

МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

И. Ю. СИТНИКОВ Т. Е. САМСОНОВА П. Н. КОЗЛОВА

**ЛАБОРАТОРНЫЙ ПРАКТИКУМ
ПО КУРСУ «ИНФОРМАТИКА»**

учебное пособие

Санкт-Петербург
2022

УДК 004.42

ББК 32.973

Л12

Ситников И. Ю., Самсонова Т. Е., Козлова П. Н.

Л12 Лабораторный практикум по курсу «Информатика»: учеб. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2022. 75 с.

ISBN 978-X-XXXX-XXXX-X

Содержит пояснения и рекомендации по выполнению лабораторных работ по курсу «Информатика» для студентов 1 курса кафедры РЭС. Подробно рассмотрен порядок изучения языка программирования, приведены примеры выполнения отдельных этапов работ. Лабораторные работы выполняются в течение первого семестра и рассчитаны на приобретение навыков выполнения проектов по созданию ПО и программирования на языке C++ в среде Microsoft Visual Studio для ОС Windows.

УДК 004.42

ББК 32.973

Рецензенты:

Утверждено
редакционно-издательским советом университета
в качестве учебного пособия

ISBN 978-X-XXXX-XXXX-X

© СПбГЭТУ «ЛЭТИ», 2022

ВВЕДЕНИЕ

Программирование на языке Си и C++ стало важной составной частью инженерного проектирования радиоэлектронных устройств. Современные средства разработки программ представляют собой полнофункциональные комплексы, имеющие в своем составе высокопроизводительные компиляторы, компоновщики и отладчики кода. Они позволяют инженеру освоить основы программирования, начиная с близких к технической реализации вычислительных устройств примеров, вплоть до новейших стандартов объектно-ориентированного программирования.

На сегодняшний день существует множество литературы по программированию. Задачей настоящего курса было детальное знакомство с низкоуровневыми технологиями написания программ на языке Си, который как правило используется в микропрограммных устройствах. В пособии большое внимание уделено изучению особенностей проектирования приложений на Си и введению в объектно-ориентированное программирование с учетом базовых возможностей C++.

В данный цикл входят 7 лабораторных работ, посвященных исследованию различных программ, созданию собственных библиотек и архитектуре комплексных проектов на их основе.

Данное учебное пособие ориентировано на использование версии Microsoft Visual Studio Community 19, доступной для индивидуального использования студентами.

1. СТРУКТУРА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

Ниже приведена общая структура отчета по лабораторной работе:

- титульный лист;
- содержание;
- спецификация задания;
- формализованное словесное описание алгоритма решения задачи;
- блок-схема алгоритма;
- выбор и обоснование типов переменных, разработка структур данных;
- вводимые и выводимые параметры и их типы;
- структура проекта, перечисление нужных файлов;
- текст программы и файлов заголовков с комментариями, синтаксический разбор текста в комментариях;
- рисунки с копиями экрана при работе программы;
- контрольный пример, сравнение результата с эталоном;
- выводы, включающие область применения, ограничения, достоинства и недостатки программы, размер исполняемого файла на диске.

Рассмотрим некоторые разделы отчета более подробно.

1.1. Титульный лист

Для оформления отчетов целесообразно использовать программы редактирования текста, такие как: Microsoft Word, электронные ресурсы <https://docs.google.com/>, OpenOffice (<https://www.openoffice.org/>) и подобные.

Пример оформления титульного листа приведен в Приложении.

1.2. Содержание

Содержание должно перечислять разделы отчета с указанием страниц.

Это можно сделать автоматически, для этого откройте меню *Ссылки* и выберите *Оглавление*. «Оглавление» следует переименовать в «содержание» как это принято в технической литературе. Для автоматического заполнения нужно добавить заголовки в содержание. Заголовки должны отличаться стилем. Стил заголовка можно выбрать из списка или добавить собственный стиль, если подходящего нет в шаблоне документа, которым вы пользуетесь. Обновить поля содержания – номера страниц – можно автоматически, выделив в редакторе весь текст (*Ctrl+A*) и нажав «обновить поле» в меню по правой кнопке мышки на выделенном тексте.

Разделы в отчете нумеруются начиная с 1, раздел «Содержание» номера не имеет.

1.3. Спецификация задания

В данном разделе отчета кратко формулируются требования к разрабатываемой программе. К ним относятся:

- в какой операционной системе должна работать программа;
- в какой среде разрабатывается программа;
- на каком языке программирования написана;
- какие вычисления и какие действия должна выполнять программа.

1.4. Формализованное описание алгоритма решения задачи

В разделе приводится словесное описание метода решения, объясняются используемые элементы, переменные, и функции. Описывается последовательность выполнения вычислений, объясняется, как и откуда передаются данные для вычислений на каждом шаге по схеме:

- что дано в задании, описание исходных данных и начальных условий;
- объект, описание переменных, структур и атрибутов;
- последовательность действий над объектом, какие действия выполняются и при каких условиях;
- повторение действий;
- полученный результат.

Подробный словесный алгоритм решения задачи не должен быть описанием программы. Это действия, которые может сделать, например, человек с

калькулятором. А разрабатываемая программа автоматизирует процесс выполнения разработанного алгоритма.

1.5. Блок-схема алгоритма

Блок-схема нужна для представления алгоритма в виде последовательности операций и переходов между ними. Основные вычислительные алгоритмы содержат повторяющиеся действия, при этом повторение не должно быть бесконечным – тогда программа никогда бы не заканчивалась и не позволяла получить результат. Существуют и длительно выполняющиеся программы – например, программы игр или операционных систем, которые работают все время, пока их не выключают. Это алгоритмы другого типа, по сути, не являющиеся вычислительными. В нашем курсе мы будем создавать программы конечными, предназначенными для решения конкретной задачи: когда результат получен, вычислительный алгоритм тоже заканчивается.

В наших блок-схемах появляются блоки «НАЧАЛО» и «ЗАВЕРШЕНИЕ», по правилам оформления блок-схем это прямоугольники со скругленными углами. У «НАЧАЛА» один выход и ни одного входа, у «ЗАВЕРШЕНИЯ» – один вход и ни одного выхода. Другие блоки содержат один вход и как минимум один выход. Если у блока нет ни одного выхода, он «висит в воздухе», то про-

грамма, попав в этот блок, останется там навсегда – **проверяйте ваши схемы на наличие блоков, «висящих в воздухе»** [1].

Блоки соединяются линиями, показывающими последовательность действий. Линия, соединяющая блоки, должна быть со стрелкой, показывающей, откуда и куда переходит действие.

Перед началом вычислений нужно ввести исходные данные, а в конце работы вывести результат. Блоки ввода и вывода изображаются параллелограммами.

Наконец, в алгоритме бывают ветвления, то есть выполнение действий может идти, например по двум путям (простое условие). Блок проверки условия содержит описание проверки (что проверяется) и два выхода, которые подписываются «ДА» и «НЕТ» и соответствуют результату проверки. Пример блок-схемы приведен на Рис. 1.

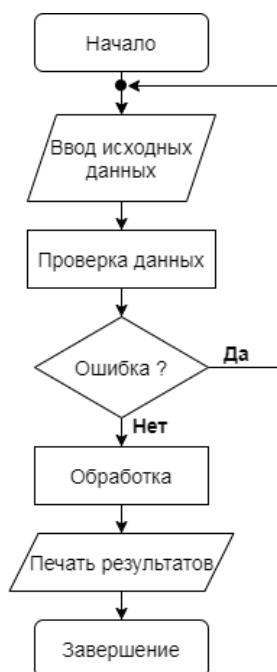


Рис. 1. Блок-схема алгоритма

1.6. Выбор и обоснование типов переменных.

В программе вычисления выполняются над переменными – числами, записанными в памяти компьютера, и называемыми операндами. Например, « $C=A+B$ » – простое выражение, которое означает: взять число из ячейки A , сложить его с числом из ячейки B и записать результат в ячейку C . В этой записи, несмотря на ее простоту, есть неопределенность для программы, заключающаяся в том, что числа в программе могут быть представлены в различных форматах, проще говоря, занимать разное по размеру место в памяти машины, и это может существенно влиять на результат. В дальнейшем мы рассмотрим детально эти представления и правила преобразования типов, однако отчет должен точно описывать используемые в программе типы переменных для правильной работы с обоснованием использования выбранного типа. Например: A , B и C имеют целочисленный тип со знаком *int*. Кроме того, при использовании сложных типов данных раздел должен содержать описание структур данных, используемых в программе.

1.7. Вводимые и выводимые параметры и их типы

В этом разделе приводится полный перечень данных, которые программа получает от оператора, в нашем случае, через ввод с клавиатуры. Также должен быть описан вывод результатов: какие условные обозначения, сообщения и другую вспомогательную информацию может выдавать программа. В отчете необходимо также привести необходимые пояснения.

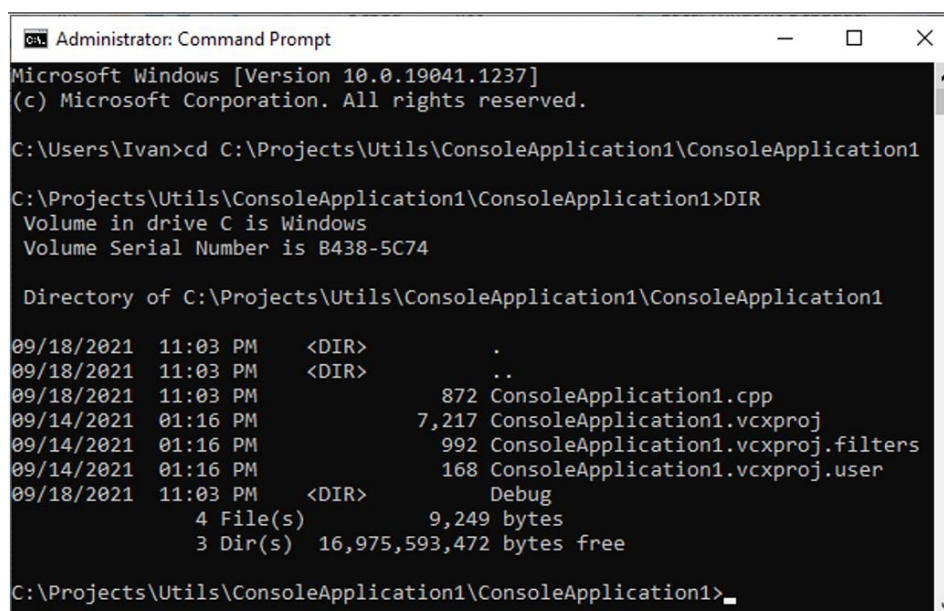
Например, в программе вводятся слагаемые A и B , оба представляют собой целые числа со знаком, не превышающие по модулю значения 200000000. Результат расчета выводится в строке « $A+B=$ », и программа ожидает нажатия клавиши оператором для завершения.

1.8. Структура проекта, перечисление нужных файлов

Программа разрабатывается в современной среде разработки Microsoft Visual Studio. Это очень хороший комплекс программ и вспомогательных средств для профессионального программирования. Создание программы – это написание текста на одном из языков программирования, мы будем использовать языки Си и C++. Сам текст является только средством для облегчения восприятия человеком последовательности машинных команд, которые «видит» машина в программе. Поэтому текст с языка Си должен быть

«переведен» на язык машинных команд. Этот перевод должен быть не вольным, а точным, процесс перевода называется компиляцией, а выполняет его специальная программа – компилятор. Для нас важно, что тексты в программе – это текстовые файлы на языке Си или C++, их имя может быть любым (избегайте использования кириллицы), а расширение *.cpp* для текста C++ или *.c* для классического Си. Кроме файлов текста, которые обычно содержат тексты выполняемых операций, в проекте есть файлы заголовков, которые в основном являются пояснениями компилятору, какие определения программист использует в своих текстах. Файлы заголовков имеют расширение *.h* от английского *headers*. Ну, и раз текстов уже несколько, должен быть файл проекта, который содержит в себе как минимум перечень файлов текста, файлов заголовков, а также инструкции, в какую форму наш проект должен исходные файлы собрать. Расширение этого файла проекта *.vcxproj* от *Visual C Extended Project*. Этап сборки скомпилированных файлов и приводит в случае успеха к созданию исполняемого файла, с расширением *.exe* (*executable*). Исполняемый файл не универсален, он строится с учетом особенностей операционной системы и, как правило, не работает в другой операционной системе (ОС).

В ОС есть возможность выводить содержимое папки – команда *DIR*. Для запуска команды *DIR* нужно запустить окно командного процессора ОС. В строке поиска набираем «CMD» и открывается окно, (Рис. 2).



```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.19041.1237]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Ivan>cd C:\Projects\Utils\ConsoleApplication1\ConsoleApplication1

C:\Projects\Utils\ConsoleApplication1\ConsoleApplication1>DIR
Volume in drive C is Windows
Volume Serial Number is B438-5C74

Directory of C:\Projects\Utils\ConsoleApplication1\ConsoleApplication1

09/18/2021  11:03 PM    <DIR>          .
09/18/2021  11:03 PM    <DIR>          ..
09/18/2021  11:03 PM                872 ConsoleApplication1.cpp
09/14/2021  01:16 PM             7,217 ConsoleApplication1.vcxproj
09/14/2021  01:16 PM             992 ConsoleApplication1.vcxproj.filters
09/14/2021  01:16 PM             168 ConsoleApplication1.vcxproj.user
09/18/2021  11:03 PM    <DIR>          Debug
               4 File(s)              9,249 bytes
               3 Dir(s)  16,975,593,472 bytes free

C:\Projects\Utils\ConsoleApplication1\ConsoleApplication1>
```

Рис. 2. Окно команд со структурой проекта

Находим в Visual Studio папку, где расположены файлы исходного кода – для этого в закладках меню по правой кнопке позволяет скопировать путь к файлу (Рис. 3):

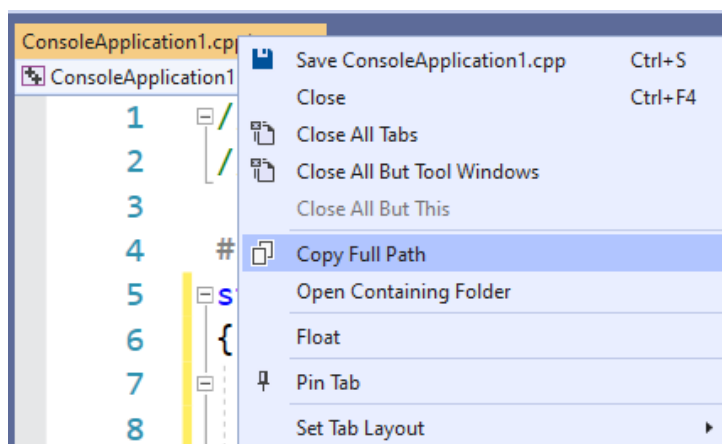


Рис. 3. Меню копирования пути к файлу исходного кода

Переходим в рабочую папку проекта командой *cd*:

```
cd C:\Projects\Utils\ConsoleApplication1\ConsoleApplication1
```

Набираем «*DIR*» и получаем список с вложенными папками. Ваша задача: скопировать список (выделяем мышкой и *Ctrl+C*) и поместить его в отчет, отредактировав ненужные файлы.

Вопрос студентам: как автоматизировать создание списка файлов проекта в вашем проекте?

1.9. Результаты лабораторной работы

К результатам работы относятся характеристики разработанной программы, такие, как:

- тип и версию ОС, для которой разработано приложение;
- тип приложения (консольное, оконное и т. д.);
- путь, имя и размер исполняемого *.exe* файла;
- ограничения на исходные данные;
- результаты проверки выходных значений по независимому источнику (обязательно приведите копии экрана);
- достоинства и недостатки программы.

2. СОЗДАНИЕ ПРОЕКТА В СРЕДЕ РАЗРАБОТКИ MICROSOFT VISUAL STUDIO

Visual Studio Community 2019 можно загрузить по ссылке ниже:

<https://visualstudio.microsoft.com/ru/vs/>

При установке нужно указать следующие компоненты: C++

Запустите программу после ее установки, откроется окно, представленное на Рис. 4.

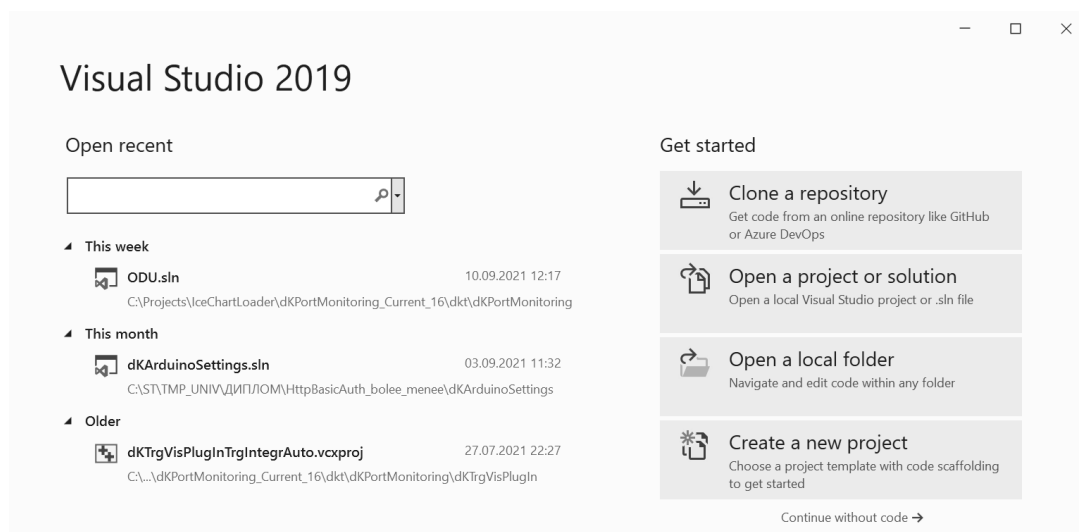


Рис. 4. Запуск Visual Studio 2019

В этом окне можно открыть существующий файл или проект, создать новый или «продолжить без кода». Главный вид окна программы представлен на Рис. 5.

