Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ
(ТУСУР)**

Кафедра компьютерных систем в управлении и проектировании
(КСУП)

А.Е. Горяинов

Введение в программирование на языке Си++

Учебно-методическое пособие

2015

Горяинов А.Е.

Введение в программирование на языке Си++: Учебно-методическое пособие / А.Е. Горяинов. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2015. – 126 с.

Пособие содержит теоретический материал по языку Си++ и основам программирования, требования к выполнению лабораторных работ, примеры написания программ, задания (часть) по дисциплине «Объектно-ориентированное программирование» для направления направлений 220400.62 «Управление в технических системах» и 230100.62 «Информатика и вычислительная техника» профиль САПР.

Оглавление

1 Введение 5
2 Основные элементы языка Си++ 7

2.1 Среда разработки Visual Studio ... 8
2.2 Программа «Hello, World!» ... 12
2.3 Типы данных и переменные ... 15
2.4 Блок кода ... 19
2.5 Ввод и вывод данных в консоли ... 21
2.6 Ветвление и операторы выбора ... 22
2.7 Операторы цикла ... 26
2.8 Операторы передачи управления ... 28
2.9 Явное и неявное преобразование типов данных ... 29
2.10 Этапы сборки программы на языке Си++ и её выполнение ... 31

3 Массивы и работа с адресами 34

3.1 Указатели и адресная арифметика ... 34
3.2 Массивы ... 39
3.3 Строки ... 42
3.4 Генерация случайных чисел ... 45

4 Функции 47

4.1 Определение, описание и вызов функций ... 47
4.2 Способы передачи переменных в функции ... 52
4.3 Рекурсивные функции ... 57
4.4 Перегрузка функций ... 59
4.5 Глобальные переменные ... 61

5 Пользовательские типы данных 63

5.1 Перечисления ... 63
5.2 Структуры ... 67
5.3 Объединения ... 71
5.4 Динамическая память ... 71
5.5 Классические структуры данных ... 76

6 Техники написания качественного кода 85

6.1 Правила именования и стандарты оформления кода 86
6.2 Комментирование кода 93
6.3 Инструменты отладки в среде разработки 94
6.4 Деление исходного кода на множество файлов 99
6.5 Разделение Модель-Вид 102
6.6 Юнит-тесты 104

7 Задания к лабораторным работам 108

7.1 Задание к лабораторной работе №1 108
7.2 Задание к лабораторной работе №2 110
7.3 Задание к лабораторной работе №3 112
7.4 Задание к лабораторной работе №4 114
7.5 Указания к составлению отчетов 116

8 Заключение 119

Список рекомендуемой литературы 120

Приложение А. Защита от неправильного ввода данных пользователем 121

1 Введение

Данное учебно-методическое пособие составлено для бакалавров направления 220400.62 «Управление в технических системах» и 230100.62 «Информатика и вычислительная техника» профиль САПР для прохождения курса «Объектно-ориентированное программирование». В данном пособии содержатся теоретический материал и лабораторные, являющиеся введением в язык Си++, необходимым для успешного освоения объектно-ориентированного программирования.

Преимуществом данного пособия является направленность на изучение не только синтаксиса языка, но и воспитание в студентах практических навыков разработки программного обеспечения, таких как владение современными средами разработки, автоматизированное тестирование, соблюдение правил оформления кода и другие техники написания качественного кода. Всё это является обязательными требованиями к современным программистам.

Данный курс предполагает, что студент в достаточной степени владеет знаниями в области алгоритмизации, способен прочесть блок-схемы, а также способен к самостоятельному составлению алгоритмов.

Язык Си++ выбран по нескольким причинам. Во-первых, Си++ является базовым языком для языков *Java*, *C#*, *Objective-C*, *php*. На данных языках программирования ведется разработка свыше 90% всех программных продуктов. Изучение языка Си++ в данном случае позволит легко освоить другой язык программирования из списка, что немаловажно при будущем трудоустройстве. Во-вторых, Си++ даёт больше возможностей для работы с памятью персонального компьютера, что крайне важно для понимания работы программ. Это несколько усложняет процесс изучения языка, однако приобретенное знание значительно повысит эффективность создаваемых вами алгоритмов и программных решений.

Стоит отметить, что данное пособие излагает лишь краткий материал по основам языка Си++, необходимого для выполнения лабораторных работ. Для

более основательного изучения языка и основ программирования, студентам предлагается ознакомиться с рекомендуемой литературой, а главное, как можно больше писать кода!

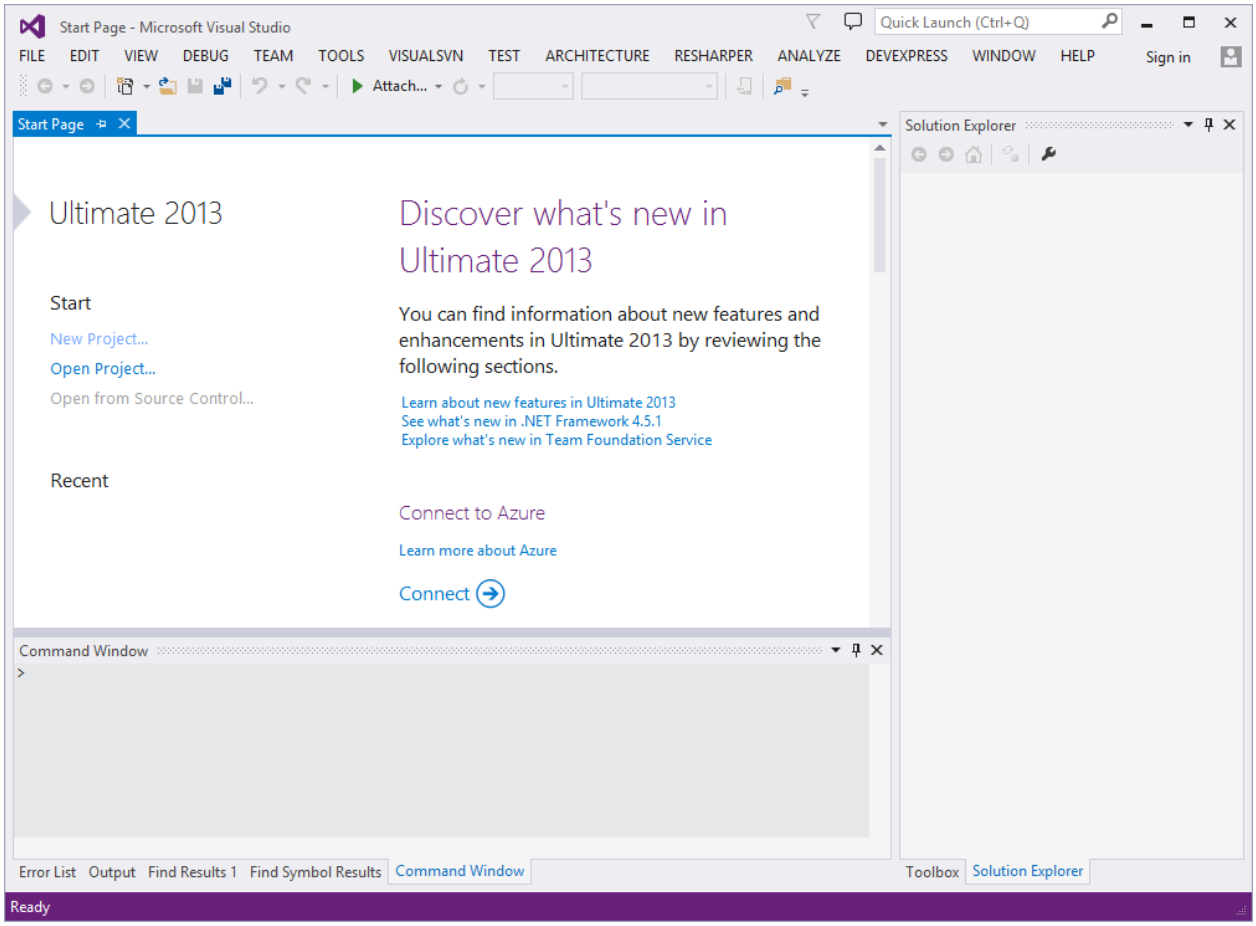
2 Основные элементы языка Си++

В данной главе рассмотрены базовые алгоритмические элементы языка Си++, необходимые для реализации простейших программ - от создания переменных до работы с консолью. Однако в первую очередь читателю предлагается ознакомиться со ***интегрированной средой разработки*** *Visual Studio.*

Интегрированная среда разработки, *IDE* (англ. *Integrated development environment*) – это комплекс утилит, используемых программистами для разработки программного обеспечения, таких как редактор текста исходного кода, компилятор, средства отладки и т.п. Существует множество сред разработки в зависимости от языка программирования, платформы разработки или сложности разрабатываемых программных решений, однако принцип работы в них схожий. В данном курсе будет рассмотрена среда разработки *Visual Studio* от компании *Microsoft*, однако читатель может изучать язык Си++ в любой другой *IDE*.

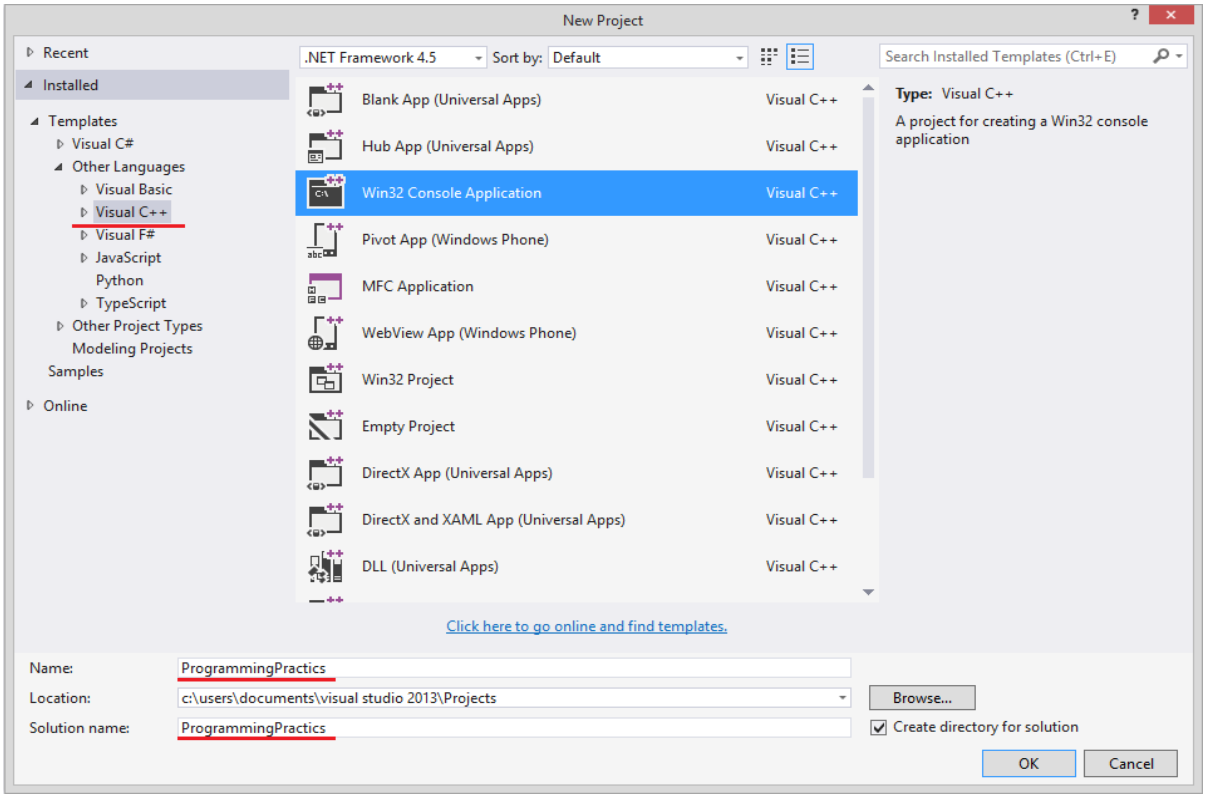
2.1 Среда разработки Visual Studio

После запуска Visual Studio перед нами возникнет следующее стартовое окно:

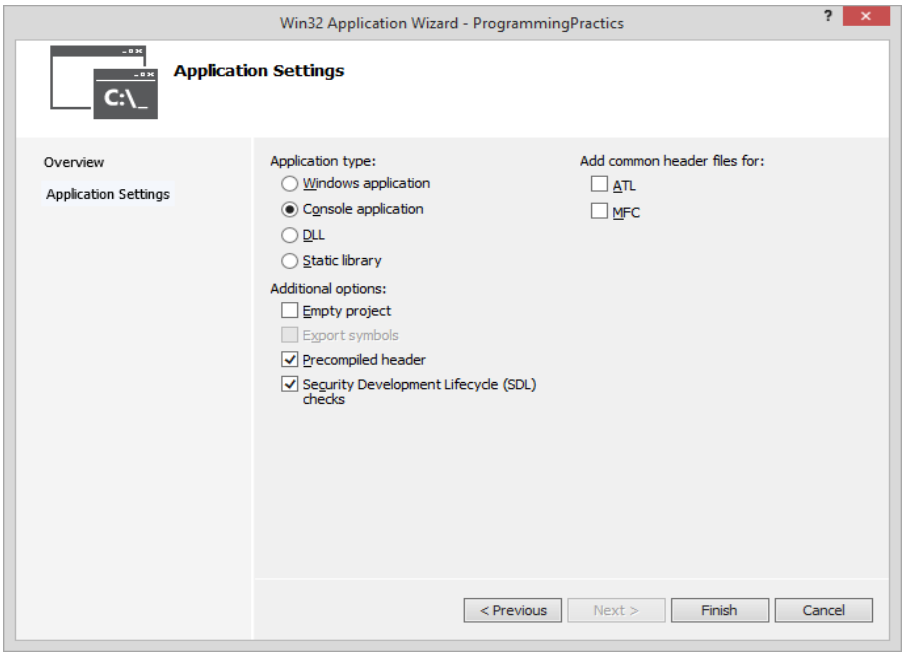


Программирование в среде *Visual Studio* выполняется в так называемых ***решениях***, состоящее из одного и более ***проектов***. Каждый проект содержит файлы исходного кода, а также настройки для их компиляции. Создадим проект, нажав *New Project* на стартовом окне или в меню *File*. Появится следующее окно создания проекта. В среде *Visual Studio* существует множество готовых шаблонов для различных проектов. Шаблоны определяют язык программирования, платформу или операционную систему будущей программы, вид выходной сборки – исполняемое приложение или подключаемая библиотека - , пользовательские интерфейсы – консоль, графические интерфейсы Winforms, WPF или других технологий, а также содержит базовые настройки компиляции исходного кода. В рамках данного курса рассматривается проект консольного исполняемого приложения для языка Си++ *Win32 Console Application*, т.е.

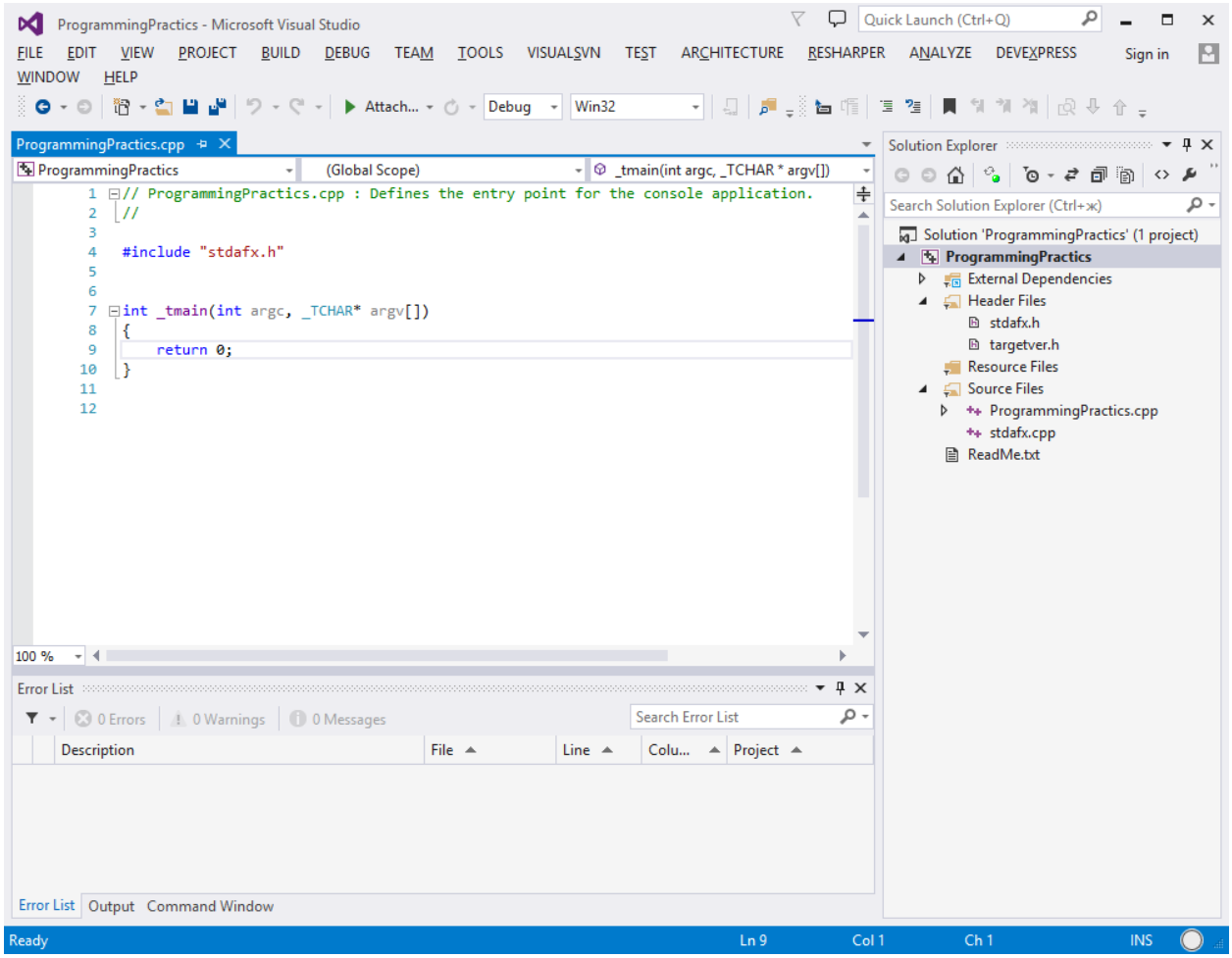
результатом компиляции будет исполняемый файл с расширением *.exe*. Выбрав необходимый шаблон проекта, требуется задать имя проекта и решения, а также их расположение на компьютере.



Имя проекта должно отражать назначение программы. Так как данный проект создается для прохождения курса по программированию, предлагается название *ProgrammingPractics*. При нажатии клавиши *Ok* появится мастер создания консольного проекта:



В данном мастере используйте значения, установленные по умолчанию, и просто нажмите кнопку *Finish* для завершения процесса создания проекта. После завершения перед вами откроется главное окно *Visual Studio* с новым решением:



Главное окно представлено:

1) Редактор исходного кода.
2) Панель «Обозреватель решения» (*Solution Explorer*) – показывает все файлы, хранящиеся в решении и проекте. Двойной щелчок по файлу откроет файл в редакторе исходного кода.
3) Панель «Список ошибок» (*Error List*) – список текущих ошибок, возникших в процессе последней компиляции решения. Отображает название ошибки, файл её расположения и номер строки, в которой ошибка располагается. Двойной щелчок по ошибке позволяет быстро перейти с нужной строке в редакторе исходного кода.

4) Главное меню и панели инструментов.

Если какие-либо панели отсутствуют по умолчанию, вы можете включить их в меню *View*. Остальные панели в рамках данного курса либо не понадобятся, либо об их использовании будет рассказано отдельно.

Скомпилируем проект, нажав кнопку *Attach* на панели инструментов вверху, либо нажав клавишу *F*5, либо выбрав пункт *Bebug→Start Debugging*. Это приведет к компиляции текущего проекта и его автоматическому запуску. Так как в настоящем проекте пока нет никаких инструкций, программа появится на экране и тут же завершится.

Теперь зайдем в каталог с созданным решением через файловую систему, чтобы посмотреть представление решения на жестком диске. В папке с решением *ProgrammingPractics* можно найти папку *Debug*, папку проекта *ProgrammingPractics* и файл *ProgrammingPractics.sln*. Указанный файл содержит основную информацию о вашем решении, в частности, данные о версии программы и список проектов. Во вложенной папке *ProgrammingPractics* находятся все файлы исходного кода вашего проекта, а также файл *ProgrammingPractics.vcxproj*, хранящий данные для *Visual Studio* о всех файлах исходного кода в проекте и настройки компиляции проекта.

Стоит сразу обратить внимание на тот момент, что физическое копирование файла в папку с проектом не добавит исходный код в проект. Для добавления файла в проект обязательно необходимо воспользоваться панелью *Solution Explorer*, щелкнув правой кнопкой по имени проекта и выбрав пункт «Добавить существующий файл». В противном случае, проект не будет участвовать в процессе компиляции программы.

Папка *Debug* содержит последний результат компиляции программы. Если компиляция прошла успешно, то в папке будет присутствовать файл *ProgrammingPractics.exe*. Это и есть исполняемый файл вашей программы, который может быть скопирован в другое расположение для работы с приложе-

нием. В папке также могут присутствовать файлы других расширений, это временные файлы, необходимые для отладки или других вспомогательных целей среды разработки. Для работы исполняемого файла они не нужны.

2.2 Программа «Hello, World!»

Рассмотрим простейшую программу на языке Си++, выполняющую вывод на экран текста «*Hello, World!*»:

program.cpp

```
#include "stdafx.h"
#include <iostream.h>

using namespace std;

int _tmain()
{
      cout << "Hello, World!\n";
      return 0;
}
```

Первая строка

```
#include "stdafx.h"
```

выполняет подключение библиотеки *stdafx.h* с помощью директивы препроцессора *#include*. ***Препроцессор*** – программа, подготавливающая код программы к компиляции. После обработки кода препроцессором, итоговый код передается компилятору. Так называемые ***директивы препроцессора*** позволяют программисту указать какие конкретно действия препроцессор должен сделать с исходным кодом. Формат директив выглядит следующим образом:

```
#ключевоеСлово параметры
```

Таким образом, строка должна начинаться с возможных пробельных символов и обязательным символом #. Далее идёт одно из предопределенных ключевых слов и параметры, связанные с данным ключевым словом. Ключевое слово *include* выполняет подстановку текста из указанного файла на своё место. Имя подставляемого файла может быть указанного либо в кавычках, либо в угловых скобках. Указание имени в кавычках, как это сделано для файла

stdafx.h, означает, что подключаемый файл находится в том же каталоге, что и данный файл, в котором выполняется директива. В данном случае, файл *stdafx.h* находится в том же каталоге, что и файл *program.cpp*. Имя файла также может быть указано в угловых скобках, так, как это сделано во второй строке программы:

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

Угловые скобки означают, что подключаемый файл находится в каталоге среды разработки. В каталоге среды разработки находятся файлы стандартных библиотек языка Си++. Например, стандартная библиотека *iostream.h* (*input/output stream*) содержит функции для вывода данных на экран и ввод данных с клавиатуры.

Далее идёт пустая строка. Пустые строки не имеют никакого значения с точки зрения выполнения программы, однако они служат хорошим визуальным разделением различных частей программы. В данном случае, пустая строка визуально отделяет подключение библиотек от основной программы.

Строка

```
using namespace std;
```

указывает, что следующий код будет использовать пространство имен *std*. Данная строка необходима для сокращения обращения к функциям вывода данных на экран.

В следующей строке

```
int _tmain()
```

происходит объявление главной функции программы. Подробно об объявлении функций рассказано в п.4.1, сейчас же стоит отметить, что функция *_tmain* является обязательной для исполняемой программы, так как она является ***точкой входа*** – с этой функции начинается выполнение скомпилированной программы.

В следующей строке находится открывающаяся фигурная скобка, обозначающая начало реализации функции *_tmain*. Каждой открывающейся фигурной скобке должна соответствовать закрывающаяся, в противном случае компиляция программы приведет к ошибке. Фигурные скобки и всё, что находится между ними, называется ***блоком кода***. В блоке кода последовательно располагаются ***операторы*** – выражения языка Си++, выполняющие некоторое действие. Последовательность операторов составляют алгоритм, т.е. последовательность действий, выполняемых компьютером. Все операторы выполняются строго друг за другом в том порядке, в котором они записаны в исходном коде. Любой оператор начнет своё выполнение только после завершения выполнения предыдущего. В нашем примере, весь алгоритм состоит только из одного оператора – следующая строка

```
cout << "Hello, World!\n";
```

является ключевой, так как именно она выполняет вывод текста на экран. В данной строке выполняется передача экранированного в двойных кавычках текста "*Hello, World*!\n" потоку вывода в консоль *cout* (*console output*) с помощью оператора ≪, таким образом текст будет выведен на экран. Текст в языке Си++ заключается в кавычки. Обратите внимание на *n* в конце строки. Со знака \ начинаются спецсимволы, используемые при выводе текста на экран. Например, *n* не будет выведен на экран в качестве текста, вместо него каретка вывода текста в консоли будет перемещена на новую строку.

