Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

А. Б. Гомбоин
Е. В. Рогожников

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Методические указания для выполнения лабораторных работ для студентов направления 11.03.02 «Инфокоммуникационные технологии и системы связи»

Томск
2021

УДК 004.43
ББК 32.973.2
Г 64

Рецензент:
Абенов Р.Р., доцент кафедры телекоммуникаций и основ радиотехники ТУСУРа, канд. техн. наук

Гомбоин, Александр Булатович
Г 64 Языки программирования: Методические указания для выполнения лабораторных работ для студентов направления 11.03.02 «Инфокоммуникационные технологии и системы связи» / А. Б. Гомбоин, Е. В. Рогожников. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2021. – 48 с.

Настоящее методическое пособие для студентов направления 11.03.02 «Инфокоммуникационные технологии и системы связи» посвящено изучению языков программирования в программной среде Qt Creator 5.0. В нём представлены указания для выполнения лабораторных работ, позволяющих углубленно изучить принцип работы языков программирования. В методическом пособии представлены как основы структурного программирования, так и работа с объектно-ориентированным программированием.

Одобрено на заседании кафедры ТОР, протокол № 1 от 31 августа 2021 г.

УДК 004.43
ББК 32.973.2

Оглавление

ВВЕДЕНИЕ

В наше время широко распространены различные языки программирования, с их помощью создано всё разнообразие программного обеспечения для компьютера, мобильного телефона, Интернета вещей и т.д. Сейчас уже создано более восьми тысяч различных языков программирования и данное число продолжает расти. Некоторыми языками умеют владеть лишь ограниченное количество людей, являющимися разработчиками данного языка, другие языки становятся известны миллионам людей, такими языками являются C/C++, Java, JavaScript, Python и т.д. Для того, чтобы охватить большее количество языков программирования было решено выбрать для данного пособия язык C++, так как многие языки основываются на языке C и имеют похожий синтаксис и структуру написания кода.

В данных лабораторных работах представленного методического пособия, приведены примеры, позволяющие углубленно изучить принцип работы языков программирования. В методическом пособии представлены как структурное программирование, так и основы объектно-ориентированного программирования.

Методическое пособие начинается с ознакомления с графическим интерфейсом программы Qt Creator, где в дальнейшем будут писаться программы лабораторных заданий.

ЛАБОРАТОРНАЯ РАБОТА №1
Создание консольного приложения в Qt Creator 5.0.0 Community. Ввод-вывод данных. Переменные и типы данных

Цель работы: ознакомиться с созданием проекта в Qt Creator 5.0.0 и изучить его структуру, основные функции ввода и вывода данных в C++ и создать простую программу на языке C++. Изучить переменные и типы данных в C++.

1.1 Создание консольного приложения в Qt Creator 5.0.0 Community. Ввод-вывод данных

1) Запустите Qt Creator. Для этого на панели задач откройте меню «Пуск», пролистайте до папки Qt, запустите Qt Creator 5.0.0 (Community), рисунок 1.1.1.

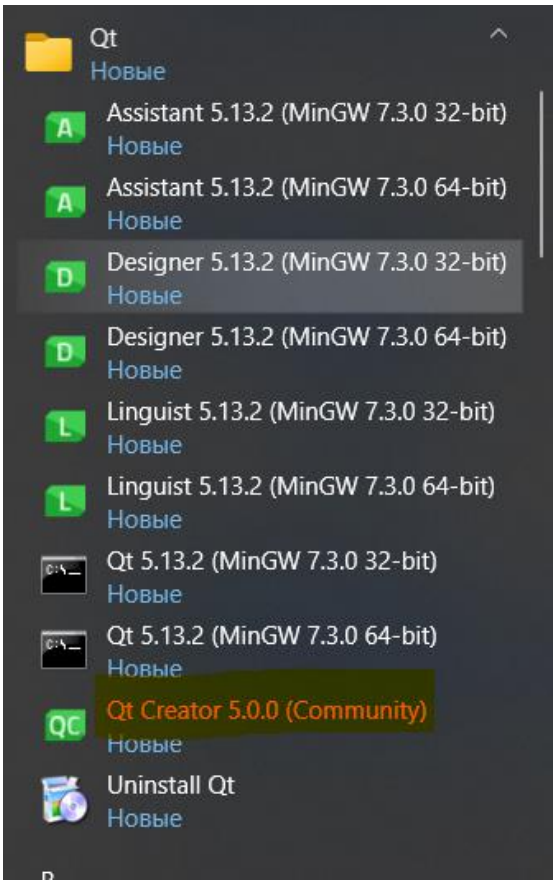

Рисунок 1.1.1 – Папка Qt

На рисунке 1.1.2 представлено основное окно фреймворка.