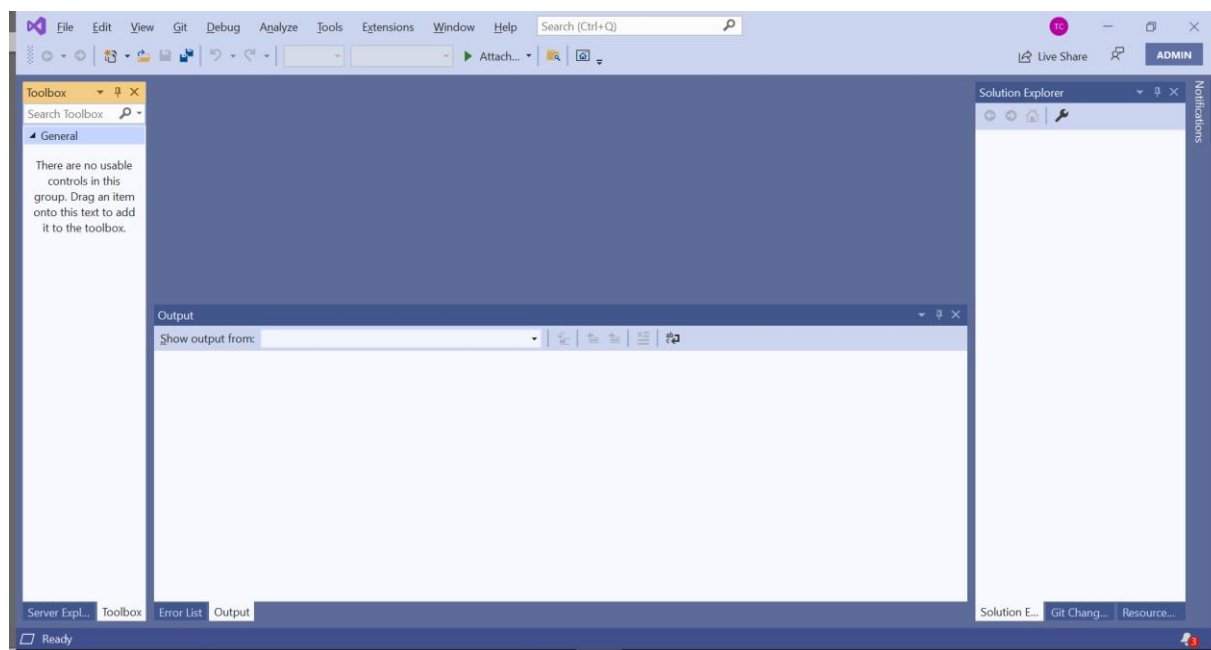


Рис. 5. Внешний вид главного окна Visual Studio 2019

Для создания проекта перейдите в *File -> New -> Project* (Рис. 6).

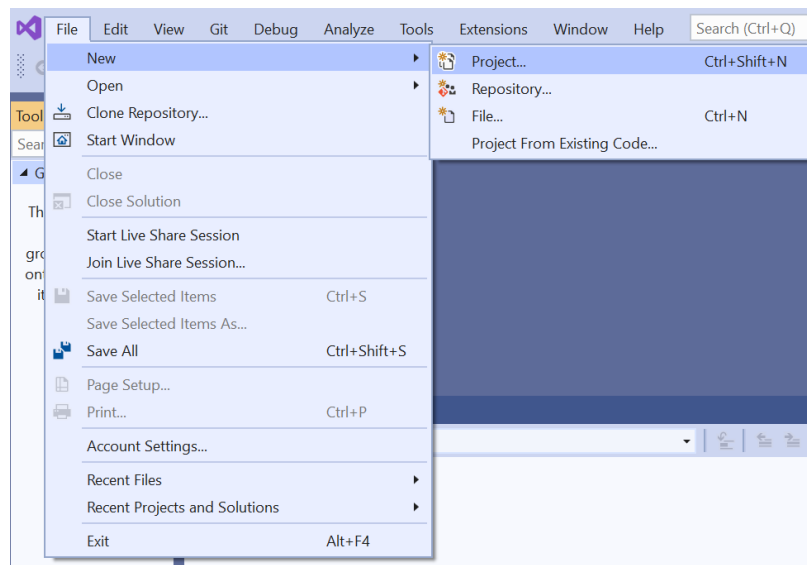


Рис. 6. Создание нового проекта

Выберите тип проекта: в данном курсе это должно быть консольное приложение для Windows (Рис. 7). Язык программирования – Си/C++.

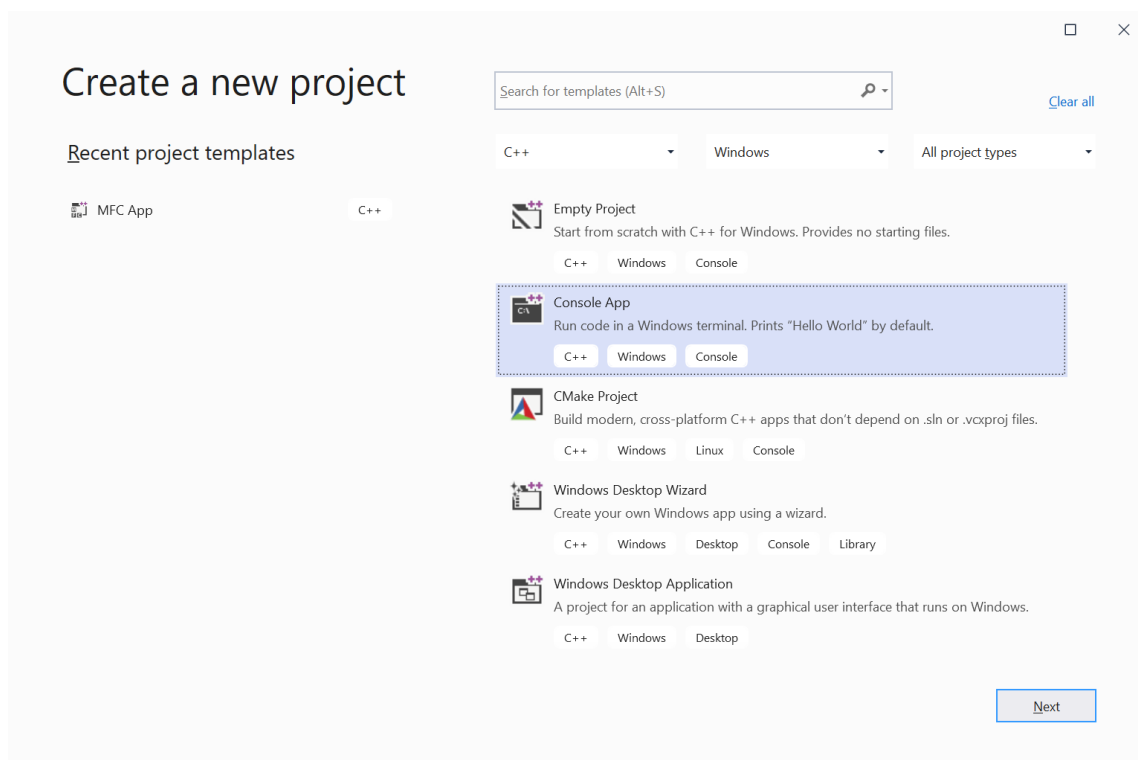


Рис. 7. Выбор шаблона консольного приложения для Windows

Далее укажите имя проекта и путь к папке, в которой будет храниться проект (Рис. 8).

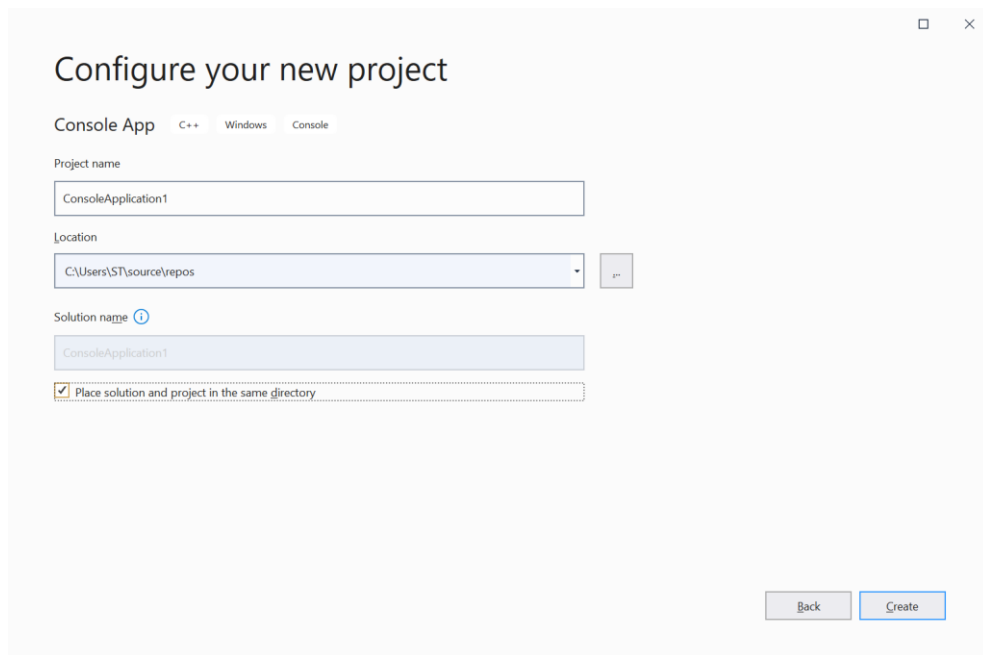


Рис. 8. Конфигурация проекта

Завершите создание проекта, нажав кнопку «Create».

В центральной части окна программы («1» на Рис. 9) откроется на редактирование исходный текст в *cpp*-файле из вашего проекта.

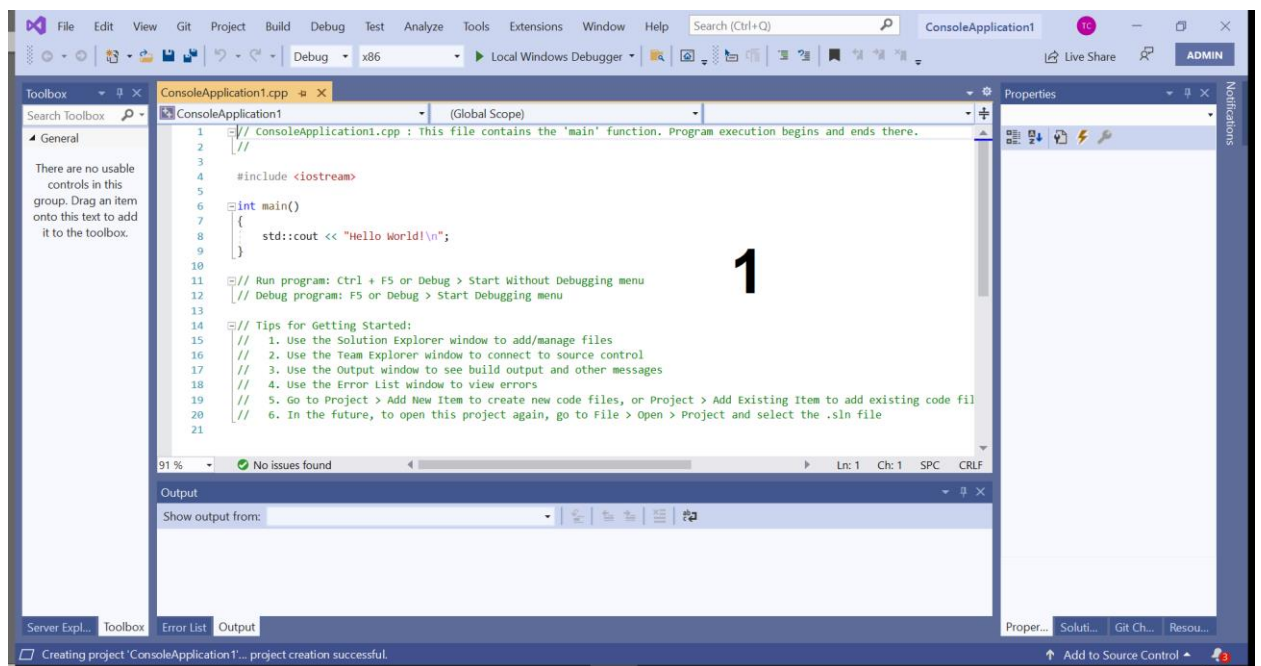


Рис. 9. Начало работы

Компиляция проекта запускается нажатием комбинации клавиш *Ctrl+F7*. С помощью локального отладчика Local Windows Debugger также можно запустить отладку с последующим просмотром результатов работы написанной программы (Рис. 10).

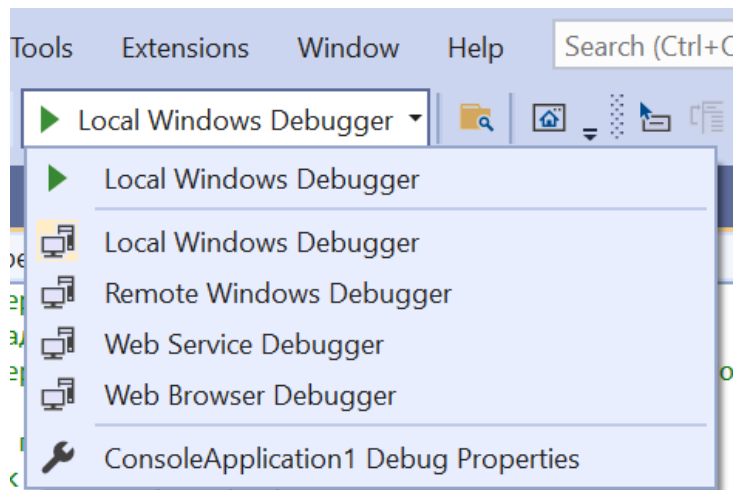


Рис. 10. Локальный отладчик Windows

Ошибки и предупреждения, возникающие при компиляции, можно посмотреть в окне «Список ошибок» (*Error List*), которое обычно находится в нижней части главного окна («1» на Рис. 11). Также, и даже несколько проще, использовать окно «Вывод» (*Output*).

Местоположение ошибок и/или комментариев в тексте программы также отображается цветными квадратиками в области слайдера вертикальной прокрутки страницы справа в окне с текстом.

Структура проекта представлена в окне *Solution Explorer* («2» на Рис. 11).

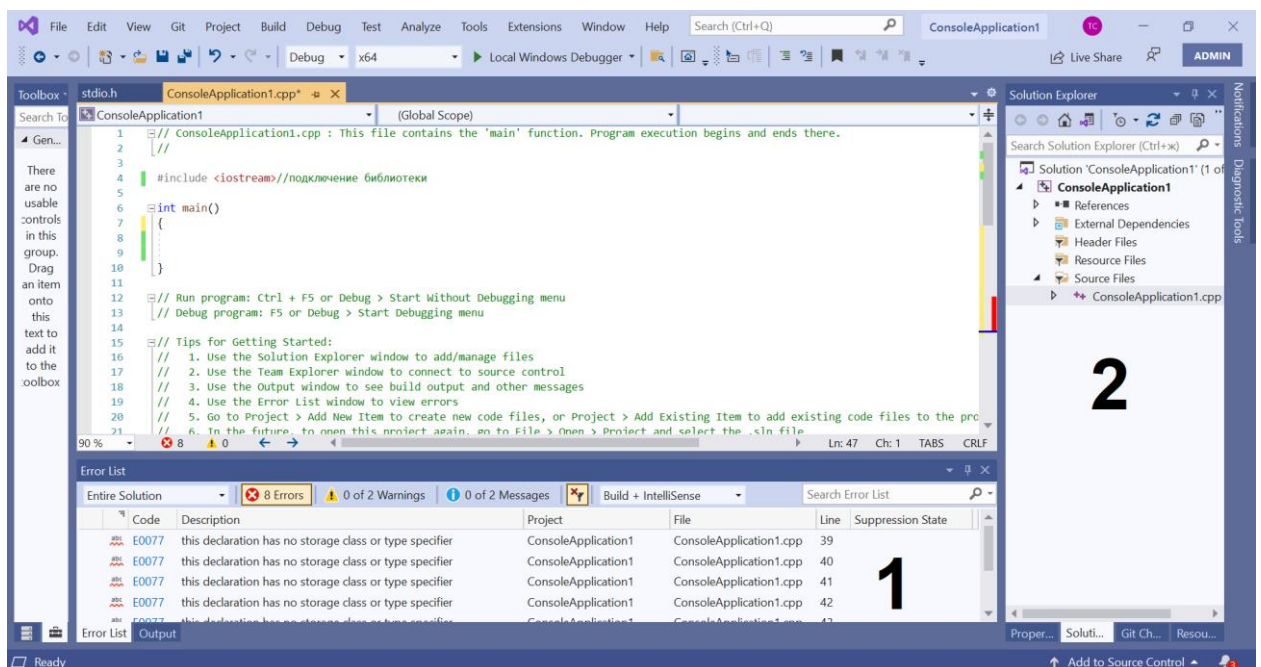


Рис. 11. Структура проекта и список ошибок

Чтобы посмотреть, как работает программа в конкретной строчке кода (и попадет ли она вообще в эту точку), нужно создать в этом месте точку останова. Для этого перед запуском отладчика нужно щелкнуть левой кнопкой

мышью на серой вертикальной линии окна редактирования текста рядом с номером интересующей вас строки кода («1» на Рис. 12).

Тогда после запуска отладки, если программа дойдет до этого места в коде, на красной точке появится желтая стрелка, («2» на Рис. 12).

Текущие значения переменных отображаются в окне «Локальные переменные» («Local», «3» на Рис. 12).

