int i, int *j, int &k) {  
    i++; (*j)++; k++;  
}  
int main () {  
    int i=1, j=2, k=3;  
    cout << "i j k\n";  
    cout << i << " " << j << " " << k << "\n";  
    f(i, &j, k);    //второй и третий параметр передаются по адресу  
    cout << i << " " << j << " " << k << "\n";  
    return 0;  
}
```

Результат работы этой программы:

```
i j k
```

1 2 3
1 3 4

В этом примере первый параметр функции **f** – **i** – при вызове функции передается по значению. Его изменение в функции не влияет на исходное значение. Второй параметр **j** передается по адресу с помощью указателя, при этом для передачи в функцию адреса фактического параметра используется операция взятия адреса, а для получения его значения в функции требуется операция разыменования.

Третий параметр **k** передается по адресу с помощью ссылки. При такой передаче в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

Примечание: использование ссылок вместо указателей улучшает читаемость программы, избавляя от необходимости применять операции получения адреса и разыменования.

Использование ссылок вместо передачи по значению также более эффективно, поскольку во время выполнения не требует копирования параметров, что имеет значение при передаче структур данных большого объема.

Если требуется явно запретить изменение параметра внутри функции, используется модификатор **const**, например:

```
int f(const char*);  
char *t(char* a, const int* b);
```

Примечание: по умолчанию параметры любого типа, кроме массива и функции, передаются в функцию по значению.

Задание 5.2. Функция определения простого числа

Самостоятельно напишите программу для вывода на экран первых 100 простых чисел. Для этого задайте логическую функцию для определения простого числа, которую затем вызывайте в основной подпрограмме **main**. В функции используйте код, полученный в работе №2 (листинг 2.3 в редакции задания 2.3.в).

За рамками этого пособия остался механизм *перегрузки функций* (методов), т.е. реализация одной и той же функции (метода) с разным количеством входных параметров. Перегрузка функций в большей степени востребована в ООП.

Передача массивов и строк в функцию

В языке C++ при передаче в функцию *массива* реально передается указатель на его первый элемент (массив всегда передается по адресу). При этом информация о количестве элементов массива теряется, и следует передавать его размерность через отдельный параметр (см. ниже строки 8 и 16 в листинге 5.2 – это

программа нахождения суммы элементов линейного массива).

При передаче **многомерных массивов** в C++ все размерности, если они неизвестны на этапе компиляции, должны передаваться в качестве параметров:

```
int sum (int **a, const int nstr, const int nstb);
```

Один из вариантов представления строки в языке C++ – это массив символов. Пример передачи такой строки в функцию:

```
void showText (char *str) // *str будет указывать на адрес первого символа в строке
{ cout << str << "\n"; }
```

Примечание: в этом примере длина строки при передаче в функцию не указывается, т.к. фактическую длину можно определить по положению нуль-символа.

Листинг 5.2.

```
1 //Передача массива в функцию
2 #include <iostream>
3 #include <cstdlib>
4 #include <Windows.h>
5
6 using namespace std;
7
8 int sum (const int* mas, const int n);
9 int const n = 10;
10
11 int main()
12 {
13     SetConsoleCP(1251);
14     SetConsoleOutputCP(1251);
15     int marks [n] = {2,7,5,0,4};
16     cout << "Сумма элементов массива:" << sum(marks,n) << "\n";
17     system("PAUSE");
18     return 0;
19 }
20
21 int sum (const int* mas, const int n)
22 {
23     int s = 0;
24     for (int i = 0; i < n; i++) s += mas[i];
25     return s;
26 }
```

Задание 5.3. Приёмы создания и использования функций в C++

5.3.а. Реализуйте программу из листинга 5.2, проверьте её работу.

5.3.б. Самостоятельно реализуйте программу для нахождения наибольшего простого числа в квадратной матрице порядка n. Для этого определите в программе функции:

- определения простого числа,
- нахождения наибольшего простого числа в двумерном динамическом массиве,

с) void-функцию (процедуру) вывода двумерного массива в консоль.

5.3.в. Самостоятельно реализуйте средствами C++ программу для вычисления десятичного значения числа по заданной с клавиатуры его шестнадцатеричной записи (определите соответствующую функцию с передачей в неё введённой строки).

Рекурсивные функции

Рекурсивной называется функция, в теле которой имеется обращение к ней самой.

Вспомним программу вычисления факториала числа обычным способом (листинг 2.5 из работы №2):

```
14     int n; //переменная для ввода
15     double f = 1; //переменная для накопления факториала
16     cout << "Введите целое число (>=0) >"; cin >> n;
17     for (int i = 2; i <= n; i++) f = f * i;
18     cout << n << "! = " << f << "\n"; //вывод результата
```

Здесь $n!$ получается как произведение $1*2*\dots*n$.

Но значение факториала можно представить и как $n! = n * (n - 1)!$, т.е. факториал очередного числа легко получить, зная факториал предыдущего. Тогда становится понятным рекурсивный подход к вычислению факториала (см. листинг 5.3).

Листинг 5.3.

```
long double fact(int n)
{
    if (n < 0) // если пользователь ввел отрицательное число
        return 0; // возвращаем ноль
    if (n == 0) // если пользователь ввел ноль,
        return 1; // возвращаем 1
    else // Во всех остальных случаях
        return n * fact(n - 1); // делаем рекурсию.
}
```

Любая функция может вызываться рекурсивно. При этом в стеке выделяется новый участок памяти (стековый кадр) для размещения копий аргументов, а также адреса возврата, поэтому предыдущее состояние выполняемой функции сохраняется и к нему впоследствии можно вернуться.

Преимуществом использования рекурсии является лаконичность и хорошая читаемость кода программы.

Но у рекурсии есть и недостатки: программу труднее отлаживать (требуется контролировать глубину рекурсии), при большой глубине стек может переполниться; также накапливаются расходы времени на каждый рекурсивный вызов функции. Рекурсию следует применять с осторожностью.

Задание 5.4. Приёмы работы с рекурсивными функциями

5.4.а. Реализуйте программу вычисления факториала с помощью рекурсивной

функции, проверьте её работу.

5.4.6. Самостоятельно реализуйте рекурсивный алгоритм Хоара быстрой сортировки массива целых чисел [7, с. 95]. Объясните работу алгоритма Хоара.

Индивидуальное задание №5

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №5** на подпрограммы. Снабдите вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата.

Примечание: в задачнике [3] термин «процедура» используется в смысле «подпрограмма».

Протестируйте работу программы на нескольких наборах входных данных.

Вопросы для самоподготовки

- ✓ Можно ли объявить основную подпрограмму **main** в C++ по типу процедуры?
- ✓ Какие параметры можно передавать в функцию **main**?
- ✓ Куда возвращает значение функция **main**?
- ✓ В чём отличие формальных и фактических параметров?
- ✓ Назовите способы передачи данных в подпрограмму.
- ✓ Назовите преимущества и недостатки использования рекурсии.
- ✓ Средняя сложность алгоритма сортировки Хоара $O(n \cdot \log n)$. Что это значит?
- ✓ Что такое заголовочный файл? Где определяется всё, объявленное в заголовочном файле?
- ✓ Что такое линкинг?

6. ФАЙЛЫ

Цель: познакомиться с особенностями реализации в языке C++ файлового ввода-вывода.

Ввод-вывод: понятие

Ввод-вывод (англ. input/output, I/O) – это информационное взаимодействие компьютера с внешним миром (человеком или другим компьютером по сети).

Ввод – получение данных компьютером, **вывод** – передача данных от компьютера.

Часто под «компьютером» понимают связку «CPU – основная память», тогда всё остальное – это периферия. В этом контексте любой обмен информацией между этой связкой и любым другим устройством можно рассматривать как ввод-вывод. Например, обмен данными между «компьютером» и любым внешним накопителем – это тоже ввод-вывод.

Современные операционные системы реализуют абстракцию внешней памяти в виде **файловой системы**. Здесь **файл** – это именованная конечная порция информации на внешнем носителе (жестком, гибком, оптическом диске, флэш-карте, USB-флэш-накопителе, твердотельном, ленточном накопителе).

С точки зрения приема и передачи информации файл представляется как конечная последовательность байт.

Примечание: это представление справедливо и для многих устройств ввода-вывода (дисплей, клавиатура, принтер, параллельный и последовательный порты). Доступ к таким устройствам операционная система позволяет осуществлять как к специализированным файлам.

Передача информации из файла называется **чтением** (вводом), в файл – **записью** (выводом).

Поток (stream) – это абстрактное понятие в современных языках высокого уровня, обозначающее любой процесс последовательного переноса данных от источника к приёмнику. Поток не зависит от конкретного источника или приемника (файла или устройства), т.е. потоковый ввод-вывод – это обобщённый механизм.

Любая операция чтения из файла или записи в файл может быть представлена как поток байт (из файла или в файл).

Основные операции при работе с файлом:

1. Создание потока и связывание его с файлом, расположенном на физическом носителе (открытие потока).
2. Обмен (ввод-вывод).

3. Заккрытие файла.

При открытии потока с ним связывается область оперативной памяти, называемая **буфером**. При выводе вся информация направляется в буфер и накапливается там до заполнения буфера или до закрытия потока. Буферизация позволяет более быстро и эффективно обмениваться информацией с внешними устройствами.

Примечание: при аварийном завершении программы или выходе из программы без закрытия потока выходной буфер может быть не выгружен на внешнее устройство, и возможна потеря данных.

Работа с файлами в C++

Файловый ввод/вывод в C++ реализуется либо а) с помощью функций, унаследованных от языка C, либо б) с помощью **потоков** C++.

Каждый способ имеет свои преимущества. Одно из преимуществ использования потоков C++ в том, что они легче в использовании в простых случаях ввода/вывода, не требующих форматирования. Но использование C-функций позволяет использовать миллионы строк кода, уже написанных на языке C.

В данной работе рассмотрим функции файлового ввода-вывода в стиле языка C++. Для использования функций ввода-вывода необходимо подключить к программе заголовочный файл <fstream> (см. листинг 6.1).

Работа с потоком начинается с его **открытия**. Поток можно открыть для чтения и/или записи в **двоичном** или **текстовом** режиме. Открытие потока включает в себя А) создание объекта и Б) связывание с ним определённого файла (открытие). Пример открытия потока **для чтения**:

```
ifstream fIn; // А
```

```
fIn.open (fName); //Б
```

Здесь fIn – экземпляр класса ifstream, реализующий поток входных данных, fName – строковое значение, содержащее полный путь к файлу.

Связать файл с объектом можно и прямо в конструкторе:

```
ifstream fIn (fName); //А и Б
```

Например:

```
ifstream fIn;
```

```
fIn.open ("d:\\temp\\proba.txt");
```

Или то же самое:

```
ifstream fIn ("d:\\temp\\proba.txt");
```

Примечание: символ слэш \ является специальным символом в коде C++, поэтому при использовании слэша в строке (в качестве разделителя в пути к файлу) его необходимо дублировать.

При указании имени файла без пути, программа на этапе исполнения станет искать файл в папке проекта (в папке с исходным кодом).

Есть причины, по которым файл может не открыться, что сделает невозможным дальнейшую работу программы. Поэтому имеет смысл сделать проверку – открылся ли файл. Можно использовать имя потока в логическом условии (если файл открыт, то вернётся непустое значение, интерпретируемое как true, иначе вернётся NULL, понимаемое как false):

```
if ( fIn )           //то же, что и if ( fIn !=NULL )
{
    // файловые операции
}
else //сообщение об ошибке
```

Можно также использовать метод is_open().

После успешного открытия, данные из файла можно считать. Считывание из текстового файла можно организовать оператором >>, который указывает в какую переменную будет произведено считывание, например:

```
double d;
int i;
string s;
fIn >> d >> i >> s;
```

Считать можно вещественные, целые числа и строковые значения. Считывание очередного значения закончится, если появится пробел или конец строки.

Считывание целой строки до перевода каретки производится (так же как и в iostream) методом getline(). Если считывается в строку типа string, то рекомендуется использовать переопределённую версию getline() в виде *функции*. Если считать нужно в массив символов char[], то используются либо get(), либо getline() именно как *методы* объекта потока.

Считывание из бинарного файла лучше производить с помощью метода read() (хотя текстовый файл – это частный случай бинарного).

Операции ввода-вывода выполняются, начиная с текущей **позиции потока**, определяемой положением **указателя потока**. Указатель устанавливается при открытии на начало или конец файла (в соответствии с режимом открытия) и изменяется автоматически после каждой операции ввода-вывода.

Поток закрывается либо при завершении программы, либо явным образом с помощью метода close():

```
fIn.close();
```

Перед закрытием потока данные из буферов выгружаются на диск. Рекомендуется **всегда явным образом закрывать потоки**, открытые для записи, чтобы избежать потери данных.

Среди полезных методов для работы с файлами назовём следующие:

- eof() – проверяет, не достигнут ли конец файла;
- seekg() – позиционирование в файле (абсолютное – относительно начала или конца файла и относительное – сдвиг на заданное количество байт относительно текущей позиции);
- tellg() – возвращает информацию, сколько байт уже прочитано.

Открытие потока *для записи* (в конструкторе):

```
ofstream fOut ("c:\\RML\\output.txt");
```

Выгрузку данных в текстовый файл можно организовать оператором <<, который указывает, из какой переменной будет произведена запись, например:

```
int a=5, b=7;
```

```
fOut << a << "+" << b << "=" << a + b;
```

```
fOut.close();
```

В листинге 6.1 приведён код программы, которая суммирует целые числа из входного файла и записывает результат сложения в выходной файл. В этом коде приведены основные операции с файлами, которые были описаны выше.

Листинг 6.1.