Передача данных в поток *cout* завершается символом ;. Данный символ называется ***разделителем*** – он разделяет операторы языки Си++ друг от друга. Разделителем должен завершаться любой оператор.

Строка

```
return 0;
```

обозначает завершение функции *_tmain* и, таким образом, всей программы. 0 будет возвращен в операционную систему как результат выполнения всей программы. Принято возвращать 0 в случае успешного завершения

программы. Возвращение другого значения, например, 1, обозначает, что программа завершилась некорректно, при этом конкретное значение может обозначать конкретный вид ошибки. Таким образом, пользователь, запустивший программу по вернувшемуся в операционную систему значению может определить отработала ли программа корректно или нет.

Вопросы для проверки:

1) Как в языке Си++ объявляется комментарий? Для чего нужны комментарии?

2) Что такое директива препроцессора? Что выполняет директива препроцессора *include*?

3) Чем отличается для директивы *include* указание названия библиотеки в кавычках (например, “*stdafx.h*”) и в угловых скобках (например, <*stdio.h*>)?

4) Для чего необходимо подключать библиотеку <*stdio.h*>? Какие функции содержит эта библиотека?

5) Что такое функция *Main*()? Для чего она нужна?

6) Что выполняет функция *printf*(“*Hello, World*!\n”)? Как расшифровывается *printf*? Для чего нужен спецсимвол “\n” в конце строки, если при выполнении программы он не отображается на экране?

7) Что делает оператор “;”? Для чего нужно ставить точку с запятой в языке Си++?

2.3 Типы данных и переменные

Язык Си++ относится к типизированным языкам программирования, то есть любые данные в программе относятся к некоторому типу данных, определенному в системе типов данных языка. Тип данных определяет представление данных в виде двоичного кода, а также множество допустимых операций над ними. Например, для целочисленного типа данных определяются арифметические операции (сложение, умножение, деление), когда для логиче-

ского типа данных определены логические операции (конъюнкция, дизъюнкция, отрицание). Также для каждого типа данных определяется размер одной переменной в байтах. Так, для хранения одного целого числа требуется 4 байта, логической переменной 1 байт, а для вещественных чисел с плавающей точкой необходимо 8 байт. Далее перечислены основные типы данных в языке Си++, их название – ключевое слово, используемое для создания переменных -, размер в байтах для представления одного значения, а также диапазон допустимых значений.

Тип данных	**Ключевое слово**	**Размер**	**Диапазон**
Целочисленный	int	4	от -2 147 483 648 до 2 147 483 647
Вещественный	float	4	$\pm 3{,}4\times10^{\pm38}$ (7 знаков)
Вещественный двойной точности	double	8	$\pm 1{,}7\times10^{\pm308}$ (15 знаков)
Логический	bool	1	false или true
Символьный	char	1	от -128 до 127

Представление вещественных типов данных отличается от хранения целочисленных или дискретных типов. Любое вещественное число можно записать в виде $n = m\times10^{p}$, где m – множитель, содержащий все цифры числа (мантисса), а p – целое число, называемое порядком. Например, число 314,159 будет представлено как $3{,}14159\times10^{2}$. Число -0,00256196 будет представлено как $-2{,}56196\times10^{-3}$. В случае машинного представления вещественных чисел типа float, для хранения мантиссы выделяется 3 байта, а для хранения порядка 1 байт. Это позволяет хранить дробную часть с точностью до 7 знака после запятой, а порядок в диапазоне от -38 до 38. Округление дробных значений до 7 знака после запятой необходимо учитывать при реализации программ. Так, с точки зрения языка Си++ числа 1,6180339887 и 1,6180339895 будут округлены до 1,6180340 и, соответственно, станут равными при сравнении.

Также существует тип данных void, множество значений этого типа пусто, то есть переменной данного типа нельзя присвоить какое-либо значение. Использование такого специфического типа данных будет объяснено в п.4.1.

Создание переменных выполняется согласно следующему формату:

```
типДанных имяПеременной;
```

где *типДанных* – это ключевое слово, название типа данных, а *имяПеременной* – уникальный, не использованный ранее идентификатор переменной. В конце ставится разделитель «;», обозначающий завершение инструкции. Например, инструкция

```
int a;
```

создаст переменную с именем *a*, при условии, что в программе отсутствует переменная с таким именем. Аналогично создание переменных других типов:

```
float b;
char c;
bool d;
```

После создания переменной ей можно присваивать значения. Присвоение значения переменной выполняется с помощью оператора присваивания «=»:

```
a = 5;
b = 3.17;
c = ‘A’;
d = true;
```

Оператор выполняет присваивание выражения, стоящего справа от оператора, в переменную, стоящую слева. Подробнее о типах данных, создании и работе с переменными, а также о допустимых операциях можно прочитать в [1-3]. Далее представлена программа, выполняющая создание двух целочисленных переменных и вывод их суммы на экран:

program.cpp

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

int _tmain()
{
   int a;  // Объявление целочисленной переменной
   a = 5;  // Инициализация переменной значением
   int b = 3;   // Допустимо одновременное объявление и инициализация

   // Вывод значения переменной на экран
   cout << "\n Variable a equals " << a;

   // Значение переменной после запятой подставится вместо %d
```

```
    cout << "\n Variable b equals " << b;

    // Можно выводить сразу результат сложения
    cout << "\n Summ of a and b equals" << a + b;
}
```

Обратите внимание на форму записи *cout*. Знаки ≪ отделяют экранированный текст и имя переменной. Такая форма записи равносильна следующей:

```
cout << "\n Variable a equals ";
cout << a;
```

Ранее отмечалось, что для каждого типа данных существует собственный набор операций и эти операции могут отличаться друг от друга. Например, следующий код:

```
int a = 5;
int b = 3;

float z = 5.0;
float y = 3.0;

cout << "\nFloat variables division: "<< z/y;
cout << "\nInteger variables division: " << a/b;
```

В результате выполнения программы на экране будут представлены два результата деления, однако, несмотря на одинаковые исходные значения переменных, результаты деления будут отличаться. При делении вещественных чисел результат будет равен 1.666, а при делении целочисленных переменных результат будет равен 1. Это объясняется тем, что целочисленные переменные не могут хранить дробные значения, таким образом, любой остаток от деления целочисленных значений будет отброшен.

Вопросы для проверки:

1) Что такое переменная в языке Си++? Что такое тип данных?

2) Чем отличаются целочисленные типы данных от вещественных (с плавающей точкой)?

3) Как в языке Си++ происходит объявление и инициализация переменной какого-либо типа?

2.4 Блок кода

Каждый оператор языка Си++ заканчивается и идентифицируется разделителем «;». Любое выражение, после которого поставлен символ «;», воспринимается компилятором как отдельный оператор. Все операторы выполняются последовательно, т.е. один оператор начинает своё выполнение только после завершения выполнения предыдущего. Ранее в п.2.2 уже упоминалось понятие блока кода при описании работы функции *main*(). Однако блок кода существует не только в рамках функции, но может использоваться отдельно.

Блоки кода обладают следующими свойствами:

- Блок кода позволяет объединить несколько операторов в один.
- Один блока кода может содержать в себе любое количество других блоков кода.
- Каждый блок кода создает уникальную ***область видимости*** – участок кода, ограничивающий уникальность имен переменных и время их жизни.
- В любом вложенном блоке кода доступны переменные, находящиеся во внешнем блоке кода.
- Блок кода не может быть закрыт, пока не будут закрыты все внутренние блоки кода.

Рассмотрим на следующем примере:

```
int _tmain()
{                       //Создание первого блока кода
   int a;
   a = 5;
   {                    //Создание второго блока кода
      int b = 3;
      int c = a + b;
      a = 7;
      int e;
   }                    //Закрытие второго блока кода
   //int d = c;         //Ошибка компиляции!
   int e
   {                    //Создание третьего блока кода
      int b = 7;
      //int a = 10;     //Ошибка компиляции!
   }                    //Закрытие третьего блока кода
}                       //Закрытие первого блока кода
```

Первый блок кода создается для функции *main*(). Первыми его инструкциями выполняется объявление и инициализация переменной *a*. Далее создается второй, вложенный блок кода. В нём создается новая переменная *b* и инициализируется значением. Во вложенном блоке кода доступны все переменные, созданные во внешнем блоке кода, например, переменная *a*. Внутри вложенного блока кода можно получить значение переменной *a*, а также его изменить.

Вложенный блок кода определяет уникальную область видимости для переменных *b, c* и *e*. Это означает, что: а) эти переменные перестанут существовать после закрытия блока кода; б) после закрытия блока кода имена *b, c* и *e* можно будет вновь использовать для других переменных.

После закрытия блока кода есть закомментированная строка кода. Если её раскомментировать и попытаться скомпилировать программу, компилятор сообщит об ошибке. Ошибка связана с тем, что после закрытия второго блока кода переменная *c* уже не будет существовать, и, соответственно, использовать её для инициализации переменной *d* будет невозможно.

В следующей строке создается новая переменная с именем *e*. Во втором блоке кода уже создавалась переменная с таким именем, но так как второй блок кода на момент выполнения данной инструкции уже завершен, имя *e* свободно и может быть использовано для создания новой переменной, в том числе и другого типа данных.

В третьем блоке кода, опять же являющегося вложенным по отношению к первому происходит создание переменной *b*. И опять же имя переменной *b* свободно, так как после закрытия второго блока кода, данное имя освободилось. Однако создать переменную с именем *a* невозможно, так как переменная *a* была объявлена в первом блоке кода, а значит, доступна в третьем. Поскольку в одной области видимости не может существовать двух переменных с одинаковым именем, такая строка приведет к ошибке компиляции. Читателю предлагается набрать указанный пример и попробовать раскомментировать указанные строки для лучшего понимания работы блоков кода.

Вопросы для проверки:

1) Что такое область видимости?
2) Как создать область видимости?
3) Поясните понятие времени жизни переменной?

2.5 Ввод и вывод данных в консоли

В языке Си++ существует два способа работы с консолью: форматный и потоковый. В данном пособии будет рассмотрен только потоковый ввод и вывод как более безопасный, однако читателю будет полезно изучить оба подхода, а главное, их отличия, преимущества и недостатки.

Вывод на экран текста и значений переменных уже был рассмотрен ранее. Выполняется вывод с помощью потока *cout* (*console output*):

```
cout << “Hello, ”;
cout << “ World!”;
```

Два последовательных вывода на экран можно объединить в одной строке:

```
cout << “Hello, ” << “World!”;
```

Вывод значений переменных осуществляется следующим образом:

```
int a = 7;
cout << a;
```

Так можно вывести на экран значения переменных всех базовых типов данных.

Программа должна уметь не только выводить данные на экран монитора, но и получать данные, вводимые пользователем с клавиатуры. Для этого используется поток *cin* (*console input*):

```
int a;
cin >> a;
float b;
cin >> b;
```

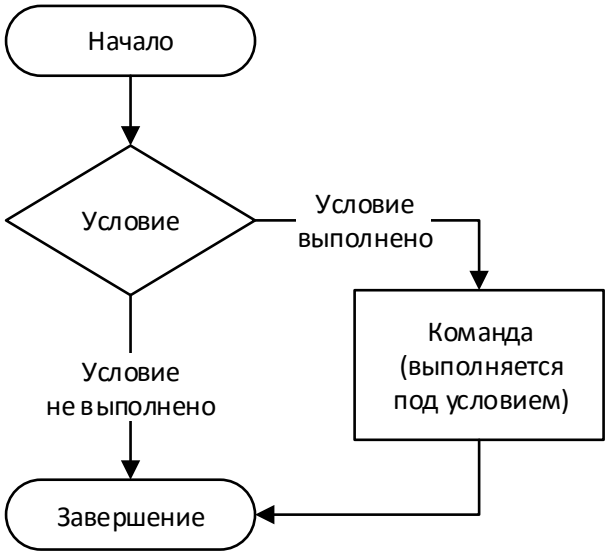
Чтение данных с клавиатуры может быть выполнено как в только что объявленную, но не инициализированную переменную, так и в ранее созданную. При этом значение, которое хранилось в переменной ранее, будет потеряно.

Вопросы для самопроверки:

1) Что будет, если попытаться вывести на экран значение переменной, которая еще не была инициализирована никаким значением?

2.6 Ветвление и операторы выбора

Ветвление является важным механизмом в алгоритмизации. ***Ветвление*** - это конструкция языка программирования, обеспечивающая выполнение определенной команды или набора команд только при условии истинности некоторого логического выражения, либо выполнение одной из нескольких команд (наборов команд) в зависимости от значения некоторого выражения.



Ветвление в языке Си++ реализуется с помощью ключевого слова *if*:

```
if (условие)
{
   //Операторы под условием
}
```

где условие – логическое значение или выражение. Например:

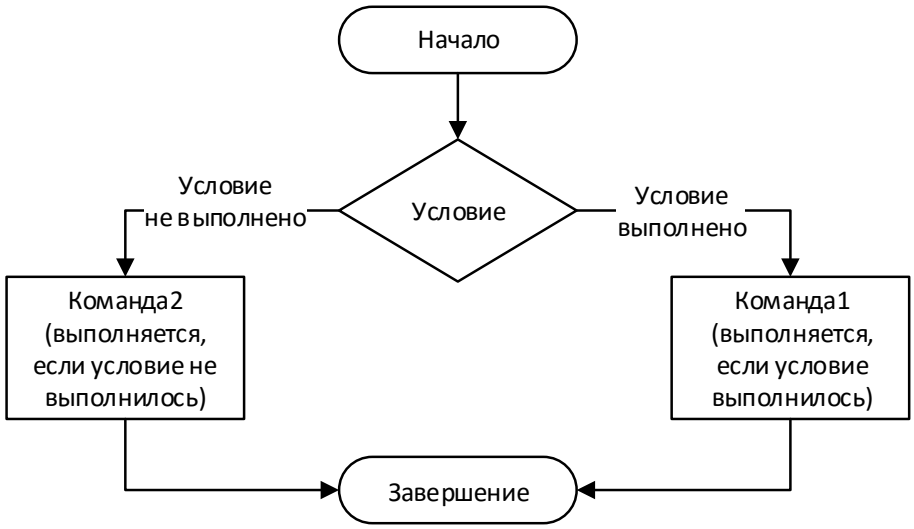
```
bool condition = true;
if (condition) //используйте логическую переменную как условие
{
  cout << "Условие истинно";
}

condition = 10 < 7;
if (condition)
{
```

```
  cout << "Этот оператор не сработает, так как 10 больше 7";
}

int value;
cin >> value;
if (value > 10) //используйте логические выражения как условия
{
  cout <<"Этот оператор сработает, если пользователь введет число больше 10";
}
```

Если необходимо сделать механизм, когда в случае выполнения условия выполняется один набор команд, а в случае невыполнения – другой, используется ключевое слово *else*:




```
int value;
cin >> value;

if (value > 10)
{
  cout <<"Этот оператор сработает, если пользователь введет число больше 10";
}
else        //обратите внимание на использование слова else
{
  cout <<"Этот оператор сработает, если пользователь введет число меньше или
          равное 10";
}
```

Возможно создание вложенных условий. Для этого второе условие должно быть написано в блоке кода под условием или блоке *else*. Также возможна организация ***лестничной***, или ***каскадной***, структуры:

```
if (condition1)
{
  if (condition2)
  {
    cout <<"Этот оператор сработает, если истинны оба условия";
```

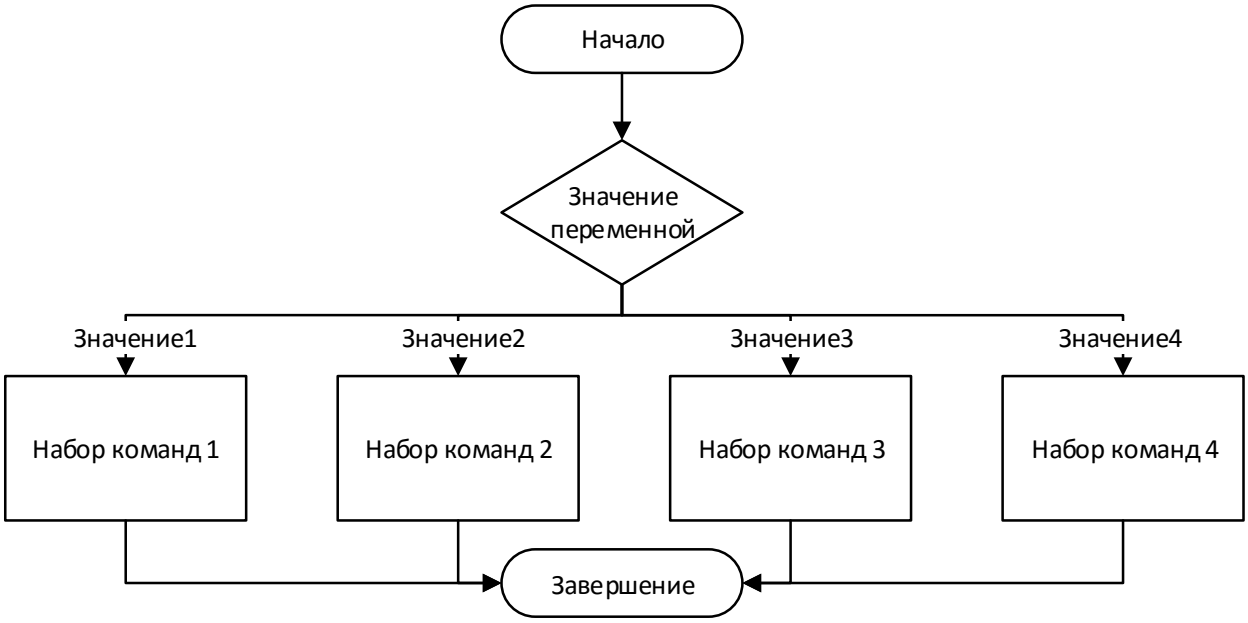
```
  }
  else            //else второго условия
  {
    cout <<"Этот оператор сработает, если истинно первое условие, а второе
           ложно";
  }
}
else        //else первого условия
{
  cout <<"Этот оператор сработает, если первое условие ложно";
}
```

Вложенность условий может быть неограниченной. Однако учтите, что уже вложенность третьего порядка становится сложной для восприятия человеком. На практике старайтесь не превышать вложенности больше четырех. Лестничная структура:

```
if (condition1)
{
  cout <<"Этот оператор сработает, если истинно первое условие";
}
else if (condition2)
{
  cout <<"Этот оператор сработает, если первое условие ложно, а второе ис-
тинно";
}
else
{
  cout <<"Этот оператор сработает, если оба условия ложны";
}
```

Лестничная структура может быть продолжена любым количеством условий.

Также в языке Си++ существует оператор ветвления switch, выполняющий один из наборов команд, связанных со определенным значением переменной:



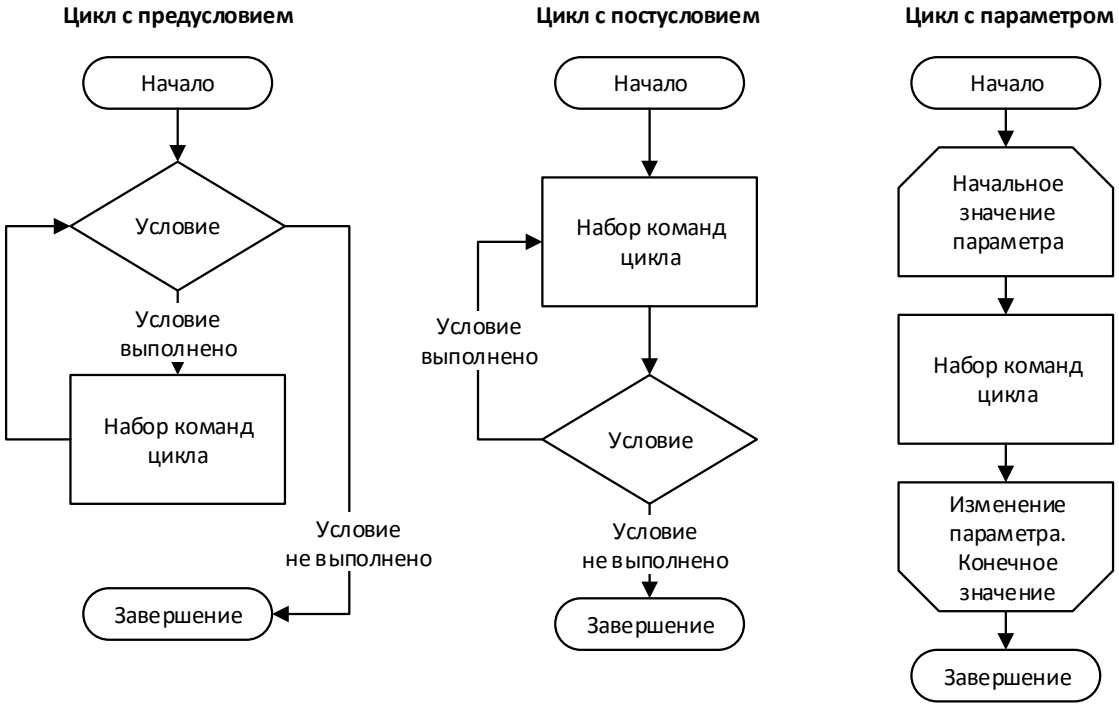
Пример использования оператора switch:

```
switch ( c )
{
  case 'A':
    cout << "Переменная с равна ‘A’";
    break;
  case 'B':
    cout << "Переменная с равна ‘B’";
    break;
         default:
  default:
    cout << "Переменная с равна любому другому значению";
    break;
}
```

После ключевого слова *switch* записывается имя переменной, от значения которой будет зависеть выполнения наборов инструкций. Ключевое слово *case* с указанием конкретного значения переменной, например, ‘A’ для символьной переменной, определяет набор значений, который будет выполняться при условии, если переменная под *switch* будет равна данному значению. После двоеточия указываются команда или набор команд для выполнения под этим условием. Последним оператором является оператор *break*, обозначающий завершения выполнения условного оператора *switch*. Вариантов case может быть указано неограниченное количество. Последним, но необязательным блоком идет блок с ключевым словом *default*: этот набор операторов выполнится, если значение переменной не подходит не под один случай *case*.

2.7 Операторы цикла

Цикл – конструкция языка программирования, реализующая повторное выполнение команд или набора команд до выполнения определенного условия. В языке Си++ есть три вида циклов – ***цикл с предусловием***, ***цикл с постусловием***, ***цикл с параметром***.



Цикл с предусловием реализуется с помощью ключевого слова while с последующим условием цикла и блоком кода:

```
int a = 0;
while (a < 5) // условный цикл с предусловием
{
  cout << "Value is " << a << " now" << endl;
  a++;
}
```

Содержимое фигурных скобок будет повторяться до тех пор, пока условие после while является истинным. Условие цикла проверяется перед каждой итерацией. Обратите внимание, что внутри цикла происходит увеличение переменной a на 1. В противном случае, цикл будет бесконечным.

Цикл с постусловием реализуется с помощью ключевого слова *do* с последующим блоком кода и ключевого слова *while* с условием:

```
int a = 0;
```

```
do // условный цикл с постусловием
{
  cout << "Value is " << a << " now" << endl;
  a++;
}
while (a < 5);
```

Важным отличием цикла с постусловием от цикла с предусловием является то, что условие будет проверяться после итерации. Таким образом, цикл с постусловием гарантировано выполнит хотя бы одну итерацию.

Цикл с параметром реализуется с помощью ключевого слова *for* и имеет следующий синтаксис:

```
for (инициализация параметра; условие цикла; изменение параметра)
{
  //тело цикла
}
```

Пример:

```
for (int i = 0; i < 10; i++)
{
  cout << "Value is " << i << " now" << endl;
}
```

Здесь в круглых скобках под *for* происходит объявление ***итератора*** – переменной, используемой для управления числом итераций цикла -; условие выполнения цикла, где итератор должен быть меньше 10; и способ изменения итератора – в данном случае, это увеличение итератора на 1.

Объявление и инициализация выполняются перед выполнением всего цикла, проверка условия выполняется перед каждой итерацией, изменение параметра выполняется после каждой итерации.

В качестве параметра может быть задана и ранее созданная переменная. В таком случае, переменную нужно только инициализировать значением без объявления. Преимуществом инициализации итератора в самом цикле является окончание области видимости итератора вне цикла, таким образом, имя переменной-итератора будет свободно для других циклов или участков кода. В качестве итератора могут быть переменные и других типов данных, например, вещественные или символьные, однако на практике такое решение встречается редко.

Условие выполнения цикла может быть сколько угодно сложным и составным и даже не использовать в своём расчете значение переменной-итератора. Опять же, на практике подобные решения встречаются редко, так как усложняют восприятие кода.

Итератор может меняться любыми другими операторами. Например:

```
i++
i--
i = i + 2
i *= 2;
i = SomeFunction(i)
```

Это расширяет возможности использования цикла с параметром, позволяя делать цикл с обратным отсчетом или цикл по четным/нечетным итерациям.

Циклы с параметрами удобно использовать для работы с массивами и другими структурами данных.

2.8 Операторы передачи управления

В языке Си++ существуют три оператора передачи управления в алгоритмах, необходимые для управления работой циклов, операторов ветвления и функций.

Оператор continue используется в теле цикла и выполняет переход на следующую итерацию:

```
for (int i = 0; i < 10; i++)
{
  cout << "Текст, выводимый в каждой итерации";
  if (i % 2 == 0) // Условие, при котором будет переход на следующую итерацию
  {
    continue; // Досрочный переход на следующую итерацию
  }
  cout << "Текст, выводимый только в нечетных итерациях";
}
```

Очевидно, что оператор *continue* следует применять только под некоторым условием. В примере выше показано, как оператор continue позволяет от-

сечь выполнение некоторых операторов в отдельных итерациях. В случае вложенных циклов оператор *continue* будет переходить на следующую итерацию только для внутреннего цикла.

Оператор *break* используется в теле цикла или оператора ветвления *switch* для указания завершения цикла или оператора ветвления. Пример использования оператора break:

```
int a = 10;
for (int i = 0; i < 10; i++)
{
  a--;
  if (i >= a)
  {
    cout << "I is more or equal A! Break of cycle";
    break; // этот оператор досрочно завершает цикл
  }
}
cout << " A is " << a;
```

Несмотря на то, что цикл с параметром объявлен на выполнение 10 итераций, оператор break прервет выполнения цикла на пятой итерации.

2.9 Явное и неявное преобразование типов данных

Язык Си++ относится к языкам с нестрогой типизацией. Это означает, что все данные в программе относятся к одному из возможных типов данных, однако проверка соответствия типов данных при выполнении различных операций осуществляется нестрого. Например, в целочисленную переменную можно поместить только целочисленное значение, а в вещественную переменную можно поместить только вещественное значение:

```
int a = 5;
double b = 7.5;
```

В случае языков со строгой типизацией операции между разными типами данных недопустимы. Например, такая простая операция, как сложение целого и вещественного чисел невозможна. На практике подобные ограничения сильно осложняют процесс написания кода. Для возможности осуществления операций между различными типами данных в языке Си++ существует механизмы явного и неявного преобразования типов данных.

Неявное преобразование для стандартных типов выполняется компилятором автоматически при условии, что преобразование не приведет к потере данных. Например:

```
int a = 5;
double b = a;
```

Обратите внимание на операцию присваивания при инициализации вещественной переменной. Слева от знака присваивания стоит вещественная переменная, а справа целочисленная. Поскольку для операции присваивания необходимо совпадение типов данных по обе стороны присваивания, перед операцией присваивания срабатывает механизм неявного преобразования целочисленного значения в вещественное. Данный механизм также будет срабатывать в следующей строке:

```
double value = 7.16f+15+'a'+true;
```

Под операциями сложения находятся четыре разных типа данных, а их результат сложения должен быть присвоен в переменную пятого типа данных. Однако благодаря неявному преобразованию типов данное выражение корректно отработает.

Явное преобразование выполняется с явным указанием целевого типа данных. Синтаксис явного преобразования:

```
(целевойТипДанных)имяИсходнойПеременной
```

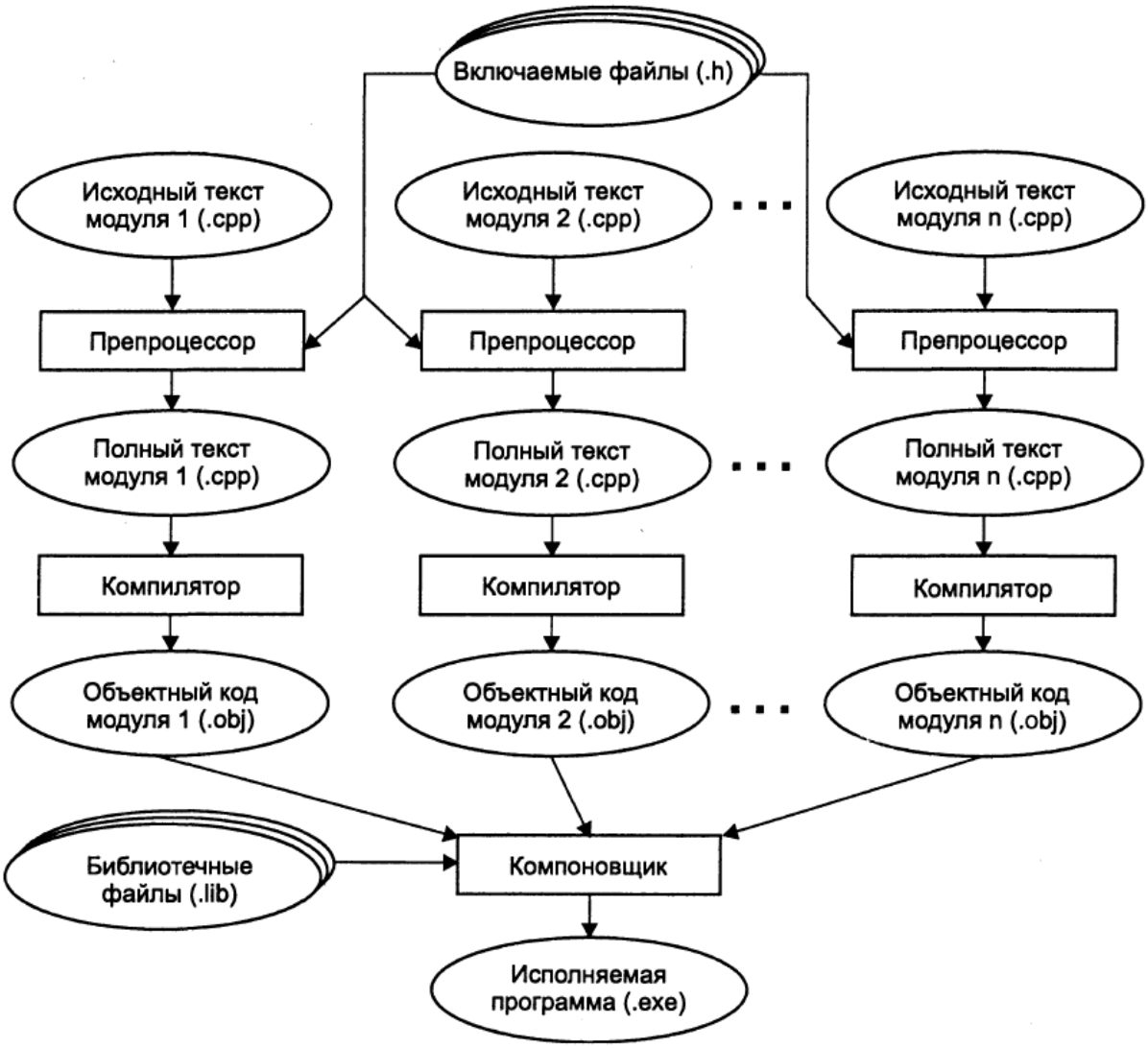
Пример:

```
double a = 7.16;
int b;
b = (int)a;   // Явное преобразование типа данных
```

Поскольку переменная *b* является целочисленной, она не может хранить дробную часть. Следовательно, в переменную *b* присвоиться только целая часть, а дробная часть будет отброшена. Поскольку явное преобразование связано с потерей данных, оно не выполняется автоматически компилятором, а только при явном указании программистом. Таким образом, разработчик показывает, что он осознанно идет на возможную потерю данных. Подробно преобразования типов данных описаны в [1].

2.10Этапы сборки программы на языке Си++ и её выполнение

Любой язык программирования является лишь средством описания алгоритмов, однако для процессора не существует понятия языка программирования. Процессор может выполнять только предопределенный набор команд, выраженных в машинном коде. Машинный код сложен для восприятия человеком, писать программы в машинных кодах тяжело для большинства людей. В определенный момент процесс написания кода был переосмыслен и был реализован новый подход: программисты будут писать исходный код на специальном языке программировании, а специальная программа – компилятор – будет выполнять перевод исходного кода с языка программирования в машинный код, понятный процессору. Такой подход в значительной степени упростил процесс написания кода и позволил создавать более сложные приложения. Таким образом, появились этапы сборки исполняемой программы из исходного кода.



Этапы создания исполняемой программы

Сначала заголовочные файлы и файлы исходного кода передаются ***препроцессору***. Задача препроцессора подготовить исходный код для компиляции. Подготовка заключается в выполнении ***директив препроцессора***, которые программист может написать в исходном коде, например, директива препроцессора *include*, выполняющая включение указанного текстового файла на собственную позицию. [1]

Обработанный препроцессором текст программы передается ***компилятору***, который выполняет лексемы, а затем на основе грамматики языка распознает выражения и операторы, построенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит

объектный модуль [11]. В случае возникновения ошибок процесс сборки завершается, а компилятор возвращает как результат список полученных ошибок. Ошибки компилятора можно отличить по букве ‘C’ в коде ошибки.

Объектные модули передаются ***компоновщику*** (англ. «*linker*»), формирующему ***исполняемый модуль*** программы. Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. На этапе компоновки могут возникнуть собственные ошибки, не связанные с синтаксисом языка, но также требующие устранения программистом. Ошибки компоновщика обозначаются буквами ‘LK’, как правило, их причины могут быть нетривиальны, и рекомендуется обращение к документации по ошибкам, например, [7].

Вопросы для самопроверки:

1) Перечислите основные этапы компиляции программы? Что выполняется на каждом этапе?

3 Массивы и работа с адресами

3.1 Указатели и адресная арифметика

Указатель – специальный тип данных языка Си++, предназначенный для хранения и работы с адресами памяти других переменных. Объявление переменной-указателя выполняется согласно следующему синтаксису:

```
ТипДанных* имяПеременнойУказателя;
```

Отличие объявления указателя от обычной переменной заключается в использовании знака * после имени данных. Примером объявления указателя является переменная *pointer* – обратите внимание на звездочку, установленную после типа данных *int*:

```
int* pointer;
```

В данном случае это означает что переменная-указатель *pointer* может хранить в себе адрес целочисленной переменной, но только целочисленной!

Рассмотрим следующий код:

```
int value;
int* pointer; // Объявление указателя на целочисленный тип данных
int anotherValue;

value = 5;
pointer = &value;   // Операция взятия адреса переменной
          // pointer может хранить адрес только
          // целочисленной переменной

anotherValue = *pointer; // Операция разыменования

// Вывод значения, полученного при разыменовании указателя
printf("Value, stored in pointer: %d \n",anotherValue);
// Вывод адреса, который хранится в указателе
printf("Address in pointer: %p \n", pointer);
// Вывод результата разыменования указателя
printf("Value in pointer: %d \n", *pointer);
```

Работа с указателями происходит с помощью специальных операций – разыменования и взятия адреса. Операция ***взятия адреса*** имеет следующий синтаксис:

```
&имяПеременной
```

Результатом данной унарной операции является получение адреса указанной переменной. Данный адрес можно присвоить в указатель:

```
int value;
int* pointer;
pointer = &value;
```

В данной строке сначала выполняется операция взятия адреса переменной *value*, а затем присвоение полученного адреса в указатель *pointer*. Операция взятия адреса может быть осуществлена над любой переменной, однако, как отмечалось ранее, указатель может хранить адрес переменной только того же типа данных, что и сам указатель.

Операция разыменования производится над указателем и позволяет получить доступ к значению, хранящемуся по адресу в указателе. Синтаксис унарной операции разыменования:

```
*имяПеременнойУказателя
```

Например:

```
*pointer
```

Операция разыменования может использоваться как для получения значения по указанному адресу, так и для изменения значения по указанному адресу.

Например:

```
int value2;
value2 = *pointer;       // Равносильно value2 = value;
*pointer = 5;            // Равносильно value = 5;
```

Как представлено в примере, операция разыменования используется как слева, так и справа от знака присваивания.

Также существуют операции ***инкремента*** и ***декремента адреса***:

```
pointer++;
pointer--;
```

Данные операции осуществляют увеличение адреса, хранящегося в указателе на одно дискретное значение. Важно понимать, что под дискретным значением подразумевается не 1, а количество байт, необходимое для хранения типа данных указателя:

```
int value1;
double value2;
int* pointer1 = &value1;
double* pointer2 = &value2;
pointer1++;    //Адрес увеличится на 4 для типа данных int
pointer2++;    //Адрес увеличится на 8 для типа данных double
```

Более подробно назначение такого специфического и мощного инструмента как указатель будет рассмотрено в следующих лабораторных работах. Важно быть внимательным и не запутаться в порядке выполнения операций. Например, в исходном коде можно встретить подобные записи:

```
pointer++;
*pointer++;
```

Под данными строками подразумеваются совершенно разные действия. В первом случае выполняется изменение адреса, хранящегося в указателе, т.е. после инкремента указатель будет указывать на другую ячейку памяти. Во втором случае сначала выполняется операция разыменования, а затем инкремент. Таким образом, будет изменен не адрес в указателе, а значение, которое хранится по этому адресу.

Для указателей также существуют операции сложения и вычитания. Сложение может быть осуществлено между двумя указателями:

```
int* pointer1;
int* pointer2;
int* pointer3;
…
pointer3 = pointer1 + pointer2;
```

Результатом сложения, как не трудно догадаться, будет сумма двух адресов. Однако указатели также могут складываться с целыми числами:

```
int* pointer1;
int* pointer2;
…
pointer2 = pointer1 + 5;
```

В случае сложения адреса с целочисленным значением осуществляется сдвиг указателя на число, равное целому числу, умноженному на размер типа данных указателя. Для данного примера это будет увеличение указателя на 5*4=20. Данные операции сложения и вычитания адресов становятся очень удобными при работе с адресами, однако для начинающих программистов это

может создать проблемы, так как, учитывая механизм приведения типов данных, адреса указателей могут по ошибке быть сложены с целочисленной или символьной переменной, и компилятор не будет считать это синтаксической ошибкой.

Обратите внимание на именование переменных. В данном примере такие имена переменных как *value*, *pointer* дают представление о том, что это за переменная и для чего она нужна. Любая создаваемая вами переменная должна иметь четкое понятное имя, которое должно отражать назначение переменной. Подробнее об именовании можно ознакомиться в 6.1.

Ссылки. Помимо указателей, в языке Си++ существует понятие ссылки. ***Ссылка*** - это константный указатель. Её объявление и инициализация использует знак & выглядит следующим образом:

```
int& aReference = a;
```

Ссылки схожи с указателями, однако, в отличии от указателей, ссылка: а) обязана быть проинициализированной при объявлении; б) не может быть изменена другим адресом; в) работа с её значением выполняется как с обычной переменной и не требует операций разыменования и взятия адреса.

```
int a = 3;
int b = 5;
int& ref = a;    //Инициализация без операции взятия адреса
//ref = &b;      //Ошибка! Нельзя присвоить другой адрес
b = ref + 5;     //Для получения значения из ссылки не нужно разыменование!
ref = 5;         //Опять же не используется разыменование
```

Не путайте операцию объявления ссылки с операцией взятия адреса! Отличие заключается в том, что объявление ссылки происходит с указанием типа данных и нового, ранее неиспользованного идентификатора, а операция взятия выполняется над любой ранее объявленной переменной.

Вопросы для проверки:

1) Что такое указатель? Продемонстрируйте синтаксис объявления переменных-указателей.

2) Что такое операция разыменования? Что такое операция взятия адреса?
3) Сколько байт выделяется под переменную-указатель? Одинаковое ли количество памяти выделяется для указателей на целочисленные типы данных и для указателей на вещественные типы данных? Для ответа на вопрос используйте функцию *sizeof*().
4) Продемонстрируйте на примере программы, чем отличается объявление указателя от операции разыменования? Как отличить друг от друга эти две операции?
5) Можно ли применить операцию взятия адреса к указателю?
6) Можно ли применить операцию разыменования к обычной переменной?
7) Ниже представлен отрывок исходного кода с объявлением различных переменных и указателей. Над ними совершаются разные действия, некоторые из которых выполнятся, другие же недопустимы. Поясните, почему те или иные операции допустимы или наоборот:

```
int integerValue;
int* integerPointer;
int* anotherIntegerPointer;

float floatValue;
float* floatPointer;

integerPointer = &integerValue;      // Допустимая операция
floatPointer = &floatValue;        // Допустимая операция
integerPointer = &floatValue;      // Недопустимая операция
floatPointer = &integerValue;      // Недопустимая операция
anotherIntegerPointer = integerPointer;     // Допустимая операция
floatPointer = integerPointer;      // Недопустимая операция
```

8) Что такое ссылка?
9) Как происходит объявление ссылки? В чем особенности объявления ссылок?
10) Напишите пример, показывающий разницу между объявлением ссылки и операцией взятия адреса.

3.2 Массивы

Массивы – структура данных, объединяющая конечное количество однотипных элементов, хранящихся последовательно в определенной области данных. В этом определении есть несколько ключевых моментов, на которые следует обратить внимание. Первое – массивы это структура данных, то есть некоторый способ организации каких-либо данных, но ни в коем случае не сами данные и ее тип данных. Второе то, что хранящиеся в массиве данные должны быть однотипные и их должно быть заранее определенное конечное количество. Третье – это их последовательное хранение в памяти. Это очень важный момент, связанный с выделением памяти под массивы и доступа к элементам массива. Один массив не может быть “разбросан” по оперативной памяти, он хранится целиком в одном месте.

Подробно о работе с массивами вы можете прочесть в [1], здесь же рассмотрим только основные моменты. Объявление и инициализация массива выглядит следующим образом:

```
  int integerArray[5];      //объявление массива
  integerArray[3] = 7;      //присвоение элементу массива значения
  int anotherArray[] = {3, 4, 1, 2};  // еще один способ объявления с инициа-
лизацией
  anotherArray[2] = 0;
```

Важно отметить, что количество элементов массива должно заранее определено и указано в квадратных скобках при объявлении. Доступ к элементам массива осуществляется также с помощью квадратных скобок, где в квадратных скобках указывается индекс нужного элемента. Второй представленный способ инициализации массива отличается от первого тем, что для него изначально не указано количество элементов. Однако оно определяется автоматически и будет равно 4 – ровно тому количеству элементу, которое указано в фигурных скобках после знака присваивания. При обращении к массиву по несуществующему индексу (например, отрицательное число или число большее, чем количество элементов массива) возникнет ошибка на этапе выполнения программы.

Индексация элементов массива начинается с нуля, а при объявлении массива без инициализации элементы будут представлять то значение, которое хранилось в памяти на момент создания массива (как и для любых других переменных, объявляемых в языке Си++). Для работы с массивом удобно использовать операторы цикла, например, *for*, однако для этого в отдельной переменной необходимо хранить размер массива, чтобы не выйти за пределы массива. В следующем примере происходит объявление массива размером в пять элементов и инициализация его случайными значениями с использованием оператора цикла:

```
  const int n = 5;
  int integerArray[n];
  for (int i = 0; i < n; i++)
  {
    integerArray[i] = rand();
  }
```

Фактически, с этим связаны два главных неудобства работы с массивами в языке Си++: это необходимость константного предопределенного значения элементов для объявления массива и необходимость хранения в дополнительной переменной размера данного массива. Попробуйте создать массивы целочисленных, вещественных, символьных типов данных и проинициализировать их значениями.

В языке Си++ можно создавать массивы любых типов данных, в том числе и массивы указателей и ссылок.

Передача массива в функцию. В языке Си++ существует три способа передачи массива в функцию. Рассмотрим на примере:

```
double SummArray(double doubleArray[5])
{
  double summ = 0.0;
  for (int i = 0; i < 5; i++)
  {
    summ += doubleArray[i];
  }
  return summ;
}

double SummArray(double doubleArray[], int arraySize)
{
```

```
  double summ = 0.0;
  for (int i = 0; i < arraySize; i++)
  {
    summ += doubleArray[i];
  }
  return summ;
}

double SummArray(double* arrayPointer, int arraySize)
{
  double summ = 0.0;
  for (int i = 0; i < 5; i++)
  {
    summ += arrayPointer[i];
  }
  return summ;
}
```

Все три способа имеют свои преимущества и недостатки. В первом варианте мы передаем в функцию массив строго определенного размера, при этом его размер мы видим из объявления функции.

Минусом подобного способа является то, что функция не будет работать с массивами другого размера, а значит у данной функции очень ограниченные варианты использования. Два других подхода могут использоваться для массивов любого размера, однако размер массива должен передаваться в функцию в виде второй переменной. Иначе вы не сможете определить пределы массива и работать с ним. При работе с указателем дополнительно следует учесть, что указатель является не указателем на массив, а указателем на целочисленный тип данных, то есть только на одно конкретное значение, в случае массивов, на первый элемент массива. Следовательно, в значение указателя можно присвоить адрес обычной целочисленной переменной и потерять доступ к целому массиву данных!

Многомерные массивы. Работа с многомерными матрицами мало чем отличается от работы с одномерными. Объявление многомерных массивов выглядит следующим образом:

```
int multiDimensionArray[3][4];
```

где в начале указывается тип данных массива, затем уникальный идентификатор массива, а затем в квадратных скобках указывается количество элементов по каждому из измерений. При этом количество измерений определяется количеством квадратных скобок и, теоретически, ничем неограниченно. С точки зрения выделения памяти многомерные массивы представляются в виде одномерного массива с размерностью, равной произведению размеров всех измерений. То есть приведенный двумерный массив в памяти будет представлен как единая область размером в 12 целочисленных переменных, а при обращении по индексу [2][1] необходимое значение будет определяться по формуле (1*3+2), где 1 – указанный индекс второго измерения, 3 – максимальная размерность первого измерения, и 2 – индекс первого измерения. Более подробно с работой двумерных массивов вы можете ознакомиться в [1-3]. Учтите, что при передаче двумерных массивов в функцию, вам нужно дополнительно передавать размеры каждого измерения массива.

Вопросы для проверки:

1) Что такое массив? Поясните представление массива в памяти? Как происходит механизм выделения памяти под массив? Как предоставляется доступ к нужному элементу массива?

2) Напишите несколько способов объявления и инициализации массива средствами Си++.

3) Объясните, что представляет из себя идентификатор массива? На какую область памяти он указывает? Как передать массив в функцию? В чем заключается особенность передачи массива в функцию тем или иным способом?

3.3 Строки

Строки – это массивы символов, предназначенные для хранения текста. Текст можно представить как последовательность символов, а значит, его

можно удобно хранить в массиве, где каждая последующая ячейка хранит следующий символ текста:

```
char text[20];
text[0] = 'H';
text[1] = 'e';
text[2] = 'l';
text[3] = 'l';
text[4] = 'o';
text[5] = '!';
```

Обратите внимание, что в примере мы создали массив на 20 символов, однако поместили в него текст только на 6 символов, т.е. 14 элементов массива не используются. При работе с массивами это обычная практика, так как сложно заранее предугадать, сколько символов займет результирующий текст. Таким образом, строки создают с расчетом на максимально необходимую для того или иного текста длину.

Однако, как мы знаем, элементы созданного массива, если их не переинициализировали, хранят случайные значения, хранившиеся в оперативной памяти по указанным адресам. В случае, если мы инициализируем текстом не весь массив, а только его часть, как определить, где заканчивается полезный текст, а где начинается неинициализированный мусор? Для этого существует договоренность, что полезный текст должен заканчиваться нулевым символом '\0', обозначающим конец строки:

```
char text[20];
text[0] = 'H';
text[1] = 'e';
text[2] = 'l';
text[3] = 'l';
text[4] = 'o';
text[5] = '!';
text[6] = '\0';  // Заканчиваем строку нулевым спец-символом
```

Таким образом, при обработке строк можно легко определить конец полезного текста:

```
for (int i = 0; i < 20; i++)
{
   if (text[i] = '\0') //Если встретили конец строки – заканчиваем цикл
      break;
   cout << text[i]; // иначе выводим символ на экран
}
```

В результате выполнения данного цикла на экран будет выведен весь полезный текст из массива *text*.

Очень часто со строками работают через указатели:

```
char* textPointer = text;
```

Поскольку конец строки может быть определен через нулевой спец-символ, передача строк в функцию может осуществляться без указания размера строки.

Стандартные функции для работы со строками в библиотеке *string.h*

int strlen(*char* string*) – определение длины строки.

char strcat*(*char* string1, char* string2*) – объединение двух строк в одну строку.

int strcmp(*char* string1, char* string2*) – сравнение двух строк. Функция возвращает 0, если строки равны (совпадают все символы двух строк).

void strcpy(*char* destination, char* source*) – копирование одной строки во вторую.

char strrev*(*char* string*) – инвертирование строки.

char strlwr*(*char* string*) – приведение строки к нижнему регистру.

char strup*(*char* string*) – приведение строки к верхнему регистру.

float atof(*char* string*) – распознавание строки в вещественное число. Функция выполнится без ошибок только при условии, что входная строка действительно представляет вещественное число, т.е. содержит только числа – мантиссу с возможной дробной частью и порядок числа, указанный через знак экспоненты.

int atoi(*char* string*) – распознавание строки в целое число.

long atol(*char* string*) – распознавание строки в целое число двойной точности.

double ecvt(*char* string*) – преобразование строки в вещественное число двойной точности.

char gcvt*(*double value*) – преобразование вещественного числа двойной точности в строку.

char itoa*(*int value*) – преобразование целого числа в строку.

char ltoa*(*long value*) – преобразование целого числа в строку.

Автор обращает внимание читателя на именование стандартных функций и еще раз напоминает о важности ***правильного***, ***грамотного*** именования функций.

3.4 Генерация случайных чисел

Довольно часто в программировании требуется инициализировать начальные данные какими-либо значениями. При этом задание всегда одних и тех же значений может привести к ситуации, когда программа отлажена только для одного определенного набора начальных данных. Поэтому для инициализации данных часто используют генерацию случайных чисел. Для генерации случайных чисел в Си++ необходимо подключить две библиотеки – *stdlib.h* и *time.h*. После подключения первой библиотеки вы сможете использовать функцию *rand*() для генерации случайного целого числа от 0 до 32767. Если же вам нужно сгенерировать число в заданном диапазоне, например, от 30 до 40, то воспользуйтесь следующим способом:

```
int randomNumber = 30 + rand() % 10;
```

Где 30 – начало вашего диапазона, а 10 – величина диапазона (40 - 30). Помните, что перед вызовом функции *rand*() необходимо хотя бы один раз вызвать функцию *srand*(*int seed*), которая задает основу для всех последующих генерируемых чисел. Чтобы при каждом новом запуске программы основа была новой, а следовательно и генерируемые числа, в качестве входного аргумента функции *srand*() предлагается подавать текущее время, получаемое вызовом функции *time*(*NULL*), лежащей в библиотеке *time.h*. Генерация случайных чисел очень удобна для инициализации больших массивов данных, например многомерных массивов, которые будут рассмотрены далее. Однако рассмотрим использование случайных чисел для написания вашей первой игры «Угадай число»:

program.cpp

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
  srand(time(NULL));        // для задания случайного начального числа
  printf("\n---Game: Guess the Number---\n");
  int guessNumber = rand()%10;     // генерация угадываемого числа
  int enteredNumber = -1;      // вводимое пользователем число
  int shots = 0;         // количество попыток
  printf("\nEnter number from 0 to 9: ");
  scanf("%d", &enteredNumber);
  while (guessNumber != enteredNumber)
  {
    shots++;
    printf("\nWrong!!! Try again!\nEnter number from 0 to 9: ");
    scanf("%d", &enteredNumber);
  }
  printf("\nCorrect! You win in %d shots!\n", shots);
  return 0;
}
```

Это ваша первая компьютерная игра, которая взаимодействует с пользователем. Попробуйте написать свою версию данной игры, ограничив количество попыток угадывания до трёх или сообщая пользователю фразы «больше» или «меньше» при угадывании. Придумайте свои дополнительные возможности в игре.