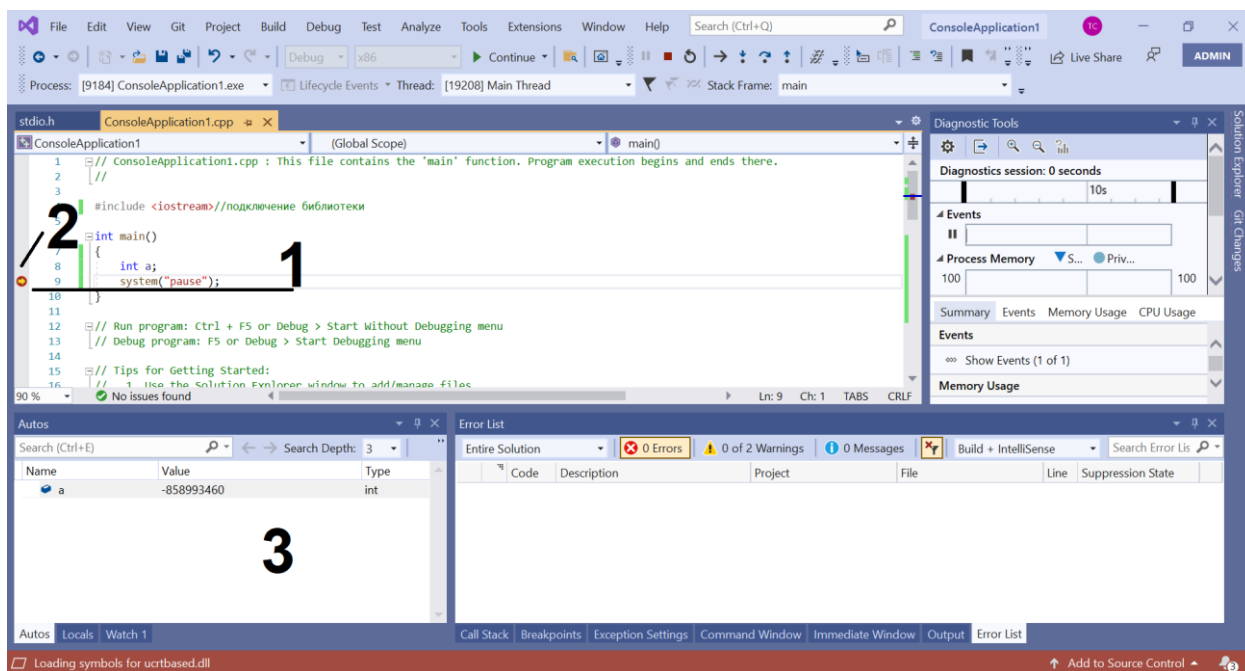
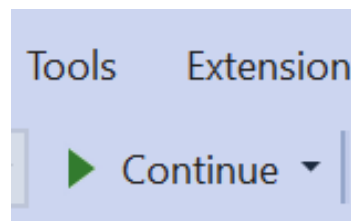


Рис. 12. Создание точки останова

Перемещаться по коду относительно точки останова можно при помощи кнопок, расположенных в верхней части главного окна (Рис. 13, а). Данные кнопки становятся доступными только после запуска отладки и при наличии точек останова.



а



б

Рис. 13. Инструменты для отладки программы

Продолжить выполнение программы после завершения работы с точками останова можно, нажав «Continue» (Рис. 13, б).

Пример программы с синтаксическим разбором:

```
#include process.h" //для работы с системными командами
int main()
```

```

{ //инициализация переменных
    short int a = -12345; //числовая переменная – целое число со знаком из
диапазона [-32768, +32767]
    int b = 123456; //числовая переменная – длинное целое число со знаком из
диапазона [-2 147 483 648, +2 147 483 647], можно использовать long см.
limits.h
    char c = -123; //числовая переменная – маленькое целое число из диапа-
зона [-127, 128]
    char d[5] = "1234"; //символьный массив, строка, нумерация символов с
0, в конце строки символ ноль, в тексте можно использовать d[4]=0; или
d[4]='\0';
    unsigned int e = 12345; //числовая переменная – целое число без знака из
диапазона [0, + 4 294 967 295]
    float f = -1234.5; //числовая переменная – вещественное число с плаваю-
щей запятой в диапазоне 3.4e-38 ... 3.4e+38 примерно 7 значащих цифр, dou-
ble – тоже число с плавающей точкой, примерно 15 значащих цифр, диапа-
зон 1.7e-308 ... 1.7e+308
    bool g = 1; // логическая переменная – значение true(1)/false(0)
    bool i = false;
    char n[3] = "12";
    //операции над переменными
    int h = a + e; //сложить две целочисленные переменные со знаком и без
знака
    long j = b + c; //сложить две целочисленных переменных со знаком и без
знака разной длины
    bool k = g || i; //выполнить логическое ИЛИ над переменными g и i
    int v = d[3] * n[0]; //умножить код (52) четвертого символа (4) массива
d на код (49) первого символа (1) массива n
    //напечатать в консоли значение переменных с пояснительными ком-
ментариями
    printf("Znakomstvo s Visual Studio");
    printf("\n\n");
    printf("h= %i", h); //как десятичное число целого типа со знаком
    printf("\n"); //перевод строки
    printf("v= %i", v); //как десятичное число целого типа со знаком
    printf("\n"); //перевод строки
    printf("k= %d", k); //как десятичное число целого типа со знаком

```

```
}  
    system("pause");//ждем действий оператора  
}
```

Результат работы примера программы представлен на Рис. 14.

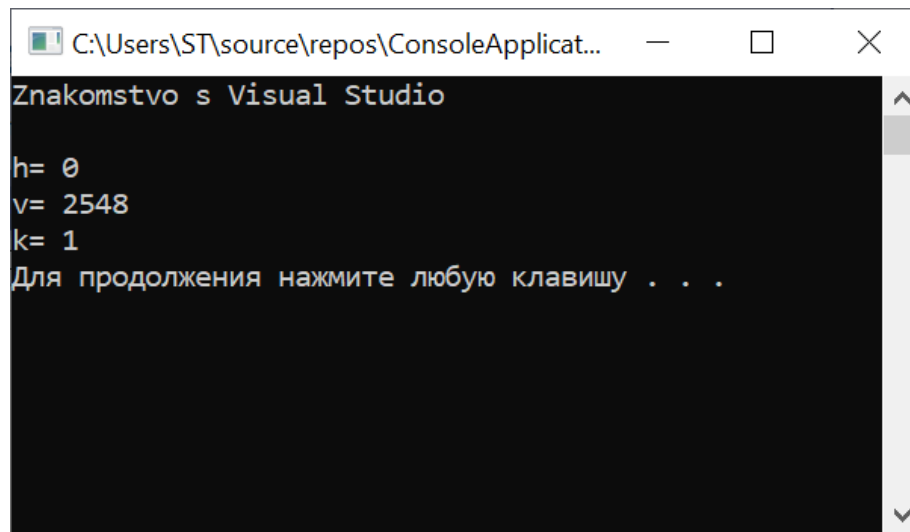


Рис. 14. Результат работы программы.

Обратите внимание, что над числовыми переменными разных типов можно выполнять математические операции, однако **необходимо учитывать какого типа и размера может получиться результат вычислений**. Так, например значение переменной «v» согласно логике работы программы получилось равным 2548, а не 4.

3. ЭЛЕМЕНТЫ ЯЗЫКА Си

Наше восприятие текста (а числа у нас – часть текста) не делает разницы между символом и его значением. Совсем другое дело – вычислительная система. Как мы выяснили там циркулирует только двоичные коды, а поскольку в наших программах встречаются различные данные, в языке Си обязательно определяются типы данных при объявлении переменных. Например:

```
int a,b;//объявляет целочисленные переменные a и b.
```

Здесь:

- разделитель, точка с запятой «;»;
- перечисление, запятая «,»;
- комментарии: «/* комментарий */» или «// комментарий до конца строки».

3.1. Типы данных

Для логических операций используется булевский тип данных *BOOL*, который в Си представлен целым числом и имеет два значения *TRUE* (1) и *FALSE* (0).

Целый тип данных *int* – это число со знаком, которое в нашей (x86) системе имеет 32 двоичных разряда. Диапазон от –2 147 483 648 до +2 147 483 647. Из них один старший разряд играет роль знака (ведь в двоичном коде нет «минуса»): если значение старшего разряда 1 – это «минус», число отрицательное, если 0 – неотрицательное число.

Целые могут быть: короткие *short* – 16 двоичных разрядов, *long* – 32 двоичных разряда (полный аналог *int*, *long long* – 64 двоичных разряда) и самые короткие *char* – всего один байт или 8 двоичных разрядов. Аналогия *char* – символ, и действительно, в простейшем случае стандартной ANSI-таблицы символов одного байта достаточно для кодирования латиницы, цифр и символов нашей клавиатуры.

Текстовые константы в коде: строка, “А” – в двойных кавычках, либо один символ из строки, ‘А’ – в одиночных кавычках

Для работы с перечисляемыми данными достаточно натуральных чисел. Тогда отпадает необходимость в использовании старшего разряда числа под знак, а диапазон значений становится от 0 до 4 294 967 295 (а комбинаций нулей и единиц столько же). Соответствующий тип *unsigned int* – беззнаковое целое, аналогично *unsigned short*, *unsigned long long* и *unsigned char*.

Правило: без *unsigned* все целые типы считаются числами со знаком.

Серьезный вопрос: что делает наша программа, когда выполняет действия с используемыми типами данных, например, при перемножении двух целых чисел типа *int* выполняется приведение результата к типу *int*. При этом достаточно большие числа 65 535 и 65 536 вызовут переполнение разрядной сетки. В программе следует для такого случая использовать результат с типом *long long* и прямо указать, что множитель имеет тот же тип (*long long*): $65535 \cdot 65536$ будет вычислено правильно. В Visual Studio тип `__int64` это аналог типа *long long*, который соответствует 64-битному целому числу.

3.2. Переменные

Переменная – это буквенно-цифровое обозначение операнда, то есть ячейки памяти, из которой будут читаться и в которую могут писаться значения и результаты вычислений. Допускаются буквы латиницы, цифры и символы подчеркивания, причем начинаться имя переменной должно с буквы. Заглавные и строчные буквы считаются разными в именах переменных:

int A; //объявление целой со знаком переменной A в программе, ее значение пока не определено.

3.3. Арифметические операции

В качестве основных арифметических операций можно выделить следующие:

- присвоение « $A = 2$ » (слева результат, справа значение, которое будет записано в A (2), это уже определение значения переменной);
- сложение « $+$ »;
- вычитание « $-$ »;
- умножение « $*$ »;
- деление « $/$ »;
- остаток от целочисленного деления « $\%$ ».

Это операции, выполняющиеся над парой переменных или констант (например, $A \cdot 2$ или $2 \cdot 2$), называются *бинарными*.

Кроме того, возможны операции с одной переменной – *унарные*. К таким, например, относится инкремент « $++$ » (увеличение на 1).

$A++$ и $++A$ – это префиксная и постфиксная формы инкремента.

Пример:

$A = 2;$

$A++;$ // A стало равно 3.

4. ЧИСЛА И ИХ ПРЕДСТАВЛЕНИЕ В МАШИННЫХ КОДАХ

Остановимся на представлении чисел в двоичном коде, то есть попробуем взглянуть на мир со стороны нашего вычислителя. Например, как выглядит число, соответствующее 2021 году в машинном представлении, в двоичном 32 битном коде. Двоичный код – позиционная система счисления, в которой старшие разряды находятся слева и весовой коэффициент каждого разряда равен степени числа «2» (основания системы счисления). Показатель степени на единицу меньше номера цифры в числе, считая от младших разрядов. То есть для цифры младшего разряда это $2^0 = 1$. Старшие разряды имеют коэффициенты: 2, 4, 8, 16, 32, 64....

Как можно разложить 2021 по разрядам в десятичной системе?

Например, делим число на весовой коэффициент 4-го разряда десятичной системы: $10^{(4-1)} = 1000$ и получаем 2. Далее остаток от деления делим на 10^2 , получаем 0 и 21 в остатке и так далее до 10^0 , в результате получим 2021.

Применим ту же технику к числу 2021 и двоичному коду. Так как наш старший разряд 32, то начинаем делить 2021 на 2^{31} – не делится, пишем 0 и в остатке 2021. Далее, пока не дойдем до 11-го разряда, результат будет 0, и только $2^{10}=1024$ даст при делении 1 и 997 в остатке. Продолжив, получим код 11111100101. Выполнение таких действий – скучная процедура, в ней слишком много повторяющихся действий. Хорошо бы их описать коротким текстом программы для нашей машины, чтобы та посчитала и напечатала двоичный код неотрицательного числа в консольном приложении.

Алгоритм в данном случае содержит выполнение повторяющихся действий, пока мы не пробежим по всем разрядам нашего числа от 32 до 1. Тут возникает необходимость в управляющих элементах языка Си.

4.1. Управляющие элементы языка Си

К ним относятся операторы условия *if*, переключатели *switch* и операторы цикла *for*, *do while*. Задачей элементов управления является выполнение перехода к следующему действию в программе в зависимости от определенных условий, которые считаются булевыми типами. Выражение проверки простого условия:

if (*A*)

B = 2;

Это означает, что программа проверит *A*, приведет его к булевскому типу, и, если *A* не 0 (любое число отличное от 0), то выполнится следующий за условием оператор – переменной *B* будет присвоено значение 2. То есть в ячейку

памяти *B* программа запишет число 2. Точка с запятой в *C* используется как символ конца операции. Если по условию выполняются несколько действий, то эти действия (тело цикла) должны быть заключены в фигурные скобки «{» и «}».

4.2. Блоки кода

Блоки в фигурных скобках группируют код и одновременно являются фрагментом кода, в котором могут определяться временные переменные. Этот блок будет областью видимости такой переменной.

Правило: объявленная в блоке переменная за пределами блока становится неопределенной.

Оператор проверки нескольких вариантов значений нашей переменной *switch (A) { case 1: B=2; break; case 2: B = 3; break; }* проверяет *A* на равенство значениям 1, а потом 2 и если выполнится одно из условий (компилятор не разрешит одинаковых проверок), то будет выполнен следующий за условием код. Здесь *break;* означает прекращение действий, выполняемых по условию *case*.

Оператор цикла *for* выполняет заданное количество действий:

```
for ( i=0; i<32; i++ ) { /* операторы, выполняемые в цикле */ }
```

Программа выполнит операции в фигурных скобках ровно 32 раза, где *i* – параметр цикла, целое число. Первый элемент цикла – задание начального значения – *i* (равного 0). Второй – проверка выполнения условия цикла *i<32* – пока это *TRUE* (истинно), цикл выполняется, а при *FALSE* (ложно) переходим к следующему за скобками действию. Третий – изменение параметра цикла на каждом проходе – в данном случае «*i++*» – инкремент параметра цикла, увеличение *i* на 1 при каждом проходе по циклу.

Часто допускают ошибку – ставят знак точку с запятой «;» между круглыми скобками и фигурными скобками:

```
for ( i=0; i<32; i++ ); { /* операторы, выполняемые в цикле */ }
```

Это создает так называемый пустой цикл, точка с запятой считается в нем оператором, который ничего не делает, он повторится 32 раза, после чего действие в скобках будет выполнено только один раз.

Для лучшего структурирования кода в языке определяется блок операций, в коде блок заключается в фигурные скобки:

```
{ /* переменные, которые определены в этом блоке, операторы, выполняемые в блоке */ }
```

Блок в свою очередь может содержать вложенные блоки. Количество вложений не ограничено. Вложенные блоки для лучшего восприятия кода принято смещать на табуляцию вправо, то есть «внутри» текста внешнего блока:

```
{//начало блока верхнего уровня  
int A; // переменная A определена и в этом и во вложенном блоках  
  
...  
{//начало вложенного блока  
int B; // переменная B определена только в этом блоке  
  
...  
}//конец вложенного блока, переменная B освобождается и далее //не  
определена  
}//конец блока верхнего уровня
```

Блок является хранилищем операций, выполняемых во фрагменте кода, называемом *функцией*. Функция, как правило, пишется для тех одинаковых действий, которые выполняются в различных частях программы.

4.3. Функции и блоки

Функции в языке C играют очень важную роль. Функция имеет имя и список параметров (аргументов), передаваемых в функцию, который заключается в круглых скобках:

```
main(int arg)  
{ /*блок: переменные и операторы, выполняемые в программе */  
} //пример функции.
```

Приведенная выше функция является обязательной. Это главная функция кода в консольном приложении, с которой начинается выполнение вашей программы. Эту функцию для нас помощник создает автоматически при создании проекта консольного приложения.

Ввод и вывод в консольных приложениях в Си выполняется функциями *scanf(<format>, address)* и *printf(<format>/ value)*. Эти функции являются частью стандарта языка Си и осуществляют много операций – чтение кодов нажатых клавиш при вводе, генерацию символов шрифта при выводе, освобождая вас от такой необходимости. Ввод и вывод в Си работает с переменными, а переменные имеют строго определенный тип, поэтому аргументы функций сообщают какой именно тип будет считываться с клавиатуры или выводиться на экран. Этот спецификатор типа называется *форматом* – вот откуда взялась буква *f* в конце имени функций. Функция ввода принимает

формат в качестве аргумента – это текстовая константа. Например, ввод и вывод целого десятичного числа имеет формат «%d».

Также при вводе мы должны передать в функцию адрес переменной:

scanf(“%d”, &A); //считывает с клавиатуры символы, пытается перевести их в десятичное число и записать результат в переменную A. Адрес переменной в языке Си – оператор «&». &A – получить адрес переменной.

printf(“%d”, A);//тут проще: взять значение из переменной A и напечатать его в текущую позицию курсора на экране консоли.

Для Visual Studio безопасным будет использование функций *scanf_s* и *printf_s*, в противном случае компилятор выдаст ошибку/предупреждение.

4.4. Задание на лабораторную работу № 1

Написать консольное приложение на языке Си, которое переводит десятичное число, введенное оператором, в 32 битный двоичный код с использованием функций ввода *scanf*, цикла *for*, оператора условия *if* и функции вывода *printf*.

5. ОТРИЦАТЕЛЬНЫЕ ЧИСЛА В ДВОИЧНОМ КОДЕ

Основной тип *int* – число со знаком, и мы уже знаем, что в самом старшем бите кода находится отведенный под знак числа разряд (знаковый разряд), который принимает значение 1 для отрицательных чисел. Разберемся, откуда происходит подобное представление.