```
1 //Файловый ввод-вывод
2 #include <cstdlib>
3 #include <iostream>
4 #include <Windows.h>
5 #include <fstream> //для работы с файлами
6
7 using namespace std;
8
9 int main() {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12
13     ifstream fIn; //объект - входной поток данных
14     ofstream fOut; //объект - выходной поток данных
15     int x, count = 0;
16     fIn.open("input.txt"); //открытие входного потока
17     if (fIn) {
18         while (fIn >> x)
19             count += x;
20         fIn.close();
21         fOut.open("output.txt"); //открытие выходного потока
22         fOut << "Result: " << count;
23         fOut.close();
24         cout << "Данные сохранены в файле 'output.txt'.\n";
25     }
26     else
27         cout << "Не удалось открыть файл 'input.txt'.";
28     system("PAUSE");
29     return 0;
30 }
```

Задание 6.1. Пример простого файлового ввода-вывода в C++.

6.1.а. Создайте в Блокноте текстовый файл **fio.txt**, **запишите** в него свои фамилию, имя и отчество. **Реализуйте** программу вывода содержимого этого файла на

экран консоли. Добейтесь корректного отображения кириллических символов. Отредактируйте код, чтобы содержимое файла **fiо.txt** при запуске вводилось с клавиатуры.

6.1.б. Создайте программу, генерирующую числовой файл, заполненный 25 случайными значениями.

6.1.в. На основе кода листинга 6.1 реализуйте программу, которая формирует выходной файл с числовыми значениями, вдвое большими чисел входного файла.

Битовое представление чисел и битовые операции

При работе с битовыми представлениями чисел можно использовать **битовые операции**, определённые в языке C++ (см. табл. 6).

Таблица 6. Битовые операции

$x \ll n$	Сдвиг влево двоичного кода (умножение на 2^n)	char x=7; x=x<<2 //00000111<<2=00011100
$x \gg n$	Сдвиг вправо двоичного кода (деление на 2^n)	char x=7; x=x>>1 //00000111>>1=00000011
$x \& \text{mask}$	Поразрядное И (запись в бит 0)	//111&100=100 char x=0xFFFFFFFF, mask=0x1F; x=x & mask; //=0x0000001F
$x \text{mask}$	Поразрядное ИЛИ (запись в бит 1)	//111 100=111 char x=0xFFFFF00, mask=0x1F; x=x mask; //=0xFFFFF1F
$x \wedge \text{mask}$	Исключающее ИЛИ (поразрядное)	char x=0x1F, mask=1; x=x ^ mask; //1111^0001=1110
$\sim x$	Инверсия	char x=0x0F; x=~x; //=0xF0

Пример – как установить пятый бит целого числа в 0:

```
unsigned char x=255;           //8-разрядное число
unsigned char mask =0x01;      //1=00000001 – 8-разрядная маска
x = x & (~ (mask<<4));        //результат 239
```

Задание 6.2. Приёмы использования битовых операций.

6.3.а. Реализуйте по аналогии с вышеприведённым примером решение установки седьмого бита числа в 1.

6.3.б. Реализуйте код листинга 6.2, объясните полученный результат.

```

1  //Битовые операции
2  #include <cstdlib>
3  #include <iostream>
4  #include <Windows.h>
5  #include <bitset>
6  using namespace std;
7
8  int main()
9  {
10     SetConsoleCP(1251);
11     SetConsoleOutputCP(1251);
12
13     unsigned int x = 25;
14     const int n = sizeof(int)*8; //32 - количество разрядов в числе типа int
15     unsigned maska = (1 << n - 1); //1 в старшем бите 32-разрядной сетки
16     cout << "Начальный вид маски: " << bitset<n>(maska) << endl;
17     cout << "Результат: ";
18     for (int i = 1; i <= n; i++) //32 раза - по количеству разрядов:
19     {
20         cout << ((x & maska) >> (n - i));
21         maska = maska >> 1; //смещение 1 в маске на разряд вправо
22     }
23     cout << endl;
24     system("pause");
25     return 0;
26 }

```

Задание 6.3. Задача сортировки числового файла с помощью битового массива

Пусть даны не более 24 чисел со значениями в диапазоне от 0 до 23, например, {1, 20, 5, 9, 12, 8}. Подобный набор чисел удобно представить 24-разрядной битовой последовательностью. В ней единичные биты (флаги) отражают наличие в исходном наборе числа, равного номеру этого бита в последовательности.

В нашем примере получим следующую битовую последовательность: 010001001100100000001000. Последовательное считывание бит этой последовательности и их отображение в соответствующее число позволит естественным образом получить исходный набор чисел *в отсортированном виде* – {1, 5, 8, 9, 12, 20}.

На этой идее можно реализовать сортировку большого объёма числовых данных во внешней памяти (внешнюю сортировку). Достаточно один раз считать содержимое файла, заполнить при этом в памяти ЭВМ битовый массив и на его основе быстро сформировать новое содержимое файла в уже отсортированном виде.

Постановка задачи:

Входные данные: файл, содержащий до $n=10^6$ неотрицательных целых чисел, среди них нет повторяющихся.

Результат: упорядоченный по возрастанию список чисел в выходном файле.

Максимальный объём используемой оперативной памяти: 1 Мб.

Время работы программы: ~10 с.

При использовании битового массива для представления сортируемых чисел, программу можно представить как последовательность из трёх подзадач:

- а) Инициализация битового массива нулевыми значениями.
- б) Считывание из файла с установкой в единицу тех битов, индексы которых равны входным числам.
- в) Формирование упорядоченного выходного файла путём последовательной проверки битов массива с выводом в файл индексов тех, которые установлены в единицу.

6.4. Реализуйте задачу сортировки файла с заданными условиями. **Добавьте** в код возможность определения времени работы программы. **Подсчитайте** объём оперативной памяти, занимаемый битовым массивом.

Примечание: битовый массив можно реализовать через массив беззнаковых целых чисел (типа char, int или long long int), вектором логических значений (vector<bool>) или экземпляром класса bitset.

Индивидуальное задание № 6

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №6** на файловый ввод-вывод. **Снабдите** вывод программы (интерфейс пользователя) описанием задачи, вводимых данных и результата. Программа должна корректно работать с кириллическими символами в файле.

Вопросы для самоподготовки

- ✓ Что такое файл? Что такое каталог (папка, директория)?
- ✓ Какая структура реализуется в современных файловых системах?
- ✓ Что такое последовательный и произвольный доступ к файлу?
- ✓ Что такое поток?
- ✓ Когда разрывается связь потока с файлом? Какая команда это делает?
- ✓ Можно ли сбросить данные из буфера на диск без закрытия файла?
- ✓ Почему при открытии двоичного файла в текстовом редакторе, как правило, отображается хаотичный набор случайных символов?
- ✓ Как вернуться в начало файла после считывания его части?
- ✓ Что такое внешние сортировки? В чём их особенность?

7. ОКОННЫЙ ИНТЕРФЕЙС

Цель: познакомиться с особенностями реализации средствами C++ приложений с оконным интерфейсом.

Создание нового проекта Windows Forms в среде Visual Studio

Для среды **Visual Studio** версий 2017 и более поздних:

А) С помощью программы **Visual Studio Installer** в рабочей нагрузке «Разработка классических приложений на C++» проверьте, что в перечне её компонентов отмечена «Поддержка C++/CLI» (иначе доустановите этот компонент).

Б) В окне запуска **Visual Studio** выберите **Создание проекта**. В открывшемся окне в поле «Поиск шаблонов» наберите **CLR**, затем в открывшемся перечне выберите «Пустой проект CLR (.NET Framework)». В следующем окне «Настроить новый проект» укажите его имя и расположение.

Для среды **Visual Studio** версий 2015 и более ранних:

Откройте диалоговое окно создания нового проекта (**Файл** → **Создать проект**), в разделе **Установленные** → **Шаблоны** → **Visual C++** → **CLR** выберите «Пустой проект CLR», задайте имя проекта и нажмите «ОК» (рис. 17).

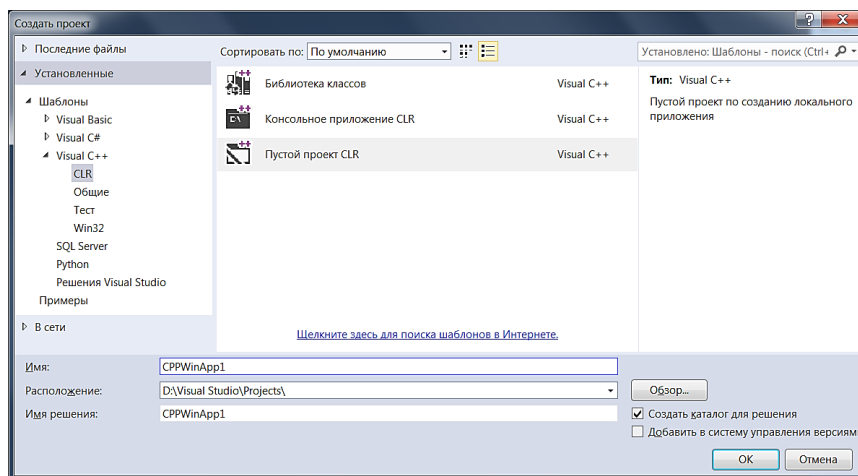


Рис. 17. Диалоговое окно создания нового проекта.

CLR (Common Language Runtime – общезыковая исполняющая среда) – среда исполнения *байт-кода CIL*, получающегося в результате компиляции исходного кода программы, написанного на .NET-совместимых языках программирования (в т.ч. Visual C++). CLR является одним из основных компонентов программной платформы Microsoft .NET Framework для исполнения *управляемого кода*. CLR компилирует байт-код приложения во время его исполнения в *машинный код* (native code).

Примечание: подробнее про разработку CLR-проектов Windows-приложений

на языке Visual C++ см., например, [4, 10].

В обозревателе решений сделайте правый щелчок по созданному **проекту**. В открывшемся контекстном меню выберите **Добавить** → **Создать элемент** и в открывшемся меню в разделе **Visual C++** → **Среда CLR** (или **UI** в более старых версиях) выберите **Форма Windows Forms**.

Это действие позволяет добавить в проект пользовательский интерфейс оконных форм (Windows Forms).

Обратите внимание на поле **Имя**. Щёлкните **Добавить** – откроется вкладка с новой формой.

Примечание: когда форма будет добавлена, возможен вывод сообщения об ошибке «Данные, необходимые для завершения этой операции еще не доступны. (Исключение из HRESULT: 0x8000000A)». При этом достаточно закрыть решение (меню **Файл** → **Закрыть решение**) и открыть его заново.

В обозревателе решений двойным щелчком откройте файл исходного кода **MyForm.cpp**. Откроется новая вкладка с единственной строчкой кода в нём:

```
#include "MyForm.h"
```

В этот файл добавьте следующий код из листинга 7.1 (строки 2-18). Обратите внимание, что в строке 15 нужно указать имя **вашего** проекта (в примере – **CPPWinApp1**) и **вашей** формы (в примере – **MyForm**).

Листинг 7.1.