Вопросы для проверки:

1) Что выполняет функция *rand*()?

2) Как произвести генерацию случайных чисел в заданном диапазоне?

3) Для чего нужна функция *srand*()? Почему на вход данной функции обычно подают результат функции *time*(*NULL*)?

4) Как с помощью функции *rand*() сгенерировать случайные символьные значения? Вещественные значения? Запишите пример подобного кода.

5) Приведите несколько примеров, где может потребоваться генерация случайных чисел?

4 Функции

Функция в программировании - фрагмент программного кода (подпрограмма), к которому можно обратиться из другого места программы. Как и переменная, функция обладает уникальным идентификатором, позволяющим ***вызвать функцию***. Идентификатор, или имя функции, связано с адресом первой инструкции, входящей в функцию, которой передается управление при обращении к функции. После выполнения функции управление возвращается обратно в адрес возврата – точку программы, где данная функция была вызвана.

Функции служат важным инструментом программирования, предназначены для уменьшения дублирования кода, тем самым позволяют управлять сложностью программного кода. Однако для правильного использования функций необходимо не только знать средства языка программирования, но и правильно декомпозировать задачи на отдельные подпрограммы. Данный навык можно приобрести только после знакомства с курсом алгоритмизации и только при постоянной практике.

4.1 Определение, описание и вызов функций

В языке Си++ реализация какого-либо метода имеет следующий вид:

```
ВозвращаемыйТипДанных Функция(ТипВходнойПеременной1 имяВходнойПеременной1, …)
{
список выполняемых инструкций
  …
  return возвращаемоеЗначение;
}
```

Где *возвращаемыйТипДанных* – это указание того типа данных, которое будет возвращаться в результате выполнения функции. Это может быть *int*, *char*, *double*, *void* или любой другой тип данных, который вы захотите. Функция – имя функции, уникальное в данном пространстве имен. По правилам языка Си++, вы не можете создать две функции с одним и тем же именем, возвращаемым типом данных и набором входных переменных.

По правилам *RSDN* функции именуются согласно стилю *Pascal*, т.е. каждое новое слово в имени начинается с заглавной буквы, например, *CalculateTriangleArea*. Первая буква также должна быть заглавной (в отличии имен переменных, где первая буква должна быть прописной).

Помимо стиля *Pascal*, имя функции должно начинаться с глагола, объясняющего действие, которое выполняет данная функция.

После имени функции в скобках через запятую указываются входные переменные. Чтобы указать входную переменную, вы должны указать для неё тип данных, а также её имя, которое будет видно только внутри данной функции. В дальнейшем, вы можете использовать входные переменные внутри функции. Далее идет тело функции. Тело функции обозначается фигурными скобками, внутри которых вы можете написать любой код.

Обратите внимание на оформление тела функции в примере ниже. Фигурные скобки идут ровно под началом объявления функции, а всё содержимое тела смещено на одну табуляцию относительно фигурных скобок.

Как уже было сказано, в теле функции вы можете написать любой код, используя входные переменные, либо объявляя новые. Однако выполнение функции должно заканчиваться оператором *return* и указанием через пробел возвращаемого значения, которое должно строго соответствовать возвращаемому типу данных. Пример функции в языке Си++:

```
double CalculateTriangleArea(int width, int height)
{
  double area = width * height / 2.0;
  return area;
}
```

Из данного примера видно, что функция в результате своего выполнения должна вернуть вещественное значение типа *double*. Название функции говорит о том, что в нем рассчитывается площадь треугольника, а двумя входными переменными являются его длина и высота. Внутри функции объявляется новая переменная вещественного типа, в которую тут же присваивается рассчи-

танное значение площади. Далее происходит возвращение значения переменной *area* в место вызова функции. Обязательным условием является соответствие типа данных значения под оператором return и типа данных, указанного при объявлении функции.

Стоит отметить, что если функция возвращает какое-либо значение, то в месте вызова функции это значение должно быть куда-то записано, например, в какую-нибудь переменную. Исключение составляют функции, возвращающие тип *void*, то есть ничего. При этом указание оператора *return* внутри подобной функции необязательно. Напишем пример функции, выводящий текст на экран.

Поскольку метод должен просто вывести произвольный текст на экран, то значит он не обязан возвращать какое-либо значение, а также иметь входных переменных. Примером подобной функции может послужить следующая:

```
void PrintHelloWorld()
{
  cout << "Hello, World!\n";
}
```

Полностью наша программа будет выглядеть следующим образом.

```
#include "stdafx.h"
#include <iostream.h>

using namespace std;

//Реализация функции, выводящей на экран текст.
void PrintHelloWorld()
{
  cout << "Hello, World!\n";
}

int main()
{
  PrintHelloWorld(); //Вызов функции в теле программы.
  return 0;
}
```

Обратите внимание, что функция должна быть описана до того, как её вызовут. В данном случае, наша функция должна быть описана до функции

main. Также не забывайте указывать комментарий перед функцией, описывающей её назначение – это значительно увеличивает читаемость программного кода.

В языке Си++ существует ограничение: функция должна быть объявлена до того, как её вызовут. В данном примере это означает, что функция PrintHelloWorld должны быть описана до функции main, в которой происходит её вызов. Если же описать функцию ниже main, как это сделано в следующем примере, компилятор сообщит об ошибке:

```
#include "stdafx.h"
#include <iostream.h>

using namespace std;

int main()
{
  PrintHelloWorld(); //Вызов функции в теле программы.
  return 0;
}

//Реализация функции, выводящей на экран текст.
void PrintHelloWorld()
{
  cout << "Hello, World!\n";
}
```

К сожалению, при разработке программ даже средней сложности (от 50 000 строк кода) данное требование практически невозможно выполнить. Но в языке Си++ есть механизм, позволяющий обойти данное ограничение. Это прототипы функций. ***Прототип функции*** – это объявление функции, возвращаемого типа данных, имени функции и её входных переменных без тела самой функции. При этом подразумевается, что само тело функции будет обязательно описано в другом месте.

Следующий пример демонстрирует работу прототипов:

```
#include "stdafx.h"
#include <iostream.h>

using namespace std;

void PrintHelloWorld();       //Прототип функции

int main()
```

```
{
  PrintHelloWorld(); //Вызов функции в теле программы.
  return 0;
}

//Реализация функции, выводящей на экран текст.
void PrintHelloWorld()
{
  cout << "Hello, World!\n";
}
```

Как видите, теперь реализация функции находится после её первого вызова в методе *main*, благодаря прототипу функции, описанного перед главной функцией. К сожалению, данный пример надуман и не демонстрирует реальной необходимости в прототипах. Однако при разработке различных библиотек или собственных программных модулей данный механизм становится просто необходим.

В завершение читателю предлагается самостоятельно реализовать функцию *double MakeCalculation*(*int value1, int value2, char operationKey*), которая принимает на вход две целочисленные переменные и одну символьную. Если в качестве символьной переменной передается «+», «-», «*», «/», «%», то должно выполниться соответствующее действие над двумя целочисленными переменными. Результат выполненной операции возвращается из функции.

Вопросы для проверки:

1) Что такое функция?
2) Как выглядит объявление функции в языке Си++? Какие обязательные элементы объявления функции?
3) Что такое входные переменные функции? Какие типы данных могут использоваться в качестве входных переменных? Что такое *void*?
4) Что такое возвращаемое значение? Какие типы данных могут использоваться в качестве возвращаемого значения из функции?
5) В чем особенность функции *Main*?

6) Обратите внимание, что функция *Main* также может обладать входными и выходными переменными. Откуда в функцию *Main* приходят входные переменные, и куда передается возвращаемое значение?
7) Что такое прототип функции? Для чего он нужен?
8) Имеет ли смысл описывать прототип функции ниже реализации данной функции?

4.2 Способы передачи переменных в функции

Теперь пришло время изучить один из важнейших механизмов языка Си++ - передача переменных в функцию. Мы уже с вами рассмотрели то, как передаются в функции обычные переменные и использовали их для вычисления новых значений. Однако здесь есть некоторые нюансы, которые необходимо знать. Рассмотрим следующий пример:

```
void IncrementInteger(int value)
{
  cout << "\nValue of variable in function is " << value;
  value++; //увеличиваем входное значение на 1.
  cout << "\nValue of variable in function is " << value;
}

int main()
{
  int a = 5;
  cout << "\nValue of variable before function is " << a;
  IncrementInteger(a); //увеличиваем значение a на 1.
  cout << "\nValue of variable after function is " << a;
  return 0;
}
```

В этом примере мы создаем целочисленную переменную со значением 5 и передаем её в функцию, которая должна увеличить это значение на 1. На каждом шаге будем выводить на экран значение переменных. Однако после запуска программы мы увидим следующее:

```
Value of variable before function is 5
Value of variable in function is 5
Value of variable in function is 6
Value of variable after function is 5
```

Из примера видно, что функция получила значение переменной *a*, равное 5, правильно. После чего внутри функции происходит изменение переменной

на 1. Однако после завершения работы функции значение переменной *a* остается равным 5, хотя переменная *value* очевидно была равна 6.

Дело в том, что при передаче переменных в функцию передаются не сами переменные, а только их значения. Фактически, для каждой входной переменной будет создаваться её локальная копия с таким же значением. Таким образом, внутри функции вы сможете работать только с локальной копией входной переменной. Вы можете присваивать в локальную копию любое значение, однако значение исходной переменной при этом останется нетронутым. Данный механизм передачи переменных называется передачей по значению. Убедиться в этом можно, если вывести адреса переменных:

```
void IncrementInteger(int value)
{
  cout << "\nAddress of variable in function is " << &value;
  cout << "\nValue of variable in function is " << value;
  value++; //увеличиваем входное значение на 1.
  cout << "\nValue of variable in function is " << value;
}

int main()
{
  int a = 5;
  cout << "\nAddress of variable before function is " << &a;
  cout << "\nValue of variable before function is " << a;
  IncrementInteger(a); //увеличиваем значение a на 1.
  cout << "\nValue of variable after function is " << a;
  return 0;
}
```

Результатом работы станет:

```
Address of variable before function is 00DD6412
Value of variable before function is 5
Address of variable before function is 00BC0027
Value of variable in function is 5
Value of variable in function is 6
Value of variable after function is 5
```

Как видно из примера, переменные *value* и *a* имеют два совершенно разных адреса в оперативной памяти и, следовательно, каждая переменная может хранить собственное значение, и при изменении значения переменной *value* значение переменной *a* останется прежним.

Рассмотрим другой пример:

```
void IncrementIntegerPointer(int* pointer)
{
  cout << "\nAddress of variable in function is " << aPointer;
  cout << "\nValue of variable in function is " << *aPointer;
  (*pointer)++; //увеличиваем входное значение на 1.
  cout << "\nValue of variable in function is " << *aPointer;
}

int main()
{
  int a = 5;
  cout << "\nAddress of variable before function is " << aPointer;
  cout << "\nValue of variable before function is " << *aPointer;
  int* aPointer = &a;
  IncrementIntegerPointer(aPointer); //увеличиваем значение a на 1.
  cout << "\nValue of variable after function is " << a;
  return 0;
}
```

Данный пример отличается тем, что в метод мы передаем не целочисленную переменную, а указатель на целочисленную переменную. То есть, на этот внутри функции мы будем работать не со значением, а с адресом некоторой переменной. И если теперь в функции мы увеличим значение, хранимое по адресу в указателе, то тем самым мы увеличим значение исходной переменной *a*, а при выполнении программы мы увидим следующее:

```
Address of variable before function is 00DD6412
Value of variable before function is 5
Address of variable before function is 00DD6412
Value of variable in function is 5
Value of variable in function is 6
Value of variable after function is 6
```

Данный механизм называется передачей ***переменных по указателю***. И в нем уже не будет создаваться какой-либо локальной копии входной переменной, здесь мы будем работать непосредственно с адресом, в котором хранится какое-то значение – результат работы программы показывает, что в функции мы работаем с тем же самым адресом, что и был до вызова функции. Следовательно, если мы изменим значение по данному адресу внутри функции, то оно изменится и вне данной функции.

Учтите, что внутри функции вы можете изменить не только значение, хранимое по адресу в указателе, но и сам адрес в указателе. В итоге это может

привести к тому, что внутри функции вы будете работать не с тем адресом-переменной, с которой вы предполагали работать изначально.

Рассмотрим еще один пример:

```
void IncrementIntegerReference(int& reference)
{
  cout << "\nAddress of variable in function is " << &reference;
  cout << "\nValue of variable in function is " << reference;
  reference++; //увеличиваем входное значение на 1.
  cout << "\nValue of variable in function is " << reference;
}

int main()
{
  int a = 5;
  cout << "\nAddress of variable before function is " << &a;
  cout << "\nValue of variable before function is " << a;
  int& aReference = a;
  IncrementIntegerReference(aReference); //увеличиваем значение a на 1.
  cout << "\nValue of variable after function is " << aReference;
  return 0;
}
```

В данном примере продемонстрирована передача ***переменной по ссылке*** – входной аргумент функции объявлен как переменная-ссылка. Результат работы программы:

```
Address of variable before function is 00DD6412
Value of variable before function is 5
Address of variable before function is 00DD6412
Value of variable in function is 5
Value of variable in function is 6
Value of variable after function is 6
```

Таким образом, в примере выше мы также будем передавать в функцию не значение переменной *a*, а её адрес, что позволит нам изменить её значение внутри функции. Заметьте, что результат работы данного примера полностью совпадает с результатом работы программы, где демонстрировалась передача переменной по указателю. Однако есть важное отличие в этих двух способах передачи переменных в функцию. Работа по ссылке считается более безопасной, чем работа по указателю, так как вы не сможете случайно изменить адрес, хранимый по ссылке. Представленный пример можно записать более кратко:

```
void IncrementIntegerReference(int& reference)
{
  cout << "\nAddress of variable in function is " << &reference;
```

```
  cout << "\nValue of variable in function is " << reference;
  reference++; //увеличиваем входное значение на 1.
  cout << "\nValue of variable in function is " << reference;
}

int main()
{
  int a = 5;
  cout << "\nAddress of variable before function is " << &a;
  cout << "\nValue of variable before function is " << a;
  //Не будем создавать переменную-ссылку
  IncrementIntegerReference(a); //Передаем сразу исходную переменную
  cout << "\nValue of variable after function is " << a;
  return 0;
}
```

То есть, не обязательно заводить отдельную переменную-ссылку, а можно сразу же передать саму переменную.

Важно усвоить разницу между всеми тремя способами передачи переменных в функцию и понять, в каких ситуациях должен использоваться каждый из них:

1) Передача переменной по значению используется, когда в функции требуется только значение исходной переменной, однако нужно гарантировать, что сама исходная переменная не будет изменена.
2) Передача переменной по ссылке используется, когда предполагается, что функция может изменить исходное значение входной переменной.
3) Передача переменной по указателю используется, когда предполагается не только изменение исходного значения входной переменной, но возможно и изменение самого объекта, на который ссылается указатель.

Одно из интересных способов использования передачи переменных в функцию по ссылке и по указателю это для возвращения из функции более одного значения. Например, необходимо написать функцию, которая бы рассчитывала корни квадратного уравнения. Однако в зависимости от коэффициентов квадратного уравнения, корней может быть один, два, или может не быть вовсе. Таким образом, чтобы было можно использовать результаты расчета корней, функция должна вернуть: количество корней, значение первого

корня и значение второго корня. Всего три значения. Однако мы знаем, что функцию может вернуть только одно значение. Данное ограничение можно обойти следующей функцией:

```
int GetRoots(int a, int b, int c, double& x1, double& x2){…}

int Main()
{
  double x1;
  double x2;
  int rootsCount = GetRoots(1, 3, 2, x1, x2);
  if (rootsCount == 2)
    cout << "x1 =" << x1 << "; x2 = " << x2;
  else if (rootsCount == 1)
    cout << "x =" << x1;
  else
    cout << "No roots";
}
```

Функция своим результирующим целочисленным значением будет возвращать количество корней – ноль, один или два. Входными переменными будем передавать три коэффициента уравнения, но помимо них, передадим еще две переменных по ссылке. В эти переменные внутри функции мы присвоим значения рассчитанных корней. Таким образом, после вызова функции мы можем использовать рассчитанные значения корней. Фактически, наша функция вернула три значения вместо одного.

Вопросы для проверки:

1) В чем отличие между передачей переменных по значению и указателю? Продемонстрируйте на примере.
2) В чем отличие между передачей переменных по указателю и ссылке?

4.3 Рекурсивные функции

Рекурсивная функция – функция, которая в процессе своего выполнения вызывает саму себя. Как правило, выполнение рекурсивных алгоритмов на практике сложно для понимания, поэтому их стараются использовать только при необходимости. Любая правильно написанная рекурсия должна содержать в себе условный оператор с ***условием останова***. При этом вызываться эта же

самая функция должна только в одной из веток условного оператора, иначе это приведет к бесконечной рекурсии, а значит к «зависанию» программы. Пример рекурсивной функции, вычисляющей значение факториала:

```
int GetFactorial(int value)
{
  if (value == 1)
  {
    return 1;
  }
  else
  {
    return value*GetFactorial(value-1);
  }
}

int main()
{
  cout << "!5" = " << GetFactorial(5);
  return 0;
}
```

Функция будет вызывать себя с уменьшением значения до тех пор, пока значение не станет равным 1. Равенство входного значения 1 и есть условие останова.

Данная функция не безопасна и при определенных условиях может привести к бесконечной рекурсии. Например, если попробовать вычислить факториал отрицательного числа. Объясните, почему это приведет к бесконечной рекурсии, и как исправить функцию, чтобы избежать этого.

Также существует понятие псевдорекурсии. ***Псевдорекурсия*** создается, когда выполнение нескольких функций можно рассматривать как одну рекурсию. Псевдорекурсия демонстрируется на следующем примере:

```
void Function1();
void Function2();

void Function1()
{
  …
  Function2();
  …
}

void Function2()
{
```

```
  …
  Function1();
  …
}
```

Здесь рекурсия создается в результате выполнения двух функций, которые вызывают друг друга.

Вопросы для проверки:

1) Что такое рекурсия?
2) Что такое условие останова? Почему оно обязательно для рекурсии?
3) Почему представленный пример рекурсии является небезопасным?
4) Объясните необходимость написания прототипов в этом случае реализации псевдорекурсии.

4.4 Перегрузка функций

Как говорилось ранее, нельзя создать две функции с одинаковым именем и совпадающими входными и выходными параметрами.

Однако, в некоторых случаях становится очень удобным, чтобы имена функций, выполняющую одну и ту же задачу, но для переменных разного типа, были названы одинаково – это упрощает запоминание имен функций для разработчика. Для этого в языке Си++ существует механизм перегрузки функций. Данный механизм позволяет создавать функции с одинаковым именем, но отличающимся набором входных данных. Рассмотрим пример:

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

//Рассчитать сумму двух целочисленных переменных
void SummNumbers(int value1, int value2)
{
  printf("Summ of integer is %d\n", value1+value2);
}

//Рассчитать сумму двух целочисленных переменных
void SummNumbers(double value1, double value2)
{
  printf("Summ of double is %f\n", value1+value2);
}
```

```
//Рассчитать сумму двух целочисленных переменных
void SummNumbers(int value1, double value2)
{
  printf("Summ of integer and double is %f\n", value1+value2);
}

int main()
{
  int a = 1;
  int b = 2;
  SummNumbers(a, b);

  double x = 3.0;
  double y = 4.0;
  SummNumbers(x, y);

  SummNumbers(a, y);

  float m = 5.0;
  float n = 6.0;
  SummNumbers(m, n);

  return 0;
}
```

Все три функции имеют одно и то же имя, однако отличаются набором входных переменных. При этом при вызове данных функций в функции *main* программа сама определяет, какая из трех версий функции лучше всего подходит для передаваемых переменных. Особое внимание обратите на последний вызов функции *SummNumbers*(). На вход передаются два вещественных числа типа *float*, при этом вызывается версия функции, принимающая на вход два вещественных числа типа *double*.

В данном случае программа отрабатывает корректно, потому что происходит неявное преобразование типов. Данную особенность следует учитывать, так как при использовании перегруженных функций вызывается та версия, для которой входные параметры лучше всего подходят или возможно неявное преобразование типов.

Вопросы для проверки:

1) Что такое перегрузка функций?

2) Как определяется выбор версии перегруженной функции?

3) Как используется механизм неявного преобразования типов при перегрузке функций?

4.5 Глобальные переменные

В п.2.4 рассказывалось о блоках кода, обозначаемых фигурных скобках {}, которые создают новые уникальные области видимости для переменных. Однако, переменные могут быть созданы в так называемой ***глобальной области видимости***, т.е. вне блоков кода.

Глобальные переменные – переменные, доступные в любой области видимости программы. Это значит, что вы можете обратиться к значению данной переменной из любого места программы, или даже изменить это значение. Объявление глобальной переменной аналогично объявлению обычной переменной, только расположение объявления переменной находится за пределами каких-либо функций. Следующий пример демонстрирует создание и использование глобальных переменных:

```
#include "stdafx.h"

//объявление глобальной переменной.
int globalVariable = 7;

void GlobalPlusTwo()
{
  globalVariable += 2;
}

void GlobalMultiplyThree()
{
  globalVariable *= 3;
}

void GlobalEqualsOne()
{
  globalVariable = 1;
}

int main()
{
  cout << "\nGlobal Variable: " << globalVariable;
  GlobalPlusTwo();
  cout << "\nGlobal Variable: " << globalVariable;
  GlobalMultiplyThree();
  cout << "\nGlobal Variable: " << globalVariable;
```

```
  GlobalEqualsOne();
  cout << "\nGlobal Variable: " << globalVariable;
  globalVariable = 5;
  cout << "\nGlobal Variable: " << globalVariable;
  return 0;
}
```

Несмотря на все удобства, которые дает нам использование глобальных переменных (например, отсутствие необходимости передавать переменные внутрь функции), использование глобальных переменных строго запрещено и является результатом плохого продумывания программы. Использование глобальных переменных запрещено, потому что: 1) их значение можно изменить из любого места программы, что в итоге не позволяет контролировать хранимое в ней значение и затрудняет понимание программы 2) нельзя создать переменную с аналогичным именем, что приводит к придумыванию сложных имен переменных.

Поэтому, в качестве глобальных переменных допустимы только константные переменные – те переменные, значения которых после инициализации изменить нельзя. Во всех остальных случаях следует избегать использования глобальных переменных.

Вопросы для проверки:

4) У оператора цикла for существует две области видимости. Покажите их на примере.
5) Что такое глобальная переменная?
6) Почему использование глобальных переменных запрещено и может привести к ошибкам в программе?

5 Пользовательские типы данных

Ранее была представлена работа с базовыми типами данных и работа с адресами памяти, являющиеся основой алгоритмизации. Однако при разработке приложений необходимы более высокоуровневые конструкции и типы данных, описывающих объекты реального мира.

В языке Си++ возможно создание собственных типов данных, которые будут относиться к одной из категорий: перечисления, структуры, классы и объединения. Рассмотрим создание составных типов данных для каждой категории.

5.1 Перечисления

Предположим, что вы разрабатываете программу «Планировщик задач», где пользователь может распланировать свои задачи на неделю. В отдельной переменной вам необходимо хранить день недели, на которую назначена та или иная задача. Как это сделать?

Самый простой способ, это представить день недели как целое число от 0 до 6.