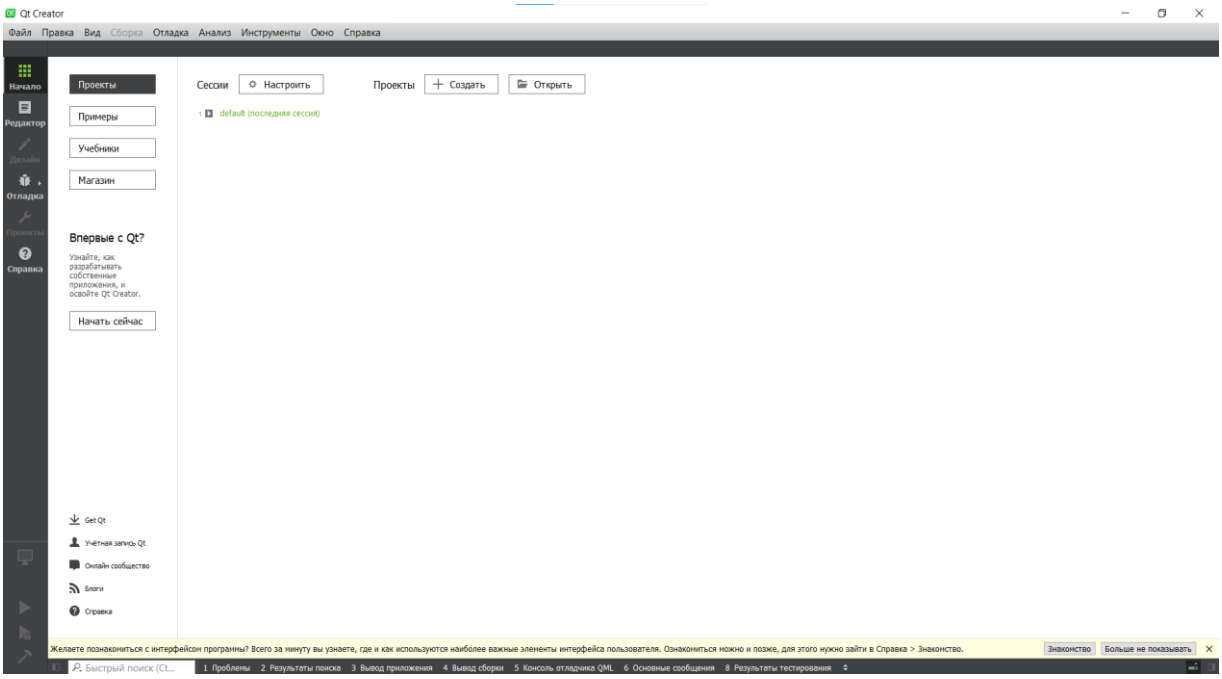

Рисунок 1.1.2 – Основное окно фреймворка

В основном окне вам необходимо выбрать пункт «Проекты», как показано на рисунке 1.1.3.