```
1  #include "MyForm.h"
2  //необходимые адресные пространства:
3  using namespace System;
4  using namespace System::Windows::Forms;
5  [STAThreadAttribute] //STA - Single Thread Apartment, т.е. это
6  //атрибут, задающий для точки входа однопоточную модель выполнения
7  int main(array<String^>^ args) //передача в главную функцию
8  //CLR-массива строковых аргументов (отслеживаемых дескрипторов в CLR)
9  {
10     Application::EnableVisualStyles(); //включаем визуальные стили
11     //метод, задающий технологию визуализации графического интерфейса, когда
12     //элементы управления используют класс GDI, основанный на TextRenderer:
13     Application::SetCompatibleTextRenderingDefault(false);
14     //запуск приложения с заданной формой в качестве главного окна:
15     CPPWinApp1::MyForm form; Application::Run(%form);
16
17     return 0; //нормальное завершение приложения
18 }
```

В обозревателе решений в свойствах проекта выберите **Компоновщик** → **Система**, далее в поле **Подсистема** из выпадающего меню выберите **Windows (/SUBSYSTEM:WINDOWS)** и нажмите **Применить**.

Не закрывая окно свойств проекта, перейдите в раздел **Компоновщик** → **Дополнительно** и в поле **Точка входа** напишите **main** (это имя главной функции), далее нажмите **ОК** (рис. 18).

Начальные настройки проекта Windows-приложения закончены. Для

редактирования внешнего вида формы, нужно перейти во вкладку **MyForm.h [Конструктор]** или кликнуть дважды по файлу **MyForm.h** в обозревателе решений.

Обратите внимание, что программный модуль к созданной форме **MyForm.h** открывается в двух режимах: **редактирования кода** и **визуального проектирования формы**.

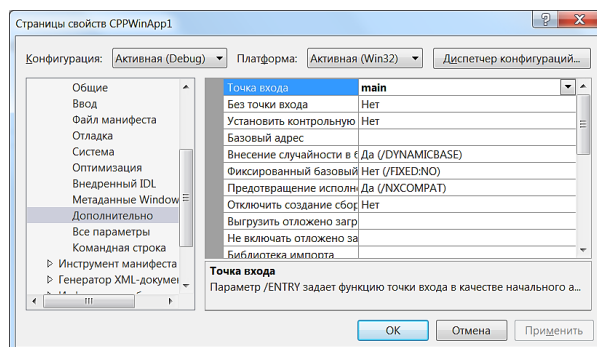


Рис. 18. Диалоговое окно свойств проекта.

Запустите программу. Завершите выполнение программы, закрыв окно формы.

Примечание: рекомендуется сделать резервную копию папки этого проекта (сохранив его перед этим) для использования в следующих заданиях в качестве уже готового шаблона.

Этапы реализации Windows-приложения

Процесс создания Windows-приложения включает два основных этапа:

- 1) разработка формы (форм) – **визуальное проектирование**;
- 2) программирование **поведения** приложения (создание процедур обработки событий).

На первом этапе, когда форма добавлена в проект, в нее в соответствии с имеющимся алгоритмом помещают необходимые элементы управления (кнопки, флажки, переключатели, поля ввода и пр.) и придают свойствам элементов необходимые значения.

На втором этапе определяют реакции на события (поведение) компонентов. Реакции задаются в программах, которые называются обработчиками событий. Все программы-обработчики событий элементов управления, расположенных на данной форме, помещаются в тот же программный модуль, который создается вместе с добавлением формы (MyForm.h, например).

Один из способов размещения элементов управления на форме – с помощью панели элементов (**Просмотр** → **Панель элементов**). Приемы размещения:

- двойной щелчок на элементе в списке, затем перетащить появившийся элемент в нужное место на форме, придать ему необходимый размер;

- один щелчок на элементе в списке, затем второй щелчок на нужном месте формы с одновременным приданием элементу необходимого размера.

Настройка элементов управления на форме (задание свойств) осуществляется с помощью окна свойств (рис. 19). Свойства отображаются либо в алфавитном порядке, либо по категориям (выбирается на панели инструментов окна свойств).

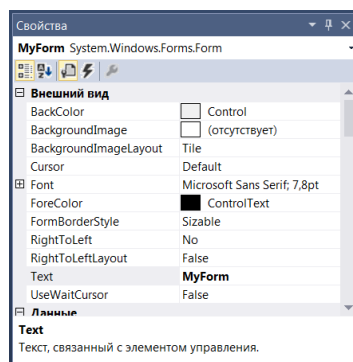


Рис. 19. Окно свойств

Свойства элементов задаются выбором из готовых вариантов или заданием значения с клавиатуры.

Определение **поведения программы** начинают с принятия решений, для каких элементов на какие события какие действия, реализующие функциональность, должны выполняться.

Вкладка **События** (Events) в окне свойств (рис. 20) содержит список возможных событий, которые могут происходить с выделенным на форме (активным) элементом. Она позволяет связывать каждое событие с программой-обработчиком этого события.

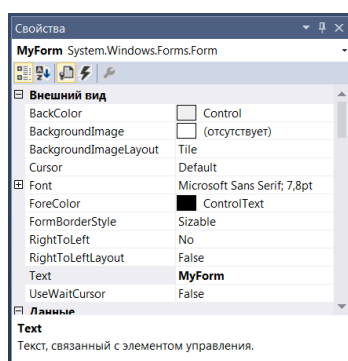


Рис. 20. Вкладка События в окне свойств.

Из списка выбирается то событие, которое требуется обработать. Например, нажатие кнопки, помещенной на форму (т.е. щелчок мыши на кнопке). После определения события следует дважды щелкнуть мышью в поле рядом с именем события.

При этом инструментальная среда создаст в программе формы, в которую

помещен компонент, обработчик этого события. Это будет функция с заголовочной частью, но с пустым телом — не программа, а заготовка программы. В это пустое тело заготовки записываются свои команды, которые будут определять реакцию элемента на данное событие (поведение) с учетом передаваемых функции фактических значений ее параметров.

Например, для элемента «кнопка» заготовка обработчика события **Click** будет выглядеть так:

```
91 | #pragma endregion
92 | private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
93 | }
```

В качестве параметров обработчику передается объект-источник события и тип события. Имя обработчика (в нашем примере **button1_Click**) формируется автоматически из имени объекта и имени события.

Для каждого типа элементов формы определен свой набор событий. Наиболее часто используемые события:

- Activated – получение формой фокуса ввода;
- Click, DoubleClick – одинарный и двойной щелчок мыши;
- Closed – закрытие формы;
- Load – загрузка формы;
- KeyDown, KeyUp – нажатие и отпускание любой клавиши и их сочетаний;
- KeyPress – нажатие клавиши с ASCII-кодом;
- MouseDown, MouseUp – нажатие и отпускание кнопки мыши;
- MouseMove – движение мыши;
- Paint – при прорисовке формы.

Работать с созданной формой (в нашем примере **MyForm.h**) можно в двух режимах: редактирования кода и визуального проектирования формы.

Задание 7.1. Создание простого Windows-приложения

Разместите на созданной в предыдущем задании форме текстовое поле и кнопку (рис. 21).

Для обоих элементов настройте свойство **Anchor** в значение **None** (щелчками убрать **Top** и **Left**), затем с помощью меню **Формат** → **По центру формы** выровняйте их расположение по горизонтали.

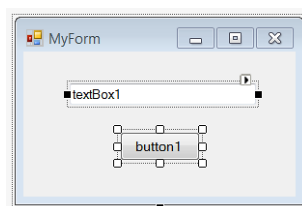


Рис. 21. Простой пример формы с двумя элементами управления.

Исправьте надпись на кнопке (свойство **Text**) на **Выход**, а заголовок формы – на **Клавиши и коды**.

Запустите программу, попробуйте изменить расположение и размер формы.

Сформируйте пустые обработчики событий – **Click** для кнопки и **KeyPress** для текстового поля.

Добавьте в обработчик **Click** для кнопки команду выхода из программы:

```
Application::Exit();
```

Добавьте в обработчик **KeyPress** для текстового поля вывод текста в *окно сообщений*:

```
MessageBox::Show("Нажата клавиша: " + e->KeyChar, "Клавиша",  
                MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
```

Здесь в окне сообщений задаётся заголовок «Клавиша», сообщение, что нажата определенная клавиша, кнопка закрытия «ОК» и изображение.

Проверьте работу обработчика события **KeyPress**.

Самостоятельно исправьте программу так, чтобы в окне сообщений выводился не сам символ, а его код. Соответственно отредактируйте надписи в окне.

Элементы управления на форме: метка, кнопка, поле ввода

Метка Label

Элемент для размещения текста на форме (см. рис. 22). Сам текст хранится в свойстве **Text**. Можно задавать шрифт, цвет текста, цвет фона и выравнивание. Можно разместить на метке изображение и задать прозрачность.

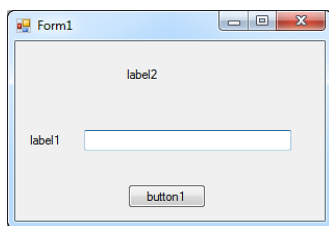


Рис. 22. Элементы управления Label (метки) на форме.

Кнопка Button

Кнопка уже была использована в первом примере (рис. 21). Основное *событие* для кнопки – щелчок мышью. Другие возможные события – нажатие клавиш на клавиатуре и мыши, изменение параметров и пр. Если кнопка имеет *фокус ввода*, нажатие пробела или <Enter> равносильно щелчку мыши. Можно изменять текст, шрифт, цвет фона и текста, добавить фоновое изображение и пр.

Примечание: кнопка по умолчанию, если в свойстве формы **AcceptButton** прописано имя кнопки – нажатие клавиши <Enter> вызовет событие **Click**, даже если кнопка не имеет фокуса ввода.

Кнопки часто используются в окнах – обычных и диалоговых (обладающих свойством модальности). В диалоговых окнах размещают кнопки, по которым окно может быть закрыто. Для кнопок имеется свойство **DialogResult** и одноименное перечисление возможных значений, позволяющее идентифицировать, как было закрыто окно (табл. 7).

Таблица 7. Перечисление DialogResult

Значение	Описание	Значение	Описание
None	Окно не закрывается	Ignore	Нажата кнопка Ignore
OK	Нажата кнопка OK	Yes	Нажата кнопка Yes
Cancel	Нажата кнопка Cancel	No	Нажата кнопка No
Abort	Нажата кнопка Abort	Retry	Нажата кнопка Retry

Поле ввода TextBox

Данный компонент (см. рис. 21) позволяет пользователю вводить текст, редактировать, защищать вводимый текст маской и статусом только для чтения. Имеется возможность многострочного ввода.

Для элемента реализованы методы очистки, выделения, копирования в буфер, вставки из буфера. Элемент реагирует на множество событий, например, **KeyPress**.

Задание 7.2. Игра «Угадай число!»

Формулировка задачи¹¹: в качестве примера использования этих элементов управления рассмотрим игру «Угадай число!» (рис. 23, справа), когда пользователь вводит число в текстовое поле (нажатием клавиши <Enter>), пока не угадает. Тогда выводится итоговое сообщение, фокус передается кнопке «Еще раз!», по итогам выводится «Коэффициент невезучести» как отношение числа попыток к максимальному значению в диапазоне.

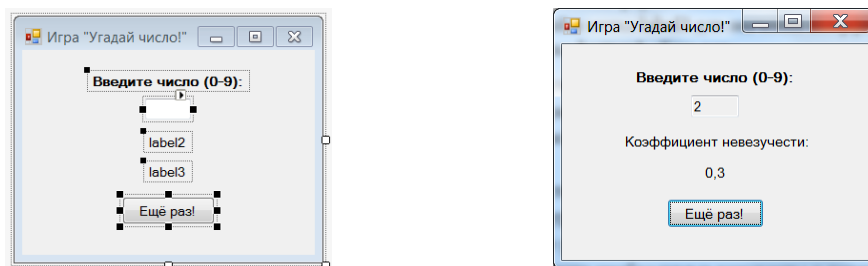


Рис. 23. Окно приложения: в конструкторе форм (слева), во время работы (справа).

Состав элементов формы: метка **label1** «Введите число (0-9):», метка **label2** для служебных сообщений и метка **label3** для значения коэффициента; текстовое поле **textBox** для ввода числа; кнопка **button** «Ещё раз!».

¹¹ В основу положен адаптированный пример из книги Павловская Т.А. С#. Программирование на языке высокого уровня: Учебник для вузов, 2014. – С. 327.

Реализуйте в новом CLR-проекте Windows-приложения форму с перечисленными выше элементами (см. рис. 23, слева), выровняйте их в форме по горизонтали. Определите соответствующие значения свойства **Text** элементов формы и самой формы (для меток **label2** и **label3** – пустая строка).

Установите свойство **Visible** для кнопки в значение **False**.

Создайте процедуру-обработчик события **Load** создания (загрузки) формы **MyForm**. В этом обработчике пропишите задание случайного числа (строки 70-71 листинга 7.2), а до обработчика объявите необходимые константы и переменные (строки 67-68).

Не забудьте подключить необходимые заголовочные файлы (строки 1-2).

Листинг 7.2. Файл MyForm.h

```
1  #include <cstdlib>
2  #include <ctime>

...

66  #pragma endregion
67  int const max = 10; //максимальное количество попыток
68  int i, k = 1; //загаданное число и счетчик попыток
69  private: System::Void MyForm_Load(System::Object^ sender, System::EventArgs^ e) {
70      srand(time(NULL)); //инициализация ДСЧ
71      i = rand()%max; //"загадывание" числа
72  }
```

Создайте процедуру обработки события **KeyPress** для элемента **textBox1**. Реализуйте в теле процедуры код реагирования на ввод числа (см. листинг 7.3).

Листинг 7.3.