```
int weekday;
weekday = 4; //значение 4 соответствует пятнице
…
switch weekday
{
   case 0: printf(“Задача запланирована на понедельник”); break;
   case 1: printf(“Задача запланирована на вторник”); break;
   case 2: printf(“Задача запланирована на среду”); break;
   case 3: printf(“Задача запланирована на четверг”); break;
   case 4: printf(“Задача запланирована на пятницу”); break;
   case 5: printf(“Задача запланирована на субботу”); break;
   case 6: printf(“Задача запланирована на воскресенье”); break;
}
```

Это вполне работоспособное решение, однако оно обладает недостатками:

а) Целочисленная переменная *weekday* может принимать не только значения от 0 до 6, но и, например, 15. Подобное значение в переменную может

быть присвоено по ошибке программиста, однако пятнадцатого для недели не существует.

б) Неоднозначная интерпретация числа. Для многих в сознании отложилось, что понедельник это первый день недели, а пятница – пятый. Однако в исходном коде понедельнику соответствует число 0, а пятнице – число 4. Несмотря на то, что программисты привычны к индексации, начинающейся с нуля, в случаях, подобных дням неделям, разработчику будет легко запутаться и в какой-то момент перепутать пятницу и субботу.

в) Плохое восприятие исходного кода. Когда разработчик будет читать данный код и встретит *switch* с целочисленными значениями, ему придется разобраться с интерпретацией ***каждого*** целого числа в условном операторе. Зачастую, это приводит к просмотру значительного объёма кода.

Для решения указанных проблем в языке Си++ существуют перечисления. ***Перечисление*** – пользовательский тип данных, способный принимать ограниченное количество именованных констант [2]. Перечисления создаются с помощью ключевого слова *enum* (от англ. «*enumeration*» - перечисление) вне блока кода какой-либо функции в формате:

```
enum ВмяПеречисления
{
   Значение1,
   Значение2,
   …
};
```

Рассмотрим на примере перечисления дней недели:

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

Как новый тип данных, имя перечисления пишется с заглавной буквы. Теперь мы можем использовать данное имя перечисления для создания переменных. Так как переменные создаются с указанием имени типа данных и некоторого уникального имени, то создание переменной перечисления будет выглядеть следующим образом:

```
Weekday day1;
```

А инициализация выполняется с указанием конкретной именованной константы:

```
day1 = Monday;
```

Теперь исправим наш исходный код планировщика задач с использованием перечисления:

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

…

Weekday weekday;
weekday = Friday; //нет проблем с интерпретацией числовых значений
…
switch weekday
{
   case Monday: printf(“Задача запланирована на понедельник”); break;
   case Tuesday: printf(“Задача запланирована на вторник”); break;
   case Wednesday: printf(“Задача запланирована на среду”); break;
   case Thursday: printf(“Задача запланирована на четверг”); break;
   case Friday: printf(“Задача запланирована на пятницу”); break;
   case Saturday: printf(“Задача запланирована на субботу”); break;
   case Sunday: printf(“Задача запланирована на воскресенье”); break;
}
```

Обратите внимание, что под значением case используется не числовое значение, а та же именованная константа. Это значительно повышает читаемость кода и уменьшает вероятность ошибки при написании кода.

На самом деле, перечисления интерпретируются компилятором как целочисленный тип данных. Поэтому существует неявное преобразование любого перечисления в *int* и обратно:

```
int a = 4;
Weekday day = a;
a = day + 2 + Monday;
```

При этом именованным константам присваиваются последовательные значения от 0 и выше. Например, в нашем перечислении *Monday* будет интерпретироваться как 0, *Tuesday* как 1, *Wednesday* как 2 и т.д. Данную интерпретацию можно изменить, явно присвоив целое значение при объявлении именованной константы:

```
enum Weekday
{
   Monday = 5,  // Теперь понедельник будет преобразовываться в 5
   Tuesday,     // Вторнику автоматически присваивается значение 6
   Wednesday,   // 7
   Thursday,    // 8
   Friday = 18, // Пятница будет преобразовываться в 18
   Saturday,    // 19
   Sunday       // 20
};
```

В данном пособии не будет рассматриваться, для чего нужны подобные преобразования и переопределения, однако на практике данный механизм бывает весьма полезен. Также важно отметить, что так как перечисления интерпретируются компилятором как целочисленный тип данных, то любое перечисление: имеет размер в 4 байта (такой же размер, что и тип *int*).

Возможно, применение перечислений будет проще понять, если сравнить их с логическим типом данных *bool*, принимающим только два возможных значения *true* и *false*.

Теперь, когда мы создали собственный тип данных, он может быть использован, например, в объявлении функций наравне с базовыми:

```
…
Weekday IdentifyTodayWeekday()
{
   …
   if (anyStrangeSituation)
      return Friday;
```

```
}
…
Weekday weekday1 = IdentifyTodayWeekday();
```

5.2 Структуры

Структура – составной тип данных, объединяющий в единое целое множество поименованных элементов в общем случае разных типов [1]. Поименованные элементы называются ***полями структуры***. Сравнивая структуру с массивом, следует отметить, что массив – это совокупность однородных объектов, имеющая общее имя – идентификатор массива. Другими словами, все элементы массива являются объектами одного и того же типа. Структура позволяет объединять объекты разных типов.

Объявление структуры выполняется с помощью ключевого слова *struct* (от англ. «*structure*» - структура) вне блока кода какой-либо функции и в следующем формате:

```
struct ИмяСтруктуры
{
   ТипДанныхПоля1 ИмяПоля1;
   ТипДанныхПоля2 ИмяПоля2;
…
};
```

Имя и поля структуры по правилам *RSDN* [9] именуются с заглавной буквы.

С помощью структур создают типы данных, соответствующие реальным объектам. Например, если мы захотим описать человека, то для описания имени, возраста, профессии понадобятся различные типы данных:

```
struct Person
{
   char Name[20];
   int Age;
   char Profession[40];
};
```

После объявления структуры можно создавать экземпляры нового типа данных и инициализировать их данными:

```
Person person1;
person1.Age = 20;
strcpy(“John McClane”, person1.Name);
```

```
strcpy(“Detective”, person1.Name);
```

Обратите внимание, что обращение к полям конкретного экземпляра выполняется через оператор «.» с указанием имени переменной до и именем поля после. Количество байт, необходимое для создания одного экземпляра структуры определяется суммой всех байт, требуемых для хранения каждого поля. Таким образом, экземпляр Person будет занимать 64 байта – массив на 20 символов по одному 1 байту, целочисленное поле в 4 байта, и массив на 40 символов по одному 1 байту.

Структура может содержать в себе и перечисления. Создадим перечисление для указания пола человека и добавим его в созданную структуру:

```
enum Sex
{
   Male,
   Female
};

struct Person
{
   char Name[20];
   int Age;
   char Profession[40];
   Sex Sex;
};
```

Обратите внимание, в добавленной в структуру строке первое слово Sex подразумевает название типа данных, а второе слово *Sex* – название поля. Таким образом, имя поля в структуре может совпадать с именем типа данных. Отчасти, это затрудняет чтение кода, однако это обычная практика.

Полям структуры можно присваивать значения по умолчанию:

```
struct Person
{
   char Name[20];
   int Age = 25; // Присвоение возраста по умолчанию
   char Profession[40];
   Sex Sex = Male; // Присвоение пола по умолчанию
};
```

Таким образом поля любого экземпляра структуры будут автоматически проинициализированы указанным значением.

```
Person person1;
```

```
printf(“Age is %d”, person1.Age); // Результат вывода: Age is 25
```

Важной особенностью структур является возможность использования другой структуры в качестве поля. Рассмотрим пример. Предположим, что нам нужно хранить больше информации о профессии человека – размер зарплаты и стаж работы. С одной стороны, мы можем добавить в структуру *Person* новые поля:

```
struct Person
{
   char Name[20];
   int Age = 25;
   Sex Sex = Male;

   char Profession[40];
   int Salary; // Размер зарплаты
   int Seniority; // Трудовой стаж
};
```

Обратите внимание, что порядок полей был изменен и сгруппирован на личные данные человека и информацию о работе. Подобная группировка данных в структуре также облегчает изучение кода, поля структуры легче запоминаются другим разработчиком.

Однако все поля о профессии можно вынести в отдельную структуру:

```
struct ProfessionInfo
{
   char Profession[40]; // Название должности
   int Salary; // Размер зарплаты
   int Seniority; // Трудовой стаж
};

struct Person
{
   char Name[20];
   int Age = 25;
   Sex Sex = Male;

   ProfessionInfo Profession;
};
```

Теперь данные о профессии содержатся в отдельной сущности, а структура Person содержит в себе экземпляр другой структуры.

Структура не может содержать экземпляр самой себя в качестве поля:

```
struct Person
{
```

```
   char Name[20];
   int Age = 25;
   Sex Sex = Male;

   ProfessionInfo Profession;
   Person Father; // Недопустимое поле
   Person Mother; // Недопустимое поле
};
```

Данное ограничение логично, так как противном случае невозможно определить размер, необходимый для хранения одного экземпляра структуры. Если представить все структуры программы в виде так называемой иерархии, показывающей включение одних структур в другие, то данная иерархия должна представлять дерево, без циклических замкнутых соединений.

Однако, структура может содержать в качестве поля указатель, в том числе и указатель на саму себя:

```
struct Person
{
   char Name[20];
   int Age = 25;
   Sex Sex = Male;

   ProfessionInfo Profession;
   Person* Father; // Допустимые поля с данными о родителях
   Person* Mother = null; // Указатели стоит инициализировать как null
};
```

Теперь, экземпляр *Person* может хранить в себе данные о родителях в качестве указателей, и это будет корректно с точки зрения синтаксиса Си++.

Работа с экземпляром структуры через указатель имеет свои особенности. Доступ к полям объекта через указатель осуществляются через оператор «→», а не «.»:

```
Person person1;
Person* personPointer = &person1; //создаем указатель
                                  //и инициализируем адресом
personPointer->Age = 25;  //Обратите внимание на обращение к полю
printf("Age is %d", personPointer->Age);
…
personPointer->Profession.Salary = 15000;
personPointer->Mother->Age = 50;
personPointer->Mother->Father->ProfessionInfo.Salary = 25000;
```

Заметьте, обращение к полю *Profession* выполняется через оператор «→», однако, так как *Profession* является экземпляром структуры, а не указателем,

но дальнейшее обращение к его внутренним полям осуществляется через оператор «.». Если бы данные о профессии хранились в структуре *Person* в качестве указателя, то обращение также выполнялось бы через оператор «→». Также стоит отметить, что теперь Person хранит информацию о родителях через указатель, и через этот указатель мы можем получить доступ к полям возраста и имени родителей, так как они также являются экземплярами структуры Person. Таким образом можно создать длинную цепочку обращений к внутренним полям, как показано в последней строке примера, где фактически происходит изменение зарплаты дедушки первоначально созданного человека.

5.3 Объединения

Объединения являются еще одной разновидностью пользовательских типов данных в языке Си++. Так как объединения предназначены для решения специфических задач, в данном пособии они рассмотрены не будут, однако их подробное описание представлено в [1].

5.4 Динамическая память

Перед изучением различных структур данных необходимо разобрать механизм того, как выполняется ваша программа и как она работает с оперативной памятью. При запуске любой программы весь исполняемый код помещается в оперативную память, а процессор начинает выполнять операцию, находящуюся в точке входа – фактически, с функции main(). Для каждой исполняемой программы можно выделить 4 области памяти:

1) Сегмент кода
2) Сегмент данных (статическая память)
3) Стек
4) Динамическая память (куча)

Сегмент кода используется для хранения кода программы – функций. Функции помещаются в сегмент кода на этапе компиляции программы и находятся там до завершения работы программы, поэтому все функции в Си++

имеют глобальное время жизни и существуют в течение всего времени выполнения программы.

Сегмент данных (статическая память) предназначена для хранения переменных в течение всего времени выполнения программы. Если для переменной в какой-либо момент работы программы выделена память в сегменте данных, она там будет находиться до завершения работы программы, даже если эта переменная больше не нужна. По умолчанию в сегменте данных хранятся глобальные переменные и константы.

Стек – это область памяти, в которой хранятся локальные переменные и параметры функций. При вызове функции ее параметры и локальные переменные помещаются в стек. Стек функционирует по принципу стакана – значение, помещенное в стек первым, оказывается на дне стека, в то время как последнее значение – на вершине стека. По завершении работы функции все данные, принадлежащие этой функции, удаляются из стека. Очистка стека начинается с вершины, т. е. со значений, помещенных в стек последними.

Динамическая память (англ. “*heap*” - куча) – свободная область памяти, незарезервированная в настоящий момент ни для одной исполняемой программы. Позволяет программисту управлять процессом выделения памяти под переменные и освобождением памяти. Переменные, размещаемые в динамической памяти, называются динамическими переменными.

Как правило, размер стека ограничен, и, следовательно, при выполнении сложных программных комплексов может произойти переполнение стека (*stack overflow*), когда в стеке уже не будет памяти для новых локальных переменных. Избежать подобной проблемы можно двумя способами:

1) Создание переменной в статической памяти.
2) Создание переменной в динамической памяти.

Очевидно, что первое решение обладает недостатками – переменные становятся глобальными, а значит доступными из любой точки программы. А

главное, подобные переменные останутся в статической памяти до конца выполнения программы. В этом случае хранение каких-либо локальных, промежуточных переменных становится невозможным.

Второе решение – выделение памяти для переменной из области динамической памяти. Суть динамическая память – это вся свободная оперативная память, не занятая в настоящий момент другими исполняющимися программами. При этом, за программистом остается возможность освобождения выделенной памяти, в отличии от памяти статической.

В языке Си++ выделение и освобождение памяти осуществляется посредством операторов *new* и *delete*, а вся работа происходит через указатели:

```
int* variable = new int;    // Выделение памяти под переменную
*variable = 7;       // Инициализация переменной значением
printf(“%d”, *variable);    // Работа со значением в переменной
delete variable;      // Освобождение памяти
```

Стоит помнить, что для каждого вызванного оператора *new* должен быть вызван свой оператор *delete*. Поясним на примере:

```
int* variable = new int;    // Выделение памяти под переменную
variable = new int;      // Повторное выделение памяти
delete variable;      // Освобождение памяти
```

В первой строке примера мы выделяем память в динамической области, а полученный адрес помещаем в указатель.

Во второй строке мы вновь выделяем память в динамической области и помещаем в тот же указатель. Однако теперь адрес выделенной в первый раз памяти потерян, и мы никак не сможем к нему обратиться, несмотря на то, что он зарезервирован операционной системой для нашей программы.

В третьей строке происходит освобождение памяти по указателю. Однако освобождается только та память, которая была выделена во второй раз, когда память, выделенная в первой строке остается неосвобожденной. Подобная ситуация называется утечкой памяти – потеря доступа к выделенной динамической памяти без возможности её восстановления. Несмотря на то, что наша программа уже не сможет получить доступ к данной памяти, операционная

система всё равно полагает, что данная область памяти используется этой программой, а значит, её нельзя зарезервировать для другой программы, пока операционная система не будет перезапущена. В данном примере мы потеряли доступ к 4 байтам оперативной памяти, однако в реальных программах подобная ошибка может стоить мегабайты или гигабайты оперативной памяти. Например, если подобная ошибка допущена на серверном приложении, то это приведет к постоянным “падениям” сервера.

Аналогичная проблема может возникнуть при повторной попытке освобождения уже освобожденной памяти. Например, следующий код приведет к ошибке:

```
int* variable = new int;    // Выделение памяти под переменную
delete variable;      // Освобождение памяти
delete variable;      // Повторное освобождение памяти
```

Выделение и освобождение памяти под, например, целочисленный массив выглядит следующим образом:

```
int size = 10;
int* intArray;
intArray = new int[size];
…
delete [] intArray;
```

Таким образом, для выделения памяти под массив данных необходимо использовать квадратные скобки с указанием числа элементов массива. Обратите внимание, что при выделении массива в качестве размера мы можем использовать значение обычной переменной, а не константной. Данная особенность позволяет создавать с помощью динамической памяти более гибкие и универсальные алгоритмы, чем со статической.

Пример:

```
int main()
{
    int n;
    printf("Input number of elements: ");
    scanf("%d", n);         //вводим количество элементов массива

    int *p = new int[n];    //выделяем память
    for(int i = 0; i < n; ++i)
```

```
        array[i] = i*10;    //присваиваем элементам массива удесятеренное
значение индекса

    for(int i = 0; i < n; ++i)
        printf("%d ", array[i]);    //выводим все элементы массива через про-
бел в одну строчку
    printf("\n");
    delete []p;    //освобождаем память
    return 0;
}
```

Ошибки при работе с динамической памятью.

1. Невозможность выделения динамической памяти операционной системой. Причины могут быть различные, одной из которых является отсутствие незанятой оперативной памяти. В случае невозможности выделения динамической памяти под переменные оператор new не прерывает выполнение программы, а возвращает *null*. Таким образом, для исключения ошибок в коде, необходимо проверять указатели на неравенство *null* после каждого выделения памяти.

2. Изменение указателя, который хранит адрес выделенной области памяти:

```
int* variable = new int;    // Выделение памяти под переменную
variable = null;      // Фактически, потеря исходного адреса
delete variable;      // Освобождение памяти
```

Если вы используете динамическую память, очень важно по окончании работы выполнить её освобождение. Изменение указателя может привести к потере адреса выделенной памяти, что сделает невозможным её освобождение.

3. Использование освобождённой области:

```
int* pointer1 = new int;
int* pointer2 = pointer1; //копируем адрес выделенной памяти
delete ponter1; //освобождаем память
*pointer2 = 7; //так как во втором указателе остался исходный адрес,
//в него можно записать значение, что будет ошибкой
```

Решением может быть обнуление указателя (указателей) после освобождения памяти.

4. Освобождение освобождённой памяти:

```
int* variable = new int;    // Выделение памяти под переменную
```

```
delete variable;      // Освобождение памяти
delete variable;      // Повторное освобождение памяти
```

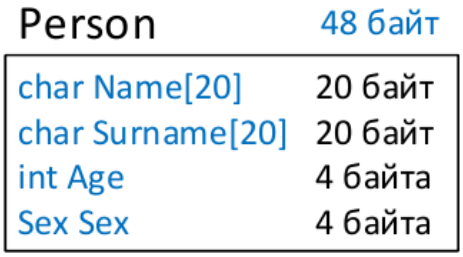
Для предотвращения ошибки также стоит обнулять указатели после освобождения памяти.

5.5 Классические структуры данных

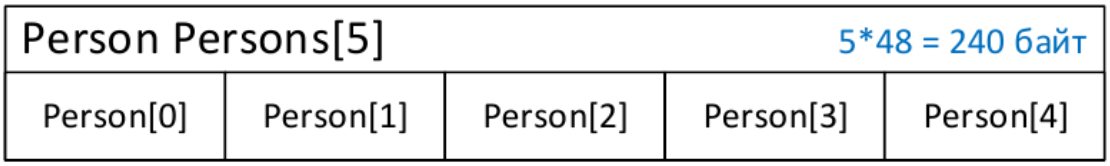
Структура данных – это способ организации каких-либо данных вне зависимости от их типа. В программировании можно перечислить следующие структуры данных:

1) Массив.
2) Односвязный и двусвязный списки.
3) Дерево.
4) Граф.
5) Словарь.
6) Стек.
7) Очередь.

Массивы были рассмотрены в п.3.2. Однако достаточно часто организация данных в виде массива не удобна. Например, при работе с массивом мы должны заранее определить максимальное количество его элементов, независимо от того, используем ли мы их все или только часть. В итоге мы либо резервируем под массив избыточный объём памяти и работаем только в его ограниченной части, не предоставляя доступа к зарезервированной памяти для других программ, либо резервируем массив недостаточного размера, что приводит к выходу за допустимые пределы и ошибке выполнения программы.



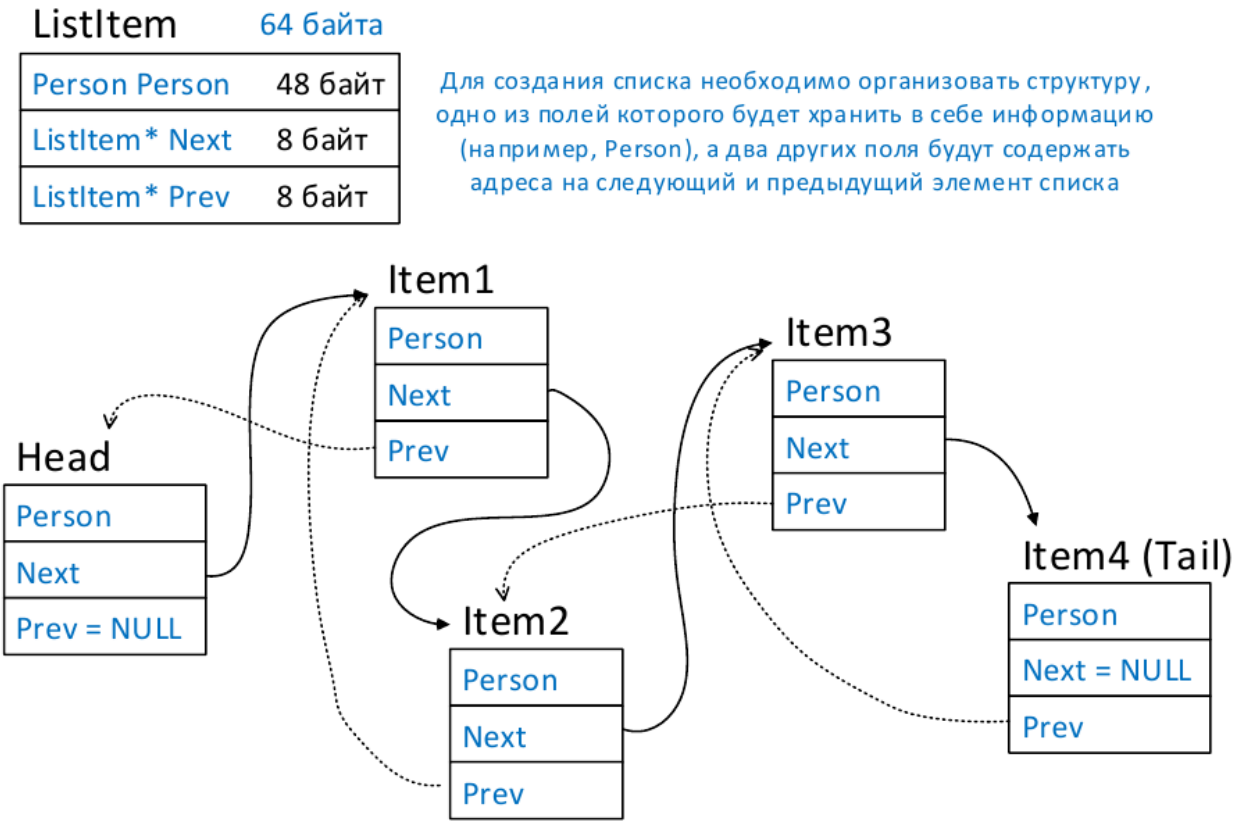
Память под переменную пользовательской структуры Person выделяется единым блоком в 48 байт
Все поля структуры в памяти хранятся последовательно



Память под массив любого типа данных также резервируется единым блоком, равным количеству элементов массива, умноженному на размер одного элемента. Все элементы массива располагаются в памяти последовательно друг за другом, обращение по индексу производит необходимый сдвиг по адресу в памяти от адреса первого элемента массива.
Поэтому при создании массива необходимо строго указывать его размер

С этой точки зрения, массивы не позволяют гибко использовать память компьютера. Однако существуют иные способы организации данных, позволяющих резервировать память исключительно под необходимое в настоящий момент количество данных и, в случае потребности, увеличить или уменьшить зарезервированный размер. Примером такой структуры являются списки.

Список – структура данных, в которой каждый элемент списка указывает на последующий элемент. В случае двусвязного списка каждый элемент списка будет указывать на последующий и предшествующий элемент списка. Таким образом, имея адрес только первого элемента списка, или ***головы списка***, можно получить доступ к любому следующему элементу списка.



В отличии от массива, память под каждый элемент массива резервируется отдельно от остальных в любом свободном пространстве памяти (кучи).
Нет необходимости заранее указывать размер списка, его расширение происходит динамически
Обратиться к любому элементу списка можно переходя по адресам указателей Next от головы до нужного элемента

Рассмотрим на примере. Допустим, нам нужно создать список, хранящий в себе данные о людях (структура *Person*). Для этого мы создадим следующую структуру:

```
struct PersonListItem
{
  Person Person;
  PersonListItem* NextItem;
}
```

Обратите внимание, что одно из полей данной структуры является указателем на такую же структуру. В данном поле мы будем хранить адрес следующего элемента списка.

Теперь, как же создать сам список? Для начала создадим первый элемент списка – голову списка:

```
PersonListItem* personListHead = new PersonListItem;
personListHead→Person = ReadPerson();    // Jack Bauer
personListHead→NextItem = NULL;
```

Обратите внимание, что в указатель следующего элемента мы присвоили *NULL*. Это логично, так как пока что в нашем списке существует только один элемент. Придерживайтесь правила, что при создании какого-либо элемента списка указатели на соседние элементы должны быть инициализированы *NULL*. В противном случае, указатель будет указывать на невыделенную область памяти, что приведет к ошибке выполнения.