5.1. Шестнадцатеричный код

Пусть дано целое положительное число 1, для сокращения записи ограничимся типом *char*, тогда двоичный код числа: 0000 0001. Этот код записан двумя порциями четырехразрядных чисел – *тетрадами*. Это удобно для представления шестнадцатеричного кода, в котором каждая цифра или буква соответствует тетраде, так что запись укорачивается до 0x01 – это то же самое число 1. Каждым разрядом шестнадцатеричного числа может быть цифра от 0 до 9 или буква от A до F (заглавная или прописная). Таким образом кодируются все 16 комбинаций в тетраде.

В качестве -1 подберем такое двоичное число, чтобы выполнялось арифметическое тождество $1 + (-1) = 0$. При выполнении суммирования двоичных чисел разряды складываются с переносом:

0000 0001

+

1111 1111

=

0000 0000 //перенос в 9 разряд, но его нет, так что единица «теряется»

Из этого следует, что число -1 выглядит как 1111 1111 в двоичном коде, или 0xFF в шестнадцатеричном представлении. Здесь обе тетрады имеют двоичный код 1111, это десятичное число 15, которому соответствует буква F в шестнадцатеричном коде.

Мы видим, что в числе, соответствующем -1 действительно 1 в старшем разряде. Тождество $-1 + 1 = 0$ именно так и выполняется двоичным сумматором, который есть в каждом процессоре каждого вычислителя.

Теперь обобщим наш результат для получения правила получения двоичного кода целого отрицательного числа. Чтобы получить ноль в сумме с таким же, но положительным числом, нужно все разряды положительного числа инвертировать (поменять 0 на 1, а 1 на 0 в каждом разряде). Однако, в этом случае сумма будет числом, во всех разрядах которого будут единицы. После этого к этому числу нужно прибавить еще 1, тогда в сумме получим 0 во всех

разрядах. Возникнет перенос в старшем разряде за пределы разрядной сетки, который не считается. Имеем тождество:

$$A + \sim A + 1 = 0,$$

где A – положительное число (и его старший разряд всегда 0), $\sim A$ – обратный код числа A (его поразрядная инверсия). Данное тождество можно переписать в виде:

$$A + (\sim A + 1) = 0,$$

или

$$A - A = 0,$$

значит

$$-A = \sim A + 1.$$

Такое представление отрицательных чисел называют *дополнительным кодом*, поскольку сперва из кода положительного числа получают обратный двоичный код, затем к нему прибавляют единицу, то есть дополняют единицей инверсию исходного кода. Интересно, как это сработает с числом 0, исходный код которого – 0000 0000, следовательно, обратный двоичный код (инверсия) – 1111 1111. В соответствии с полученным правилом, прибавляем к нему 1. Получаем 0000 0000 – то есть нуль взятый со знаком «минус» в двоичной арифметике остался тем же нулем – правильный результат.

Вернемся к нашему методу представления чисел в двоичном коде путем поразрядного деления числа на весовой коэффициент текущего двоичного разряда. По сути, этим действием мы определяли простую альтернативу: нуль или единицу нужно напечатать в данном разряде двоичного числа. Двигаясь от старшего 32-го разряда к младшему первому, мы таким образом заполняли нулями и единицами 32 разряда двоичного числа, которое и являлось ответом в задаче. Для отрицательных чисел такой алгоритм без модификаций не работает – деление будет давать отрицательное число, и результат будет неправильным.

Попробуем найти ответ на вопрос «нуль или единица стоит в данном разряде двоичного числа» другим способом. Используем наши знания о двоичной операции «И», которая дает в результате 1 только когда оба операнда равны 1. Это работает как для булевых переменных, так и для каждого из разрядов двоичного числа. Различие состоит в том, что операция «&&» сперва осуществит приведение чисел – операндов к булевскому типу (все числа отличные от нуля – *TRUE*, число 0 – *FALSE*). Операция «&» выполнит поразрядное логическое «И» (конъюнкция), то есть в каждом разряде первого и второго чисел будет

выполнено «И» только над этими значениями в данном разряде, и в этот же разряд будет записан результат операции «И».

Такое действие, относится к логическим операциям и выполняется одновременно со всеми 32 разрядами нашего числа в арифметико-логическом устройстве (АЛУ) нашего вычислителя.

Можем использовать эту операцию для ответа на вопрос: единица или ноль стоит в проверяемом разряде нашего двоичного числа. Для этого сформируем так называемую «маску» – специальное число, только в одном разряде которого (в проверяемом на данном шаге) выставляется 1, а во всех остальных разрядах – 0. Для 32-го разряда числа это 1000 0000 0000 0000 0000 0000 0000 0000 – двоичное число, которое в шестнадцатеричном коде будет гораздо компактнее – 0x80000000. Заметим, что старший разряд этого числа – 1, то есть число с типом *int* отрицательное. Для битовой маски есть смысл использовать беззнаковый тип *unsigned int* или *DWORD*, который исключает влияние знака на результат.

Тип *DWORD* так называется потому, что состоит из двух 16-разрядных слов, а каждое 16-разрядное слово – из двух байт. Когда вычислители были 16-разрядными, типы «байт» (*byte*) и «слово» (*word*) служили для различения 8-разрядных и 16-разрядных чисел-операндов. Для 32-разрядного вычислителя *double word* сократился в *DWORD*. Ну и в 64-разрядных вычислителях числа становятся *QWORD* (*Quadro word*) – четыре шестнадцатеричных слова, каждое по 2 байта – всего восемь байт в таком целом числе. Напомним, что в Си тип кодируется как *long long*.

Пока для 32-разрядной системы используем переменную типа *unsigned int*

```
Maska = 0x80000000; // начальное значение маски
```

Выполняем операцию проверки старшего бита числа (и совершенно неважно, положительное оно или отрицательное, арифметики в операции проверки теперь нет, мы заменили ее логической операцией, в которой никакого влияния на результат знак числа не оказывает, проверяются только биты):

```
if (A & Maska ) //поразрядное логическое «И», при этом само число A не
//меняется
    printf("1"); //печать 1 на консоли
else
    printf("0"); //печать 0 на консоли
```

Старший разряд мы проверили. Теперь хотим проверить 31-й разряд числа. Для этого маска должна стать 0100 0000 0000 0000 0000 0000 0000 0000.

Это соответствует результату деления *Maska* на 2, то есть можно написать унарную операцию:

Maska /= 2;

Но по виду маски можно просто сдвинуть единицу в ней вправо на 1 разряд. Оказывается, операция деления двоичного числа на 2 – это просто сдвиг всех его битов вправо на один разряд. В АЛУ эта операция называется *арифметический сдвиг вправо*, и символ ее в языке Си два символа «>>» подряд – «>>»:

Maska = Maska >> 1; //сдвинуть содержимое числа Maska вправо на 1 разряд, результат записать в эту же переменную Maska.

Короткая унарная запись того же действия:

Maska >>= 1;

В языке С есть и операция сдвига разрядов числа влево, ее синтаксическое обозначение: «<<». Например, сдвиг влево на 3 разряда можно записать так:

A <<= 3;

Логические операции проверки битов чисел в вычислителях часто называют проверкой «флагов». Аналогия заключается в следующем: каждый бит числа может отвечать за какой-то отдельный признак, например первый бит – включен мотор; второй бит – направление вращения мотора. Значение бита 1 похоже на символическое обозначение флага на флагштоке если нет ветра (флаг – хвостик у единицы, а флагшток – вертикальная линия). Процедуру проверки произвольного бита числа нужно оформить функцией.

5.2. Функции и структурирование кода

Функции позволяют структурировать код и, главное, позволяют повторно использовать написанный вами код как в той же программе, так и в других ваших программах. Представьте, что у вас есть десять мест в тексте программы, где нужно выполнить одно и то же действие. Можно скопировать текст уже написанного фрагмента во все эти места. Теперь представим, что потом выяснилось, что в первом фрагменте есть недочет, который нужно исправить. При копировании фрагментов вам придется найти и подправить все десять мест в тексте, а вот если фрагмент оформлен функцией, то исправления нужно сделать только внутри функции и только один раз. Вызываться эта функция будет в вашем коде по-прежнему десять раз, но ни искать их, ни исправлять уже не нужно.

Функцией называется блок кода, у которого есть тип и имя. Тип функции – это **одно** значение, которое функция может вернуть в вызывающий ее код:

int main(){ return 0;} //функция, типа int, которая возвращает значение 0 (нет ошибок) в конце блока.

Это главная функция нашей программы, и возвращается значение 0 во внешний код – в операционную систему. Очевидно, для этого нельзя использовать ячейки памяти нашей программы, ведь ее уже не будет в памяти машины. Технически осуществляется хранение целого числа из 4 байтов в памяти временного хранения данных, в стеке. Стек организован по принципу «последний пришел – первый вышел» (*Last In First Out – LIFO*). Такой принцип позволяет эффективно хранить переменные, как при вызове функций, так и временных, объявляющихся во вложенных блоках. Адрес таких переменных будет вычисляться относительно указателя стека.

Передача данных в функции и возврат данных из функций реализуются следующими описанными ниже способами.

5.3. Глобальные переменные

Объявляется переменная вне всех функций программы, обычно в верхней части текста (тогда по правилам эта переменная будет доступна для чтения и записи в любой точке программы, то есть становится глобальной). Способ простой, но имеет недостаток: место изменения значения такой переменной трудно отследить в длинном тексте. Если переменная нужна только для чтения, ей присваивается префикс *const*, и проблемы с поиском изменений переключаются на компилятор, который следит за переменными с префиксом *const*, изменять которые нельзя. Префикс *const* это аналог привычного «только для чтения» ограничения данных. Например:

const int Dim=10; //переменная для размера данных.

5.4. Параметры, передаваемые по значению

В списке аргументов функции через запятую перечисляются необходимые переменные с их именами и типами:

int MyFun(int A, int B); //пример функции, возвращающей целое и считывающей два целых параметра.

Такой способ называют «передачей параметров по значению», поскольку можно вызвать эту функцию, подставив вместо параметров числа:

MyFun(3, 5); //правильный вызов.

Параметры *int A*, *int B* в блоке функции становятся переменными:

*int MyFun(int A, int B){ return A*B;}*

И даже их значения можно менять в коде функции:

```
int MyFun(int A, int B){ A+=2; return A+B;}
```

Эта запись допустима, но при выходе из функции значения таких параметров будут потеряны, как и у всех локальных переменных в тексте функции.

5.5. Передача параметров по указателю

Параметры можно передавать в функцию по указателю. *Указатель* простой переменной – это адрес, в котором начинаются байты, занимаемые переменной данного типа в памяти. В Си указатель представляет собой тип данных, и поскольку разные типы данных занимают разное количество байт в памяти, указатель определяется как «тип со звездочкой»: *int ** – это тип «указатель на целое число». Если в функцию попадает не значение переменной, а ее адрес, то, изменяя значение по этому адресу, получим возможность возвращать в вызывающий код произвольное количество переменных нужных типов вдобавок к возвращаемому функцией значению.

*int MyFun(int * A, int * B);*//функция, получающая два параметра по указателю может изменить и вернуть оба параметра и еще один через return.

Вызвать ее со значениями не получится:

MyFun(3, 5);

Это вызовет ошибку компиляции. И действительно, передав адрес «3», мы попадаем в зарезервированную область памяти и получим остановку по возникшей исключительной ситуации (исключению) – нарушению прав доступа с кодом *0xC0000005*. В тексте функции менять *A* или *B* не будет ошибкой компиляции, но будет скорее всего ошибкой с нарушением памяти. Менять же значения переменных, хранящихся по указателю, можно безопасно:

**A = 2;*//запишет число 2 в ячейку по адресу A.

Синтаксис «*» и указатель на переменную называют разыменованием указателя, то есть при наличии звездочки перед указателем он становится полным аналогом обычной переменной, той, на адрес которой он указывает. Вызов такой функции возможен следующим образом:

int A1, B1; MyFun(&A1, &B1);//здесь & при переменной означает, получить адрес этой переменной.

Адреса при загрузке программы в компьютер могут различаться на соседних машинах, так как распределением памяти для программ занимается операционная система. Этот синтаксис мы видели в функции *scanf(&A)*.

5.6. Передача параметров по ссылке (C++)

Наконец в C++ вариант передачи переменной в функцию для ввода и вывода данных упростили, и это называют «передача данных по ссылке».

Технически это то же, что и передача по указателю, а синтаксически – проще при вызове и при работе с этими переменными в теле функции.

int MyFun(int &A1, int &B1);//функция с двумя параметрами, целыми, передаваемыми по ссылке.

Ссылка тоже тип данных и пишется тип и «&», например *int &* – ссылка на переменную целого типа:

int MyFun(int &A, int &B)

*{ A +=2; return A*B;}* //при выходе из функции первая переменная увеличивается на 2.

5.7. Отладка функций

Первым делом убедитесь, что ваша функция выполняется в программе. Для этого в начале исследуемой функции поставьте точку останова, см. Рис. 12. Если при запуске программы в режиме отладки произойдет остановка внутри функции, значит вызов функции в коде выполнен успешно. Правильным будет проверить значения аргументов, которые переданы в функцию. Для этого подведите курсор к аргументу или найдите значение в окне отладки *Locals* («Локальные переменные»).

Важный вопрос: из какого места кода была вызвана функция. Для этого выберете в панели отладки закладку *Call stack* («Стек вызовов»). Список вызовов интерактивный – по щелчку можно быстро перейти к строчкам, откуда была вызвана функция.

5.8. Задание на лабораторную работу № 2

Написать консольное Win32 приложение на языке C, которое переводит десятичное число со знаком, введенное оператором, в 32-битный двоичный код с использованием функций ввода *scanf_s* (это безопасный аналог *scanf*) цикла *for*, оператора условия *if* и функции форматированного вывода *printf*. Использовать метод проверки битов введенного числа поразрядной конъюнкцией (логической операцией «И») числа и «маски». Маска содержит только одну единицу в исследуемом разряде двоичного числа, а остальные биты маски – нули. Проверку произвольного бита числа оформить функцией.

5.9. Пояснения к лабораторной работе № 2

Требуется проверить каждый бит, так что маска будет выглядеть как набор из 31 нуля и 1 единицы. Единица сдвигается по разрядам по мере проверки каждого бита числа. Например, проверим первый бит числа 42:

int num = 42; //0...000101010

```
int mask = 1; //0...000000001
```

```
bool checkBit = num & mask; // checkBit = false, & – побитовый оператор «И».
```

Первый (самый правый) бит числа 42 равен нулю, так что в результате операции побитового «И» получаем ноль (*false*). Далее:

```
int num = 42; //0...000101010
```

```
int mask = 1<<1; //0...00010, сдвиг единицы в маске на один разряд
```

```
bool checkBit = num & mask; //checkBit = true.
```

Следующий бит числа 42 равен единице, так что *checkBit = true*.

Используя оператор сдвига «<<», можно проверить все 32 разряда числа, используя соответствующий цикл. По результату проверки *true* или *false* выводятся символы «1» или «0» соответственно. Целесообразно объединить их в одну строковую переменную. Обратите внимание, что при выполнении задания нужно начинать проверку со **старших разрядов**, чтобы вывести биты в правильном порядке (**слева направо**).

6. ИНДЕКСИРОВАННЫЕ ПЕРЕМЕННЫЕ

Набор переменных одного типа, которые представляют собой упорядоченный список, удобно представлять *переменной с индексом*. Индексированные переменные также часто называют массивами.

Массив – это пронумерованный набор переменных (*элементов*), фактически хранящийся в одной переменной. Доступ к отдельному элементу массива выполняется по его порядковому номеру, называемому *индексом*. А общее число элементов массива называется его *размером* [2].

Массивы удобны для хранения большого количества значений некоторого параметра. Например, температура воздуха в нашем городе по часам может храниться в массиве температур.

6.1. Классификация массивов

Основным параметром массива является его размер.

```
int A[32]; //объявлен массив из 32 целых чисел, выделена память 128 байт.
```

В зависимости от того, может ли размер массива изменяться во время исполнения программы или он задается на момент компиляции программы, массивы разделяют на *динамические* и *статические*. У статического массива размер это всегда число, оно не может быть переменной на момент компиляции.

В текстах встречаются конструкции, где размер массива указывается через подстановку макроопределением:

```
#define ARR_SIZE 32  
int A[ARR_SIZE]; // здесь тоже 32 элемента и это статический массив.
```

В предыдущей строке компилятору дается инструкция «во всем тексте ниже сделать контекстную замену: вместо символов *ARR_SIZE* подставить число 10» и только после этого компилировать полученный текст. Использование макроопределений удобно, если в нескольких местах программы мы работаем с массивами одного размера. Допустим, было 32 символа в двоичном коде. Когда мы изменим программу и будем работать с 64 символами, то изменим только строчку макроса *#define ARR_SIZE 64*, а все остальное в текстах изменит компилятор при первом проходе, когда просто выполнит контекстную замену *ARR_SIZE* на число 64.

У динамических массивов размер неизвестен на момент компиляции программы, а создавать простую индексированную переменную, у которой размер тоже переменная, в Си нельзя.

Подробнее с динамическими массивами вы познакомитесь в разделе 6.5.

По размерности массивы делят на *одномерные* и *многомерные*.

Примером одномерного массива может служить строка (**одномерный массив не обязательно строка, но строка – всегда одномерный массив**), а примером многомерного (в данном случае двумерного) массива – таблица текста или матрица чисел.

В языке С строка представляет собой массив **печатных** символов, который может включать в себя заглавные и прописные буквы, цифры, знаки препинания и разделители, а также служебные символы. Обязательной частью Си-строки является символ окончания строки (нуль-символ, «\0», символ все биты которого равны 0 – 0x00, код символа есть нуль). Не путайте этот символ с печатным нулем, код **цифры 0** равен 48 (в десятичном представлении) или 30 (в шестнадцатеричном представлении). Нуль-символ является управляющим, и как все управляющие символы не виден пользователю, однако, **программист должен всегда помнить о его существовании**.

Важно! В С/С++ индексация массивов начинается с нуля. Пример массива из 9 элементов представлен в Таблица 6.1.