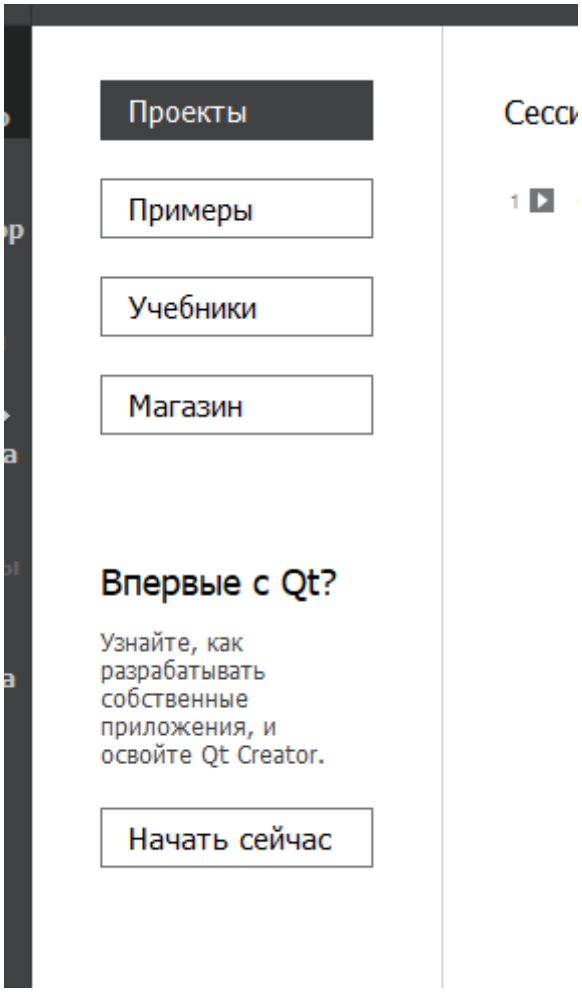

Рисунок 1.1.3 – Пункт «Проекты»

Выберите в появившемся меню «Создать», рисунок 1.1.4.

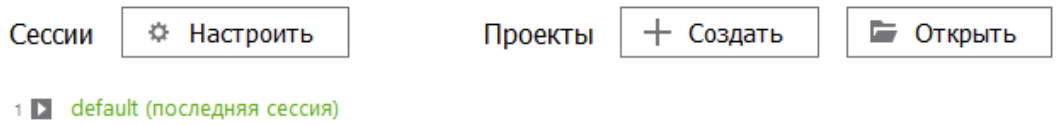

Рисунок 1.1.4 – Создание проекта

Далее откроется окно для выбора типа проекта, в списке «Проекты» выберите «Проект без Qt», в следующем окошке выберите «Приложение на языке C++» и нажмите на кнопку «Выбрать…», как показано на рисунке 1.1.5.