Теперь добавим еще один элемент списка:

```
PersonListItem* personListItem = new PersonListItem;
personListItem→Person = ReadPerson();    // Tony Almeida
personListItem→NextItem = NULL;
personListHead→NextItem = personListItem;    // связывание нового элемента с головой
```

Теперь, если мы захотим получить доступ ко второму элементу списка, это можно сделать через указатель на начало списка:

```
personListHead→Person→Name = “Tom”  // меняем имя Jack на Tom в 1 элементе
personListHead→NextItem→Person→Name = “John” //меняем имя Tony на John во 2 элементе
```

Для перехода к нужному элементу массива (например, пятому) можно использовать следующий код:

```
PersonListItem* currentItem = personListHead;
for(int i =0; i < 5; i++)
{
  currentItem = currentItem→NextItem;
}
```

После выполнения цикла указатель *currentItem* будет содержать адрес пятого элемента списка. Однако данный пример не учитывает, что в списке может просто и не существовать пятого элемента (например, элементов только три). Поэтому необходимо производить проверки следующего указателя на *NULL*. Например, переход к концу списка будет выглядеть следующим образом:

```
PersonListItem* currentItem = personListHead;
while(currentItem→NextItem != NULL)
{
  currentItem = currentItem→NextItem;
}
```

Данный пример после выполнения присвоит в *currentItem* адрес последнего элемента списка. Однако если при создании и добавлении элемента в список не присвоить в указатель на следующий элемент *NULL*, вы не сможете отловить конец списка, а данный пример превратится в бесконечный цикл. Также стоит помнить, что присвоение *NULL* в какой-либо уже добавленный элемент списка (например, *personListHead→NextItem = NULL*) приведет к потере данных всего последующего списка и утечке памяти.

Работа и организация двусвязного списка осуществляется аналогичным образом, за тем исключением, что каждый элемент списка должен дополнительно указывать на предыдущий:

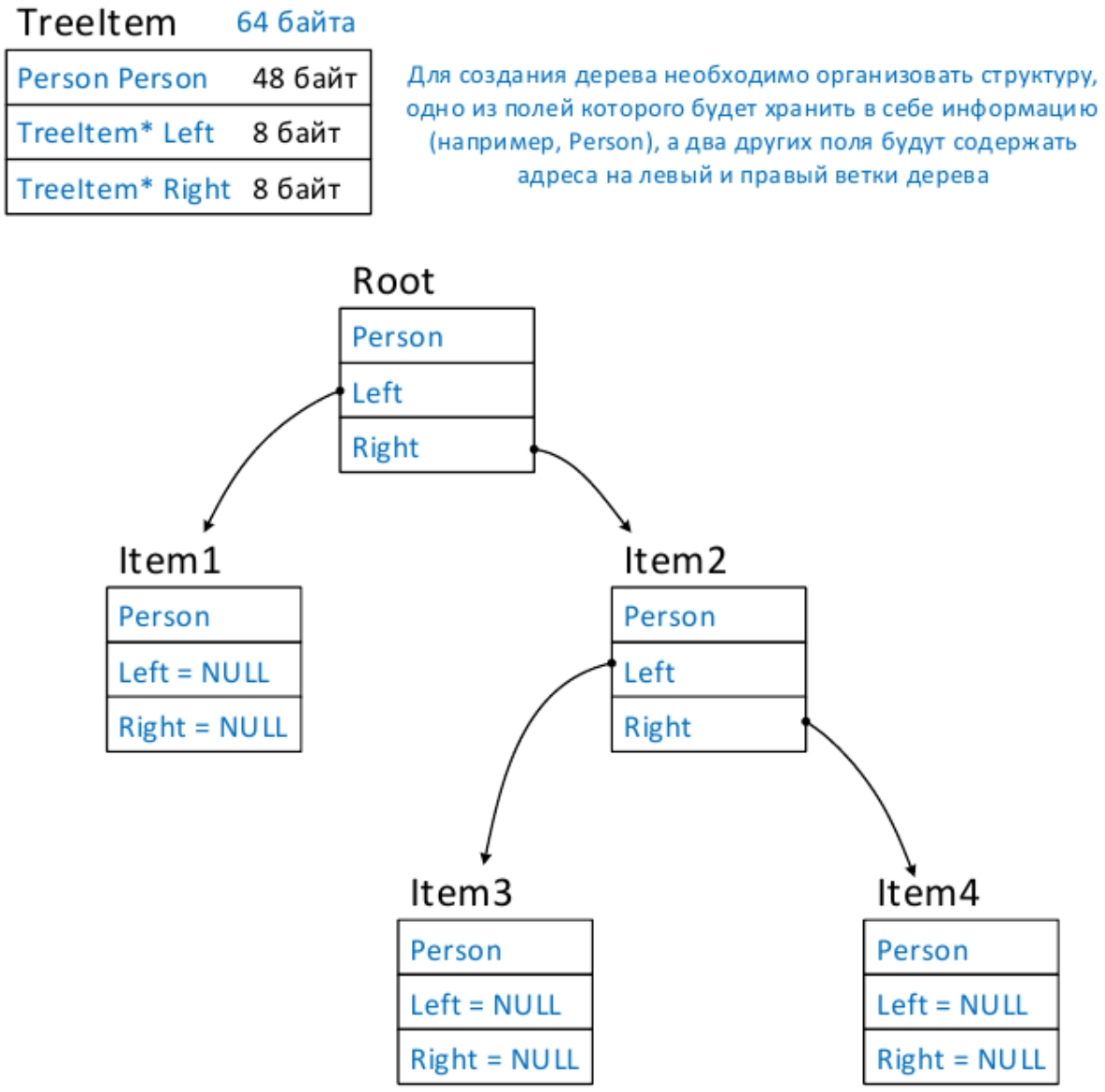
```
struct PersonListItem
{
  Person Person;
  PersonListItem* NextItem;    // Следующий элемент списка
  PersonListItem* PrevItem;    // Предыдущий элемент списка
}
```

Очевидным преимуществом списков перед массивами является то, что мы выделяем память под объекты только при необходимости, не расходуя лишней памяти. Недостатком же является простота потери данных в случае ошибки или неправильного действия с указателем.

Дерево – структура данных, в которой каждый элемент дерева имеет набор дочерних элементов, каждый из которых связан строго с одним родителем. В дереве можно выделить корневой узел – самый верхний узел, из которого выходит всё дерево – и листья – узлы дерева, не имеющие своих дочерних элементов. Также существуют понятия корня – любого рассматриваемого в настоящий момент узла – и поддеревьев. Организация элемента дерева на языке Си++ выглядит следующим образом:

```
struct PersonTreeItem
{
  Person Person;
  PersonTreeItem* LeftNode;    // Левая дочерняя ветвь
  PersonTreeItem* RightNode;    // Правая дочерняя ветвь
}
```

Здесь представлен пример дерева с двумя потомками – левой и правой ветвями. Данный вид деревьев достаточно распространен в вычислительных алгоритмах, например, задачах оптимизации, сортировки и поиска данных. Обратите внимание, что дочерний узел не хранит в себе информации о родительском узле, что позволяет достаточно быстро “переставлять” одну ветвь дерева в другую. В более общем случае элемент дерева может хранить в себе неограниченное количество дочерних элементов.



Элементы дерева также хранятся отдельно друг от друга, храня в себе указатели на дочерние узлы дерева. Зная только расположение элемента Root, можно перейти сверху вниз к любому элементу дерева. Благодаря своей структуре, можно достаточно быстро «переставлять» местами любые ветви дерева между собой вне зависимости от их размера, что позволяет использовать деревья для организации быстрых алгоритмов поиска и сортировки

Списки и деревья являются лишь частными случаями такой структуры, как граф. ***Граф*** – структура данных, каждый элемент которого связан с неопределенным количеством других элементов графа.

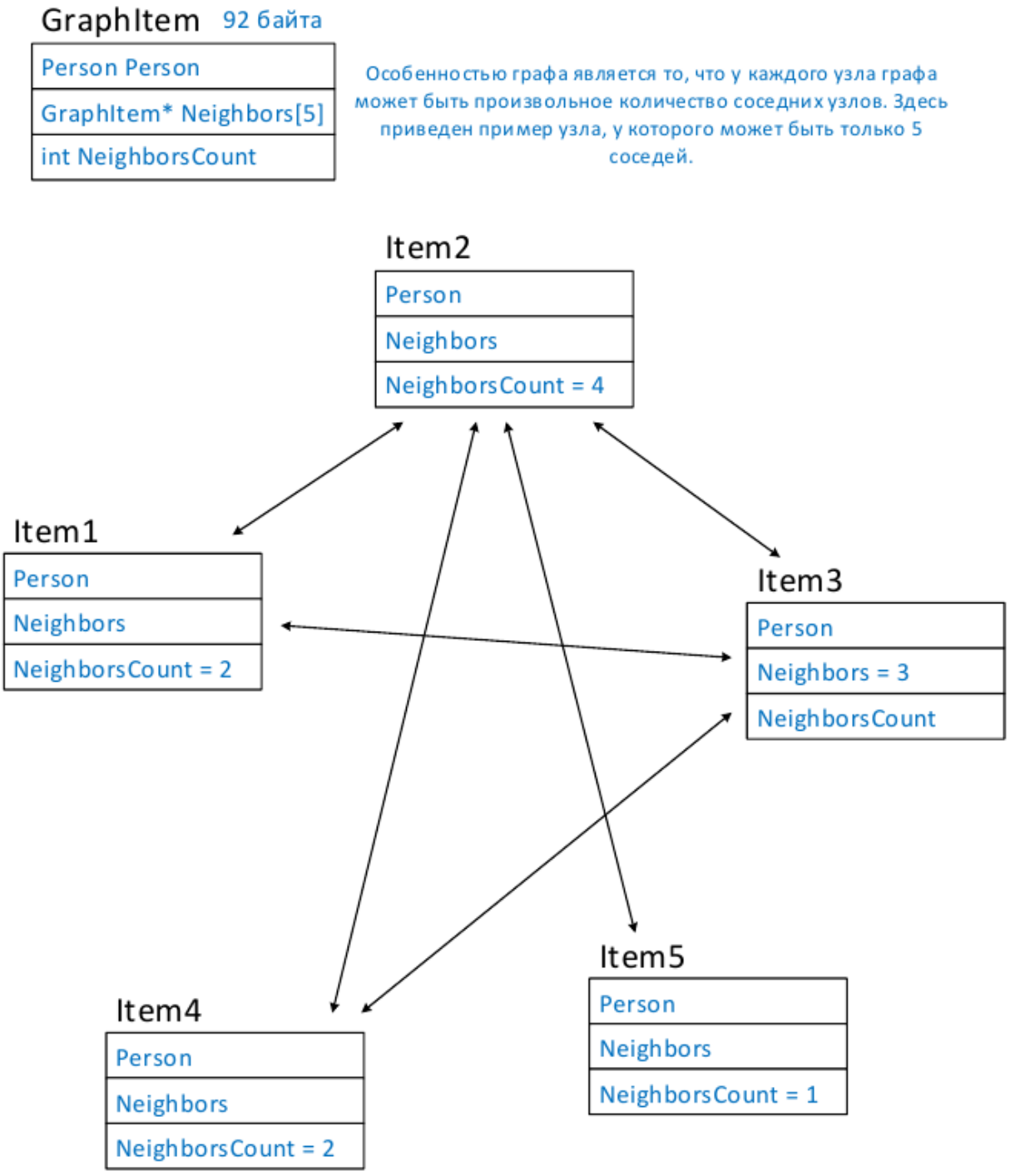
Как правило, в графе нельзя четко определить головной или корневой узел, им может быть любой элемент. По этому корневой узел выбирается условно и определяется конкретной реализацией и назначением графа. Организация элемента графа на языке Си++:

```
struct PersonGraphItem
{
  Person Person;
  PersonGraphItem* Neighbors;    // Массив «соседей» данного элемента
  int NeighborsCount;      // Количество соседей
}
```

Граф – достаточно сложная для работы структура данных, и применяется при решении нетривиальных задач, где обычные алгоритмы не справляются, либо справляются с огромными затратами вычислительных и временных ресурсов.

Словарь (отображение) – структура данных, хранящая набор уникальных пар <*ключ-значение*>, где в качестве ключа и значения могут быть любые типы данных. Для понимания данной структуры представьте себе обычный словарь. В нём определенное слово, или ***ключ***, которое является уникальным в пределах данного словаря. Данному слову соответствует некоторое описание, ***значение***, поясняющее первоначальное слово. Таким образом, словарь организует точное соответствие уникального ключа определенному значению. Обратившись к такому словарю по ключу, мы получим нужное значение, при условии существования ключа в словаре. Словарь не может содержать ключ без значения или значение без ключа.

Примером словаря можем послужить словарь отдела кадров <*Person-Double*>, где ключ *Person* указывает уникального человека в пределах предприятия, а *Double* – значение зарплаты данного человека. Или словарь расписания поездов <*int-string*[]>, где *int* – ключ-номер поезда, а *string*[] – массив строк-названий станций.



Граф – достаточно сложная структура для хранения данных, используют её крайне редко. В зависимости от задачи могут различать: направленные и ненаправленные графы, весовые и невесовые, цикличные и ацикличные и т.д.

Стек – структура данных, работа которой организована по принципу “Первым пришел, последним вышел”. Для примера возьмем стопку тарелок. Сначала вы положите на стол первую тарелку, затем на неё вторую, а затем и третью. Но когда вам потребуется взять тарелку, вы будете забирать их в обратном порядке – сначала третью, затем вторую и только потом первую. Таким

образом, тарелка, пришедшая первой, вышла последней. Работа стека организована аналогичным образом – помещая в неё последовательность элементов, вы сможете получить к ним доступ только в обратном порядке. Подобная структура данных часто используется в низкоуровневых программах, где требуется жесткий контроль памяти. Однако, данная структура может находит своё применение и пользовательских приложениях. Не путайте понятия стека как структуры данных и стека как области памяти – несмотря на идентичное название данные понятие принципиально отличаются.

Очередь – структура данных, организующая принцип «Первым пришел, первым вышел». Фактически, очередь противоположна стеку, и принцип её работы понятен из названия.

Отличным источником для знакомства со структурами данных и их применением является [4], входящей в список обязательной литературы для программистов.

6 Техники написания качественного кода

Данную главу хотелось бы начать цитатой С. Макконнелла из книги «Совершенный код» [10]: «Пишите код так, будто человек, который будет его поддерживать маньяк-психопат, который знает, где вы живете».

На самом деле, исходный код это разновидность технической документации. Любая техническая документация должна оформляться согласно стандартам, принятым в мире, государстве или конкретной компании. В Российской Федерации, требования и стандарты технической документации описаны в ГОСТах ЕСКД (единая система конструкторской документации) и ЕСПД (единая система программной документации). Придерживаться общепринятых стандартов обязан каждый инженер или разработчик. Это позволяет обеспечить простоту понимания технических проектов между инженерами, что важно, например, при разработке в команде – любая команда должна владеть одним профессиональным языком для облегчения разработки и повышения её качества – или при подключении новых разработчиков к проекту – очень важно, чтобы новый работник быстро разобрался с идеей и структурой проекта и приступил к работе.

Говоря об программном коде, то в каждой компании также существуют собственные ***стандарты оформления кода***. Стандарт оформления кода – это система правил, касающихся именования объектов и функций, стиля оформления исходного кода, требования к архитектуре проекта и т.п. При устройстве на работу в качестве разработчика, вы должны в обязательном порядке ознакомиться со стандартом оформления кода данной компании и весь дальнейший код писать строго с соблюдением данных правил. Ни при каких обстоятельствах вы не должны нарушать данные правила, даже если они кажутся вам «неудобными». То, что кажется неудобным вам, может быть удобным для остальных. Разумеется, если у вас есть предложения по улучшению оформления кода, вы можете их высказать команде или вашему непосредственному руководителю, но оформлять код, игнорируя принятый стандарт, приведет к

проблемам при дальнейшей поддержке вашего кода другими разработчиками. Разработчики в команде быстро привыкают к стандартам оформления. В итоге, код, оформленный не по принятому стандарту, приводит к увеличению времени на его чтение. Таким образом, если другому разработчику потребуется исправить ошибку в вашем коде, ему понадобится больше времени на решение проблемы. Больше времени – медленная разработка, медленная разработка – поздние выпуск программы, поздний выпуск программы – потеря прибыли.

Это может показаться странным, однако код в первую очередь должен быть правильно оформлен, и уже во вторую правильно выполнять поставленную задачу. Это иллюстрируется следующим примером. Представьте, что вы – новый молодой разработчик в команде и перед вами была поставлена задача написать часть программы. Вы изначально не уделили должное внимание оформлению кода, решив сначала написать код, а затем «оформить все по требованиям». Однако при написании кода вам потребовался совет другого, более опытного разработчика. Теперь: а) вы не можете продолжать работу, пока вам не помогут; б) другой разработчик не может быстро ответить советом, так как ему сначала нужно разобраться в вашем не оформленном коде; в) исходный код плохо оформлен и не работает, пока два работника простаивают. Из примера следует, что оформление кода при разработке важнее правильной работы, а также что оформление кода — это не процедура, выполняющаяся после написания кода, а неотъемлемая часть процесса разработки.

Далее в п.6.1-6.2 представлены стандарты оформления кода, принятые в рамках данной дисциплины, и являющиеся частью соглашения по оформлению кода *RSDN* [9]. В последующих подразделах также описываются полезные техники по написанию качественного кода.

6.1 Правила именования и стандарты оформления кода

Общие правила

1. Помните! Код чаще читается, чем пишется, поэтому не экономьте на понятности и чистоте кода ради скорости набора.
2. Не используйте малопонятные префиксы или суффиксы (например, венгерскую нотацию), современные языки и средства разработки позволяют контролировать типы данных на этапе разработки и сборки.
3. Не используйте подчеркивание для отделения слов внутри идентификаторов, это удлиняет идентификаторы и затрудняет чтение. Вместо этого используйте стиль именования Кемел или Паскаль.
4. Старайтесь не использовать сокращения лишний раз, помните о тех, кто читает код.
5. Старайтесь делать имена идентификаторов как можно короче (но не в ущерб читаемости). Помните, что современные языки позволяют формировать имя из пространств имен и типов. Главное, чтобы смысл идентификатора был понятен в используемом контексте. Например, количество элементов коллекции лучше назвать *Count*, а не *CountOfElementsInMyCollection*.
6. Когда придумываете название для нового типа данных, пространства имен или интерфейса, старайтесь не использовать имена, потенциально или явно конфликтующие со стандартными идентификаторами.
7. Предпочтительно использовать имена, которые ясно и четко описывают предназначение и/или смысл сущности.
8. Старайтесь не использовать для разных сущностей имена, отличающиеся только регистром букв. Разрабатываемые вами компоненты могут быть использованы из языков, не различающих регистр, и некоторые методы (или даже весь компонент) окажутся недоступными.
9. Старайтесь использовать имена с простым написанием. Их легче читать и набирать. Избегайте (в разумных пределах) использования

слов с двойными буквами, сложным чередованием согласных. Прежде, чем остановиться в выборе имени, убедитесь, что оно легко пишется и однозначно воспринимается на слух. Если оно с трудом читается, и вы ошибаетесь при его наборе, возможно, стоит выбрать другое.

Стили использования регистра букв

Паскаль – указание этого стиля оформления идентификатора обозначает, что первая буква заглавная и все последующие первые буквы слов тоже заглавные. Например, ***B****ack****C****olor, ****L****ast****M****odified, ****D****ate****T****ime*.

Кэмел – указание этого стиля обозначает, что первая буква строчная, а остальные первые буквы слов заглавные. Например, ***b****order****C****olor, ****a****ccess****T****ime, ****t****emplate****N****ame*.

Сокращения

1. Не используйте аббревиатуры или неполные слова в идентификаторах, если только они не являются общепринятыми. Например, пишите *GetWindow*, а не *GetWin*.
2. Не используйте акронимы, если они не общеприняты в области информационных технологий.
3. Широко распространенные акронимы используйте для замены длинных фраз. Например, *UI* вместо *User Interface* или *Olap* вместо *On-line Analytical Processing*.
4. Если имеется идентификатор длиной менее трех букв, являющийся сокращением, то его записывают заглавными буквами. Например, *System.IO, System.Web.UI*. Имена длиннее двух букв записывайте в стиле Паскаль или Кэмел, например *Guid*, *Xml, xmlDocument*.

Оформление

1. Используйте табуляцию, а не пробелы для отступов.
2. Избегайте строк длиннее 78 символов, переносите инструкцию на другую строку при необходимости.

3. При переносе части кода инструкций и описаний на другую строку вторая и последующая строки должны быть отбиты вправо на один отступ (табуляцию).
4. Оставляйте запятую на предыдущей строке так же, как вы это делаете в обычных языках (русском, например).
5. Избегайте лишних скобок, обрамляющих выражения целиком. Лишние скобки усложняют восприятие кода и увеличивают возможность ошибки. Если вы не уверены в приоритете операторов, лучше загляните в соответствующий раздел документации.
6. Не размещайте несколько инструкций на одной строке. Каждая инструкция должна начинаться с новой строки.

Примеры:

```
longMethodCall(expr1, expr2, expr3,
    expr4, expr5);

var1 = a * b / (c - g + f)
    + 4 * z;

var2 = (a * (b
    * (c + d)
    + e * (f / z))
    + 4);
```

Пустые строки и пробелы в строке

1. Используйте две пустые строки между логическими секциями в исходном файле.
2. Используйте две пустые строки между объявлениями классов и интерфейсов.
3. Используйте одну пустую строку между методами.
4. Если переменные в методе объявляются отдельным блоком, используйте одну пустую строку между их объявлением и инструкцией, идущей за этим блоком.
5. Используйте одну пустую строку между логическими частями в методе.

6. После запятой должен быть пробел. После точки с запятой, если она не последняя в строке (например, в инструкции *for*), должен быть пробел. Перед запятой или точкой с запятой пробелы не ставятся.
7. Все операторы должны быть отделены пробелом от операндов с обеих сторон.

   Примеры:

```
TestMethod(a, b, c);

a=b;                          // неверно
a = b;                        // верно
for (int i=0; i<10; ++i)      // неверно
{
      a = i;                  // содержимое фигурных скобок сдвигается
      b *= i;                 // на одну табуляцию
}
for( int i = 0 ;i< 10;++i)    // неверно

for (int i = 0; i < 10; ++i) // верно
{
}

if(a==b){}                    // неверно

if (a == b)                   // верно
{
}
```

Локальные переменные

1. Используйте стиль Кэмел для регистра букв в именах переменных.
2. Объявляйте переменные непосредственно перед их использованием.
3. Счетчики в циклах традиционно называют *i*, *j*, *k*, *l*, *m*, *n*.
4. Для временных локальных переменных, используемых в коротких участках кода, можно давать имена, состоящие из начальных букв слов имени типа.
5. Не объявляйте более одной переменной в одной инструкции.
6. Комментируйте объявления так, чтобы были понятны назначение и способы использования переменной.
7. Инициализируйте переменные при объявлении, если это возможно.

Функции

1. Используйте глаголы или комбинацию глагола и существительных и прилагательных для имен функций.
2. Используйте стиль Паскаль для регистра букв.

Примеры:

```
private int RemoveAll() {}
public void GetCharArray() {}
internal static Invoke() {}
```

Оформление *if, if-else, if-else-if else*

if:

```
// Правильное оформление
if (condition)
{
    DoSomething();
    ...
}

// Правильное оформление
if (condition)
    DoSomething();

// Неправильное оформление
if(condition) DoSomething();

// Неправильное оформление
if (condition) {
    DoSomething();
}

// Неправильное оформление
if (condition)
    {
    DoSomething();
    }
```

if с последующим *else*:

```
// Правильное оформление
if (condition)
{
    DoSomething();
    ...
}
else
```

```
{
    DoSomethingOther();
    ...
}

// Правильное оформление
if (condition)
    DoSomething();
else
    DoSomethingOther();

// Неправильное оформление
if (condition) {
    DoSomething();
    ...
} else {
    DoSomethingOther();
    ...
}

// Неправильное оформление
if (condition) DoSomething(); else DoSomethingOther();
```

Оформление *for*, *while*, *do-while*

```
for (int i = 0; i < 5; ++i)
{
    ...
}

while (condition)
{
    ...
}

do
{
    ...
}
while (condition);

switch (condition)
{
    case 1:
    case 2:
        x = ...;
        break;
    case 3:
        x = ...;
        break;
    default:
        x = ...;
        break;
}
```

Пользовательские типы данных (перечисления, структуры)

1. Все не вложенные типы (размещаемые в пространствах имен или непосредственно в глобальном пространстве), за исключением делегатов, должны находиться в отдельных файлах.
2. Старайтесь объявлять вложенные типы в начале внешнего типа.
3. Старайтесь не делать излишней вложенности типов. Помните, что вложенными должны быть только тесно связанные типы.
4. Элементы типов должны отделяться одной строкой друг от друга. Вложенные типы должны отделяться двумя строками. При объявлении большого количества полей, используемых внутри класса (не публичных), можно опускать пустую строку.

Представленные правила описывают только часть требований *RSDN*, не относящуюся к объектно-ориентированному программированию. С полным списком требований вы можете ознакомиться в [9].

6.2 Комментирование кода

Для описания сути некоторого участка кода, пояснений к алгоритму и другой важной информации используйте несколько подряд идущих однострочных комментариев (//...). Между группой комментариев и собственно кодом поставьте пустую строку. Это покажет, что комментарий относится к блоку кода, а не к конкретной инструкции. Напротив, если комментарий относится к конкретной инструкции, прижмите его вплотную к этой инструкции.