Таблица 6.1

Номер элемента массива	1	2	3	4	5	6	7	8	9
Индекс элемента в С	0	1	2	3	4	5	6	7	8

Говоря о двумерном массиве, можно также представлять его как одномерный массив, где в качестве каждого элемента выступает другой одномерный массив.

Например, у нас может быть набор (массив) элементов, каждым из которых является символьная строка. Тогда при работе с отдельными элементами каждой строки весь массив строк будет считаться двумерным массивом.

Так же, как и обычные переменные массивы могут хранить данные типов *int*, *float*, *double*, *char*.

В зависимости от этого различные массивы будут объявляться следующим образом:

int intArray[5], double doubleArray[136]; //одномерные массивы, которые хранят целочисленные элементы (*int*) и числа с плавающей точкой большого размера (*double*)

float floatArray[5][4], char charArray[11]; //двумерные массивы, хранящие числа с плавающей точкой (*float*) и строка из 10 символов (*char*).

6.2. Кодирование символов

Как вы знаете, компьютер не «понимает» символы как таковые и может оперировать только наборами единиц и нулей. Следовательно, каждому человеку-читабельному символу должна соответствовать некая двоичная комбинация. Для этого и была разработана кодировка ASCII. В этой кодировке каждый символ кодируется восемью битами, в связи с чем ее также называют однобайтной кодировкой. Следовательно, максимальное количество символов, которые можно закодировать, составляет $2^8=256$.

Ниже представлена таблица ASCII, определяющая 128 символов (Рис. 15):

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Рис. 15. Базовая таблица ASCII-символов

Исходя из представленной таблицы, видно, что в данную кодировку вошли управляющие символы, буквы латинского алфавита, знаки препинания и др. специальные символы. Всего под эти основные символы было выделено 7 бит (128 символов). Остальными битами можно было кодировать символы, например, национальных языков. Однако такого количества доступных для кодирования символов, как правило, недостаточно, в связи с чем позднее появились расширенные кодировки.

Unicode же представляет собой таблицу символов, в которую занесены знаки практически всех письменных языков в мире. Всего Юникод включает в себя 1 114 112 позиций, большая часть из которых до сих пор не заполнена

(таким образом, диапазон значений, которые могут принимать символы Юникода, простирается от 0 до *0x10FFFF*).

Существуют несколько форм представления Юникода:

1. В UTF-8 для кодирования символа используется от 1 до 4 байт. Поскольку для кодировки наиболее часто используемых символов достаточно 1 байт (в частности, для вышеописанных ASCII-символов), то UTF-8 удобно использовать для английского текста, но не для азиатского;

2. UTF-16 использует кодирование 2 или 4 байтами. Эту кодировку считают удобной для кодирования символов азиатских языков с иероглифическим письмом;

3. В UTF-32 все символы представляются 4 байтами. Данная кодировка, потребляет много памяти и, следовательно, используется не очень часто. Однако, она вполне успешно работает при наличии достаточной вычислительной мощности.

Заметим, что кодировки UTF вполне совместимы с кодировкой ASCII, поскольку общие символы и в тех, и в других кодируются одним байтом. Таким образом, все латинские и управляющие символы находятся в UTF на тех же местах, что и в ASCII. Следовательно, представление всех символов из диапазона 0–127 будет единым для всех кодировок.

В настройках проекта нужно указывать тип кодировки, multi-byte или Unicode. В рамках данного курса мы будем использовать multi-byte.

6.3. Инициализация и представление массивов в коде

Как и для всех переменных при создании массива требуется его объявление. И так же, как в случае с обычными переменными, объявление массива без его инициализации будет означать, что в выделенных под массив ячейках памяти может храниться всякий «мусор».

Можно инициализировать сразу все элементы массива при объявлении:

```
int intArray[5] = { 3, 9, 6, 4, 10 };
```

По Таблица 6.2 можно увидеть, как отличается восприятие массива символов человеком и процессором.

Таблица 6.2

Как пользователь видит номер элемента	1	2	3	4	5
Как видит массив пользователь	3	9	6	4	10
Как «видит» номер элемента	0	1	2	3	4

процессор					
Как «видит» массив процессор	0x0...03	0x0...09	0x0...06	0x0...04	0x0...05

Пример инициализации двумерного массива приведен ниже:

```
int intMatrix[2][3]={{1, 2, 3}, {4, 5, 6}};
```

Для предотвращения возможных ошибок рекомендуется или инициализировать элементы массива в явном виде, или предварительно инициализировать каждый элемент как цифру 0:

```
int intArray1[5] = { 0, 0, 0, 0, 0 };
```

```
int intArray2[100] = {0};
```

Необходимо различать случаи, когда массив типа *char* представляет собой массив целых чисел размером в 1 байт, а когда символьную строку.

Если массив является символьной строкой, при инициализации его лучше указывать как пустую строку, см. Таблица 6.3:

```
char charArray[5]="";
```

Таблица 6.3

Как пользователь видит номер элемента	1	2	3	4	5
Как «видит» номер элемента процессор	0	1	2	3	4
Как «видит» массив процессор (hex, ASCII)	0 («\0»)	Символ не определен	Символ не определен	Символ не определен	Символ не определен

Если же нужно задать символы по отдельности, обязательно указать в конце массива нуль-символ, см. Таблица 6.4:

```
char charArray1[5] = {'a', 'B', '3', '&', 0};
```

```
char charArray2[5] = "aB3&";
```

Таблица 6.4

Как пользователь представляет номер элемента	1	2	3	4	5
Как видит массив пользователь	<i>a</i>	<i>B</i>	3	&	0
Как «видит» номер элемента процессор	0	1	2	3	4
Как «видит» массив процессор (hex, ASCII)	0x61	0x42	0x33	0x26	0x0
Как «видит» массив процессор (hex, UTF8)	0x0061	0x0042	0x0033	0x0026	0x0000

Не забывайте добавлять завершающий нулевой символ к строке, которую вы создаете программой. Если при отображении строки на экране в ее

конце появились непонятные «кракозябры» или если программа почему-то зависла, проверьте, не забыли ли вы добавить в строку этот волшебный нулевой символ [3].

Если вы не хотите добавлять к строке нуль-символ, при работе вам нужно будет каждый раз передавать размер массива.

Для задания **массива символов** можно использовать указатель на символьный тип.

Итак, мы подошли к еще одному новому типу – *указателю*.

Указатель — это переменная, которая содержит адрес другой переменной (то есть ее расположение в памяти). Адреса в памяти традиционно записываются в шестнадцатеричном коде.

Вернемся к примеру. Предположим, имеется массив вида

```
char charArray[5] = { 'a', 'B', '3', '&', 0};
```

Указатели всегда имеют размер, соответствующий разрядности системы. Для 32 разрядной системы размер адреса 32 бита, или 4 байта. Указатель должен определить какого размера данные будут извлечены системой по этому адресу. Для типа *char* – это 1 байт, для *int* – 4 байта. Значит указатель должен быть связан с определенным типом, так что он пишется как тип со звездочкой, например указатель на элемент *char* или на начало массива из элементов *char* это *char **:

```
char * pointer;
```

В этом случае объявленной переменной *pointer* может быть присвоен адрес массива:

```
pointer = charArray;
```

**pointer*; //эквивалентно *charArray* [0]='a' – разыменование указателя, извлечение числа, находящегося по адресу, на который указывает указатель.

**(pointer +1)*; //эквивалентно *charArray* [1]='B' – значение второго элемента массива.

Здесь *charArray* – константа-указатель, то есть неизменяемый указатель. Нельзя изменить *charArray*, поскольку это будет означать изменение адреса массива в памяти. А *pointer* – переменная типа указатель на символ, может меняться. Так *pointer++* будет указывать на следующий (второй) элемент массива.

**(pointer++)*; //эквивалентно *charArray* [1]='B' – значение второго элемента массива.

Получить указатель на переменную можно написав «&» перед именем переменной, например:

pointer = &charArray[1]; // указатель на второй элемент массива, то есть на символ 'B'.

Массивы символьных строк, по сути, являются аналогом двумерного массива. Например:

```
char *symStrArr [3] = {"Znakomstvo", "s Visual", "Studio"};
```

Здесь *symStrArr* – массив из трех указателей на символьные строки. Каждая строка символов представляет собой символьный массив, поэтому имеется три указателя на массивы. То есть:

```
*symStrArr[0]; //соответствует 'Z',
```

```
*symStrArr[1]; //соответствует 's'.
```

Инициализация выполняется по правилам, определенным для массивов.

Тексты в кавычках эквивалентны инициализации каждой строки в массиве. Запятая разделяет соседние последовательности [4].

Существует также понятие **свободного массива**:

```
char *Array[4]; //определяет свободный массив, где длина каждой строки определяется тем указателем, который эту строку инициализирует.
```

Свободный массив позволяет экономнее использовать память [4].

Обратиться к конкретному элементу массива при работе с символьной строкой можно следующим образом:

```
char Elem1 = charArray[0]; //обращение к первому элементу
```

```
char Elem2 = charArray[3]; //обращение к четвертому элементу.
```

6.4. Выход за границы массива

В памяти машины разделяют блок исполняемого кода и блок данных. За программой присматривает система безопасности процессора и операционной системы. Программа выполняется на уровне привилегий пользователя, и у нее нет прямого доступа к системным ресурсам.

В памяти данных расположение индексированных переменных всегда обеспечивается в непрерывном интервале адресов ячеек памяти. Например, с адреса *0x40110000* до *0x401010013* будут расположены 19 элементов массива *char*.

Программа не может обращаться по адресам 0–31 (в десятичном представлении) или 0-*0x1F* (в шестнадцатеричном представлении) – это коды служебных символов – в этом случае вызывается ошибка доступа к памяти и остановка программы.

К сожалению, в C++ не проверяется выход индекса за пределы массива. Этот язык будет рад предоставить вам доступ к элементу *integerArray [200]*.

Более того, C++ позволит вам обратиться даже к *integerArray* [-15]. Угадать, куда в этом случае попадешь, довольно трудно, но хранить там что-то еще рискованнее.

Пытаясь прочитать 20-й элемент (*array[20]*) 10-элементного массива, вы получите то или другое случайное число (а, возможно, программа просто аварийно завершит работу). Записывая в *array[20]* какое-то значение, вы получите непредсказуемый результат. Скорее всего, ничего в него не запишется, но возможны и другие странные реакции вплоть до аварийного завершения программы.

Одной из самых распространенных ошибок является неправильное обращение к последнему элементу. Например, при использовании массива типа *char* это будет обращение по адресу *integerArray* [128]. Хотя это всего лишь следующий за концом массива элемент, записывать или считывать его не менее опасно, чем любой другой некорректной адрес [3].

Следите за тем, по каким адресам ячеек вы обращаетесь и куда записываете свои данные.

6.5. Динамические массивы

Очень часто мы не знаем заранее, сколько элементов будет содержаться в массиве. Например, когда пользователь вводит числа, или мы работаем с меняющимся набором объектов. В этом случае массивы можно ограничить достаточно большим числом элементов, но при этом в системе будет резервироваться лишняя неиспользуемая память, или наоборот может произойти переполнение массива, а, следовательно, и потери информации.

Динамический массив целых чисел размером *a* в языке C можно создать следующим образом:

```
int *pN = (int *)malloc(a*sizeof(int));
```

Размер памяти будет представлен в **байтах**, индекс массива будет лежать в пределах (0, *a*). Память такого массива **необходимо** освобождать функцией *free(pN)*, где – *pN* указатель на память, выделенную функцией *malloc*.

Функция *realloc()* позволяет менять размер массива. Данные при этом будут скопированы (если хватит памяти на новый блок).

В языке C++ динамический массив целых чисел размером *a* может быть создан с помощью оператора *new*:

```
int *pN = new int[a];
```

Размер памяти такого массива будет представлен в **элементах**, индекс массива также будет лежать в пределах (0, *a*). Память также требует

освобождения, что нужно сделать с помощью функции *delete [] pN*. Здесь квадратные скобки указывают на удаление всех элементов массива.

Выделенная в блоке память (*malloc*, *calloc*, *new*) **не будет освобождаться автоматически** при выходе из блока. Если в процессе работы не вызван (*free* или *delete*) то возникают **утечки** памяти.

Для компилятора нет разницы между указателем и индексированной переменной, в то время как для программиста важно корректно использовать индексы и указатели в динамических массивах.

Помните – выход за границы всегда вызывает нарушение памяти и сбой программы!

Всего у 32-битного приложения 1,5 Гб памяти данных.

6.6. Работа с массивами

Как и с любым другим объектом языка Си, работа с массивами подразумевает использование существующих библиотечных функций, а также создание своих собственных функций, а, следовательно, передачу массивов в эти функции.

Часто для ввода строки используется функция *scanf()*. Однако, нужно помнить, что функция *scanf()* в большей степени предназначена для получения слова (то есть без использования пробелов, табуляции, перевода строки). Если применять формат «*%s*» для ввода, строка вводится до (но не включая) следующего пустого символа [4]:

```
scanf("%9s", name);
```

```
char * gets(char *); //для ввода строки, включая пробелы,
```

или ее эквивалент

```
char * gets_s(char *);
```

Нужно отметить, что аргументом функции является указатель на строку, в которую осуществляется ввод. Пользователь вводит строку, которую функция будет помещать в массив, пока пользователь не нажмет Enter.

Для вывода строк можно воспользоваться функцией *printf()* [4]:

```
printf("%s", str); //где str — указатель на строку
```

или в сокращенном формате

```
printf(str);
```

также можно использовать функцию *puts()*:

```
int puts (char *s); //печатает строку s и переводит курсор на новую строку (в отличие от printf()) – функция вывода строк
```

Данную функцию также можно использовать для вывода строковых констант, заключенных в кавычки.

char getchar(); //возвращает значение символа, введенного с клавиатуры – функция ввода символов

char putchar(char); //возвращает значение выводимого символа, выводит на экран символ, переданный в качестве аргумента – функция вывода символов.

В Таблица 6.5 приведены некоторые из библиотечных функций, использующихся при работе с символьными строками (библиотека *string.h*) [4]:

Таблица 6.5

Функция	Описание
<i>char *strcat(char *s1, char *s2)</i>	присоединяет <i>s2</i> к <i>s1</i> , возвращает <i>s1</i>
<i>char *strncat(char *s1, char *s2, int n)</i>	присоединяет не более <i>n</i> символов <i>s2</i> к <i>s1</i> , завершает строку символом '\0', возвращает <i>s1</i>
<i>char *strcpy(char *s1, char *s2)</i>	копирует строку <i>s2</i> в строку <i>s1</i> , включая '\0', возвращает <i>s1</i>
<i>char *strncpy(char *s1, char *s2, int n)</i>	копирует не более <i>n</i> символов строки <i>s2</i> в строку <i>s1</i> , возвращает <i>s1</i> ;
<i>int strcmp(char *s1, char *s2)</i>	сравнивает <i>s1</i> и <i>s2</i> , возвращает значение 0, если строки одинаковы
<i>int strncmp(char *s1, char *s2, int n)</i>	сравнивает не более <i>n</i> символов строк <i>s1</i> и <i>s2</i> , возвращает значение 0, если начальные <i>n</i> символов строк эквивалентны
<i>int strlen(char *s)</i>	возвращает количество символов в строке <i>s</i>
<i>char *strset(char *s, char c)</i>	заполняет строку <i>s</i> символами, код которых равен значению <i>c</i> , возвращает указатель на строку <i>s</i>
<i>char *strnset(char *s, char c, int n)</i>	заменяет первые <i>n</i> символов строки <i>s</i> символами, код которых равен <i>c</i> , возвращает указатель на строку <i>s</i>

6.7. Задание на лабораторную работу № 3

1. Необходимо создать консольное приложение Win32, осуществляющее перевод десятичного числа со знаком в двоичное с выводом всех 32-х разрядов в прямом порядке.
2. Перевод должен быть осуществлен методом поразрядной проверки бита и логического сдвига маски.
3. На вход поступает строка, содержащая натуральное десятичное число «а» в диапазоне -2 147 483 648...2 147 483 647. В строке возможны ошибки, такие как лишние символы, буквы, превышение длины строки. Их необходимо исправить. Всего на вход может поступить до 100 строк.
4. Выход из программы происходит после ввода пользователем 0, перед выходом запрашивается подтверждение выхода.
5. Вывод сделать в виде таблицы со столбцами: номер, число, двоичный код числа.

6.8. Пояснения к лабораторной работе № 3

1. Данные вводим с клавиатуры как строку:

char sIn[100]; // размер с запасом

scanf_s("%s", sIn); // указатель на строку это имя строковой переменной.

Удаляем из строки ошибочные символы так:

- а) делаем перебор символов в строке, и если это цифра и не знак «минус», то символ записываем в исправленную строку, а иначе символ пропускается;
- б) в исправленной строке не забываем символ окончания – ноль;
- в) копируем исправленную строку в исходную;
- г) если в строке были ошибки возвращаем код ошибки.

2. Разберемся со структурой проекта. Требуется написать код самого приложения и создать отдельную библиотеку. Код приложения – это текстовый файл с расширением *.crr*. Библиотека состоит из двух файлов – файла заголовка с расширением *.h* и файла с текстом функций библиотеки (определениями функций) с расширением *.crr*. Эту структуру легко понять из примера повторного использования кода предыдущей лабораторной работы. В том коде есть только один файл *.crr* с нужной нам функцией *Fun*. Но подключить этот файл к нашему проекту помешает функция *main*, в новом коде функция с этим названием уже есть, а две функции с одинаковыми именами *main* компилятор не пропустит. Значит, нужные в других проектах функции следует выносить в отдельные файлы, а их объявления - в отдельные заголовки.

Создайте файл приложения, например, *Application.cpp*. Создайте файлы библиотеки, например, *MyLip.cpp* и *MyLib.h*. с помощью меню по правой кнопке на вашем проекте в окне решения: *Add\New item*.

В начале файла *Application.cpp* перечисляются подключаемые библиотеки, например, стандартная библиотека *stdafx.h*. Подключим и нашу собственную библиотеку:

```
#include "MyLib.h"
```

Такая же строка будет и в начале файла *MyLip.cpp*.