```
139  private: System::Void textBox1_KeyPress(System::Object^ sender,
140      System::Windows::Forms::KeyPressEventArgs^ e) {
141      int n; //вводимое игроком число
142      if ((int)e->KeyChar == 13) { //если был нажат Enter
143          n = Convert::ToInt16(textBox1->Text); //число из текстового поля (с преобразованием)
144          if (n - i != 0) {
145              label2->Left += 5;
146              label2->Text = "Не угадали!";
147              textBox1->Clear();
148              k++; //+1 в счетчик попыток
149          }
150          else {
151              label2->Left = 60; //восстанавливаем положение метки
152              label2->Text = "Коэффициент невезучести:";
153              double koef = 1.0*k / max;
154              label3->Text = koef.ToString();
155              textBox1->ReadOnly = true;
156              button1->Visible = true;
157              button1->Focus();
158          }
159      }
160  }
```

Создайте процедуру обработки события **Click** для элемента **button1**. Реализуйте в теле процедуры код в соответствии с листингом 7.4.


```

161 private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
162     i = rand() % max; //"загадывание" числа
163     k = 1;
164     textBox1->Clear(); //очистка поля ввода
165     textBox1->Focus(); //установка фокуса ввода
166     label2->Text = ""; //очистка служебных меток
167     label3->Text = "";
168     textBox1->ReadOnly = false;
169     button1->Visible = false;
170 }

```

Проверьте работу созданной программы. При необходимости измените на форме положение меток относительно левого края формы, чтобы выводимый текст располагался по центру.

Самостоятельно внесите изменения в программу так, чтобы кнопка «Ещё раз» была видна, но неактивна, пока пользователь не отгадает число.

Самостоятельно внесите изменения в программу, чтобы допустимое число попыток реально не превышало значения, заданного в переменной **max**, после чего пользователю выводилось бы сообщение о проигрыше.

Элементы управления на форме: меню, флажок, переключатель, панель группировки, список

Меню (MenuStrip, ContextMenuStrip)

Главное меню при добавлении на форму располагается под заголовком окна, среда переходит в режим редактирования пунктов меню. Пункты меню – отдельные настраиваемые объекты.

Пункт меню может быть запрещен/разрешен, видим/невидим, выбран/не выбран. Пример обработчика для пункта меню **Exit** см. в листинге 7.5.

```

125 private: System::Void exitToolStripMenuItem_Click(System::Object^ sender,
126                                                    System::EventArgs^ e) {
127     Application::Exit();
128 }

```

Контекстное меню – его состав команд зависит от того объекта, для которого оно было вызвано.

Флажок CheckBox

Флажок используется включения какого-либо режима, или для выбора одного или нескольких значений из множества. Флажок может быть установлен, не установлен, не полностью установлен. Флажки в группе устанавливаются независимо друг от друга.

Флажок может быть в виде собственно флажка или в виде «залипающей» кнопки.

Переключатель RadioButton

Переключатель позволяет выбрать только один из нескольких вариантов в группе. Один переключатель в группе устанавливается, остальные – сбрасываются.

Как и флажок, переключатель может быть в традиционном виде или в виде «залипающей» кнопки.

Панель группировки GroupBox

Панель группировки служит для группирования (визуального выделения) элементов на форме. Например, если на форме необходимо разместить несколько групп переключателей, то их помещают внутри элемента **GroupBox** (рис. 24).

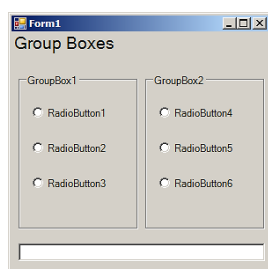


Рис. 24. Переключатели и панель группировки.

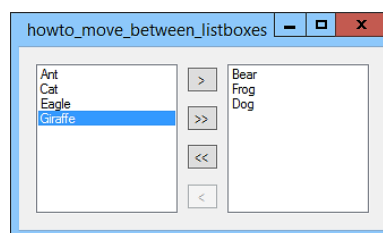


Рис. 25. Пример списка.

Список ListBox

Список позволяет представить перечни элементов, в которых пользователь может выбрать одно или несколько значений (рис. 25). Чаще всего используются списки строк, но можно выводить и изображения.

Список может быть отсортированным, состоять из нескольких столбцов. Элементы списка нумеруются с нуля и объединяются в коллекцию. Элементы при выполнении программы можно добавлять и удалять.

Задание 7.3. Практические приёмы использования управляющих элементов формы

Описание задачи: программа (по кнопке) создает массив случайных чисел заданного (в поле) пользователем размера (рис. 26). Пользователь может выбрать – сортировать массив или нет (переключателями, объединенными в панели группировки; по умолчанию выбрано не сортировать). После формирования массива в нём (по кнопке) можно вычислить максимальный элемент и/или (выбор флажком) количество положительных элементов (значения выводятся в поля только для чтения).

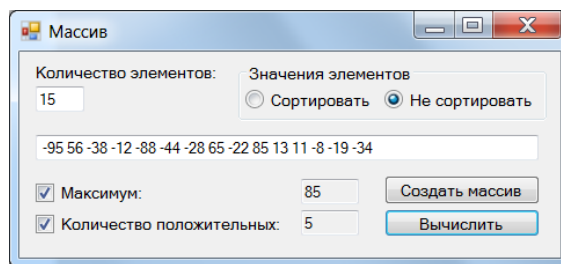


Рис. 26. Пример работы приложения «Массив».

Создайте новый CLR-проект Windows-приложения, разместите на форме и настройте необходимые элементы в соответствии с рис. 26 и листингом 7.6 (расположение на форме и размер, имена – свойство **Name**, надписи – свойство **Text**). Настройте, чтобы по умолчанию был выбран переключатель «Не сортировать». Текстовые поля **maxtextBox** и **numPositttextBox** сделайте только для чтения (Read only).

Листинг 7.6.

```

61 void InitializeComponent(void)
62 {
63     this->label1 = (gcnew System::Windows::Forms::Label());
64     this->numtextBox = (gcnew System::Windows::Forms::TextBox());
65     this->radioButton1 = (gcnew System::Windows::Forms::RadioButton());
66     this->radioButton2 = (gcnew System::Windows::Forms::RadioButton());
67     this->groupBox1 = (gcnew System::Windows::Forms::GroupBox());
68     this->arraytextBox = (gcnew System::Windows::Forms::TextBox());
69     this->maxcheckBox = (gcnew System::Windows::Forms::CheckBox());
70     this->numPositchcheckBox = (gcnew System::Windows::Forms::CheckBox());
71     this->maxtextBox = (gcnew System::Windows::Forms::TextBox());
72     this->numPositttextBox = (gcnew System::Windows::Forms::TextBox());
73     this->createbutton = (gcnew System::Windows::Forms::Button());
74     this->calcbutton = (gcnew System::Windows::Forms::Button());

```

Создайте процедуру обработки события **Click** для элемента **createbutton**. Перед обработчиком объявите исходный массив **arr** и процедуру **sort** сортировки массива (строки 215-216 листинга 7.7). Реализуйте в теле обработчика формирование массива заданной пользователем длины (строки 219-243 листинга 7.7).

Листинг 7.7. К программе «Массив» (часть 1).

```

214 #pragma endregion
215 int n; int* arr; //объявление массива
216 void sort(int* m, const int n) {} //объявление процедуры сортировки массива
217 private: System::Void createbutton_Click(System::Object^ sender,
218                                     System::EventArgs^ e) {
219     try
220     {
221         n = Convert::ToInt16(numtextBox->Text);
222     }
223     catch (Exception^ ex)
224     {
225         MessageBox::Show("Введите число!", "Ошибка",
226                             MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
227         numtextBox->Clear();
228         numtextBox->Focus();
229     }
230     arraytextBox->Clear(); //очистка текстовых полей
231     maxtextBox->Clear();
232     numPosittextBox->Clear();
233     if (n > 0) {
234         arr = new int[n]; //создание массива
235         for (int i = 0; i < n; i++) { //заполнение случайными числами
236             arr[i] = rand() % 201 - 100;
237         }
238         if (radioButton1->Checked) {sort(arr,n);}
239         for (int i = 0; i < n; i++) {
240             arraytextBox->Text += " " + arr[i];
241         }
242     }
243 }

```

Конструкция **try-catch** (строки 219-223-229) – это механизм обработки исключительных ситуаций, которые могут возникать при исполнении кода в блоке **try**. Она необходима здесь, потому что, если конвертация введённого пользователем текста в число прошла неудачно (введены, например, не только цифры), то получим лишь сообщение об ошибке без фатального сбоя (блок **catch**), и программа продолжит своё выполнение.

Самостоятельно определите процедуру **sort** сортировки массива, реализующую быструю сортировку Хоара (используйте код, уже реализованный по заданию 5.4.б в работе №5).

Создайте для формы процедуру-обработчик события **Load**, поместите в неё команды инициализации датчика случайных чисел и передачи фокуса текстовому полю **numtextBox**.

Создайте процедуру обработки события **Click** для элемента **calcbutton**, реализуйте в теле процедуры код вычислений в массиве в соответствии с лист. 7.8.

Проверьте работу программы.

Листинг 7.8. К программе «Массив» (часть 2).

```

251 private: System::Void calbutton_Click(System::Object^ sender,
252                                     System::EventArgs^ e) {
253     int max = arr[0]; //переменная для поиска максимума в массиве
254     int numPosit = 0; //для подсчета количества положительных элементов
255     for (int i = 0; i < n; i++) {
256         if (arr[i] > max) max = arr[i];
257         if (arr[i] > 0) numPosit++;
258     }
259     if (maxcheckBox->Checked) maxtextBox->Text = max.ToString();
260     else maxtextBox->Clear();
261     if (numPositcheckBox->Checked) numPosittextBox->Text = numPosit.ToString();
262     else numPosittextBox->Clear();
263 }

```

Самостоятельно добавьте на форму меню с одной командой «Выход», сформируйте соответствующий код обработчика (по примеру листинга 7.5).

Задание 7.4. Windows-приложение для работы с изображениями

В следующем примере познакомимся ещё с несколькими элементами управления и научимся динамически загружать изображения в форму.

Создайте новый CLR-проект Windows-приложения.

Добавьте на форму меню с пунктами «Файл» и «Загрузить» (рис. 27) с помощью окна **Панель элементов** (группа **Меню и панели инструментов** → элемент **MenuStrip**). В меню «Файл» создайте команду «Выход», а в меню «Загрузить» – подменю «Загрузить изображение» с командами «В формате JPG» и «В формате PNG».

Сделайте двойной щелчок левой кнопкой мыши на пункте «Выход» – среда автоматически создаст код функции обработчика и настроит событие обработки.

При этом откроется пустое тело функции-обработчика выбора этого пункта **выходToolStripMenuItem_Click()**. Добавим в него код, который будет генерировать окно с вопросом (MessageBox) о подтверждении выхода из приложения. Если пользователь подтвердит выход, приложение будет завершено (см. листинг 7.9).

Листинг 7.9.

```

141 private: System::Void выходToolStripMenuItem_Click(System::Object^ sender,
142                                                     System::EventArgs^ e) {
143     if (MessageBox::Show("Вы действительно хотите выйти?", "Внимание!",
144                         MessageBoxButtons::YesNo, MessageBoxIcon::Question) ==
145         System::Windows::Forms::DialogResult::Yes)
146         Application::Exit();
147 }

```

Окно **MessageBox** будет содержать текст вопроса, кнопки **Yes** и **No** и иконку **Question** (Вопрос). В условии проверяется результат ответа пользователя на вопрос.


Запустите приложение и проверьте работоспособность кнопки **Выход**.

Часто панель инструментов (**Toolbar**) дублирует элементы меню для быстрого к ним доступа.

С помощью окна **Панель элементов** (группа **Меню и панели инструментов** → элемент **ToolStrip**) разместите на форме панель инструментов (**Toolbar**). Элемент расположится вдоль верхней границы окна.

В окне свойств этого элемента найдите параметр **Dock** (значение по умолчанию **Top**), измените его на **Left**.

Для увеличения размера кнопок на toolbar'е в его свойствах установите параметр **AutoSize** равным **false**. Установите параметр **Size** → **Width** равным 44. Ширина поля изменится.

Добавим три кнопки на нашу панель. Для этого на панели в раскрывающемся списке элементов , которые можно добавить, выберите элемент **button**. Повторите операцию, чтобы кнопок на панели стало три. Поочередно выберите каждую кнопку и в ее свойствах установите **AutoSize**, равный **false**. После это перейдите к полю **Size** и установите высоту (**Height**), равную 42. Кнопки примут вид квадрата.

Назначим изображения для данных кнопок. В качестве изображений можно использовать все современные форматы, в том числе и PNG с поддержкой прозрачности¹².

Изображение 1 назначим кнопке, отвечающей за открытие дополнительного диалогового окна:



Изображение 2 будет назначено кнопке, отвечающей за загрузку файлов JPG:



Изображение 3 будет назначено кнопке, отвечающей за загрузку файлов PNG:



Первая кнопка будет предназначена для вызова окна с отображением картинки, которую загрузили, и две кнопки будут дублировать меню с функциями загрузки изображений.

Найдите в сети Интернет подходящие изображения в формате png с прозрачным фоном, настройте в любом графическом редакторе для них размер 35x35 точек. Для установки изображений перейдите в свойства каждой кнопки и установите для них значение параметра **Image** → **ImageScaling**, равным **none** (чтобы изображение не масштабировалось). Проверьте, что теперь в параметре **Image** можно выбрать изображение для загрузки.

¹² Готовые изображения можно взять по ссылке [\[https://cloud.mail.ru/public/co87/G1nQJ9x1F\]](https://cloud.mail.ru/public/co87/G1nQJ9x1F)

Назначьте каждой кнопке соответствующее изображение.

В результате получится форма с красивыми кнопками (рис. 27).

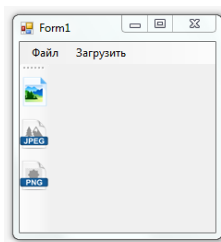


Рис. 27. Вид окна программы после компиляции и запуска.

Увеличьте размер формы **MyForm** до 380x380 точек.

Разместите на ней элемент **PictureBox** и настройте его размер, соответственно до 300x300 точек. Расположите элемент **PictureBox** в центре рабочей области окна.

В свойствах элемента **PictureBox** установите параметр **SizeMode** в значение **StretchImage** – загружаемое изображение будет масштабироваться под размеры элемента **PictureBox**.

Реализуем возможность выбора пользователем файла для загрузки через стандартное Windows-окно загрузки файлов.

Перейдите в **Панель элементов**. Перетащите элемент управления **OpenFileDialog** на форму (место на форме, на которое вы перетащите элемент, не важно – он добавится в поле под окном формы к другим «специфическим» объектам).

Двойными щелчками на командах меню и соответствующих кнопках загрузки JPEG и PNG на панели инструментов создайте 4 пока пустых обработчика нажатий (листинг 7.10).

Листинг 7.10.