Комментируйте объявления переменных, используя однострочные комментарии на той же строке, как это показано ниже.

Отделяйте текст комментария одним пробелом «// Текст комментария.».

Примеры

```
int level;
int size;  // size of table

// Line 1
//
ArrayList list = new ArrayList(10);
```

```
// Line 1
// Line 2
//
for (int i = 0; i < list.Count; i++)
    ...
```

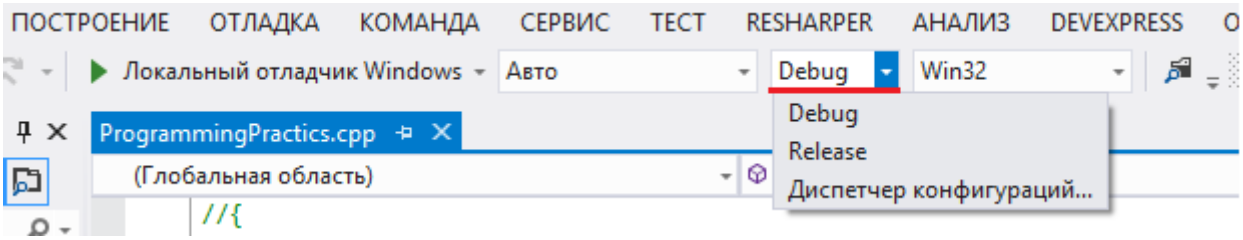
Комментируя код, старайтесь объяснять, что он делает, а не какая операция производится. Так, инструкции *if* соответствует выражение «если... то...», причем часть, идущая за «то», является кодом, который будет выполняться, если выражение в *if* будет верным. Таким образом, для конструкции «*if* (*somePath* && *File.Exists*(*somePath*))», нужно написать комментарий «// Если выбранный файл существует, то...», а не «// Производим проверку на наличие файла и, если он имеется, удаляем его». Часть предложения, идущую за «то», вписывайте непосредственно перед выполнением конкретных действий. Для инструкций, осуществляющих действия, пишите «// Производим...» или «// Делаем...», где вместо троеточия вписывайте описания действий. Описывая действия, старайтесь описывать суть происходящего, а не то, что делают те или иные операторы. Так, совершенно бессмысленны комментарии вроде «Присваиваем переменной a значение *b*» или «вызываем метод *f*».

Помните, что экономить на комментариях нельзя. Однако не стоит также формально подходить к процессу создания комментариев. Задача комментария – упростить понимание кода. Есть немало случаев, когда сам код отличным образом себя документирует.

6.3 Инструменты отладки в среде разработки

В среде *Visual Studio*, так же, как и во многих других средах разработки, существует специальный набор инструментов, предназначенный для отладки программы. С помощью данных инструментов разработчик может отследить выполнение программы шаг за шагом, наблюдая за тем, как меняются значения тех или иных переменных. Этот механизм достаточно удобен, потому что найти ошибку просто в исходном коде зачастую трудно или даже невозможно. Давайте рассмотрим некоторые из инструментов отладки.

В первую очередь, вам необходимо установить конфигурацию сборки программы в состояние *Debug*. ***Конфигурация сборки*** – это набор настроек компилятора и компоновщика, указывающего отдельные специфические возможности сборки проекта. Вы можете создавать свои собственные конфигурации сборки, если вы достаточно опытный разработчик и понимаете принципы работы компиляторов. В среде же *Visual Studio* изначально существует две конфигурации сборки – *Debug* и *Release*. Фактически, разница между ними заключается в том, что при конфигурации *Release* выполняется оптимизация кода, а при *Debug* – нет. В итоге, программа, собранная под *Release* может выполняться на машине конечного пользователя быстрее. Однако при этом отладка оптимизированного кода невозможна, а значит и средства отладки будут недоступны. Поэтому для отладки программы необходимо установить конфигурацию сборки *Debug*. Переключение конфигураций сборки представлено на рисунке ниже:



Теперь всё готово для отладки приложения. Отладку рассмотрим на следующем примере:

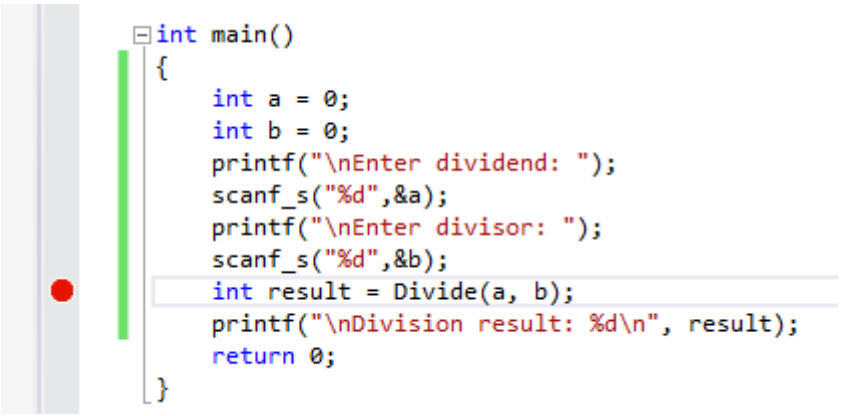
```
#include "stdafx.h"
#include <stdio.h>

//
int Divide(int dividend, int divisor)
{
int result;
result = dividend/divisor;
return result;
}

int main()
{
int a = 0;
int b = 0;
printf("\nEnter dividend: ");
scanf_s("%d", &a);
```

```
printf("\nEnter divisor: ");
scanf_s("%d", &b);
int result = Divide(a, b);
printf("\nDivision result: %d\n", result);
result 0;
}
```

Первым и основным инструментом отладки является точка останова (*breakpoint*). Точка останова позволяет приостановить выполнение программы на любом этапе и отследить текущее значение переменных. Для установки точки останова необходимо выбрать интересующую вас строчку исходного кода и кликнуть напротив неё на панели слева (или нажать *F*9). Если вы всё сделали правильно, то возле строчки кода появится красный кружочек:

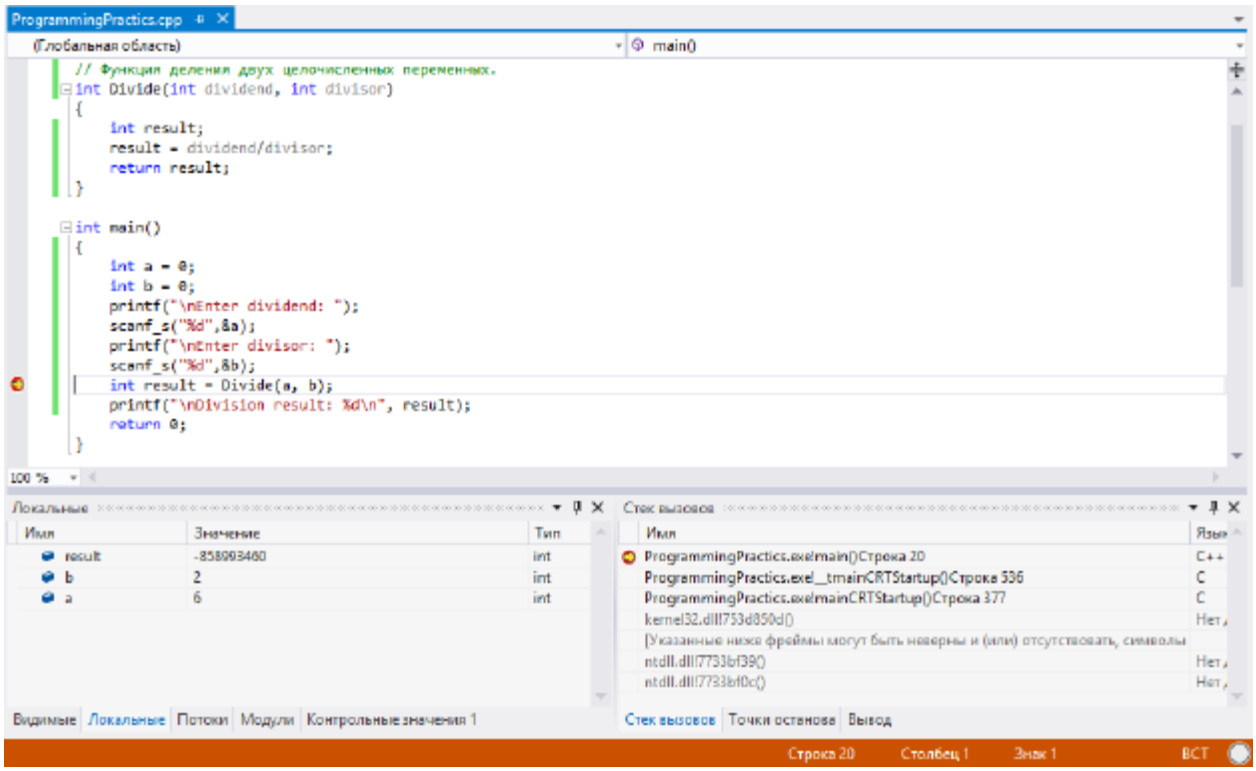


Наберите представленный код и установите точку останова на аналогичную строчку. Теперь запустим программу на исполнение.

В среде *Visual Studio* приложение можно запустить как с отладкой, так и без. Данные варианты указаны в меню *Отладка* главного меню. Если вы запустите приложение без отладки, точки останова не будут работать. Для выполнения данного примера вам нужно обязательно запускать приложение с помощью режима *Начать отладку*.

Приложение запустилось и потребовало от вас ввода значений двух переменных, значения которых будут поделены друг на друга. Однако после ввода выполнение программы приостанавливается и вас возвращает в экран *Visual Studio*. Заметьте, что на панели слева, где установлена точка останова, появилась желтая стрелка. Эта стрелка указывает на ту строчку кода, которая еще

не выполнилась, но выполнится следующей:



Теперь, если вы наведете курсор мыши на ту или иную переменную, то её текущее значение всплывет в виде подсказки. Например, две введенные с клавиатуры переменные имеют то значение, которое ввели вы. А переменная, в которую должен поместиться результат выполнения функции хранит в себе 0. В данном случае, вызов функции деления и присвоение значения в переменную только должно произойти, а значит, и результат деления еще не поместился в *result*. Более того, на панели *Локальные переменные* внизу экрана вы можете в виде таблицы-дерева посмотреть значения всех переменных, которые находятся в данной области видимости. Красным подсвечиваются те значения, которые были изменены за последний шаг – последнюю выполненную операцию. Чтобы продолжить выполнение программы нужно нажать *F5* (или команда *Продолжить* в меню *Отладка*), либо можно прервать выполнение программы, нажав *Shift+F5* (команда *Остановить*).

Однако, при работе с точками останова можно выполнять строки кода одну за другой. Перезапустите программу в режиме отладки и, когда про-

грамма вновь приостановиться в точке останова, нажмите *F*10. Программа выполнила следующую строчку кода, а желтая стрелка переместилась на строчку ниже. Теперь, если вы наведете курсор на переменную *result*, то заметите, что в неё поместился результат деления. Команда *Шаг с обходом*, вызываемая клавишей *F*10 производит выполнение одной строчки кода, позволяя построчно выполнить какой-либо алгоритм и отследить значения переменных на каждом шаге.

Перезапустите программу еще раз, но теперь, когда программа приостановится, нажмите *F*11. Теперь вы вместе с желтой стрелочкой переместились внутрь функции деления и можете с помощью нажатия *F*10 проследить выполнение внутри вызванной функции, а также значения всех локальных переменных. Команда *Шаг с заходом*, вызываемая клавишей *F*11, позволяет зайти внутрь вызываемой функции. Учтите, что зайти можно только в те функции, чей исходный код доступен. Вы не сможете зайти в функцию, находящуюся в системных библиотеках, либо сторонних *dll*-библиотеках, подключенных к вашему проекту.

Также в среде *Visual Studio* существует такой полезный инструмент, как панель *Стек вызовов* (или *Иерархия вызовов*, перевод может отличаться!). Данная панель показывает вам кто и в каком порядке вызвал текущую функцию. Например, в нашем текущем состоянии, *Стек вызовов* показывает, что сейчас выполняется функция деления, вызванная функцией *main*. Данный механизм полезен, когда необходимо посмотреть кто, когда и сколько раз вызывает ту или иную функцию, особенно если эта функция часто используется в различных частях программы.

Таким образом, точка останова, панель локальных переменных и панель стека вызовов являются мощными и эффективными инструментами при отладке программы.

Стоит помнить, что точка останова будет срабатывать только тогда, когда выполнение программы доходит до указанной строчки. Это важно, так как,

например, поставив точку останова внутри функции, которая вызывается более одного раза, то и останов программы будет не один раз, а столько, сколько эта функция вызывается. Также и с циклами: точка останова внутри цикла приведет к остановам на каждой итерации. Однако если точка останова находится в одной из веток условного оператора, то останов произойдет только при выполнении этой ветки условия.

Вопросы для проверки:

1) Что такое конфигурация сборки?

2) Чем отличается конфигурация сборки *Debug* от *Release*?

3) Что такое точка останова?

4) Что такое стек вызовов?

5) Какими способами можно узнать значение локальной переменной во время останова?

6.4 Деление исходного кода на множество файлов

В п.**Ошибка! Источник ссылки не найден.** описаны этапы сборки программы. Также в пособии не раз указывалось на ограничения языка Си++, связанные с тем, что функция или пользовательский тип данных, перечисление или структура, должны быть объявлены до момента их первого использования. Таким образом, нельзя вызвать функцию, если в исходном коде выше нет хотя бы прототипа этой функции, и нельзя создать экземпляра перечисления или структуры, если они не были объявлены выше или добавлены в код посредством директивы препроцессора *#include*. Данные ограничения создают определенные правила по ведению проекта программы в среде *Visual Studio* и других средах разработки.

Во-первых, в проекте явно присутствуют папки «Заголовочные файлы» (*Headers*) и «Файлы исходного кода» (*Source Files*). Предполагается, что в папке «Заголовочные файлы» будут храниться только так называемые ***заголовочные файлы*** – файлы с расширением *.h*, а в папке «Файлы исходного кода»

- файлы с расширением *.cpp*. Если вы раскроете содержимое данных папок, то убедитесь, что так и есть.

Заголовочные файлы должны хранить в себе только прототипы функций, объявление типов данных, а также различные глобальные переменные и константы, если они существуют в программе.

Файлы исходного кода *.cpp* хранят реализацию всех функций и, в дальнейшем, реализацию членов классов.

Деление проекта на файлы выполняется по функциональному назначению, где в одном файле группируются функции, направленные на решение схожих задач. Например, функции работы со строками, функции для подключения баз данных и т.д. Также любой пользовательский тип данных должен выделяться в отдельный заголовочный файл. Такое разделение проекта на файлы даёт максимальную гибкость при включении ранее реализованной функциональности или в случае значительного перепроектирования архитектуры программы.

В случае выполнения лабораторных работ логично вынести в отдельный заголовочный файл функции, реализованные во второй лабораторной работе, отдельным файлом хранить функции работы с массивами и строками третьей лабораторной работы, отдельные файлы для перечислений *Color*, *Weekday*, *Sex* и отдельные файлы для структуры *Person* четвертой лабораторной работы. Такая структура облегчает навигацию по проекту и поиск необходимой функциональности в программе.

Рассмотрим пример разделения программы на несколько файлов. Предположим, что у нас есть проект с единственным файлом исходного кода *program.cpp*, в котором реализовано несколько функций:

program.cpp

```
#include "stdafx.h"

void DoSomething1(){…};

int DoSomething2(){…};

double DoSomething3(int value){…};
```

```
int main()
{
  DoSomething1();
  int a = DoSomething2();
  double b = DoSomething3(a);
}
```

Вынесем функции в отдельные файлы. Для этого создадим файл *SomethingFunctions.h* в папке «Заголовочные файлы» и файл *SomethingFunctions.cpp* в папке «Файлы исходного кода».

Откроем заголовочный файл и напишем в нём директиву препроцессора #pragma once и прототипы функций:

SomethingFunctions.h

```
#pragma once

void DoSomething1();

int DoSomething2();

double DoSomething3(int value);
```

Директива препроцессора *#pragma once* контролирует выполнение директивы *#include*, не позволяя повторного добавления файла в проекте. Это гарантирует, что данный заголовочный файл не будет дважды подключен в другом файле.

В файл исходного кода добавим директиву *#pragma once*, подключение заголовочного файла *SomethingFunctions.h*, а также реализацию всех функций.

SomethingFunctions.cpp

```
#pragma once
#include "SomethingFunctions.h"

void DoSomething1(){…};

int DoSomething2(){…};

double DoSomething3(int value){…};
```

Подключение заголовочного файла *SomethingFunctions.h* необходимо для связывания реализации функций с их прототипами. После создания данных файлов можно модифицировать основной файл исходного кода:

program.cpp

```
#include "stdafx.h"
#include "SomethingFunctions.h"

int main()
{
  DoSomething1();
  int a = DoSomething2();
  double b = DoSomething3(a);
}
```

Теперь функции вынесены в отдельный файл, который может быть подключен с помощью директивы *#include* в любую точку исходного кода. Аналогичным образом происходит вынесение в отдельные файлы структур, перечислений и других пользовательских типов данных.

6.5 Разделение Модель-Вид

Мы уже изучили разделение программы на множество файлов, каждый со своим функциональным назначением. Несколько файлов могут быть объединены в так называемые модули, или пакеты. Деление может быть достаточно условным, например, файлы одного модуля могут находиться в одной директории, а может быть жесткое разделение модулей, когда модуль вынесен в отдельный проект Visual Studio. Одним из базовых делений программы на модули есть деление Модель-Вид.

Модель-Вид – паттерн проектирования, в котором функциональность, связанная с непосредственной расчетной частью и данными предметной области вынесена в модуль «Модель» (или «Бизнес-логика»), а вся работа с интерфейсом, например, консолью или графическим интерфейсом, вынесена в модуль «Вид» (или «Пользовательский интерфейс»). Данное разделение программы позволяет более гибко использовать бизнес-логику системы, а также уменьшает сложность поддержки программного кода в дальнейшем.

Рассмотрим пример. Допустим, в программе существует функция определения четности числа:

```
void IsEven()
{
  int value;
```

```
  cout << "Enter a value: ";
  cin >> value;
  if (value % 2 == 0)
    cout << "Value is even";
  else
    cout << "Value is odd";
}
```

Данная функция выполняет свою задачу, однако, если подумать, то функция жестко привязана к конкретному пользовательскому интерфейсу. Например, мы не сможем использовать функцию IsEven(), если захотим проверить на четность число, созданное программно, или считанное из файла. Пользователь должен будет обязательно ввести число с клавиатуры, а результат посмотреть на экране. Для работы с файлами или для работы с графическим пользовательским интерфейсом нужно будет писать новые функции вычисления четности числа, что фактически является дублированием функциональности.

Для решения данной проблемы разделим функцию на части Модель-Вид. Для этого вынесем из функции весь код, связанный с консолью в отдельную функцию, которая будет выводить сообщения на экран, считывать значения с клавиатуры и передать их в функцию IsEven(). Сама функция IsEven() будет выполнять только определение четности числа и возвращать результат не в консоль, а в качестве возвращаемого значения функции.

```
bool IsEven(int value)
{
  if (value % 2 == 0)
    return true;
  else
    return false;
}

void IsEvenConsole()
{
  int value;
  cout << "Enter a value: ";
  cin >> value;
  if (IsEven(value))
    cout << "Value is even";
  else
    cout << "Value is odd";
}
```

С одной стороны, может показаться, что теперь у нас стало больше функций и больше кода. Однако данное разделение позволяет гораздо шире использовать функцию определения четности. Например, теперь мы можем проверять на четность не только числа, введенные из консоли, но и внутренних значений переменных, значений из файлов и т.д.

```
int main()
{
  IsEvenConsole(); // Определяем четность числа, введенного пользователем
  …
  cout << IsEven(4); // Определяем четность числа, заданного программно
  cout << IsEven(5); // Определяем четность числа, заданного программно
  cout << IsEven(6); // Определяем четность числа, заданного программно
  …
  int value;
  fscanf("%d", value, inputFile);
  fprintf("%d", IsEven(value), outputFile; // Определяем четность числа,
                    //считанного из файла и результат записываем в файл
}
```

Фактически, мы расширили варианты использования разработанной функциональности, не ограничивая разработчика работой только в консоли. Подобное разделение даёт и другие преимущества, связанные с поддержкой кода. Например, возможность автоматизации тестирования с помощью юнит-тестов.

6.6 Юнит-тесты

Как инженер, обязанный гарантировать качество собственных решений, программист обязан уметь тестировать программный код. Задачей тестирования является ***контроль качества*** программного продукта. Результатом тестирования является информация, существуют ли ошибки в программном коде в настоящий момент и, если существуют, то в какой части программы. Определение места ошибки в программном, или ***локализация ошибки***, очень важно, так как мало знать, что в программе есть ошибка, но нужно эту ошибку и исправить. Таким образом, любое тестирование должно быть построено таким образом, чтобы максимально точно определить причину возникновения

ошибки. Юнит-тестирование – один из множества способов тестирования программного обеспечения, и его преимуществом является автоматизированность, т.е. с минимальным участием разработчика на этапе выполнения.

Юнит-тесты – тесты, направленные на тестирование отдельного элемента программы изолированно от остальных элементов программы. В случае структурного программирования элементом системы считается функция. Под изоляцией от остальных элементов подразумевается такое тестирование, при котором тестирование одной функции не будет зависеть от результатов работы других функций. В противном случае, юнит-тест не позволяет локализовать ошибку.

В предыдущем подразделе рассказывалось о разделении программы на Модель-Вид. Подобное представление программы в значительной степени упрощает написание юнит-тестов. Рассмотрим написание юнит-тестов на примере функции расчета корней квадратного уравнения из лабораторной работы №2. В лабораторной работе предлагается протестировать функцию на множестве исходных данных. Вводить все наборы тестовых данных каждый раз вручную может отнять значительное время. Однако можно автоматизировать тестирование функции на всех тестовых данных за счет юнит-тестирования:

```
//Тестируемая функция, возвращающая количество корней
//и рассчитывающая значения корней в переменные x1 и x2
int GetRoots(int a, int b, int c, double& x1, double& x2){…}

//Юнит-тест, на вход которого подаются исходные данные
//и ОЖИДАЕМЫЕ результаты выполнения теста
bool Test_GetRoots(int a, int b, int c, int rootsCountExpected, double x1Ex-
pected, double x2Expected)
{
   double x1Actual;
   double x2Actual;

   // Вызываем функцию и рассчитываем корни для исходных данных
   int rootsCountActual = GetRoots(a, b, c, &x1Actual, &x2Actual);

   // Если рассчитанное количество корней
   // отличается от ожидаемого количества корней,
   // то тест провален и надо вернуть false
   if (rootsCountActual != rootsCountExpected)
   {
      return false;
```

```
   }

   // Если корней должно быть два,
   // то нужно проверить каждый рассчитанный корень с ожидаемым значением
   if (rootsCountActual == 2)
   {
      return (x1Actual == x1Expected) && (x2Actual == x2Expected)
   }
   // Если корень должен быть только один,
   // то нужно проверить корень с ожидаемым значением
   else if (rootsCountActual == 1)
      return (x1Actual == x1Expected);
   // Если корней нет и не должно быть, то тест выполнен успешно
   else
   {
      return true;
   }
}

int main()
{
   // Вызываем юнит-тест с каждым тестовым набором данных
   // и со значениями корней, которые ожидаем получить
   cout << Test_GetRoots(1, 3, 2, 2, -2, -1) << endl;
   cout << Test_GetRoots(1, 4, 0, 2, -4,  0) << endl;
   cout << Test_GetRoots(0, 1, 2, 1, -2,  0) << endl;
   cout << Test_GetRoots(0, 0, 3, 0,  0,  0) << endl;
   cout << Test_GetRoots(0, 1, 0, 1,  0,  0) << endl;
   cout << Test_GetRoots(4, 1, 4, 0,  0,  0) << endl;
   // Если в результате выполнения будет хотя бы одно значение false,
   // значит, тест был провален и в функции есть ошибка.
   // Такое тестирование гораздо быстрее ручного ввода значений с клавиатуры!
   // Плюс при необходимости легко расширяется новыми тестовыми данными
}
```

Функции, выполняющие тестирование, называются ***тестами***. Наборы входных и выходных данных для теста называются ***тестовыми случаями*** или ***тест-кейсами***. Все тесты должны быть вынесены в отдельный файл, содержащий в название слово *Test*, это упростит навигацию по проекту. Для программных решений, состоящих из множества проектов, тесты выделяются в отдельные проекты.

Существуют специальные библиотеки и фреймворки для автоматизации тестирования и написания юнит-тестов, например, *CppUnit* и встроенные средства Visual Studio (пространство имен Microsoft::VisualStudio::CppUnitTestFramework). Они содержат дополнительные инструменты, позволяющие создавать более сложные варианты тестирования. Владение навыком написания

юнит-тестов и знание специальный фреймворков является обязательным для современного разработчика. Автор рекомендует читателям ознакомиться с задачей юнит-тестирования самостоятельно.

7 Задания к лабораторным работам

7.1 Задание к лабораторной работе №1

Название: основные элементы языка Си++.

Цель работы: изучить базовые конструкции языка Си++ и средства среды разработки Visual Studio, необходимые для реализации простейших алгоритмов.