Целесообразно в файл *MyLip.cpp* перенести все сложные самостоятельно созданные функции, например: функцию ввода числа и его проверки на ошибки, функцию вывода в виде строк таблицы и т. д. Определения функций выглядят так:

```
//определение функции
```

```
int FunctionName(int a, int b)//тип функции(int, bool, void и т.д.), ее имя и  
аргументы на входе
```

```
{  
    int c = a + b;//содержание функции – что она делает  
    return c;//возвращаемое значение  
}
```

В файле заголовка *MyLib.h* помещаются объявления функций, которые выглядят так:

```
int FunctionName(int a, int b); //объявление функции
```

Все функции, определенные в *MyLip.cpp*, должны быть объявлены в *MyLib.h*, иначе компилятор их не увидит. Также ошибки возникают, если аргументы функции в определении и в объявлении не совпадают.

3. По условию задачи на вход поступает число в формате строки, при этом в строке могут быть ошибки. Строка – набор символов, каждый из которых можно проверить. Сделать это можно с помощью функции *isdigit()*. Функция возвращает *TRUE*, если символ на ее входе является цифрой:

```
//пример использования функции isdigit()  
char str[6] = "12abc";//пример строки  
if (isdigit(str[1])) // проверка, является ли символ с индексом [1] цифрой  
    printf("This is digit");//результат проверки  
else  
    printf("Not digit");
```

Обратите внимание, что символ минус «-» не является цифрой, но должен быть сохранен, так как пользователь может ввести отрицательное число.

При правильном вводе строка может содержать до 12 символов (10 цифр, знак «-» и 0 – символ конца строки). Из-за ошибок длина вводимой строки может оказаться больше, так что максимальную длину строки можно сделать 20–30 символов.

4. В процессе проверки строки лишние символы исключаются, в результате останется новая строка, содержащая число и, возможно, знак «-». Ее можно преобразовать в целое число *int* , воспользовавшись функцией *atoi()*:

```
//пример использования функции atoi()  
char str[3] = "42"; // в памяти записаны ASCII-коды цифр 4 и 2  
int num = atoi(str); //num = 42, в памяти записано само число 42.
```

Функция *atoi()* корректно преобразует и положительные, и отрицательные числа. В примере в ячейках памяти, выделенных для хранения числа 42, будет записано: 0000000000000000000000000000101010. Это и есть двоичный код числа, который требуется вывести в таблицу. Для того чтобы «достать» этот двоичный код и вывести его в виде строки, воспользуемся методом поразрядной проверки бита и логического сдвига маски.

5. Результаты требуется вывести в таблицу. Известно, что в таблице будет не более 100 записей. При этом в одной строке таблицы будет располагаться двоичный код числа (32 символа), само число (до 11 символов) и номер строки (до 3 символов), всего 46 символов. Между столбцами для удобства чтения следует сделать пробелы. Таким образом, для одной строки будет достаточно взять 50 символов: 46 для информации, 3 для пробелов и 1 для перевода на новую строку. Такая таблица представляет собой двумерный массив:

```
char Tab[100][50];
```

Заполнение строк таблицы происходит по ходу обработки вводимых пользователем чисел. Ввод происходит циклически, условий окончания два – введено 100 строк, либо введено число нуль. Для такого ввода удобно использовать цикл с постусловием *do while*:

```
int strNumber = 0; //переменная для индекса строки (номер строки – на 1  
больше!)  
do  
{  
//Происходит ввод числа и запись его в таблицу  
...  
//StrNumber++ – каждый цикл увеличивается индекс строки  
} while (Value!=0 && StrNumber < 99); //пока число не ноль и введено  
меньше 100 строк, продолжить.
```

Не забудьте перед выводом таблицы вывести названия столбцов «Номер», «Значение», «Двоичный код», :

```
printf("Num Value Bin\n");
```

Далее нужно вывести таблицу по строкам. Напомним, что вся таблица *Tab* в этом случае считается массивом строк. Чтобы таблица была аккуратной, каждую строку следует заполнять текстом, выделяя для каждой колонки достаточно символов.

Колонка с порядковым номером, например, содержит текст длиной от одного до 3-х символов, колонка числа от одного до 11 символов, включая знак, двоичный код фиксированный размер 32 символа. Колонки заканчиваются разделителем – символом пробела или вертикальной чертой.

Вывод текста заданного размера в колонку можно реализовать, добавляя в строку с текстом пробелы по количеству недостающих символов. Оформите в вашей библиотеке функцию вывода десятичного числа в поле заданного размера.

7. СТРУКТУРЫ

Иногда требуется описать несколько свойств одного и того же объекта. Например, в списке студентов вуза может быть такая информация: имя (набор символов), номер группы и номер в списке (целые числа). Эти переменные имеют различные типы. Можно объявить их индивидуально: *int group*, *int number*, *string name* – и так для каждого студента. Но с увеличением числа студентов в списке количество отдельных переменных будет расти, всем им потребуется присвоить различные имена, так что работать с таким кодом будет затруднительно.

Казалось бы, можно создать массив из трех элементов, в первый его элемент записать имя, во второй – номер группы и так далее. Но наши переменные имеют различные типы, а массивы могут содержать элементы только одного типа. Так что такой способ не подойдет.

Другой вариант – использовать различные массивы: в одном хранить все имена, в другом – все номера. Но эти массивы надо синхронизировать, ведь изменения в одном массиве должны учитываться во всех остальных, что создает дополнительные сложности. Поэтому для решения данной задачи в Си существуют структуры данных.

Структурой в языке Си называется совокупность переменных различного типа, формирующих составной тип, который программист может объявлять самостоятельно.

Структуру можно объявить тремя способами, которые приведены в Таблица 7. 1:

Таблица 7. 1

Способ 1 (C++)	Способ 2 (Си)	Способ 3 (комбинированный)
<pre>struct MyStruct_s { int group, number; char name[200]; };</pre>	<pre>typedef struct { int group, number; char name[200]; } MyStruct_t;</pre>	<pre>typedef struct MyStruct_s { int group, number; char name[200]; } MyStruct_t;</pre>

Рассмотрим первый способ объявления структуры. В нем используется ключевое слово *struct*, и выбирается имя структуры, например, *MyStruct_s*. Внутри записываются поля (элементы) структуры и их типы. Мы создали

новый составной тип данных – шаблон, который программа будет использовать, когда мы объявим переменную-структуру: *MyStruct_s Student*.

Второй способ использует ключевое слово *typedef*. **Здесь тип данных не имеет названия, но имеет псевдоним *MyStruct_t***. Этот псевдоним можно использовать так же, как и обычный тип данных. С помощью *typedef* можно задавать псевдонимы и для стандартных типов, важно не запутаться в переопределениях.

Третий способ совмещает два предыдущих – создается тип данных *MyStruct_s*, и ему присваивается псевдоним *MyStruct_t*.

При создании переменной *Student* типа *MyStruct_s* в памяти выделяется место для нее. В данном случае будет выделено по 4 байта для элементов *group* и *number*, 200 байт для символов поля *name* – всего 208 байт, что проиллюстрировано на Рис. 16.

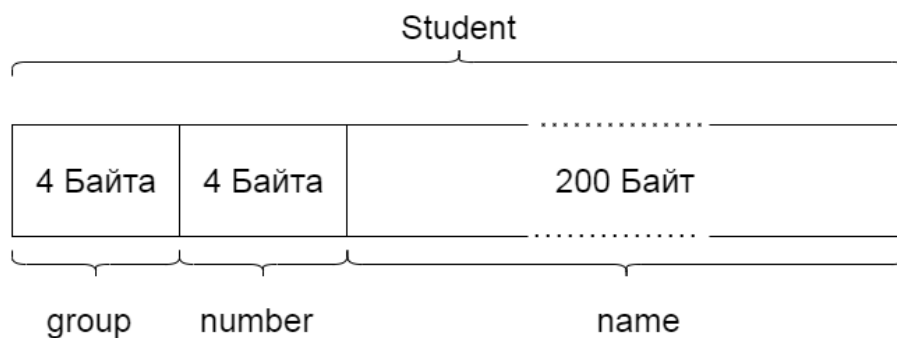


Рис. 16. Расположение структуры в памяти

Для доступа к элементам структуры указывается имя переменной и название элемента: *Student.group = 4321*. Это значит, что в переменную типа *int*, занимающую первые 4 байта переменной *Student*, запишется число 4321. Адрес переменной *&Student.group* – это адрес первого байта памяти, в которой хранится переменная *Student*.

Пример заполнения строки имени: *strcpy_s(Student.name, 200, "Peter")*. При этом от начала переменной *Student* отсчитывается 8 байт, в память копируется строка «*Peter*», а затем добавится 0 – символ конца строки.

К элементам структуры можно получить доступ с помощью указателя:

```
MyStruct_s Student;
```

```
MyStruct_s *pStudent = &Student;
```

```
strcpy_s(pStudent->name, 200, "Peter");
```

Структуры могут содержать десятки элементов. Чтобы не присваивать каждому из них значение индивидуально, можно инициализировать всю структуру сразу: *MyStruct_s Example = {4, 2, "QWERTY" };*

Если при инициализации в скобках записаны не все переменные, то оставшимся числовым переменным будут присвоены нулевые значения.

7.1. Состав структуры

Структура может содержать следующие элементы:

- простые переменные (*int*, *double*, *char* и т. д.);
- индексированные переменные (в качестве элементов структуры могут использоваться массивы);
- другие структуры (не допускается рекурсия типов – внутри структуры не может содержаться такая же структура);
- указатели.

7.2. Особенности работы со структурами

Так же, как и обычные переменные, структуры одного типа можно копировать:

```
MyStruct_t A = { 1,2, "ABC" };
```

```
MyStruct_t B = { 0,0, "0" };
```

```
B = A; //копирование содержимого одной структуры (A) в другую (B)
```

Можно создавать массивы из структур. При этом используются квадратные скобки, как и для обычного массива, а номер элемента массива пишется перед точкой:

```
MyStruct_t Array[5]; //массив, состоящий из 5 структур
```

```
Array[2].number = 15; //во второй элемент третьей структуры в массиве записывается 15
```

По аналогии с обычными индексированными переменными, *Array* – это указатель на первый элемент массива, так что *Array->group* это то же самое, что и *Array[0].group*.

7.3. Передача структур в функции

Аргументы типа структур передаются в функции разными способами:

1. По значению – *function(MyStruct_s F)*. В этом случае в памяти создается копия структуры. Все изменения, происходящие с копией, не затрагивают исходную структуру. Такой способ может быть неэффективен, если структура очень большая.

2. По указателю – *function(MyStruct_s *F)*. В этом случае передается только адрес структуры, копия в памяти не создается. Чтобы действия внутри

функции не изменяли содержимого структуры, может использоваться модификатор *const*: *function (const MyStruct _s *F)*.

3. По ссылке – *function(MyStruct_s &F)*. Здесь тоже можно использовать модификатор *const*, если надо подчеркнуть, что память переменной не будет изменена при работе функции.

Аналогично, в функцию можно передавать только части структур:

1. По значению – *function (F.number)*. Будет создана копия переменной.
2. По указателю – *function (&F.number)*. Передается только адрес.
3. По ссылке – *function (F.number)*. Тоже передается только адрес.

7.4. Битовые поля и объединения

Битовые поля позволяют определить точное количество бит, которые используются в памяти для каждого из элементов структуры. При этом через двоеточие указывается количество бит:

```
struct date
{
    unsigned short day : 5; //максимальное число дней в месяце: 31, доста-
точно 5 разрядов
    unsigned short month : 4; //максимальное число месяцев: 12, доста-
точно 4 разрядов
};
```

Полученный тип данных занимает 2 байта в памяти, размер его равен размеру *unsigned short*. При этом модификатор *unsigned* важен, иначе один бит будет зарезервирован под знак числа.

Использование битовых полей удобно для экономии памяти.

Существуют особые структуры – объединения. Для их создания используется слово *union*, например: *union MyUnion {int n; char ch;};*. Такая конструкция позволяет интерпретировать одну и ту же память или как целое, или как символ, как показано в примере ниже:

```
MyUnion Un;
Un.n = 75; //записываем в память целое число 75
int a = Un.n; //a = 75, память интерпретируется как целое число 75
char b = Un.ch; //b = 'K', число 75 интерпретируется как ASCII-код
буквы 'K'
Un.ch = 'G'; //записываем в память букву 'G'
```


int c = Un.n; //c = 71, ASCII-код буквы 'G' интерпретируется как целое число 71

char d = Un.ch; //d = 'G', память интерпретируется как ASCII-код 'G'

Здесь используется один и тот же байт памяти, поэтому изменение элемента структуры *ch* одновременно изменяет и элемент *n*.

7.5. Задание на лабораторную работу № 4

1. С использованием двух структур создать консольное приложение для записи телефонной книжки.
2. Ввод записей осуществляется, пока не будет введен телефонный номер 0;
3. Выводить записи телефонной книжки в таблицу:
 - порядковый номер;
 - ник;
 - телефон.
4. Таблицу выводить построчно;
5. Использовать библиотеку и общее с лабораторной №3 решение.

7.6. Пояснения к лабораторной работе № 4

1. Номер телефона состоит из нескольких цифр и символа «+». Для его хранения создадим структуру:

```
struct PhoneNo_s // это определение нашего собственного типа  
{  
    long long Number : 40; // почему 40 бит достаточно для телефона?  
    unsigned int Plus : 1; // поле для плюса  
};
```

Чтобы сэкономить место в памяти, выделяем на номер 40 бит и один бит выделяем в качестве признака «+». Такая структура займет 8 байт, но использоваться будет только 41 бит из 64 доступных.

2. Запись в телефонной книжке тоже является структурой – она содержит номер и ник:

```
struct MyPhRec_s //определяет структуру тип с именем MyPhRec_s  
{  
    PhoneNo_s PhoneNo; //номер  
    char Nic[MY_MAX_STR_LEN]; //ник  
};
```

Поля данной структуры: ранее созданная структура номера телефона *PhoneNo_s* и индексированная переменная *char[]* для имени абонента. Размер этой переменной можно изменять, меняя определение *MY_MAX_STR_LEN*. Для этого в файле заголовка пишется следующий макрос:

```
#define MY_MAX_STR_LEN 40; // размер строки – 40 символов
```

Если потребуется изменить количество символов, достаточно поменять значение в строке с *#define*. Везде, где записано *MY_MAX_STR_LEN*, будет использовано новое значение.

Вопрос для студентов: почему для этой же цели нельзя использовать обычную переменную?

Итак, в структуре есть номер телефона, занимающий 8 байт, и имя размером до 40 символов по одному байту. Всего эта структура займет 48 байт.

3. Создадим переменную, имеющую описанный тип:

```
MyPhRec_s rec;
```

Чтобы сохранить в нее номер, указываем названия нужных полей структуры через точку:

```
rec.PhoneNo.Number = 79219222222;
```

Поле *Number* находится в начале структуры *PhoneNo*. Структура *PhoneNo* находится в начале структуры *rec*. Число 79219222222 будет записано в переменную типа *_int64*, расположенную в начале памяти, занятой переменной *rec*. К этой переменной также можно обратиться, используя адрес *&rec.PhoneNo.Number*.

Заполним строку имени:

```
strcpy_s(rec.Nic, 40, "Peter");
```

Поле *Nic* располагается после поля *Number*. От начала переменной *rec* отсчитывается 8 байт, занятых полем *Number*, а дальше в память копируется строка "Peter", в конце строки добавляется символ 0.

4. Если используется указатель на структуру *MyPhRec_s *prec*, доступ к ее элементам можно получить так:

```
prec->PhoneNo.Number = 79219222222;
```

```
strcpy_s(prec->Nic, 40, "Peter");
```

Чтобы инициализировать сразу все элементы структуры, используются фигурные скобки:

```
MyPhRec_s F = {{7921, 1}, "ABC"};
```

Здесь внешние скобки принадлежат структуре *MyPhRec_s*, а внутренние – структуре *PhoneNo_s*.

8. СТРУКТУРЫ C++ И КЛАССЫ

Структура, как и всякая другая переменная в коде имеет область видимости от того места, где она определяется до конца блока. Тут мы встречаем операцию создания структуры, в C++ это функция, называемая конструктор. Она может быть явно не задана, тогда это просто выделение памяти под структуру. В конце блока память, занимаемая структурой, освобождается, и в C++ эта операция тоже может быть дополнена некоторыми действиями в функции, называемой деструктором.

8.1. Конструкторы в структурах

В описании структуры как типа данных можно определять функции, инициализирующие данные – *конструкторы*. Использование конструкторов упрощает задание начальных значений элементов структур, которые иначе пришлось бы перечислять по одному, присваивая им начальные значения в тексте после объявления переменной данного типа. Если не присвоить начальные значения, в переменной может оказаться «мусор», оставшийся от предыдущего владельца этой памяти. Пример:

```
struct PhoneNo_s // определение нашего собственного типа  
{  
    long long Number : 40; //40 бит достаточно для номера телефона  
    unsigned int Plus : 1; // поле для плюса, 1 бит  
    PhoneNo_s(long long Num, unsigned int Pls) : Number(Num), Plus(Pls) {}  
//инициализирующий конструктор, оба поля структуры PhoneNo_s будут за-  
писаны значениями из списка параметров конструктора  
    PhoneNo_s() : Number(0), Plus(0) {} // конструктор без параметров,  
записывает в поля 0  
};
```

Имя функции-конструктора должно совпадать с именем типа, которому она принадлежит. В нашем примере *PhoneNo_s* – единственно возможное имя конструктора. Мы написали два варианта конструкторов: конструктор с параметрами *long long Num* и *unsigned int Pls*, которые будут записаны в элементы структуры, и конструктор без параметров, в котором элементы инициализируются нулями. Конструкторы без параметров еще называются *конструкторами по умолчанию*.

8.2. Полиморфизм конструкторов

Свойство языка различать две и более функций с одинаковыми именами в C++ называется полиморфизмом. Хотя нельзя изменить имя функции (как в случае с конструктором), можно изменить количество или тип параметров функции. В зависимости от типов параметров при вызове в коде будет вызван нужный вариант конструктора:

PhoneNo_s P(); // будет вызван конструктор без параметров (по умолчанию) и элементы Number и Plus будут инициализированы нулями

PhoneNo_s P(799211234567, 1); // будет вызван конструктор с параметрами и будут инициализированы элементы Number числом 799211234567 и Plus числом 1.