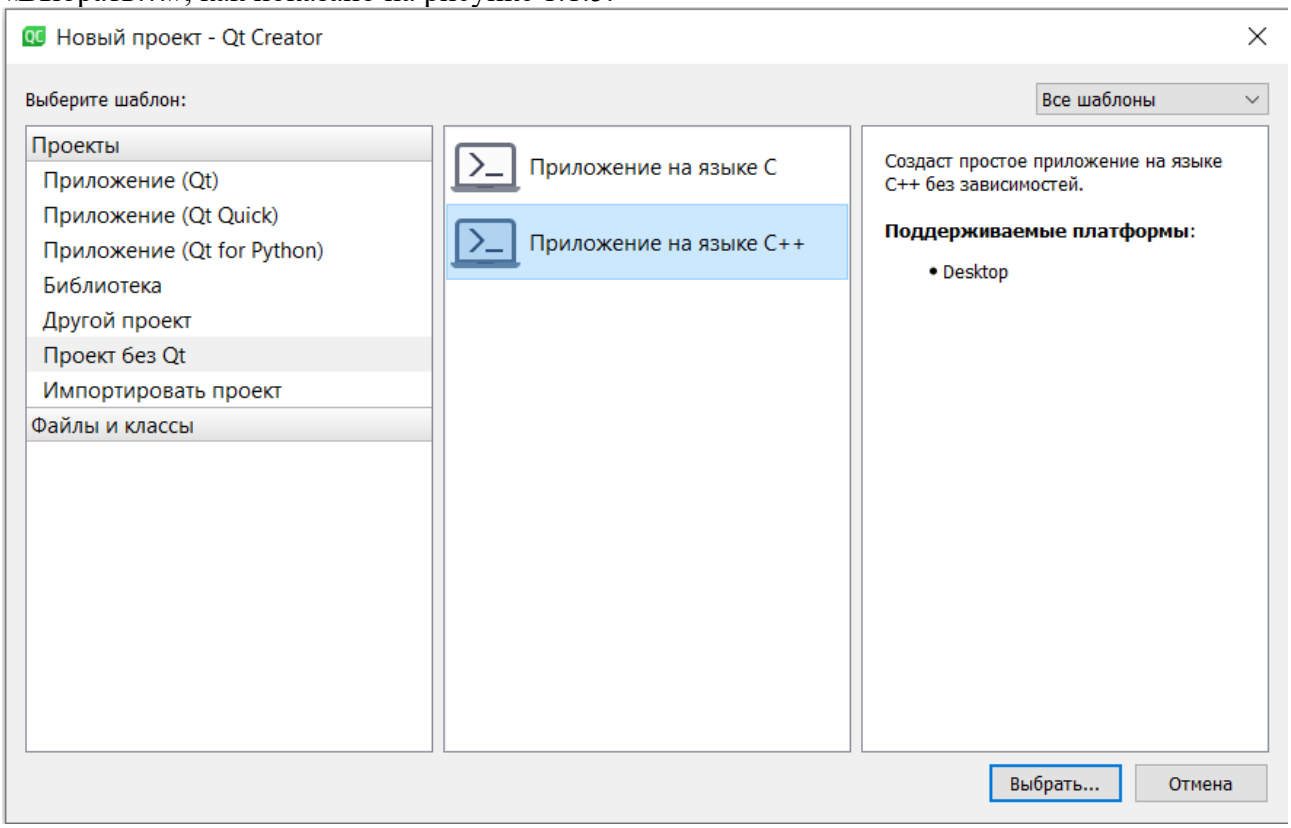

Рисунок 1.1.5 – Выбор проекта и языка

Далее откроется окно с выбором названия проекта и его размещения на компьютере, рисунок 1.1.6. После того, как вы назвали проект и выбрали его размещение, нажмите на кнопку «Далее».