Задание:

1. Создать решение *Practics* с консольным проектом *CppElements* в среде разработки *Visual Studio Express*. Работа в среде *Visual Studio Express* описана в п. 2.1.

2. Написать программу, выводящую на экран фамилию, имя, отчество и номер группы студента. Пример реализации представлен в п. 2.2.

3. Реализовать все примеры, представленные в п.2.3-2.9. Для выполнения примеров в рамках одного проекта и одной функции *main*(), рекомендуется использовать отдельные блоки кода для каждого примера. Такой подход визуально разделит примеры и разрешит потенциальные конфликты имен переменных.

4. Пример:

```
int _tmain()
{
      printf(“Student: Surname Name Patronymic - Group: 5XX”);
      //Типы данных и переменные
{
      …
}
      //Ввод/вывод данных
{
      …
}
…
      //Явное и неявное преобразование типов
{
      …
}


}
```

5. Разработать программу, запрашивающую у пользователя целочисленное значение и проверяющее, является ли введенное значение простым числом. Простое число, это число, которое делится без остатка только на само себя и на единицу. Таким образом, если пользователь ввел число *N*, программа должна проверить в цикле все числа от 2 до *N*-1, является ли хотя бы одно из них делителем без остатка. Алгоритм программы следующий:

6. Вывести на экран сообщение «Проверка простого числа. Введите число: ».

7. Считать значение переменной *n*.

8. Если значение отрицательное или равно нулю, сообщить пользователю об этом и запросить ввод числа еще раз.

9. Создать булеву переменную *isValuePrime* и присвоить в неё значение *true*. Эта переменная понадобится далее для запоминания, является ли введенное число простым.

10. Создать цикл по переменной *i* от 2 до n-1.

11. В цикле сделать проверку – если остаток от деления *n* на *i* **не равен** нулю, то присвоить в *isValuePrime* значение *false* и прервать выполнение цикла.

12. После цикла сделать проверку – если *isValuePrime* истинно, то вывести на экран текст «*n* является простым числом», иначе вывести на экран текст «*n* не является простым числом».

13. Удостовериться, что весь код лабораторной работы соответствует требованиям к оформлению кода, описанным в п.6.1 и 6.2.

14. Ответить на все вопросы для самопроверки, представленные в п. 2.2-2.9.

7.2 Задание к лабораторной работе №2

Название: функции, указатели и адресная арифметика

Цель работы: изучить конструкции языка Си++, связанные с декомпозицией программы на функции, а также рассмотреть вопросы работы с адресами памяти.

Задание:

1. Очистить функцию main в файле *program.cpp* от кода предыдущей лабораторной работы.

2. Ознакомиться с примерами в п.3.1 и ответить на вопросы для самопроверки. Для выполнения этой и всех последующих лабораторных **очень важно** понять работу с указателями и адресную арифметику. Уделите этой теме особое внимание!

3. Реализовать функцию расчета корней квадратных уравнений *int GetRoots*(*int a, int b, int c, double* x*1, *double* x*2). Функция возвращает количество рассчитанных корней – ноль, один или два -, а в переменные *x*1 и *x*2 помещает значения рассчитанных корней. Функцию протестировать на следующих входных данных:

Входные данные			**Выходные данные**	
a	***b***	***c***	***x*1**	***x*2**
1	3	2	-2	-1
1	4	0	-4	0
0	1	2	-2 (одно решение)	
0	0	3	Нет решений	
0	1	0	0 (одно решение)	
4	1	4	Нет решений	

Если хотя бы при одном из тестовых наборов программа возвращает иное значение, значит, программа работает неправильно и её необходимо исправить.

4. Реализовать функцию расчета корней квадратного уравнения *int GetRoots*(*int a, int b, int c, double*& *x*1, *double*& *x*2). Данная функция отличается от предыдущей тем, что корни передаются в функцию как ссылки, а не указатели. Функцию протестируйте на тех же тестовых наборах, что и предыдущую функцию.

5. Объяснить разницу в способах передачи переменных в функцию.

6. Объяснить механизм перегрузки функций на примере двух ранее созданных функций.

7. Написать рекурсивную функцию возведения целого числа в степень *int GetPower*(*int base*, *int power*).

8. Написать пример, демонстрирующий понятие глобальных переменных.

9. Написать функцию-игру «Угадай число» void *GuessNumber*(). Функция генерирует случайное число от 0 до 9, затем предлагает пользователю его угадать. У пользователя может быть неограниченное количество попыток, количество попыток подсчитывается функцией. Если пользователь угадал число, функция сообщает затраченное количество попыток и завершается. Можете модифицировать игру на своё усмотрение. Например, ограничив количество попыток до трёх или сообщая пользователю фразы «больше/меньше» при угадывании.

10. Для всех реализованных функций создать прототипы. Объяснить назначение прототипов.

11. Ответить на все вопросы для самопроверки в п.4.

12. Изучить механизм отладки программ, описанный в п.6.3 и рассмотреть его работу на примерах функций *GuessNumber*() и *GetPower*().

13. Изучить технику разделения программы на модель и вид, описанную в п.6.5, а также технику написания автоматизированных тестов, описанную в п.6.6.

7.3 Задание к лабораторной работе №3

Название: массивы и строки

Цель работы: изучить понятие массивов и строк в языке Си++ и работу с динамической памятью.

Задание:

1. Создать в проекте заголовочный файл *functions.h* и файл исходного кода *functions.cpp*, как это описано в п.6.4. Вынести в заголовочный файл все прототипы функций, написанных в предыдущей лабораторной работе, а в файл исходного кода их реализацию. Деление исходного кода на множество файлов облегчит навигацию в проекте и упростит разработку дальнейших лабораторных работ.

2. Создать аналогичные заголовочные файлы и файлы исходного кода *arrays.h, arrays.cpp, strings.h, strings.cpp*. Дальнейшие задания выполнять в соответствующих файлах исходного кода.

3. Изучить создание и работу с массивами и строками в п.3.2-3.3.

4. Реализовать функцию *void Sort*(*double* array, int arrayLength*), выполняющую сортировку входного массива вещественных чисел длиной *arrayLength*. Продемонстрировать работу функции на трёх массивах разных размеров, заданных случайными числами. Генерация случайных чисел описана в п.3.4. Демонстрация должна показать исходный массив, записанный в одну строку, и массив отсортированный.

5. Реализовать функцию перемножения двумерных матриц *bool MultiplyMatrices*(*int** matrixA, int aRows, int aCols, int** matrixB, int bRows, int bCols, int** resultMatrix*). В алгоритме обязательно должно быть учтено, что две матрицы могут быть перемножены только при условии *aCols = bRows*. Также размеры матриц не могут быть отрицательны или равны нулю. Функция возвращает истину, если умножение выполнено, и ложь, если умножение по какой-либо причине не удалось.

6. Продемонстрировать работу функции перемножения матриц: на квадратных матрицах 3×3, на прямоугольных матрицах 2×5 и 5×3, и матрицах со сторонами 3×4 и 3×5.

7. Объяснить, какими преимуществами и недостатками обладают способы передачи массивов в функцию с фиксированным размером и передачи массивов в функцию по указателю.

8. Реализовать функцию определения длины строки *int GetLength*(*char* string*). Если указатель равен *null*, функция должна вернуть значение -1.

9. Реализовать функцию определения индекса вхождения символа в строку *int IndexOf*(*char* string, char c*). Если такого символа нет в строке, то вернуть значение -1. Если символ встречается в строке более одного раза, вернуть индекс первого вхождения.

10. Реализовать функцию определения индекса последнего вхождения символа в строку *int LastIndexOf*(*char* string, char c*). Если такого символа нет в строке, то вернуть значение -1. Если символ встречается в строке более одного раза, вернуть индекс последнего вхождения.

11. Реализовать функцию получения подстроки из строки *bool GetSubstring*(*char* string, char* substring, int startIndex, int substringLength*). Функция получает исходную строку string и переписывает из неё *substringLength* символов в строку substring, начиная с символа *startIndex*. Если получить подстроку по какой-либо причине нельзя (отрицательные значения, пустые строки, недостаточно элементов в исходной строке), функция возвращает *false*. После формирования новой строки (в данном случае, подстроки) последним символом обязательно необходимо записать символ конца строки ‘\0’.

12. Реализовать три функции:

bool GetFilename(*char* fullFilename, char* filename*)

bool GetFileExtension(*char* fullFilename, char* fileExtension*)

bool GetFilepath(*char* fullFilename, char* filepath*)

Функции позволяют получить по полному имени файла (включая путь) отдельно имя файла, его расширение и путь соответственно и помещать их во

вторую входную строку через указатель. В случае отсутствия какой-либо из частей (пути или расширения), в результирующий указатель должно помещаться *null*. При реализации можно использовать ранее разработанные функции, что позволит уменьшить данные функции до пяти строк кода. Обратите внимание, что исходная и результирующая строки являются входными аргументами функции. Это подразумевает, что объявление этих переменных и выделение памяти под эти строки выполняется ***вне*** данных функций.

7.4 Задание к лабораторной работе №4

Название: пользовательские типы данных, структуры данных и динамическая память.

Цель работы: освоить механизмы языка Си++, предназначенных для создания пользовательских типов данных, таких как перечисления, структуры и объединения, изучить принципы работы с динамической памятью, а также стандартные структуры данных, используемых в программировании.

Задание:

1. Изучить теорию в п.5.1-5.5.

2. Создать перечисления цветов *Color* (порядка 5 цветов), дней недели *Weekday*, месяцев *Month*, пола человека *Sex* (воспользуйтесь словарем, чтобы написать дни недели, месяцы и пол по-английски правильно). Для каждого перечисления (а также для любого нового типа данных, который вы создадите в будущем) создайте отдельный заголовочный файл с названием по имени данного перечисления. Может показаться избыточным создание отдельного файла для каждого перечисления, занимающего 10 строк кода. Однако такой подход значительно облегчает навигацию по проекту и изучение сторонних проектов. Гораздо удобнее, когда все созданные типы данных можно посмотреть в дереве проекта, а не просматривать каждый файл исходного кода. Поэтому выделение новых типов данных в отдельные файлы является обязательной практикой среди разработчиков ПО.

3. Написать программу, которая запрашивает у пользователя ввести целое число от 1 до 365 – день года, а программа выводит соответствующий день недели, подразумевая, что первый день в году был понедельник. Программа должна быть реализована с использованием условного оператора *switch* и ранее созданного перечисления.

4. Реализовать аналогичную программу для определения месяца.

5. Реализовать программу, которая запрашивает у пользователя номер цвета. Перед запросом программа должна отобразить в качестве меню, какому числу соответствует тот или иной цвет. После ввода пользователем числа, программа должна запросить цвет еще раз, однако вывести меню и остальной текст тем цветом, который выбрал пользователь.

6. Создать в отдельном заголовочном файле структуру *Person*. Структура должна содержать поля имени, фамилии, возраста, пола.

7. Создать в функции *main*() несколько экземпляров структуры Person и инициализировать их поля конкретными значениями.

8. Вывести на экран одно из полей созданных персон, например, фамилию или возраст.

9. Попробовать изменить данное поле в структуре и снова вывести на экран.

10. Создать указатель на *Person* и поместите туда адрес одной из созданных персон. Выведите адрес, хранящийся в указателе, на экран.

11. Вывести на экран одно из полей созданной персоны, но сделайте это через указатель. Отметить для себя разницу в способах обращения к внутренним полям структуры через обычную переменную и через указатель.

12. Попробовать изменить значения полей персоны через указатель.

13. Создать функции *void PrintPerson*(*Person*& *person*) и *ReadPerson*(*Person* person*), выводящие человека на экран и считывающие человека с клавиатуры соответственно. Для дальнейшего удобства восприятия данных с экрана, вывод данных о человеке должен выполняться в одну строку, например:

```
John McClane      Age: 35     Sex: Male
```

14. Поместить в указатель новую персону, но созданную в динамической области памяти с помощью ключевого слова *new*. Считать данные о человеке в эту переменную с клавиатуры, и затем вывести их на экран. Затем освободить память созданной переменной с помощью ключевого слова *delete*.

15. Объявить ***массив указателей*** на *Person* из пяти элементов. Далее в цикле в каждый элемент массива создать динамически новую персону и считать её с клавиатуры. Вторым циклом вывести на экран данные из массива в формате «{индекс элемента}.{данные о персоне}». Третьим циклом освободить память для каждого элемента массива.

16. Подключить стандартную библиотеку односвязного списка *vector*. Создать экземпляр списка целых чисел *vector<int>*. Поместить программно в список пять целых чисел, удалить третий элемент списка, добавить на вторую позицию новое целое число. Вывести все элементы списка на экран и убедиться, что удаление и добавление нового элемента выполнилось корректно.

17. В качестве необязательного задания попробовать создать список персон и поработать с ним аналогичным путем. Также попробовать создать экземпляры стандартных словарей и других структур данных. В реальной практике разработчиков очень часто приходиться работать с вложенными структурами данных, например, список массивов указателей, словарь списков или список списков словарей, и это не самые сложные или хитрые примеры. Таким образом, получение дополнительного опыта работы с подобными структурами будет для начинающих программистов полезным.

7.5 Указания к составлению отчетов

Типовое содержание отчета по лабораторной работе:

1. **Титульный лист**.
2. **Содержание**.

3. **Цель работы:** четко сформулировать объект изучения. Объектом изучения могут выступать элементы языка Си++, механики и техники написания кода.

4. **Постановка задачи:** привести текст выполняемых задач так, как он представлен в пособии.

5. **Блок-схема:** в случае, когда задачей лабораторной работы является придумать алгоритм, в отчете должна быть представлена блок-схема разработанного алгоритма.

6. **Исходный код:** в случае, когда задачей лабораторной работы является освоение конкретных элементов языка Си++, в отчете должен быть представлен исходный код, демонстрирующий данный элемент.

7. **Вывод:** краткий текст в виде одного-двух абзацев, доказывающий, была ли достигнута первоначальная цель работы, какие новые понятия и техники были изучены в ходе работы и для чего нужны данные элементы языка Си++.

При проверке отчета преподавателем особое внимание уделяется правильности оформления. Отчет должен соответствовать стандарту оформления вуза [8] (можно найти на сайте кафедры или университета). Обращайте внимание на размеры шрифтов и межстрочных интервалов, размеры полей и абзацных отступов, указания номеров страниц и т.д. В качестве примера оформления титульного листа, содержания, текстового параграфа и рисунков-графиков использовать приложения В, Д, И, К, Л [8].

У многих студентов возникает мнение, что написание отчетов по лабораторным работам является бессмысленным занятием, а сами отчеты никому не нужны. Так и есть. В подавляющем большинстве случаев вся эта макулатура уходит на черновики через пять минут после её сдачи преподавателю. Преподавателю отчеты и не нужны, так как отчеты не являются формой проверки знаний студентов. Однако они хорошо подходят для наказания студентов, плохо справляющихся с выполнением лабораторных. Так или иначе, оформление отчетов вырабатывает навыки написания технической документации.

Если вы действительно планируете связать свою жизнь с программированием или инженерией, работа с технической документацией, в том числе и её создание, станет неотъемлемой частью избранной вами профессии.

8 Заключение

В пособии рассмотрены базовые элементы языка Си++, позволяющие разрабатывать простейшие алгоритмы и программы, соответствующие процедурной и структурной парадигмам программирования. Их успешное понимание является основой для дальнейшего развития навыков разработки программного обеспечения. В частности, важными навыками являются умение вести проект, соблюдать требования к оформлению исходного кода, декомпозировать программный код на составные логические модули, выполнять тестирование разрабатываемой функциональности.

Однако, на практике от современных разработчиков требуется владение объектно-ориентированной парадигмой. Именно данные подходы в проектировании и разработке программного обеспечения позволяют создавать действительно сложные решения, исходный код которых исчисляется сотнями тысяч строк кода и более. По этому следующим направлением развития навыков программирования для студентов предлагается выбрать именно объектно-ориентированное проектирование на языке Си++ или другом Си-подобном языке.

Список рекомендуемой литературы

1. Подбельский В.В. Язык Си++: учеб. пособие. – 5-е изд. – М.: Финансы и статистика, 2003. – 560 с.
2. Лафоре Р. Объектно-ориентированное программирование в С++. – 4-е изд. – С.-П.: Питер, 2004. – 924 с.
3. Шилдт Г. С++: базовый курс. - 3-е изд., пер. с англ. – М.: Издательский дом «Вильямс», 2010. – 624 с.
4. Вирт Н. Алгоритмы и структуры данных. – Пер. с англ. Ткачев Ф.В. – С.-П.: ДМК Пресс, 2010. – 274 с.
5. Страуструп Б. Язык программирования С++. – М.: Бином, 2011. – 1136 с.
6. Visual Studio: офиц. страница [Электронный ресурс]. – Режим доступа: https://www.visualstudio.com/ - (Дата обращения: 15.11.2015)
7. Microsoft Developer Network: форум поддержки разработчиков Microsoft MSDN [Электронный ресурс]. – Режим доступа: https://www.social.msdn.microsoft.com/ - (Дата обращения: 15.11.2015)
8. Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления: образовательный стандарт вуза ТУСУР 01-2013. – Томск: ТУСУР, 2013. – 57 с.
9. Russian Software Developer Network: Соглашения по оформлению кода команды RSDN [Электронный ресурс]. - http://rsdn.ru/article/mag/200401/codestyle.xml/ - (Дата обращения: 15.11.2015)
10. Макконнелл С. Совершенный код / С. Макконнелл, пер. с англ. под ред. В.Г. Вшивцева. – С.-П.: Питер, 2005. – 896с.

Приложение А.
Защита от неправильного ввода данных пользователем

При разработке программного обеспечения важно учитывать, что пользоваться программой будут обычные люди. Человек склонен совершать ошибки, например, при вводе данных в приложение, и это необходимо учитывать в пользовательских интерфейсах. Чтобы защитить программу от пользователя следует выполнять следующие шаги:

1. При запросе данных от пользователя всегда сообщайте что пользователь должен ввести, для чего и в каком формате.
2. Постарайтесь исключить неправильный ввод данных программно. Например, при вводе пользователем номера телефона позвольте пользователю вводить с клавиатуры только цифры, игнорируя нажатия клавиш букв и других символов.
3. Если исключить неправильный ввод данных невозможно или неоправданно сложно, реализуйте проверку введенных данных и, в случае ошибки, сообщите пользователю, в чем заключается ошибка, и запросите повторный ввод.

Далее представлен примеры функций, считывающей с клавиатуры целые или вещественные числа, запрещая пользователю вводить иные символы:

protectedInput.cpp

```
//Набор функций ввода стандартных типов данных с защитой от неправильных
данных пользователя
//Файл добавляется в любой проект в папку Headers Files и подключается че-
рез директиву include к конкретному файлу исходного кода,
//либо подключается в файле stdafx.h для доступности во всем проекте
#pragma once

#include <stdio.h> //Библиотека стандартного потока ввода/вывода (подключа-
ется для printf, scanf)
#include <conio.h> //Библиотека ввода/вывода в консоли (подключается для
getch, putch)
#include <stdlib.h> //Библиотека стандратных функций (подключается для
atof)

//Получить целое число из ввода консоли. Функция запрещает ввод любых сим-
волов, кроме цифр, знака минус
int GetInt()
```

```
{
	int inputedKey; //Последний считанный с клавиатуры символ
	int value = 0; //Считанное число, формируемое в процессе ввода с кла-
виатуры
	bool isMinusInput = false; //Флаг, был ли введен знак минус (является
ли число отрицательным)
	int inputedKeyIndex = 0; //Индекс текущего введенного символа (важен
для реализации backspace)
	int valueOrder = 0; //Порядок числа (может отличаться от индекса в
случае ввода знака минус)

	//Создаём "бесконечный" цикл
	while (true)
	{
		//Считываем символ с клавиатуры без отображения в консоли
		inputedKey = _getch();
		//Если был введен минус (и если он был введен только первым
символом), то запомнить его ввод и отобразить на экране
		if ((inputedKey == '-') && (inputedKeyIndex == 0) && (!is-
MinusInput))
		{
			isMinusInput = true;
			_putch('-');
			inputedKeyIndex++;
		}

		//Если введена цифра, то рассчитать новое целочисленное значе-
ние с учетом позиции введеной цифры
		if ((inputedKey >= '0') && (inputedKey <= '9'))
		{
			printf("%c", inputedKey);
			value = value * 10 + inputedKey - '0';
			inputedKeyIndex++;
			valueOrder++;
		}

		//Если был нажат Backspace, то стираем последний символ, а ра-
нее полученное число делим без остатка на 10 (потеряли один порядок)
		if (inputedKey == 8)
		{
			_putch(8);
			_putch(' ');
			_putch(8);
			value = value / 10;
			valueOrder--;
			if (valueOrder < 0)
			{
				valueOrder = 0;
			}
			inputedKeyIndex--;
			if (inputedKeyIndex == 0)
			{
				isMinusInput = false;
			}
		}
```

```
            //Если нажали Enter и есть хоть какое-то введенное число, то
завершить бесконечный цикл
            if ((inputedKey == 13) && (inputedKeyIndex != 0) && (valueOrder
!= 0))
            {
                  break;
            }
      }

      if (isMinusInput)
      {
            value = -value;
      }
      return value;
}

//Получить целое число из ввода консоли. Функция запрещает ввод любых сим-
волов, кроме цифр, знака минус
float GetFloat()
{
      //DESCRIPTION: Здесь реализован иной подход, нежели в GetInt(). Если
в GetInt() считывание происходит непосредственно
      //в целочисленную переменную, то здесь сначала формируется массив
символов с числом, затем массив символов
      //преобразуется в число с помощью стандартных функций
      //1) Считываем с клавиатуры введенный символ
      //2) игнорируем на ввод любые символы, кроме цифр, одной точки и од-
ного знака минус на первой позиции
      //3) Если символ правильный, запоминаем его в массив символов
      //4) Когда пользователь закончил ввод, преобразуем массив символов в
число, используя стандартную функцию

      //TODO: Реализовать ввод экспоненциальной части
      //TODO: Учесть, что экспоненциальная часть может быть введена как 'e'
и 'E'
      //TODO: Учесть, что экспоненциальная часть тоже может содержать знак
плюс или минус
      //TODO: Учесть, что точка не может быть в экспоненциальной части
      //TODO: Учесть, что перед экспоненциальной частью обязательно должно
быть число
      //TODO: Учесть, что после 'e' обязательно должно быть число
      //TODO: При реализации экспоненциальной части сделать более грамотный
расчет максимальной длины массива символов
      //TODO: Правильно выстроить систему условий, сделать код лаконичнее

      const int valueMaxLength = 10;
      char inputedKey; //Последний считанный с клавиатуры символ
      int inputedKeyIndex = 0; //Индекс текущего введенного символа (важен
для реализации backspace)
      char valueAsString[valueMaxLength];
      bool isMinusInput = false; //Флаг, был ли введен знак минус
      bool isCommaInput = false; //Флаг, был ли введен знак разделителя
дробной части
      //Создаём "бесконечный" цикл
```

```
      while (true)
      {
            //Считываем символ с клавиатуры без отображения в консоли
            inputedKey = _getch();

            //Если был введен минус (и если он был введен только первым
символом), то запомнить его ввод и отобразить на экране
            if ((inputedKey == '-') && (inputedKeyIndex == 0) && (!is-
MinusInput))
            {
                  if ((inputedKeyIndex >= (valueMaxLength - 1)))
                  {
                        continue;
                  }
                  isMinusInput = true;
                  _putch('-');
                  valueAsString[inputedKeyIndex] = '-';
                  inputedKeyIndex++;
            }

            if ((inputedKey == '.') && (!isCommaInput))
            {
                  if ((inputedKeyIndex >= (valueMaxLength - 1)))
                  {
                        continue;
                  }
                  isCommaInput = true;
                  _putch('.');
                  valueAsString[inputedKeyIndex] = '.';
                  inputedKeyIndex++;
            }

            //Если введена цифра, то рассчитать новое целочисленное значе-
ние с учетом позиции введеной цифры
            if ((inputedKey >= '0') && (inputedKey <= '9'))
            {
                  if ((inputedKeyIndex >= (valueMaxLength - 1)))
                  {
                        continue;
                  }
                  printf("%c", inputedKey);
                  valueAsString[inputedKeyIndex] = inputedKey;
                  inputedKeyIndex++;
            }

            //Если был нажат Backspace, то стираем последний символ
            if (inputedKey == 8)
            {
                  _putch(8);
                  _putch(' ');
                  _putch(8);
                  valueAsString[inputedKeyIndex] = '\0';
                  inputedKeyIndex--;
                  if (inputedKeyIndex == 0)
                  {
```

```
                        isMinusInput = false;
                }
                if (valueAsString[inputedKeyIndex] == '.')
                {
                        isCommaInput = false;
                }
        }

        //Если нажали Enter и есть хоть какое-то введенное число, то
завершить бесконечный цикл
        if ((inputedKey == 13) && (inputedKeyIndex != 0))
        {
                break;
        }
    }
    valueAsString[inputedKeyIndex++] = '\0';
    printf("\n%s\n", valueAsString);

    return atof(valueAsString);
}
```

Представленные функции имеют ряд недостатков, они перечислены в комментариях с меткой *TODO*. Однако они дают наглядный пример того, как можно выполнить проверку вводимых пользователем данных. Функции могут быть использованы при выполнении лабораторных работ.