Аналогично и для структуры телефонной книжки, которая содержит элемент *PhoneNo* с типом структуры телефонного номера *PhoneNo_s*:

```
#define MY_MAX_STR_LEN 50
```

```
struct MyPhRec_s //определяет структуру тип с именем MyPhRec_s  
{
```

```
    PhoneNo_s PhoneNo; //номер
```

```
    char Nic[MY_MAX_STR_LEN]; //короткое имя контакта
```

```
    MyPhRec_s(long long Num, unsigned int Pls, const char* Nm) : PhoneNo(Num, Pls) { strcpy_s(Nic, MY_MAX_STR_LEN, Nm); } {}
```

// это инициализирующий конструктор, одно поле структуры MyPhRec_s и два поля PhoneNo_s будут записаны значениями из списка параметров конструктора

```
    MyPhRec_s(): PhoneNo(), Nic("") {} // это конструктор без параметров, записывает в поля 0
```

```
};
```

Здесь конструкторы (с параметрами и без) структуры *MyPhRec_s* сперва вызывают соответствующие конструкторы вложенной структуры *PhoneNo_s*. В синтаксисе конструктора после знака двоеточия «:» можно перечислять через запятую конструкторы элементов этой структуры. В C++ конструкторы есть и у простых типов. Например, имя простой переменной со скобками *Number(Num)* – это вызов конструктора типа *long long* со значением *Num*. Это значение *Num* и будет записано в элемент телефонного номера нашей структуры. В блоке кода конструктора также можно присваивать значения элементам.

Вызывать другие конструкторы этой же структуры из блока кода конструктора небезопасно.

8.3. Наследование в типах C++

Обратим внимание, что действия в конструкторах структуры *MyPhRec_s* вызывают конструкторы ее вложенной структуры. Здесь просматривается иерархия наших данных, которой в Си не было. Можно считать структуру *MyPhRec_s* некоторой надстройкой над структурой *PhoneNo_s*. В *MyPhRec_s* появляется еще один атрибут *Nic* – имя контакта в телефонной книжке.

Итак, мы пришли к еще одному важному свойству данных в C++, которое называется *наследованием*. Предназначение базового типа *PhoneNo_s* – работа с данными телефонного номера. Его наследник *MyPhRec_s* должен оперировать еще и именем контакта. Иерархия типов удобна, она экономит код, автоматически включая в наследников функционал базовых типов.

8.4. Инкапсуляция переменных в классах

Дальнейшее развитие языка C при переходе к C++ основано на принципе объединения переменных и действий над ними в классы. Действия реализуются с помощью функций и операторов, определяемых внутри этих классов. То есть, если структуры могли иметь только функции-конструкторы, то для классов допустимы любые другие функции. Эти функции должны быть объявлены в определении класса.

Класс объявляется словом *class*, за ним следует имя этого класса. Действуют те же ограничения на имена, что и у простых переменных. Затем следует блок текста C++, содержащего объявление переменных и функций класса. **Классы не поддерживаются компиляторами простого Си.** Если в текстах файлов заголовков вам нужно использовать классы, а сами заголовки подключаются в тексты простого Си, возникает конфликт. Он разрешается заключением C++ определений классов в блок с условием для препроцессора: *#ifdef __cplusplus ... #endif*. Полезные сведения о классах можно подчерпнуть из [5]

8.5. Ограничения доступа к элементам и функциям классов

В C++ введено понятие прав доступа к содержимому класса. Права бывают:

- *public* – доступ без ограничений;
- *protected* – доступ из внутренних функций этого класса и его наследников;
- *private* – доступ только из внутренних функций этого класса, наследникам не разрешен.

Public, protected, private – это спецификаторы типа доступа. Группы переменных и функций в объявлении класса располагаются ниже соответствующего спецификатора.

Основными функциями в классах являются конструктор и деструктор. Конструктор вызывается при объявлении переменной нашего класса. Деструктор вызывается, когда переменная класса выходит из области видимости в тексте.

Во время сборки компилятор проверяет, есть ли в коде переменные с типом нашего класса. Если ни одной такой переменной нет, код класса не будет включен в код проекта. Никакие функции этого класса работать не будут.

8.6. Переменные внутри классов

Все переменные, которые вы делаете элементами класса, имеют область видимости, включающую все функции данного класса, независимо от уровня доступа. При этом возникает аналогия с глобальными переменными в простом Си, только глобальность ограничивается функциями класса.

В функциях класса переменные класса не нужно передавать как параметры – они уже доступны для чтения и записи. Если запись не предполагается, переменная объявляется с префиксом *const*. Если переменные объявлены в классе как *public*, то они доступны из внешнего по отношению к классу кода. Пример:

```
class CPhoneNo // определение нашего класса телефонного номера
{
public:
    long long Number : 40; // номер
    unsigned int Plus : 1; // поле для плюса
    CPhoneNo(long long Num, unsigned int Pls) : Number(Num), Plus(Pls) {}
// инициализирующий конструктор, оба элемента класса CPhoneNo будут
записаны значениями из списка параметров конструктора
    CPhoneNo():Number(0), Plus(0) {} // конструктор без параметров, за-
писывает в элементы класса 0
    ~CPhoneNo() {} // деструктор
    bool Input(); // функция ввода номера с клавиатуры
    bool Input(const char* strAsk); // функция ввода номера с клавиатуры с
запросом оператору
};
```

Инициализация тех переменных класса, которые имеют фиксированные значения, должна быть явно выполнена в конструкторе, иначе значения параметров будут неопределенными.

Два экземпляра одноименной функции допустимы, так как различаются списком параметров:

```
bool Input(); //функция без параметров  
bool Input(const char* strAsk); //функция с параметром – строкой (для приглашения)
```

Это правило распространяется на все функции, как описанные в классе, так и внешние обычные функции.

Рассмотрим, почему это удобно. Пусть есть одно и то же действие, которое по-разному выполняется для разных типов данных: *ToBin(int N)* и *ToBin(long long N)*. Типы аргументов у этих функций разные, так что получаются два экземпляра функции перевода в двоичный код – для 32-битных и для 64-битных целых чисел.

Рассмотрим пример кода:

```
CPhoneNo PhNo(); // создается переменная объекта с типом CPhoneNo  
нашего класса.
```

```
//в скобках нет аргументов: вызывается конструктор без параметров,  
в переменные класса записываются 0
```

```
//переменная объявлена и класс «инстанцирован», т.е. его переменные  
доступны в коде как данные, и функции класса доступны в вызывающем коде
```

```
PhNo.Number = 792312345678; // присваиваем значение, так как теле-  
фонный номер доступен для записи (public)
```

```
if( PhNo.Number ) {/*код, выполняемый по условию*/} // оператор условия  
использует разрешенный доступ на чтение к переменной Number
```

Классы идеально подходят для повторного использования кода. Класс может определяться как наследник другого класса, называемого в этом случае базовым или родительским. Это значит, что наследующий класс может включить все переменные (как это было в структурах) и функции базового класса автоматически, без дублирования кода. В нашем случае класс записи телефонной книжки может быть наследником класса телефонного номера:

```
class CMyPhRec : public CPhoneNo // запись телефонной книжки насле-  
дует свойства номера телефона
```

```
{  
public:
```

```

    CMyPhRec(__int64 Num, unsigned int Pls, const char* Nm) : CPhoneNo(Num, Pls) { strcpy_s(Nic, MY_MAX_STR_LEN, Nm); }
    // конструктор явно вызывает конструктор базового класса
CPhoneNo
    ~CMyPhRec() {} // деструктор
    char Nic[MY_MAX_STR_LEN]; // переменная определенная в классе
};

```

Классы как параметры передаются в функции теми же способами, что и обычные переменные, в частности структуры (см. п. 7.3). Передача элементов класса в функцию возможна только для *public* элементов.

Классы удобно помещать в отдельные файлы. В файле заголовка (.h) находятся объявления классов. Объявление класса содержит его компоненты, то есть переменные и объявления функций, но не содержит код (определение) этих функций. Код функций (методов класса) помещается в файл .cpp.

Все переменные класса и его базовых классов являются локальными переменными внутри этого класса. Например, *Number* и *Plus* в классе *CPhoneNo* будут доступны (видны) внутри функции *Input()*. Следовательно, при использовании классов сокращается список параметров функции.

Вызов функции класса осуществляется так же, как и обращение к переменной класса. Например:

```

F.Input(); // вызывает функцию ввода телефонного номера для переменной F класса CPhoneNo.

```

В файле .cpp определение функции дается с указанием имени класса и разделителем пространства имен «::»:

```

CPhoneNo::Input()
{ // здесь записывается код функции Input() }

```

Наконец, классы позволяют определять функции – действия над переменными, объявленными принадлежащими к этому классу. Такие функции называются операторами. Примером может быть оператор присваивания «=». Он позволяет нам сделать копирование $A = B$; для переменных класса. При этом выполняется «осмысленное» копирование переменных из экземпляра *B* в экземпляр *A* класса. Например, возможен такой оператор копирования:

```

CPhoneNo operator =(CPhoneNo& in) { this->Number = in.Number;
this->Plus = in.Plus; return *this; } // this – это указатель на сам экземпляр
класса, который стоит в левой части оператора.

```


Для класса существует пространство имен (*namespace*). Переменные и функции класса находятся внутри этого пространства имен. Это значит, что к имени каждой переменной или функции добавляется имя класса, например:

```
CPhoneNo::Input();
```

В программе могут быть и другие функции *Input()*, однако имя класса делает нашу функцию *Input()* уникальной и позволяет компилятору отличать ее от других. Компилятор будет искать определение такой функции в первую очередь внутри данного класса.

Внутри класса пространство имен присваивается по умолчанию. Например, если вы при описании функции класса вызываете другую функцию или переменную этого же класса, префикс *CPhoneNo::* не нужен. Его можно добавить, и это не будет ошибкой.

8.7. Классы как переменные и массивы

Переменные класса можно присваивать:

```
CMyPhRec A(), B(79219222222, 1, "ABC" );
```

A = B; // содержимое нормально копируется из B в A, включая внутреннюю структуру, если в классе есть оператор присваивания.

Индексированные переменные – это массивы, элементами которых являются переменные с типом этого класса. Синтаксис будет таким же, как и у структур: квадратные скобки с номером, затем точка. Например:

```
CMyPhRec Ar[2]; //массив классов длиной 2 элемента;
```

Ar[1].Number = 10; //во второй элемент массива и в первый элемент класса в первый элемент PhoneNo записать 10

По аналогии с обычными индексированными переменными *Ar* – указатель на первый элемент массива, а *Ar->Nic* – это то же, что и *Ar[0].Nic*.

8.8. Задание на лабораторную работу № 5

1. С использованием двух классов создать консольное приложение для записи телефонной книжки.
2. Ввод записей осуществлять, пока не будет введен телефонный номер «0».
3. Выводить записи телефонной книжки в таблицу в алфавитном порядке:
 - порядковый номер;
 - ник;
 - телефон.
4. Таблицу выводить построчно.

5. Использовать библиотеку и общее с лабораторными № 3–4 решение.

8.9. Пояснения к лабораторной работе № 5

В наше решение, где уже есть часть нужных функций, добавляем проект лабораторной работы. Для этого в представлении *Solution* (*Ctrl+Alt+L*) щелкаем правой кнопкой мыши на имени нашего решения и в вызванном меню выбираем пункты *Add* и *Project*. Далее создаем консольное приложение как в разделе 1 «Создание проекта в среде разработки Microsoft Visual Studio».

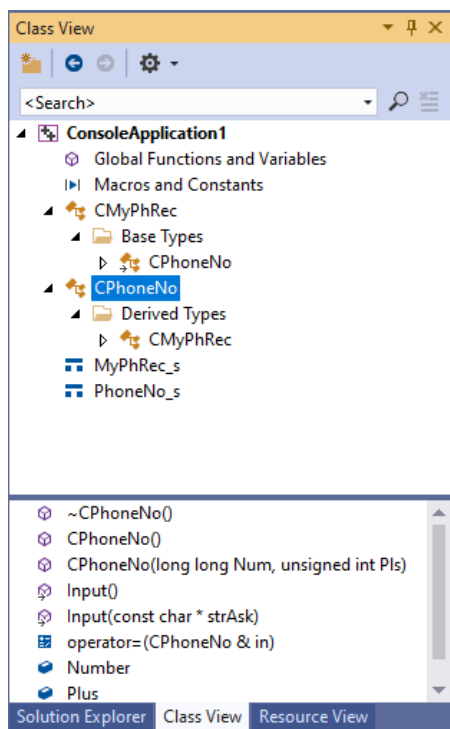


Рис. 17. Представление классов

Создание классов в Microsoft Visual Studio облегчается встроенным помощником. Для этого перейдите в представление классов, обычно оно располагается в закладке под структурой решения, *Ctrl+Shift+C* (Рис. 17).

В этом представлении можно видеть уже имеющиеся классы нашего приложения и добавить новый класс, для чего меню по правой кнопке мышки на нашем проекте имеет пункт *Add* и подменю *Class*. Появляется окно создания класса (Рис. 18).

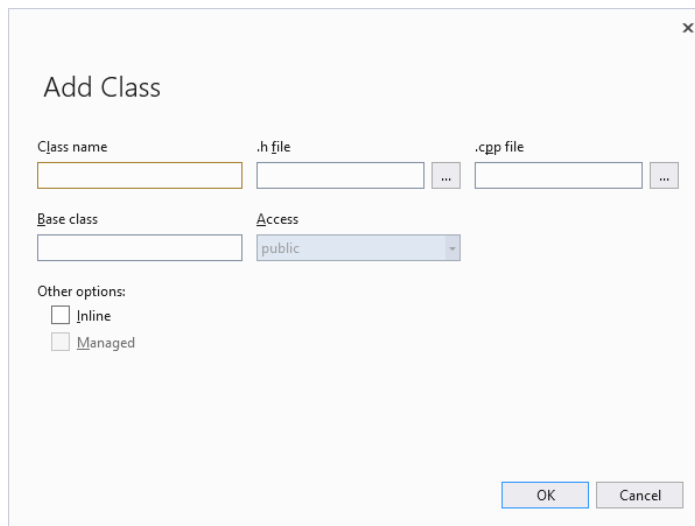


Рис. 18. Окно создания класса

В этом окне вводим имя класса и определяем файлы заголовка (*.h*) и текста (*.cpp*). Объявление класса помещается в файл *.h*, а конструктор – в файл *.cpp*. При наличии базового класса автоматически будет добавлен заголовок, где этот класс объявлен и заполнены соответствующие зависимости в объявлении классов. Аналогично добавляем функции и переменные в класс через пункт меню *Add*, щелкнув правой кнопкой мыши на строчке нужного класса в

окне представления классов. Заготовки для функций с типами и списками добавленных вами параметров будут созданы автоматически.

Сортировку массива можно выполнить по алгоритму из пункта 10.3 (сортировка методом «камешка»).

9. СТАНДАРТНЫЕ БИБЛИОТЕКИ C++

Стандартные библиотеки C++ предоставляют пользователям большое количество полезных классов и функций, выполняющих типовые алгоритмы. Алгоритмы хранения данных обобщаются в классах, которые относят к группе «коллекции». Стандартные классы и функции помещаются в пространстве имен «*std::*», для сокращения текста вверху обычно помещают директиву *using namespace std*.

9.1. Класс *vector* – векторы, динамические массивы

Динамический массив экономит память в процессе работы программы, а автоматический еще и освобождает память по окончании использования.

В языке C++ введено понятие шаблона – это возможность описать действия над неопределенным типом данных. При этом компилятор сам допишет нужный код с учетом конкретного типа данных, который передается в шаблон как параметр. Шаблон использует для определения типа синтаксис угловых скобок *<T>*.

Шаблон классов динамических массивов выглядит так: *std::vector <T>*, где *T* – тип данных, которые хранит массив.. Библиотеку векторов нужно подключить в виде заголовка следующим образом:

```
#include <vector>
```

Вектор назван так, поскольку представляет собой одномерный массив. Вектор из векторов масштабирует функционал в два измерения.

Конструкторы *std::vector* создают объект этого типа в коде и инициализируют переменную данными:

```
vector<CMyPhRec> phBook; // вводится переменная без параметров,  
компилятор вызывает конструктор без параметров, так что phBook – ну-  
стой массив длиной 0;
```

```
vector<int> vecInt(vecIntOther); // копирующий конструктор: вводится  
новая переменная, которую компилятор заполняет копией другого вектора  
(можно не целиком, а в выбранном диапазоне);
```

```
vector<int> vecInt(100, 1); //заполняющий конструктор: вводится пере-  
менная, компилятор заполняет массив единицами;
```

Когда переменная класса *std::vector* выходит из области видимости, вызываются деструкторы всех элементов массива, сам массив тоже удаляется.

Векторы поддерживают функции, которые синтаксически тождественны операциям:

```
std::vector<int> A,B(5,1); //создаем два объекта
```

A=B; //присваиваем одному объекту значения другого

Здесь вызывается оператор «=» класса *std::vector*. Этот оператор на самом деле – функция класса. Обычно функции имеют буквенные имена, но эта функция имеет имя «=», то есть выглядит как знак равенства. Это очень похоже на присваивание в Си. В результате содержимое *B* будет скопировано в *A*, причем с сохранением порядка следования.

Существует также оператор вида «*[]*», тогда *B[0]* будет содержимым первого элемента массива, в нашем случае числом 1. То есть наши привычные действия с вектором, как с массивом, работают через эту функцию-оператор.

Важнейшая операция – добавление в вектор нового значения. Для этого используется функция *push_back(value)*. Чтобы стереть последнее значение в векторе, используется *pop_back()*. При этом размер вектора *size()* в первом случае увеличится, а во втором – уменьшится на 1. Все действия по выделению и освобождению памяти будут выполнены автоматически.