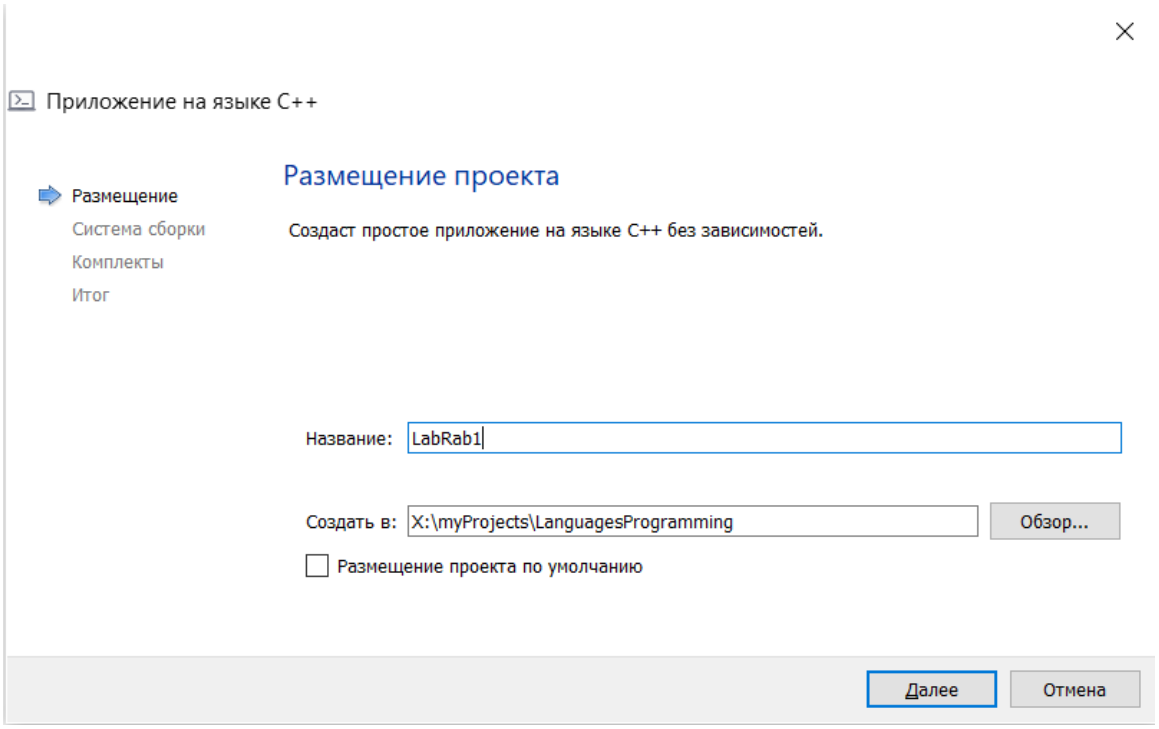

Рисунок 1.1.6 – Размещение и название проект

Далее откроется окно с выбором системы сборки. Необходимо выбрать «qmake», рисунок 1.1.7.

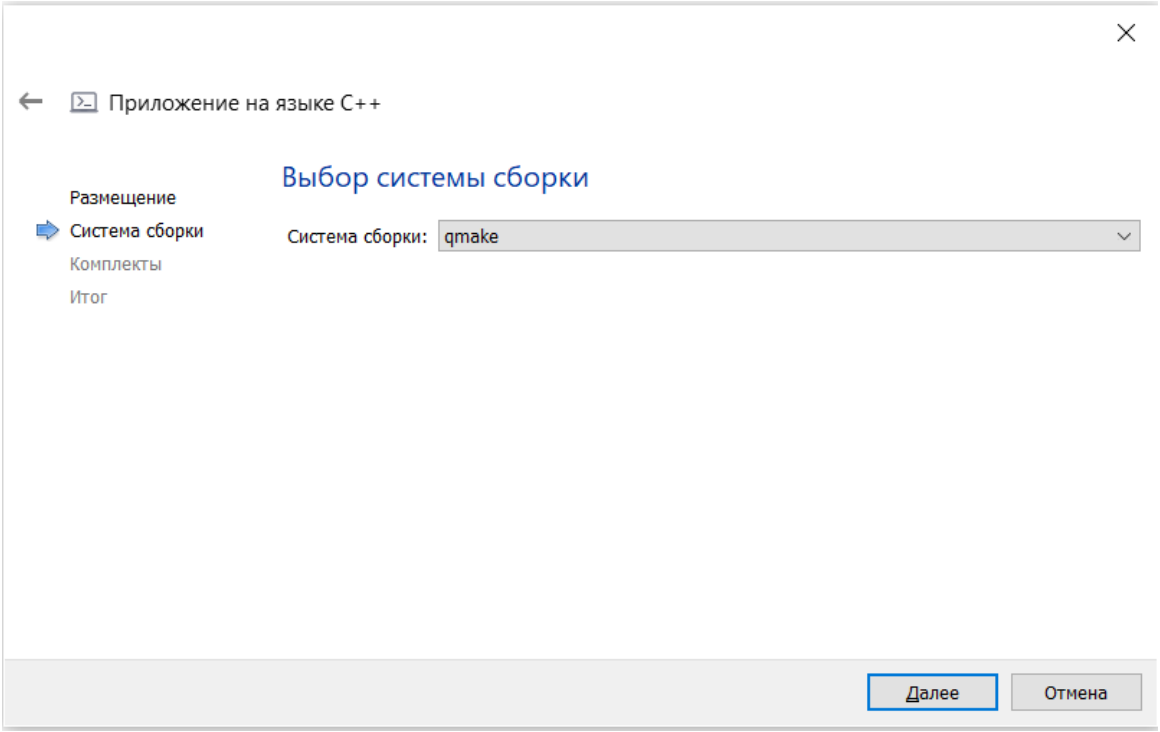

Рисунок 1.1.7 – Система сборки

Далее необходимо выбрать комплект, нужно выбрать комплект «Desktop Qt 5.13.2 MinGW 64-bit», смотрите рисунок 1.1.8, здесь необходимо учесть вашу текущую версию Qt и установленного компилятора. Компилятор на вашей машине может отличатся от компилятора, приведенного в данном методическом пособии.