```
233 private: System::Void вФорматеJPGToolStripMenuItem_Click(System::Object^ sender,
234                                     System::EventArgs^ e) {
235 }
236 private: System::Void вФорматеPNGToolStripMenuItem_Click(System::Object^ sender,
237                                     System::EventArgs^ e) {
238 }
239 private: System::Void toolStripButton2_Click(System::Object^ sender,
240                                     System::EventArgs^ e) {
241 }
242 private: System::Void toolStripButton3_Click(System::Object^ sender,
243                                     System::EventArgs^ e) {
244 }
```

Далее объявим функцию **LoadImage**, которая в качестве входного параметра будет получать признак (флаг) того, какой фильтр (JPG или PNG) для выбора файлов необходимо выбрать. В функции будет вызываться окно выбора

файла, и, если оно при закрытии возвращает результат **DialogResult::OK** (пользователь выбрал файл и нажал **OK**), то будет попытка загрузить и установить выбранную картинку в поле **PictureBox** (листинг 7.11).

Конструкция **try-catch** необходима здесь по следующей причине: если загрузка прошла неудачно, то получим сообщение об ошибке, но фатального сбоя не возникнет, и программа продолжит своё выполнение.

Реализуйте функцию **LoadImage()** в файле MyForm.h (листинг 7.11).

Листинг 7.11.

```
245 Image^ MemForImage; //переменная для загружаемого изображения
246 private: void LoadImage(bool jpg) { //процедура загрузки изображения
247     //установка начальной папки:
248     openFileDialog1->InitialDirectory="c:\\";
249     if (jpg) //если выбор jpeg-файла:
250         //установка формата jpg:
251         openFileDialog1->Filter = "image (JPEG) files (*.jpg)|*.jpg|All files (*.*)|*.*";
252     else
253         //установка формата png:
254         openFileDialog1->Filter = "image (PNG) files (*.png)|*.png|All files (*.*)|*.*";
255     //если файл в окне выбран:
256     if (openFileDialog1->ShowDialog() == System::Windows::Forms::DialogResult::OK)
257         try { //безопасная попытка загрузки
258             MemForImage = Image::FromFile(openFileDialog1->FileName);
259             //установка изображения в элемент формы PictureBox:
260             pictureBox1->Image = MemForImage;
261         }
262         catch (Exception^ ex) {
263             MessageBox::Show("Не удалось загрузить файл: "+ex->Message, "Ошибка",
264                 MessageBoxButtons::OK, MessageBoxIcon::Asterisk);
265         }
266     }
```

Добавьте в обработчики нажатий вызов объявленной выше функции **LoadImage()** с соответствующими признаками формата искоемых файлов (листинг 7.12).

Листинг 7.12.

```
233 private: System::Void вФорматеJPGToolStripMenuItem_Click(System::Object^ sender,
234     System::EventArgs^ e) {
235     LoadImage(true);
236 }
237 private: System::Void вФорматеPNGToolStripMenuItem_Click(System::Object^ sender,
238     System::EventArgs^ e) {
239     LoadImage(false);
240 }
241 private: System::Void toolStripButton2_Click(System::Object^ sender,
242     System::EventArgs^ e) {
243     LoadImage(true);
244 }
245 private: System::Void toolStripButton3_Click(System::Object^ sender,
246     System::EventArgs^ e) {
247     LoadImage(false);
248 }
```

Запустите приложение и попробуйте загрузить изображение.

В окне изображение масштабируется под размер элемента **PictureBox**.

Поэтому добавим в проект еще одну форму, на которой будем отображать картинку в ее истинном размере.

Перейдите в окно **Обозреватель решений**, после чего щелкните на названии проекта (не решения) правой кнопкой мыши и в открывшемся контекстном меню выберите **Добавить** → **Создать элемент** и в открывшемся меню в разделе **Visual C++** → **UI** выберите **Форма Windows Forms**. В открывшемся окне в поле **Имя** введите название файла – **Preview.h**.

Увеличьте размер окна **Preview** (рис. 28). Затем на форме разместите элемент **Panel**. Внутри элемента **Panel** разместите элемент **PictureBox** и кнопку «Закрыть».

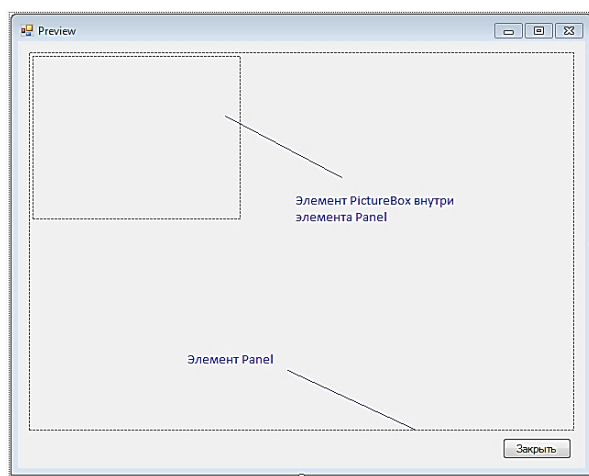


Рис. 28. Форма вспомогательного окна программы.

Данное окно будет: получать при загрузке в качестве параметра ссылку на загружаемое изображение; затем устанавливать границы элемента **pictureBox**, равными размерам полученного изображения, после чего устанавливать его в элемент **pictureBox**.

Перейдите к свойствам элемента **Panel** и установите значение параметра **AutoScroll**, равное **true**. Теперь в том случае, если размер изображения будет превышать размер элемента **panel1**, будут появляться вертикальная и горизонтальная полосы прокрутки, с помощью которых можно будет просмотреть всё изображение.

Сформируем код, необходимый для реализации заявленных возможностей.

Щелкните дважды по первой кнопке на панели задач главного окна, после чего откроется код функции, которая будет вызвана при щелчке по данной кнопке.

Код обработчика в файле главной формы **MyForm.h** сформируйте в соответствии с листингом 7.13. Здесь в строке 2 связываются файлы двух форм.

Листинг 7.13.

```
1 | #pragma once
2 | #include "Preview.h"
```



```

...
273 private: System::Void toolStripButton1_Click(System::Object^ sender,
274                                             System::EventArgs^ e) {
275     //создаем вспомогательную форму
276     Preview^ PreviewF = gcnew Preview(MemForImage);
277     //вызываем форму в модальном режиме
278     PreviewF->ShowDialog();
279 }

```

Новая форма – **PreviewF** (экземпляр класса **Preview**) получает в качестве параметра загружаемое изображение (**MemForImage**). Для правильной работы программы изменим конструктор **Preview**, а также реализуем остальную функциональность программы.

Измените код конструктора **Preview** в файле **Preview.h** в соответствии с листингом 7.14.

Листинг 7.14.

```

15 public ref class Preview : public System::Windows::Forms::Form
16 {
17     Image^ PicToView; //для хранения изображения
18     //модифицированный конструктор окна
19 public:
20     Preview(Image^ View)
21     {
22         PicToView = View; //получаем из главного окна изображение
23         InitializeComponent();
24     }

```

Перейдите к окну **Preview** и сделайте двойной щелчок на нём (а НЕ на размещённых на нём объектах). Откроется для редактирования обработчик открытия второй формы **Preview_Load()**, сформируйте его код в соответствии с лист. 7.15.

Листинг 7.15.

```

92 private: System::Void Preview_Load(System::Object^ sender,
93                                   System::EventArgs^ e) {
94     //устанавливаем размеры pictureBox1 по размерам изображения:
95     pictureBox1->Width = PicToView->Width;
96     pictureBox1->Height = PicToView->Height;
97     //помещаем изображение в pictureBox1
98     pictureBox1->Image = PicToView;
99 }

```

Сформируйте код обработчика кнопки «Заккрыть» на форме **Preview**.

Проверьте работу приложения на различных рисунках (рис. 29).

Самостоятельно доработайте код программы:

- исправьте текст всплывающих подписей к кнопкам на панели инструментов основного окна, а также заголовок основного окна;
- реализуйте, чтобы при загрузке нового рисунка в окне диалога автоматически открывалась папка предыдущего загруженного рисунка;
- реализуйте, чтобы при перетаскивании мышкой правой границы окна **Preview** кнопка **Заккрыть** также смещалась (т.е. положение кнопки **Заккрыть** не должно изменяться относительно правой границы окна **Preview**).

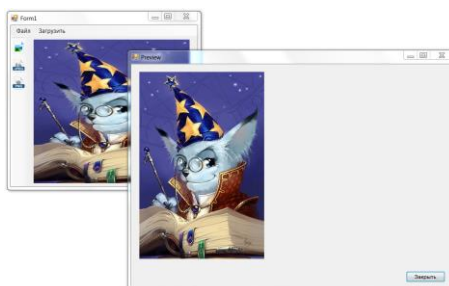


Рис. 29. Пример работы программы.¹³

Взаимодействие между формами

Часто для реализации вычислительной задачи в программе должно быть несколько окон, причём содержимое в одном окне может зависеть от содержимого других. Т.е. информация из одного окна может использоваться в других окнах. Подобное *информационное взаимодействие* реализуется несколькими способами, рассмотрим один из них на примере простой программы с двумя формами.

Пусть есть форма **MyForm** (родительская) с кнопкой (рис. 30), по которой должна открыться другая форма-потомок **MyForm1** (рис. 31).

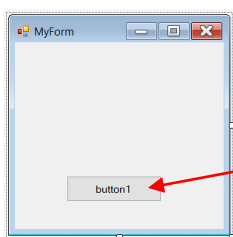


Рис. 30. Первая (родительская) форма.

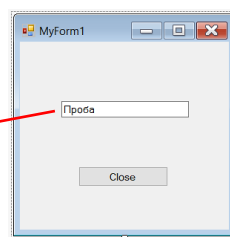


Рис. 31. Вторая форма (потомок).

На второй форме есть текстовое поле, содержимое которого по нажатию на кнопку на этой второй форме **MyForm1** должно стать подписью кнопки родительской формы **MyForm**.

Для реализации этого:

1. В файле **MyForm.h** оставляем только описание обработчика **button1_Click** кнопки родительской формы:

```
77 | #pragma endregion
78 | private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e);
79 |
```

Определение этого обработчика позже поместим в **MyForm.cpp** – см. пункт 3. Это позволит избежать ошибки – перекрёстных ссылок в **MyForm.h** и **MyForm1.h**.

2. В файле **MyForm1.h** подключим **MyForm.h** (см. листинг 7.16, строка 2).

И переопределим конструктор формы **MyForm1**, добавив параметр – форму-родителя (им будет **MyForm**) – см. листинг 7.16, строки 19-29.

¹³ Иллюстрация художника А.В. Иванченко (Anru).

Это позволит в обработчике нажатия кнопки второй формы обратиться к родительской форме (листинг 7.16, строки 95-98).

3. Остаётся в файле **MyForm.cpp** подключить **MyForm1.h** и **MyForm1.h** (листинг 7.17, строки 1-2) и добавить определение обработчика нажатия на кнопку родительской формы (листинг 7.17, строки 7-11).

Листинг 7.16. Фрагменты файла MyForm1.h.