Ниже представлены еще несколько полезных функций:

vector::size(); //возвращает количество элементов вектора;

vector::empty(); // возвращает true для пустого вектора;

vector::clear(); //удаляет все из вектора.

Для векторов и других классов коллекций определен класс перебора элементов вектора, который называется *итератором*: *vector<T>::iterator*. Значение итератора – адрес соответствующего элемента, то есть указатель на элемент с типом *T**.

Для вектора определены итераторы начала *vector::begin()*, который указывает на первый элемент вектора, и конца *vector::end()*, который указывает **за пределы вектора**.

У итератора есть тип, полное название которого выглядит так: *std::vector<T>::iterator i*. В нашем тексте *auto i* – синоним из C++ 11, то есть компилятор автоматически подставит вместо *auto* этот длинный тип *std::vector<T>::iterator*. Так что использование синонимов в таких случаях улучшает читабельность кода.

У итератора существуют унарные операторы «*», «&», «++», а также операторы сравнения «==», «!=».

Приведем простую аналогию с циклом перебора Си:

```
for( int i = 0; i < phBook.size(); i++ )
```

```
СMyPhRec rc = phBook[i];
```

То же самое можно записать с помощью итератора:

```
for( auto i = phBook.begin(); i!= phBook.end(); i++ )
```

*CMyPhRec rc = *i;*

Использовать сравнение итераторов «<» и «>» не рекомендуется, хотя данные в массиве должны строго находиться в непрерывной области памяти, чтобы арифметика указателей работала.

В классе *std::vector* хранятся данные, и функция *data()* предоставит доступ к данным начиная с первого элемента массива. Кроме того, очевидно, в классе хранится длина вектора, но эта переменная защищена (вот для чего нужен доступ *protected*) и доступна нам только через функцию *size()*.

В C++ существует класс *std::string* для работы с текстами. У класса строк базовый класс это *vector* и, как следует из изложенного, C++ строки наследуют функции векторов (в том числе сложение, копирование, размер и пр.) [6].

9.2. Задание на лабораторную работу № 6

1. С использованием двух классов создать консольное приложение для записи телефонной книжки.
2. Ввод записей осуществлять пока не будет введен телефонный номер «0».
3. Выводить записи телефонной книжки в таблицу в алфавитном порядке, сортировка методом «камушка» (см. раздел 10):
 - порядковый номер;
 - ник;
 - телефон.
4. Таблицу выводить построчно.
5. Использовать библиотеку и общее с лабораторными работами № 3, 4 и 5 решение.

9.3. Пояснения к лабораторной работе № 6

Доступ к элементам класса *vector<CMyPhRec> phBook* производится по имени переменной с индексом. Программист при написании кода должен учитывать, есть ли такой элемент в векторе. Если элемента с таким индексом нет, возникнет исключение.

phBook[0].Number = 7921922222;

Эта строка означает, что компилятору нужно записать число 7921922222 в переменную типа *long long*, которая расположена в начале памяти, занятой переменной *phBook[0]*. Адрес этой переменной: *&(phBook[0].Number)*.

Пример заполнения строки имени:

phBook[0].Nik = "Peter";

Доступ к элементам класса, переданного итератором:

```
CMyPhRec * prec = phBook.begin();
```

```
phBook.begin() ->Number = 79219222222;
```

phBook.push_back(CMyPhRec(7921,1,"ABC")); //сконструирует объект из параметров в списке конструктора CMyPhRec и запишет его в конец вектора *phBook*.

9.4. Поточковый ввод и вывод

Поточковый ввод-вывод представляет собой функции, которые выполняют действия преобразования из/в текст переменных различного типа. При вводе и выводе источниками могут быть консоль, файл, буфер в памяти и, конечно, различные коммуникации – USB, LAN, BlueTooth, WiFi и пр., которые играют роль символьного потока.

Действия при вводе текста можно представить как чтение последовательности символов – *слова*. Слово может быть текстом или числом. В качестве разделителей выступают пробелы, перевод строки или специальные символы. **При выводе текста необходимо преобразовывать числа в строки и обеспечивать последовательный посимвольный вывод строк в нужный поток.**

В языке C++ эти действия выполняют функции и классы библиотеки *<iostream>* и их наследники. В данном курсе рассматриваются два класса библиотеки консольного ввода-вывода:

- *cin* – чтение из клавиатуры консоли;
- *cout* – запись в окно консоли на экране.

Поскольку они являются частными случаями выполнения преобразования из/в строку, действия по преобразованию перенесены в базовые классы *istream* и *ostream*, которые в свою очередь являются наследниками класса *io_base*.

Когда вы пишете в коде «*cin*» или «*cout*», там создаются, соответственно, объект ввода из консоли или объект вывода.

Оператор чтения из буфера ввода «*>>*», очевидно, имеет варианты: по типам данных (*int*, *double*, *char*) и соответствующие операторы в классах строк. Синтаксис оператора «*>>*» при этом не меняется, а нужный вариант компилятор находит из набора полиморфных операторов библиотеки. То же справедливо для оператора записи «*<<*».

При выводе в таблицу всегда возникает проблема форматирования, для решения которой применяется библиотека *<iomanip>*.

В C++ программировании существует также понятие *манипулятора* – объекта, изменяющего действие операторов чтения «>>» и записи «<<». Так `std::setw(int width)` ограничивает количество символов, которые читаются, например, в массив, а при выводе создает строку нужной ширины. Таким образом можно сделать колонки в таблице. Внутри этих колонок выравнивание при выводе обеспечивается с помощью манипуляторов *left* (по умолчанию) или *right*:

```
std::cout << setw(20) << right << "ABC"; //ABC будет в 18-20 позициях.
```

Существуют также и другие полезные функции стандартной библиотеки C++, например:

```
std::cin.get(char &ch) //прочитает каждый символ из потока, включая  
пробелы, обычный оператор «>>» пробелы и перевод строки игнорирует  
std::cin.getline( char *ch, int length) //прочитает строку с ограничением  
длины.
```

Вместе с консольным вводом-выводом можно использовать также ввод-вывод в строку/из строки (функции такого вывода принадлежат библиотеке `<sstream>`) или ввод-вывод с использованием файлов (библиотека `<fstream>`).

Работа с файлами имеет некоторые особенности. Так, например, сперва необходимо открыть файл, для чего создается объект с типом операции *ifstream* (для ввода) или *ofstream* (для вывода). Также желательно проверять открыт ли он. Это можно сделать следующим образом:

```
std::ifstream FileIn;  
FileIn.open("MyFile.txt");  
if(FileIn.is_open())//проверка что файл открыт  
{FileIn >> MyStr;}//запись из файла в строку
```

Полный путь к файлу при записи должен обязательно содержать символы «\» (двойной обратный слэш), которые разделяют папки:

```
FileIn.open("c:\\Tmp\\MyFile.txt");
```

Чтение из файла должно сопровождаться проверкой того, что файл не закончился:

```
while ( FileIn. good()){}; //один вариант проверки  
while ( !FileIn. eof() ){}; //другой вариант
```

После того как чтение или запись в файл (из файла) закончены, файл необходимо закрыть, при этом он будет сохранен на диск:

```
FileIn.close();
```

Манипуляторы *setw*, *right* и т. п. при чтении и записи из/в файл точно так же наследуются классами файлового ввода/вывода.

9.5. Задание на лабораторную работу № 7

1. С использованием двух классов создать консольное приложение для записи телефонной книжки.
2. Ввод записей осуществлять, пока не будет введен телефонный номер «0».
3. Выводить записи телефонной книжки в файл в табличном виде в алфавитном порядке, сортировка любым методом:
 - порядковый номер;
 - ник;
 - телефон;
4. Использовать библиотеку и общее с лабораторными № 3, 4, 5 и 6 решение.
5. Реализовать функцию поиска телефона по имени методом деления на 2 (см. раздел 10).

10. ИСПОЛЬЗУЕМЫЕ АЛГОРИТМЫ

В данном разделе приведен краткий обзор алгоритмов обработки данных, используемых в лабораторных работах.

10.1. Алгоритм простого перебора

Для вывода телефонной книжки в алфавитном порядке нужно упорядочить записи при выводе, так, чтобы имена абонентов следовали по возрастанию (ведь «В» > «А»). Для этого необходимо:

1. Найти индекс наименьшего имени в исходном массиве и поменять запись первого элемента с найденной местами;
2. Повторять п.1, начиная со следующего элемента, ведь предыдущий уже на месте и так до конца массива.

Чтобы найти индекс минимального элемента необходимо:

1. В цикле перебрать записи в таблице и найти наименьшее имя;
2. В новой строке запомнить имя текущего элемента, и последовательно сравнивая со следующим меньшее из них, записывать в эту строку;
3. Запомнить индекс наименьшего элемента.

10.2. Алгоритм простого перебора (вариант 2)

Чтобы поменять две записи в массиве местами, будем использовать временную переменную следующим образом:

1. Заводим временную переменную с типом записи *Temp*;
2. Копируем в *Temp* содержимое первого неупорядоченного элемента;
3. Копируем в первый неупорядоченный элемент, найденный нами наименьший из оставшихся элементов;
4. На место наименьшего элемента копируем *Temp*.

10.3. Сортировка методом «камешка»

Для вывода телефонной книжки в алфавитном порядке нужно упорядочить записи при выводе, так, чтобы имена абонентов следовали по возрастанию (ведь ASCII-код буквы В больше, чем у А и так далее).

1. Начинаем с самого верха нашего вектора: пусть $i = phBook.begin()$, тогда, сравниваем этот элемент со следующим:

```
if ((i->Nic) > (i++)->Nic)
    /*выполняем п.2*/
```

2. Если предыдущее имя больше следующего, то сразу меняем эту запись со следующей местами;

3. Повторяем п.1-2, начиная со следующего элемента, но не доходим на 1 позицию до конца массива;

4. После 1–3 утверждается, что самое дальнее по алфавиту имя находится в конце массива и его больше проверять не нужно.

10.4. Сортировка методом «камешка» (вариант 2)

1. Повторяем п.1–3 из предыдущего метода для верхних $N-2$ элементов, потом для $N-3$, и так далее, пока не останется один элемент.

2. Важно: если при каком-либо проходе не было ни одной перестановки, то массив **уже отсортирован**. Останавливаем цикл 1–3.

3. Меняем две записи в массиве местами, используя временную переменную следующим образом:

3.1. заводим временную переменную с типом записи *Temp*

3.2. копируем в *Temp* содержимое первого неупорядоченного элемента

3.3. копируем в первый неупорядоченный элемент, найденный нами наименьший из оставшихся;

3.4. на место наименьшего копируем *Temp*.

10.5. Поиск методом дихотомии (деления на 2)

Для поиска номера абонента в отсортированной в алфавитном порядке телефонной книжке можно эффективно использовать свойство последовательности, упорядоченной по возрастанию. Для этого необходимо:

1. Завести две переменных, соответствующих верхнему и нижнему индексам интервала поиска

$iTop = 0;$

$iBottom = PhoneBook.size()-1;$

2. Найти середину (разделить пополам нашу книгу)

$iMid = (iTop + iBottom)/2;$

3. Проверить, в какой половине находится искомое имя: *if* ($PhoneBook[iMid].Nik > Name$), значит в верхней, иначе:

а) если равно, то поиск закончен;

б) если меньше, то имя в нижней половине списка.

4. После 1–3 утверждается, что половину массива, где искомого имени нет, больше проверять не нужно.

10.6. Поиск методом дихотомии (вариант 2)

1. Меняем границы нашего интервала поиска на основании сравнения искомого имени с серединой интервала, см. п.10.5.
2. Если поиск продолжаем в верхней части интервала, то $iBottom = iMid$.
3. Если же поиск продолжаем в нижней части интервала, то меняется верхняя граница $iTop = iMid$.
4. Если имя найдено, то поиск заканчивается успехом.
5. Если разница между верхним и нижним индексами равна 1, то поиск заканчивается неудачей.

ЗАКЛЮЧЕНИЕ

Наиболее полные сведения о языке Си с множеством полезных примеров можно найти в классической книге Б. Кернигана и Д. Ритчи [7].

Фундаментальный труд по языку С++ и способам его применения принадлежит Б. Страуструпу [8].

Изучение этих книг позволит вам расширить знания по курсу информатики до уровня профессионала.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19.701-90 (ИСО 5807-85) Единая система программной документации (ЕСПД). Схемы алгоритмов, программ, данных и систем. // Электронный фонд правовой и нормативно-технической информации [Электронный ресурс]. URL: <https://docs.cntd.ru/document/9041994> (дата обращения: 19.10.2021).
2. Массивы // Викичтение [Электронный ресурс]. URL: <https://it.wikireading.ru/4163> (дата обращения: 19.10.2021).
3. Дэвис Стефан Р. С++ для «чайников», 4-е издание. : Пер. с англ. : — М. : «И. Д. Вильямс», 2003.
4. Обработка строк: стандартная библиотека string // Программирование [Электронный ресурс]. URL: <https://prog-cpp.ru/c-string/> (дата обращения 19.10.2021).
5. Введение в классы С++ // Программирование на С и С++ [Электронный ресурс]. URL: <http://www.c-cpp.ru/books/vvedenie-v-klassy-s> (дата обращения: 19.10.2021).
6. cplusplus reference string // cplusplus.com [Электронный ресурс]. URL: <https://www.cplusplus.com/reference/string/string/> (дата обращения: 20.10.2021).
7. Керниган Брайан У., Ритчи Дэвис М. Язык программирования С, 2-е изд. : Пер. с англ. — М.: ООО «И.Д. Вильямс», 2017.
8. Бьерн Страуструп Язык программирования С++. Специальное издание. Пер. с англ. — М.: Издательство Бином, 2011 г. — 1136 с: ил.

ПРИЛОЖЕНИЕ

**МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра РЭС**

**ОТЧЕТ
по лабораторной работе № А
по дисциплине «Информатика»
Тема:**

Студент(ка) гр.

Фамилия И.О.

Преподаватель

Фамилия И.О.

Санкт-Петербург
202А

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
1. СТРУКТУРА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ	4
1.1. Титульный лист	4
1.2. Содержание.....	5
1.3. Спецификация задания	5
1.4. Формализованное описание алгоритма решения задачи	5
1.5. Блок-схема алгоритма.....	6
1.6. Выбор и обоснование типов переменных.	7
1.7. Вводимые и выводимые параметры и их типы	7
1.8. Структура проекта, перечисление нужных файлов.....	7
1.9. Результаты лабораторной работы	9
2. СОЗДАНИЕ ПРОЕКТА В СРЕДЕ РАЗРАБОТКИ MICROSOFT VISUAL STUDIO	10
3. ЭЛЕМЕНТЫ ЯЗЫКА Си.....	17
3.1. Типы данных.....	17
3.2. Переменные	18
3.3. Арифметические операции	18
4. ЧИСЛА И ИХ ПРЕДСТАВЛЕНИЕ В МАШИННЫХ КОДАХ	19
4.1. Управляющие элементы языка Си	19
4.2. Блоки кода.....	20
4.3. Функции и блоки.....	21
4.4. Задание на лабораторную работу № 1	22
5. ОТРИЦАТЕЛЬНЫЕ ЧИСЛА В ДВОИЧНОМ КОДЕ	23
5.1. Шестнадцатеричный код.....	23
5.2. Функции и структурирование кода	26
5.3. Глобальные переменные	27
5.4. Параметры, передаваемые по значению.....	27
5.5. Передача параметров по указателю	28
5.6. Передача параметров по ссылке (C++).....	28
5.7. Отладка функций	29

5.8. Задание на лабораторную работу № 2	29
5.9. Пояснения к лабораторной работе № 2	29
6. ИНДЕКСИРОВАННЫЕ ПЕРЕМЕННЫЕ	31
6.1. Классификация массивов	31
6.2. Кодирование символов	33
6.3. Инициализация и представление массивов в коде	34
6.4. Выход за границы массива	37
6.5. Динамические массивы	38
6.6. Работа с массивами	39
6.7. Задание на лабораторную работу № 3	41
6.8. Пояснения к лабораторной работе № 3	41
7. СТРУКТУРЫ	45
7.1. Состав структуры	47
7.2. Особенности работы со структурами	47
7.3. Передача структур в функции	47
7.4. Битовые поля и объединения	48
7.5. Задание на лабораторную работу № 4	49
7.6. Пояснения к лабораторной работе № 4	49
8. СТРУКТУРЫ C++ И КЛАССЫ	51
8.1. Конструкторы в структурах	51
8.2. Полиморфизм конструкторов	52
8.3. Наследование в типах C++	53
8.4. Инкапсуляция переменных в классах	53
8.5. Ограничения доступа к элементам и функциям классов	53
8.6. Переменные внутри классов	54
8.7. Классы как переменные и массивы	57
8.8. Задание на лабораторную работу № 5	57
8.9. Пояснения к лабораторной работе № 5	58
9. СТАНДАРТНЫЕ БИБЛИОТЕКИ C++	60
9.1. Класс <i>vector</i> – векторы, динамические массивы	60
9.2. Задание на лабораторную работу № 6	62

9.3. Пояснения к лабораторной работе № 6	62
9.4. Поточковый ввод и вывод.....	63
9.5. Задание на лабораторную работу № 7	65
10. ИСПОЛЬЗУЕМЫЕ АЛГОРИТМЫ	66
10.1. Алгоритм простого перебора.....	66
10.2. Алгоритм простого перебора (вариант 2).....	66
10.3. Сортировка методом «камешка»	66
10.4. Сортировка методом «камешка» (вариант 2).....	67
10.5. Поиск методом дихотомии (деления на 2)	67
10.6. Поиск методом дихотомии (вариант 2)	68
ЗАКЛЮЧЕНИЕ	69
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	70
ПРИЛОЖЕНИЕ	71

Ситников Иван Юрьевич
Самсонова Татьяна Евгеньевна
Козлова Полина Николаевна

Лабораторный практикум по курсу «Информатика»

Учебное пособие

Редактор _____

Подписано в печать _____. Формат 60×84 1/16.

Бумага офсетная. Печать цифровая. Печ. л. 4,25.

Гарнитура «Times New Roman». Тираж XXX экз. Заказ XXX.

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5