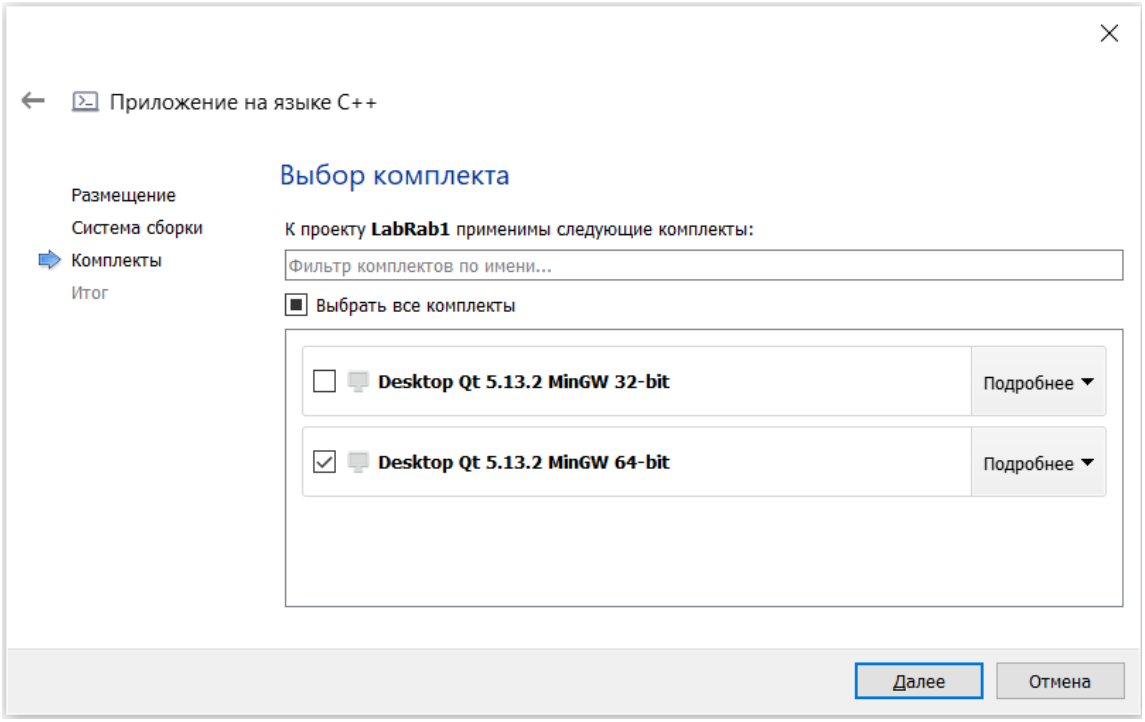

Рисунок 1.1.8 – Выбор компилятора

Также нужно настроить проект, для этого нажмите на кнопку «Подробнее» и нажмите на появившуюся кнопку «Управление…», рисунок 1.1.9. Далее перейдите в раздел «Текстовый редактор», в появившемся окне справа откройте вкладку «Поведение» и в разделе «Кодировка файлов» по умолчанию поставьте «Windows-1251/ CP-1251», смотрите рисунок 1.1.10. Это нужно для русской локализации ввода/вывода данных в консоль.