```
1  |#pragma once
2  |#include "MyForm.h"
...
18 | public:
19 |     //MyForm1(void)
20 |     MyForm1 (MyForm^ parent)
21 |     {
22 |         InitializeComponent();
23 |         //
24 |         //TODO: добавьте код конструктора
25 |         //
26 |         parentForm = parent;
27 |     }
28 | private:
29 |     MyForm^ parentForm;
...
94 | #pragma endregion
95 | private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
96 |     parentForm->button1->Text = this->textBox1->Text;
97 |     Close();
98 | }
```

Листинг 7.17. Файл MyForm.cpp.

```
1  |#include "MyForm.h"
2  |#include "MyForm1.h"
3
4  |using namespace System;
5  |using namespace System::Windows::Forms;
6
7  |System::Void ProjectPro::MyForm::button1_Click(System::Object^ sender, System::EventArgs^ e) {
8  |    //ВОТ
9  |    MyForm1^ MyF1 = gcnew MyForm1(this);
10 |    MyF1->Show();
11 | }
12
13 | [STAThreadAttribute]
14 | int main(array<String^>^ args) {
15 |     Application::EnableVisualStyles();
16 |     Application::SetCompatibleTextRenderingDefault(false);
17 |     ProjectPro::MyForm form;
18 |     Application::Run(% form);
19
20 |     return 0;
21 | }
```

Рекомендуем воспользоваться этим приёмом для решения индивидуальных задач 7 и 8.

Индивидуальное задание №7

Реализуйте средствами языка C++ назначенный преподавателем вариант

индивидуального задания №7 на GUI-приложения¹⁴, с обязательным соблюдением требований общей части.

Вопросы для самоподготовки

- ✓ Назовите пользовательские интерфейсы приложений по элементам интерфейса и по способам взаимодействия.
- ✓ Что такое спецификация CLI? Что такое CLR и CIL?
- ✓ Из каких этапов состоит разработка оконного приложения?
- ✓ Опишите назначение декларатора **handle** (^) объектного типа.

¹⁴ См. Павловская Т.А. С#. Программирование на языке высокого уровня: Учебник для вузов, 2014. – С. 412-419.

8. ЭЛЕМЕНТЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

Цель: познакомиться с возможностями и особенностями реализации в языке C++ программируемой графики¹⁵.

Графическая поверхность

Для вывода на форму графических фигур, текста и готовых изображений создается экземпляр класса **Graphics** из пространства имён **System:Drawing**.

На практике чаще используется *графическая поверхность* не самой формы, а компонента **PictureBox**, который представляет собой объект класса **Graphics**, чьи методы и обеспечивают вывод графики.

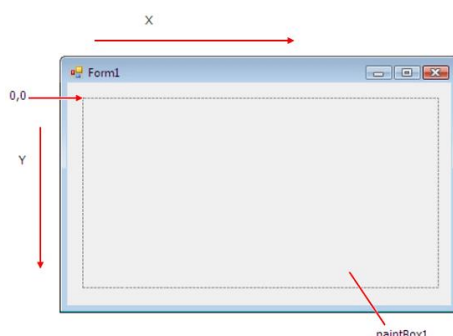


Рис. 32. Графическая поверхность.

Графическая поверхность состоит из отдельных точек (пикселей). Положение точки на графической поверхности определяется горизонтальной (X) и вертикальной (Y) координатами (рис. 32). Координаты точек отсчитываются от левого верхнего угла и возрастают слева направо (координата X) и сверху вниз (координата Y). Начало координат (0, 0) – это левая верхняя точка.

Примечание: размер графической поверхности формы соответствует размеру внутренней (клиентской) её области без учёта высоты области заголовка и ширины границ формы (свойство **ClientSize**), а размер графической поверхности **PictureBox** – размеру этого компонента.

Задание 8.1. Простой пример прорисовки на графической поверхности

Реализуем пример обработки события **Paint** для компонента **PictureBox** и прорисовки на его графической поверхности простого изображения.

Создайте новый CLR-проект Windows-приложения.

¹⁵ В рамках одного раздела можно получить лишь самое начальное представление о программируемой графике. Больше практических заданий с их более подробным описанием и справочный материал представлен в специальных печатных и электронных изданиях, например [4, 10].

Добавьте на форму с помощью окна **Панель элементов** компонент **PictureBox** (группа **Стандартные элементы управления** → элемент **PictureBox**). Растяните его на всю форму.

Создадим для компонента **PictureBox** процедуру обработки события **Paint**: Выберите компонент **PictureBox**, затем в окне **Свойства** откройте вкладку **События** и в поле события **Paint** сделайте двойной щелчок. При этом среда автоматически сформирует не только имя процедуры-обработчика, но и её код. В теле процедуры обработчика события **Paint** компонента **PictureBox** реализуйте код в соответствии с листингом 8.1.

Здесь метод **DrawRectangle** выполняет прорисовку контура объекта. Заполнение внутренней области объекта осуществляется при помощи метода **FillRectangle**.

Листинг 8.1.

```
80 private: System::Void pictureBox1_Paint(System::Object^ sender,
81                                     System::Windows::Forms::PaintEventArgs^ e) {
82     int w=80, h=60; // размер флага
83     int ws = w / 3; // ширина полосы (флаг из трех вертикальных полос)
84     int x0, y0; // координаты левого верхнего угла флага
85     int x, y; // координаты левого верхнего угла полосы
86
87     // разместим флаг в центре формы:
88     x0 = (this->ClientSize.Width - w) / 2; // ширина внутренней области формы
89     y0 = (this->ClientSize.Height - h) / 2; // высота внутренней области формы
90     // рисуем флаг
91     // зеленая полоса
92     x = x0;
93     y = y0;
94     e->Graphics->FillRectangle
95         (System::Drawing::Brushes::Green, x, y, ws, h);
96
97     // белая полоса
98     x = x + ws + 1;
99     e->Graphics->FillRectangle
100         (System::Drawing::Brushes::White, x, y, ws, h);
101     // оранжевая полоса
102     x = x + ws + 1;
103     e->Graphics->FillRectangle
104         (System::Drawing::Brushes::Orange, x, y, ws, h);
105     // окантовка
106     e->Graphics->DrawRectangle
107         (System::Drawing::Pens::Black, x0, y0, w, h);
108     // подпись (шрифтом формы)
109     e->Graphics->DrawString("Ирландия", this->Font, Brushes::Black, x0+16, y0+h+10);
110 }
```

Создайте процедуру-обработчик события для формы **SizeChanged**, возникающего при изменении её размера. Реализуйте в теле этой процедуры код лист. 8.2.

Листинг 8.2.

```
111 private: System::Void MyForm_SizeChanged(System::Object^ sender,
112                                     System::EventArgs^ e) {
113     this->Refresh(); // обновить форму
114 }
```

Здесь метод **Refresh** информирует систему о необходимости перерисовать

(обновить) форму. В результате генерируется событие **Paint**, при обработке которого произойдет вывод в форму трёх полос различного цвета в рамке с соответствующей подписью (рис. 33).

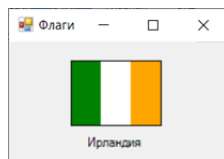


Рис. 33. Результат выполнения программы.

Определите заголовок формы в соответствии с рис. 33.

Проверьте работу программы при изменении размера окна.

Самостоятельно реализуйте под уже сформированным изображением вывод российского флага с соответствующей подписью.

Карандаши и кисти

В предыдущем примере для рисования графических примитивов использовались *карандаши* и *кисти*.

Карандаш (объект **Pen**) определяет вид линии (цвет, толщину, стиль), а кисть (объект **Brush**) – вид заливки внутренней области, ограниченной линией. Например, метод

```
e->Graphics->DrawLine(System::Drawing::Pens::Black, 10, 20, 100, 20);
```

рисует из точки (10, 20) горизонтальную линию длиной 100 пикселей, используя черный (Black) карандаш из стандартного набора карандашей (**Pens**).

Метод

```
e->Graphics->FillRectangle(System::Drawing::Brushes::Green, 5, 10, 20, 20);
```

при помощи стандартной кисти из стандартного набора кистей (**Brushes**) зеленого (Green) цвета рисует зеленый квадрат, левый верхний угол которого находится в точке (5, 10).

Карандаши и кисти определены в пространстве имен System::Drawing.

В распоряжении программиста есть два набора карандашей: стандартный и системный. Также программист может создать собственный карандаш.

Стандартный набор карандашей – это цветные карандаши (около 150), которые рисуют непрерывную линию толщиной в один пиксел. Перечень можно найти в разделе **Pens** справочника на портале Microsoft Docs [11].

Системный набор карандашей – их цвет определяется текущей цветовой схемой операционной системы и совпадает с цветом какого-либо элемента интерфейса пользователя. Например, цвет карандаша **SystemPens::ControlText** совпадает с цветом, который в текущей цветовой схеме используется для отображения текста на элементах управления (командных кнопках и др.), а цвет карандаша **SystemPens::WindowText** – цвет текста в окнах сообщений.

Карандаш из стандартного (**Pens**) и системного (**SystemPens**) наборов рисует непрерывную линию толщиной в один пиксел. Если надо нарисовать пунктирную линию или линию толщиной больше единицы, то следует использовать карандаш программиста.

Собственный карандаш программиста – это объект типа **Pen**, свойства которого определяют вид линии, рисуемой карандашом. Перечень свойств можно найти в разделе **Pen** справочника на портале Microsoft Docs [11].

Для того чтобы использовать собственный карандаш, его нужно создать конструктором объекта **Pen**, например:

```
Pen^ skyBluePen = gcnnew Pen (Brushes::DeepSkyBlue);
```

Конструктор *перегруженный*, т.е. для объекта **Pen** определено несколько конструкторов, различающихся количеством параметров. Например, конструктор **Pen(Цвет)** создает карандаш заданного цвета толщиной в 1 пиксел, а **Pen(Цвет, Толщина)** – карандаш заданного цвета и толщины. В качестве параметра **Цвет** можно использовать константу типа **Color**, значения которой (более 100) можно найти в разделе **Color** справочной системы Microsoft Docs [11].

Цвет, ширину линии и стиль собственного карандаша, созданного программистом, можно изменить, определив значение соответствующего свойства. Можно создать собственный карандаш, который рисует, например, пунктирную линию, отличную от стандартной.

В листинге 8.3 приведен пример, который демонстрирует создание и использование собственного карандаша программиста.

Кисти (brushes) используются для закраски внутренних областей геометрических фигур. Есть три типа кистей: стандартные, штриховые и текстурные.

Стандартная кисть закрашивает область одним цветом (сплошная закрашка). Всего есть почти 150 кистей. Полный список кистей см. в разделе **Brushes** справочной системы на портале Microsoft Docs [11].

Штриховая кисть (HatchBrush) закрашивает область путем штриховки. Область может быть заштрихована горизонтальными, вертикальными или наклонными линиями разного стиля и толщины. Полный список стилей штриховки см. в подразделе **HatchStyle** раздела «Пространство имён System.Drawing.Drawing2D» справочной системе на портале Microsoft Docs [12].


```

79 private: System::Void pictureBox1_Paint(System::Object^ sender,
80                                     System::Windows::Forms::PaintEventArgs^ e) {
81     // графическая поверхность
82     System::Drawing::Graphics^ g = e->Graphics;
83     //создаём карандаш толщиной 2 пиксела
84     System::Drawing::Pen^ aPen;
85     aPen = gcnew Pen(Color::Red,2);
86     //рисует этим карандашом
87     g->DrawRectangle(aPen,10,10,100,100);
88     //изменяем толщину и цвет карандаша
89     aPen->Width = 4;
90     aPen->Color = Color::Green;
91     //рисует
92     g->DrawRectangle(aPen, 20, 20, 100, 100);
93     //снова меняем толщину, цвет, а также стиль
94     aPen->Width = 1;
95     aPen->Color = Color::Purple;
96     aPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash; //пунктир
97     //рисует пунктиром
98     g->DrawRectangle(aPen, 30, 30, 100, 100);
99 }

```

В листинге 8.4 приведена функция, демонстрирующая процесс создания и использования штриховой кисти. При создании кисти конструктору передаются: константа `HatchStyle`, которая задает вид штриховки, и две константы типа `Color`, первая из которых определяет цвет штрихов, вторая – цвет фона.

```

79 private: System::Void pictureBox1_Paint(System::Object^ sender,
80                                     System::Windows::Forms::PaintEventArgs^ e) {
81     // графическая поверхность
82     System::Drawing::Graphics^ g = e->Graphics;
83     //создаём HatchBrush-кисть
84     System::Drawing::Drawing2D::HatchBrush^ myBrush;
85     myBrush = gcnew System::Drawing::Drawing2D::HatchBrush
86         (System::Drawing::Drawing2D::HatchStyle::DottedGrid, //стиль штриха
87          Color::Black, //цвет штриха
88          Color::SkyBlue); //цвет фона
89
90     g->FillRectangle(myBrush,10,10,250,250);
91 }

```

Примечание: класс штриховой кисти `HatchBrush` определен в пространстве имен `System::Drawing::Drawing2D`, поэтому в программу следует добавить ссылку на это пространство имён или указать полное описание элемента (как в листинге 8.4, строки 84-86).

Текстурная кисть (`TextureBrush`) представляет собой рисунок, который обычно загружается во время работы программы из файла (bmp, jpg или gif) или из ресурса. Закраска области текстурной кистью выполняется путем дублирования рисунка внутри области.

Листинг 8.5 демонстрирует процесс создания и использования текстурной кисти. Кисть создает конструктор, которому в качестве параметра передается текстура – имя графического файла.

```

79 private: System::Void pictureBox1_Paint(System::Object^ sender,
80                                     System::Windows::Forms::PaintEventArgs^ e) {
81     // графическая поверхность
82     System::Drawing::Graphics^ g = e->Graphics;
83     float x = 20, y = 20, w = 130, h = 195;
84     //создаём текстурную кисть
85     System::Drawing::TextureBrush^ myBrush;
86     try {
87         myBrush = gcnew System::Drawing::TextureBrush
88             (Image::FromFile(Application::StartupPath + "\\brush_1.bmp"));
89         //рисует текстурной кистью
90         g->FillRectangle(myBrush, 10, 10, 250, 250);
91     }
92     catch (System::IO::FileNotFoundException^ ex) {
93         g->DrawRectangle(Pens::Black, x, y, w, h);
94         g->DrawString("Рисунок", this->Font, Brushes::Black, x + 5, y + 5);
95         g->DrawString("не найден", this->Font, Brushes::Black, x + 5, y + 20);
96     }
97 }

```

Задание 8.2. Приёмы использования карандашей и кистей

8.2.а. Реализуйте в новом проекте собственный вариант карандаша и кисти (по примеру листингов 8.3 и 8.4), нарисуйте им ваши инициалы (ФИО).

8.2.б. Реализуйте собственный вариант текстуры (по примеру листинга 8.5).

8.2.в. Самостоятельно изучите справочную информацию и реализуйте пример градиентной заливки с помощью *градиентной* кисти **LinearGradientBrush** (из пространства имён **System::Drawing::Drawing2D**).

Графические примитивы

Любое векторное изображение (чертёж или схема) – это совокупность *графических примитивов*: точек, линий, окружностей, дуг и др. Их прорисовку на графической поверхности выполняют соответствующие методы (перечень можно найти в разделе **Graphics** → **Методы** справочной системы на портале Microsoft Docs [11]).

Линия

Метод **DrawLine** рисует прямую линию. При вызове метода следует указать карандаш, которым надо нарисовать линию, и координаты начала и конца:

```
DrawLine(aPen, x1, y1, x2, y2)
```

Здесь параметр **aPen** (типа **Brush**) задаёт карандаш, которым рисуется линия, x_1 и y_1 – точку начала линии, x_2 и y_2 – точку конца линии. Параметры x_1 , y_1 , x_2 и y_2 должны быть одного типа (**int**).

Например, инструкция

```
e->Graphics->DrawLine(Pens::Green, 25, 25, 377, 10);
```

рисует зеленую линию толщиной в 1 пиксел из точки (10, 10) в точку (300, 10).

В качестве параметров методов создания графических примитивов часто используется структура **Point**. Её поля **X** и **Y** – это координаты на графической

поверхности. Например:

```
Point p1 = Point (25, 25);  
Point p2 = Point (377, 10);
```

Тогда эту же линию можно нарисовать так:

```
e->Graphics->DrawLine (Pens::Green, p1, p2);
```

Другой пример реализации подобных структур:

```
Point startPoint = Point (Button1->Size);  
Point endPoint = startPoint + System::Drawing::Size ( 140, 150 );
```

Впоследствии эти структуры можно задействовать многократно.

Ломаная линия

Метод **DrawLines** рисует ломаную линию. В качестве параметров методу передаётся карандаш (**Pen**) и массив типа **Point**, элементы которого содержат координаты узловых точек линии. Метод рисует ломаную линию, последовательно соединяя точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т.д. Например, следующий фрагмент кода рисует ломаную линию, состоящую из четырех звеньев:

```
// массив точек  
array<Point>^ p = { Point(10, 10), Point(10, 100), Point(200, 50), Point(250, 300) };  
// изменение координат точек кривой:  
p[0].X = 10; p[0].Y = 50;  
p[1].X = 20; p[1].Y = 20;  
p[2].X = 30; p[2].Y = 50;  
p[3].X = 40; p[3].Y = 20;  
// рисование ломаной линии:  
e->Graphics->DrawLines(Pens::Green, p);
```

Прямоугольник

Рамочный прямоугольник можно нарисовать методом **DrawRectangle**, которому в качестве параметров передаются координаты левого верхнего угла и размеры прямоугольника:

```
e->Graphics->DrawRectangle(Pens::Black, x, y, w, h)
```

Методу **DrawRectangle** можно передать не сами координаты и размеры, а структуру типа **Rectangle**, поля которой определяют прямоугольник (положение и размер):

```
Rectangle aRect = Rectangle (20, 100, 50, 50);  
e->Graphics->DrawRectangle (Pens::Blue, aRect);
```

Эту структуру позже можно использовать для формирования копий фигур.

Метод **FillRectangle** рисует *закрашенный* прямоугольник:

```
e->Graphics->FillRectangle (aBrush, x, y, w, h);
```

Здесь параметр **aBrush**, в качестве которого можно использовать стандартную или созданную программистом штриховую (**HatchBrush**), градиентную

(LineadGradientBrush) или текстурную (TextureBrush) кисть, определяет цвет и стиль закрашки внутренней области прямоугольника.

Аналогично методу для рамочного прямоугольника, здесь вместо x, y, w и h можно указать структуру типа **Rectangle**:

```
e->Graphics->FillRectangle (aBrush, aRect);
```

Параметр **aBrush**, в качестве которого можно использовать стандартную или созданную программистом штриховую (HatchBrush), градиентную (LineadGradientBrush) или текстурную (TextureBrush) кисть, определяет цвет и стиль закрашки области.

Многоугольник

Метод **DrawPolygon** чертит контур многоугольника. Вызов метода:

```
DrawPolygon (aPen, p)
```

Параметр **p** – это массив элементов типа **Point**, определяет координаты вершин многоугольника. Метод **DrawPolygon** чертит многоугольник, соединяя прямыми линиями точки, координаты которых находятся в массиве: первую со второй, вторую с третьей и т.д. Последняя точка соединяется с первой. Вид границы определяет параметр **aPen**, в качестве которого можно использовать стандартный или созданный программистом карандаш.

Закрашенный многоугольник рисует метод **FillPolygon**. Инструкция вызова метода в общем виде выглядит так:

```
FillPolygon (aBrush, p)
```

Параметр **aBrush**, в качестве которого можно использовать стандартную или созданную программистом штриховую (HatchBrush), градиентную (LineadGradientBrush) или текстурную (TextureBrush) кисть, определяет цвет и стиль закрашки внутренней области многоугольника.

Пример использования методов **DrawPolygon** и **FillPolygon** для рисования:

```
array<Point>^ p = {Point(0, 0), Point(10, 10), Point(10, 100), Point(200, 50),  
Point(250, 300)};  
p[0].X = 10; p[0].Y = 30;  
p[1].X = 10; p[1].Y = 10;  
p[2].X = 30; p[2].Y = 20;  
p[3].X = 50; p[3].Y = 10;  
p[4].X = 50; p[4].Y = 30;  
e->Graphics->FillPolygon(Brushes::Gold, p);  
e->Graphics->DrawPolygon(Pens::Black, p);
```

Аналогично уже рассмотренным можно задействовать и другие методы для рисования, например, эллипса и окружности, дуги и сектора, текста и пр.

Задание 8.3. Приёмы использования графических примитивов

8.3.а. В новом проекте с помощью метода **DrawLines** отобразите на форме правильный пятиугольник.

8.3.б. Реализуйте вывод на форму одной точки (у объекта **Graphics** нет отдельного метода для вывода точки).

8.3.в. Самостоятельно реализуйте программу для построения круговой диаграммы аналогично примеру на рис. 34 на произвольную тему. Вывод диаграммы сопроводите текстовым заголовком и описанием легенды.

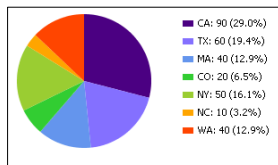


Рис. 34. Круговая диаграмма.

Задание 8.4. Пример с динамической графикой

Рассмотрим приёмы создания динамических изображений на основе программы, которая будет показывать в окне формы текущее системное время (рис. 35).

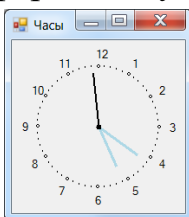


Рис. 35. Программа «Часы».

Создайте новый CLR-проект Windows-приложения.

Расположите на форме компонент **Timer**.

Изучите и реализуйте код листингов 8.6 – 8.10.

Проверьте работу программы. Дополните программу собственным вариантом цветового и стилевого оформления содержимого формы (стрелок, текста).

Листинг 8.6.

```

15 public ref class MyForm : public System::Windows::Forms::Form
16 {
17 public:
18     MyForm(void)
19     {
20         InitializeComponent();
21         // g – графическая поверхность
22         g = this->CreateGraphics();
23         R = 70;
24         // установим размер формы
25         // в соответствии с размером циферблата
26         this->ClientSize =
27             System::Drawing::Size((R + 30) * 2, (R + 30) * 2);
28         x0 = R + 30;
29         y0 = R + 30;
30         // Определить положение стрелок.
31         // Угол между метками часов (цифрами) – 30 градусов.
32         // Угол между метками минут – 6 градусов.
33         // Угол отсчитываем от 12-ти часов
34         ahr = 90 - DateTime::Now.Hour * 30 - (DateTime::Now.Minute / 12) * 6;
35         amin = 90 - DateTime::Now.Minute * 6;
36         asec = 90 - DateTime::Now.Second * 6;
37         timer1->Interval = 1000; // период сигнала от таймера 1 с
38         timer1->Enabled = true; // пуск таймера
39     }
40 private:
41     // графическая поверхность
42     Graphics^ g;
43     int R; // радиус циферблата
44     int x0, y0; // центр циферблата
45     int ahr, amin, asec; // положение стрелок (угол)

```

Листинг 8.7.

```

96     //сигнал от таймера
97 private: System::Void timer1_Tick(System::Object^ sender, System::EventArgs^ e) {
98     // нарисовать стрелки
99     DrawClock();
100 }

```

Листинг 8.8.

```

102 private: System::Void MyForm_Paint(System::Object^ sender,
103                                     System::Windows::Forms::PaintEventArgs^ e) {
104     int x, y; // координаты маркера на циферблате
105     int a; // угол между ОХ и прямой (x0,y0) (x,y)
106     int h; // метка часовой риски
107     a = 0; // метки ставим от 3 -х часов против часовой стрелки
108     h = 3; // угол 0 градусов – это 3 часа
109 #define TORAD 0.0174532
110     // циферблат
111     while (a < 360)
112     {
113         x = x0 + R * Math::Cos(a * TORAD);
114         y = x0 - R * Math::Sin(a * TORAD);
115         // Form1->Canvas->MoveTo(x,y);
116         if ((a % 30) == 0)
117         {
118             g->DrawEllipse(Pens::Black, x - 2, y - 2, 3, 3);
119             // цифры по большему радиусу
120             x = x0 + (R + 15) * Math::Cos(a * TORAD);
121             y = x0 - (R + 15) * Math::Sin(a * TORAD);
122             g->DrawString(h.ToString(),
123                 this->Font, Brushes::Black, x - 5, y - 7);
124             h--;
125             if (h == 0) h = 12;
126         }

```

```

127         else
128             g->DrawEllipse(Pens::Black, x - 1, y - 1, 1, 1);
129         a = a + 6; // 1 минута – 6 градусов
130     }
131     // стрелки
132     DrawClock();
133 }

```

Листинг 8.9.

```

135 // Рисует вектор из точки (x0,y0) под
136 // углом а относительно оси X
137 // L – длина вектора
138 void Vector(float x0, float y0, float a, float l, System::Drawing::Pen^ aPen)
139 {
140     // x0,y0 – начало вектора
141     // а – угол между осью x и вектором
142     // l – длина вектора
143     // коэфф. пересчета угла из градусов в радианы
144     #define TORAD 0.0174532
145     float x, y; // координаты конца вектора
146     x = x0 + l * Math::Cos(a*TORAD);
147     y = y0 - l * Math::Sin(a*TORAD);
148     g->DrawLine(aPen, x0, y0, x, y);
149 }

```

Листинг 8.10.

```

151 // рисует стрелки
152 void DrawClock(void)
153 {
154     // карандаши для рисования стрелок
155     Pen^ hPen = gcnew Pen(Color::LightBlue, 3);
156     Pen^ mPen = gcnew Pen(Color::LightBlue, 3);
157     Pen^ sPen = gcnew Pen(Color::Black, 2);
158     // карандаш для стирания стрелок
159     Pen^ cPen = gcnew Pen(SystemColors::Control, 4);
160     // шаг секундной и минутной стрелок 6 градусов,
161     // часовой – 30.
162     // стереть изображение стрелок:
163     Vector(x0, y0, ahr, R - 20, cPen); // часовую
164     Vector(x0, y0, amin, R - 15, cPen); // минутную
165     Vector(x0, y0, asec, R - 7, cPen); // секундную
166     // новое положение стрелок
167     ahr = 90 - DateTime::Now.Hour * 30 - (DateTime::Now.Minute / 12) * 6;
168     amin = 90 - DateTime::Now.Minute * 6;
169     asec = 90 - DateTime::Now.Second * 6;
170     // нарисовать стрелки:
171     // часовую
172     Vector(x0, y0, ahr, R - 20, hPen);
173     // минутную
174     Vector(x0, y0, amin, R - 15, mPen);
175     // секундную
176     Vector(x0, y0, asec, R - 7, sPen);
177     g->DrawEllipse(Pens::Black, x0-2, y0-2, 4, 4);
178     g->FillEllipse(Brushes::Black, x0 - 3, y0 - 3, 6, 6);
179 }

```

Битовые образы (спрайты)

Для формирования сложных изображений используют *битовые образы*. Битовый образ (спрайт) – это небольшое изображение в оперативной памяти. Содержимое памяти можно быстро вывести на экран, поэтому спрайты используются для формирования динамической графики, например, в играх.

Создать битовый образ (объект **Bitmap**) можно из файла (bmp, jpg или gif),

из ресурса или копированием из другого графического объекта (**Image**).

Загрузку битового образа из файла обеспечивает конструктор, которому в качестве параметра передаётся имя файла. Например:

```
Bitmap^ image1; // спрайт  
// загрузить из файла  
image1 = Gcnew Bitmap ("D:\\My Music\\music.bmp", true);
```

Битовый образ можно вывести на графическую поверхность формы или компонента **PictureBox** методом **DrawImage**. В качестве его параметров указывают сам битовый образ и координаты точки поверхности, от которой следует вывести битовый образ. Например, инструкция

```
e->Graphics->DrawImage (image1, 0, 0);
```

выводит на графическую поверхность битовый образ **image1**.

Примечание: вместо параметров *x* и *y* в инструкции вызова метода **DrawImage** можно указать структуру типа **Point**.

Для битового образа можно задать *прозрачный цвет*. Точки рисунка, цвет которых совпадает с "прозрачным", при выводе битового образа не отображаются. Прозрачный цвет задает метод **MakeTransparent**. При его вызове необходимо указать цвет, который следует рассматривать как прозрачный. Например, команда

```
plane.MakeTransparent(System.Drawing.Color.Magenta);
```

задаёт, что для спрайта **plane** прозрачным является цвет **Magenta** (пурпурный).

Примечание: при вызове метода **MakeTransparent** цвет можно не указывать. В этом случае прозрачным будет цвет, которым окрашена левая нижняя точка спрайта.

Следующий пример продемонстрирует загрузку и отображение битовых образов¹⁶. Пусть есть два прямоугольных изображения – небо и самолёт (**sky** и **plane**) – это битовые образы¹⁷. На картинке самолёт окружен белым полем (фоном), но его не будет видно, т.к. перед выводом битового образа **plane** для фона будет определён прозрачный цвет.

Загрузку битовых образов выполнит конструктор формы (листинг 8.11, строки 26-29), вывод – функция обработки события **Paint** элемента **PictureBox** (листинг 8.12).

¹⁶ В основу задания положен адаптированный пример из книги Культин Н.Б. Основы программирования в Microsoft Visual C# 2010. – СПб.: БХВ-Петербург, 2011. – С. 173.

¹⁷ Изображения можно скачать по ссылке: [\[https://cloud.mail.ru/public/W2zi/NjRBnXJWVW\]](https://cloud.mail.ru/public/W2zi/NjRBnXJWVW)


```

15 public ref class MyForm : public System::Windows::Forms::Form
16 {
17     public:
18         Bitmap^ sky;    //спрайт "небо"
19         Bitmap^ plane;  //спрайт "самолёт"
20         MyForm(void)
21         {
22             InitializeComponent();
23             //загрузка изображений из файлов:
24             try
25             {
26                 sky = gcnew System::Drawing::Bitmap
27                     (Application::StartupPath + "\\sky.bmp");
28                 plane = gcnew System::Drawing::Bitmap
29                     (Application::StartupPath + "\\plane.bmp");
30             }
31             catch (System::Exception^ e)
32             {
33                 MessageBox::Show("Ошибка загрузки битового образа: "
34                     + e->Message, "Полет", MessageBoxButtons::OK,
35                     MessageBoxIcon::Error);
36                 this->Close(); // закрываем форму
37                 return;
38             }
39             // размер формы = размеру фонового рисунка
40             this->ClientSize = sky->Size;
41         }

```

```

96 private: System::Void pictureBox1_Paint(System::Object^ sender,
97                                         System::Windows::Forms::PaintEventArgs^ e) {
98     if (sky != nullptr)
99     {
100         //вывести фоновый рисунок
101         e->Graphics->DrawImage(sky, 0, 0);
102     }
103     if (plane != nullptr)
104     {
105         //левый самолёт
106         e->Graphics->DrawImage(plane, 70, 50);
107         //правый самолёт
108         plane->MakeTransparent(); // сделать прозрачным фон
109         e->Graphics->DrawImage(plane, 300, 80);
110     }
111 }

```

Битовые образы можно использовать для создания динамического изображения – *анимации*.

Одним из способов анимации является перемещение объекта на фоне. Иллюзия движения создаётся выводом изображения объекта, затем, через некоторое время, его удалением и новым выводом, но уже на некотором расстоянии от первоначального положения. Подбор времени между удалением и выводом, а также расстояния между новым и предыдущим положением объекта (шага перемещения), позволит добиться эффекта равномерного движения.

Программа полёта самолёта на фоне неба покажет, как заставить объект двигаться [4, с. 114]. Код её конструктора формы и обработчиков событий приведён в листингах 8.13 и 8.14 (стр. 109-110).

Оба изображения загружаются из файлов, причём всё выводится прямо на

форму без использования элемента **PictureBox**. Конструктор формы обеспечит загрузку битовых образов, задаёт начальное положение объекта, выполняет настройку и запуск таймера.

Начальное значение поля **X** структуры **rec**, определяющее положение левого верхнего угла битового образа (движущейся картинке), – отрицательное число (строка 51), равное ширине битового образа. Потому в начале не виден, картинка прорисовывается за границей видимой области.

С каждым событием **Tick** значение координаты **X** увеличивается, и на экране появляется та часть битового образа, координаты которой больше 0. Таким образом у наблюдателя создается впечатление, что самолет вылетает из-за левой границы окна.

Скорость полета определяется периодом следования сигналов от таймера (значение свойства **timer1->Interval**) и величиной приращения координаты по **X** (строка 60).

Процедура обработки сигнала от таймера выполняет основную работу: стирает изображение объекта (восстанавливает фоновый рисунок), затем выводит изображение объекта на новом месте. Восстановление фона выполняет метод **DrawImage** путём вывода фрагмента фонового рисунка (битового образа **sky**) в область графической поверхности, в которой находится в данный момент объект.

Задание 8.5. Анимация на основе битовых образов (масок)

8.5.а. В новом CLR-проекте расположите элемент **PictureBox** так, чтобы он покрывал всё рабочее поле формы.

Реализуйте код конструктора формы и обработчика события **Paint** для элемента **PictureBox** из листингов 8.11 и 8.12 соответственно.

Скопируйте в папку **Debug** проекта необходимые файлы **sky.bmp** и **plane.bmp**.

Проверьте работу программы.

8.5.б. Добавьте на форму в новом CLR-проекте элемент **Timer**. Реализуйте код конструктора формы и обработчика события **Tick** из листингов 8.13 и 8.14 соответственно.

Скопируйте в папку **Debug** проекта необходимые файлы **sky.bmp** и **plane.bmp**.

Проверьте работу программы.

Индивидуальное задание №8

Реализуйте средствами языка C++ назначенный преподавателем вариант **индивидуального задания №8** на программируемую графику. Реализуйте общие требования для всех вариантов и условие индивидуального варианта (с учётом комментариев к нему).

```

15 public ref class MyForm : public System::Windows::Forms::Form
16 {
17 public:
18     MyForm(void)
19     {
20         InitializeComponent();
21         try
22         {
23             sky = gcnew System::Drawing::Bitmap
24                 (Application::StartupPath + "\\sky.bmp");
25             plane = gcnew System::Drawing::Bitmap
26                 (Application::StartupPath + "\\plane.bmp");
27         }
28         catch (System::Exception^ e)
29         {
30             MessageBox::Show(
31                 "Ошибка загрузки битового образа: "
32                 + e->Message,
33                 "Полет",
34                 MessageBoxButtons::OK,
35                 MessageBoxIcon::Error);
36             this->Close();// закрываем форму
37             return;
38         }
39         // сделать прозрачным фон
40         plane->MakeTransparent();
41         // установить размер формы равным
42         // размеру фонового рисунка
43         this->ClientSize = sky->Size;
44         // задать фоновый рисунок формы
45         this->BackgroundImage = gcnew Bitmap(sky);
46         // определяем графическую поверхность g
47         g = this->CreateGraphics();
48         // инициализация генератора случ. чисел
49         rnd = gcnew System::Random();
50         // исходное положение самолета
51         rct.X = -40;
52         rct.Y = 20 + rnd->Next(20);
53         rct.Width = plane->Width;
54         rct.Height = plane->Height;
55         /*
56         скорость полета определяется периодом
57         следования сигналов от таймера и величиной
58         приращения координаты X
59         */
60         dx = 2; // скорость полета – 2 пиксела/тик_таймера
61         timer1->Interval = 20;
62         timer1->Enabled = true;
63     }
64 private:
65     Bitmap^ sky;
66     Bitmap^ plane;
67     // рабочая графическая поверхность
68     Graphics^ g;
69     // приращение координаты X, определяет скорость полета
70     int dx;
71     // область, в которой находится объект
72     Rectangle rct;
73     // генератор случайных чисел
74     System::Random^ rnd;

```

```

122 private: System::Void timer1_Tick(System::Object^ sender,
123                                 System::EventArgs^ e) {
124     g->DrawImage(sky,
125                 Rectangle(rct.X, rct.Y, rct.Width, rct.Height),
126                 Rectangle(rct.X, rct.Y, rct.Width, rct.Height),
127                 GraphicsUnit::Pixel);
128     // вычислить новое положение объекта
129     if (rct.X < this->ClientRectangle.Width)
130         rct.X += dx;
131     else
132     {
133         // объект достиг правой границы,
134         // перемещаем его к левой границе
135         rct.X = -40;
136         rct.Y = 20 + rnd->Next(40);
137         // скорость полета от 2 до 5
138         // пикселей/тик_таймера
139         dx = 2 + rnd->Next(4);
140     }
141     g->DrawImage(plane, rct.X, rct.Y);
142 }

```

СПИСОК ЛИТЕРАТУРЫ

1. ISO/IEC 14882:2020. Programming languages — C++.
2. ГОСТ 19.701-90. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.
3. Абрамов С.А. и др. Задачи по программированию. – М.: Наука. Гл. ред. физ.-мат. лит., 1988. – 124 с.
4. Культин Н.Б. Microsoft Visual C++ в задачах и примерах. – СПб.: БХВ-Петербург, 2010. – 272 с.
5. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб.: Питер, 2003. – 461 с.
6. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд. – М.: ООО «И.Д. Вильямс», 2016. – 1328 с.
7. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, - СПб.: ООО «Альфа-книга», 2017. – 432 с.
8. Справочник по языку C++ [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/cpp/cpp/> (дата обращения 10.01.2022).
9. Visual Studio IDE [Электронный ресурс]. URL: <https://visualstudio.microsoft.com/ru/> (дата обращения 10.01.2022).
10. Программирование .NET с использованием C++/CLI – Microsoft Docs [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/cpp/dotnet/dotnet-programming-with-cpp-cli-visual-cpp> (дата обращения 10.01.2022).
11. Пространство имён System.Drawing – Microsoft Docs [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/dotnet/api/system.drawing> (дата обращения 10.01.2022).
12. Пространство имён System.Drawing.Drawing2D – Microsoft Docs [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/dotnet/api/system.drawing.drawing2d> (дата обращения 10.01.2022).

СВЕДЕНИЯ ОБ АВТОРАХ

Рысин Михаил Леонидович, к.п.н., доцент, работает на кафедре математического обеспечения и стандартизации информационных технологий Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет», автор более 60 научных и учебно-методических работ;

Сартаков Михаил Валерьевич, к.т.н., доцент кафедры математического обеспечения и стандартизации информационных технологий Института информационных технологий ФГБОУ ВО «МИРЭА – Российский технологический университет», автор более 40 научных и учебно-методических работ;

Макеева Оксана Валерьевна, к.т.н., проректор по качеству образования и дистанционным технологиям АНО ВО «Московский региональный социально-экономический институт», автор более 60 научных и учебно-методических работ.