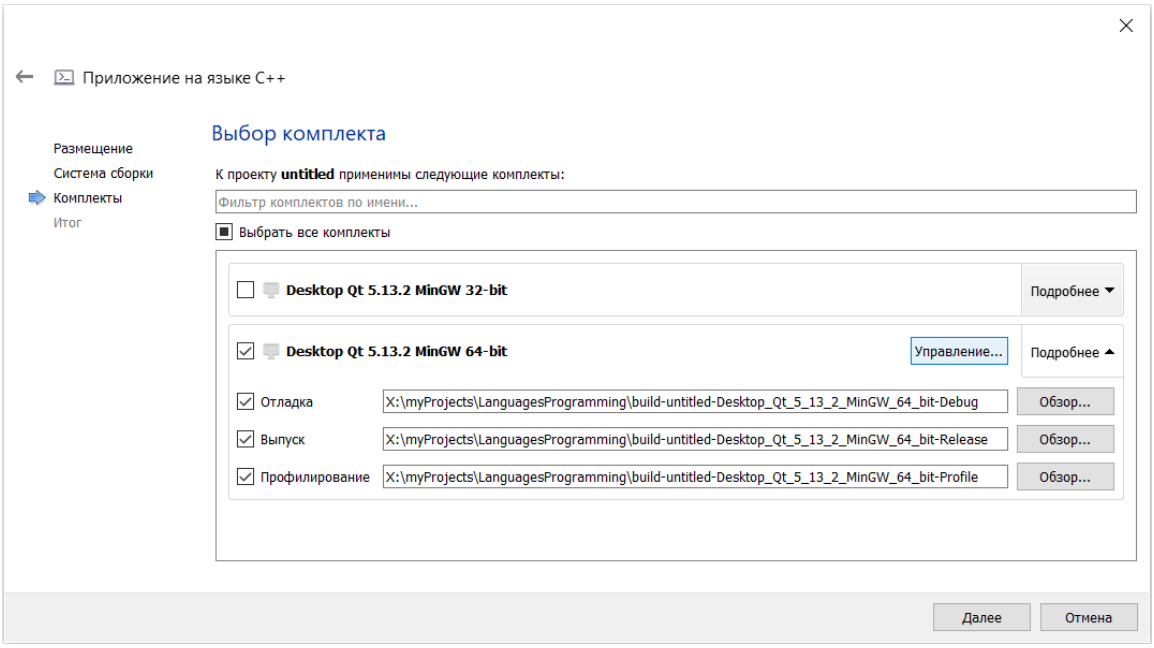

Рисунок 1.1.9 – Дополнительная настройка проекта

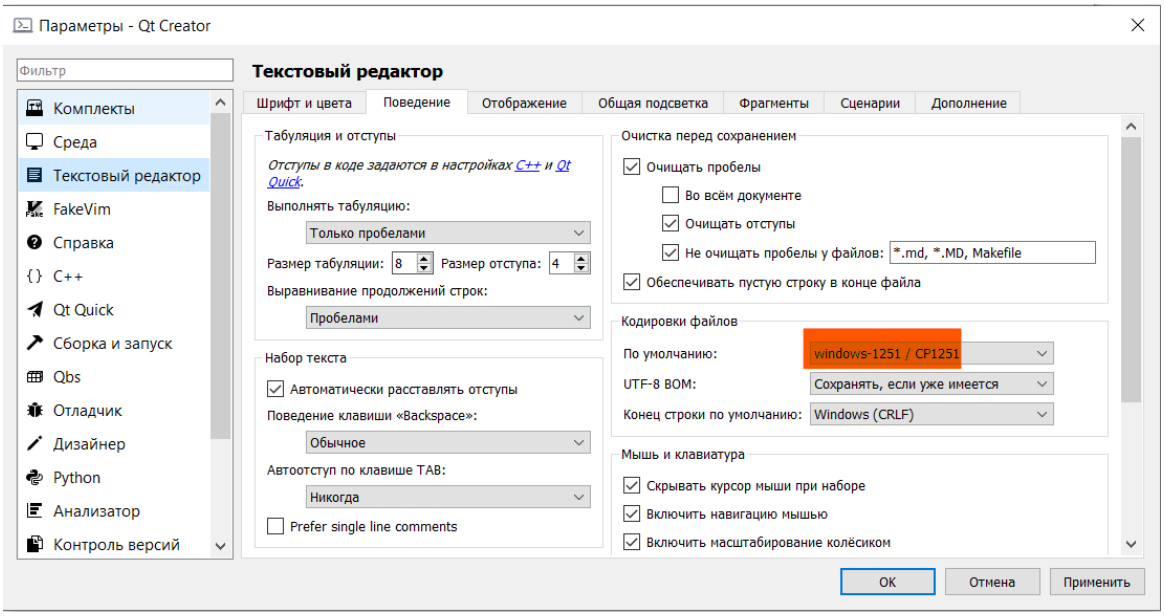

Рисунок 1.1.10 – Параметры проекта

После выполнения всех вышеописанных действий Qt создаст шаблон простейшего консольного приложения:

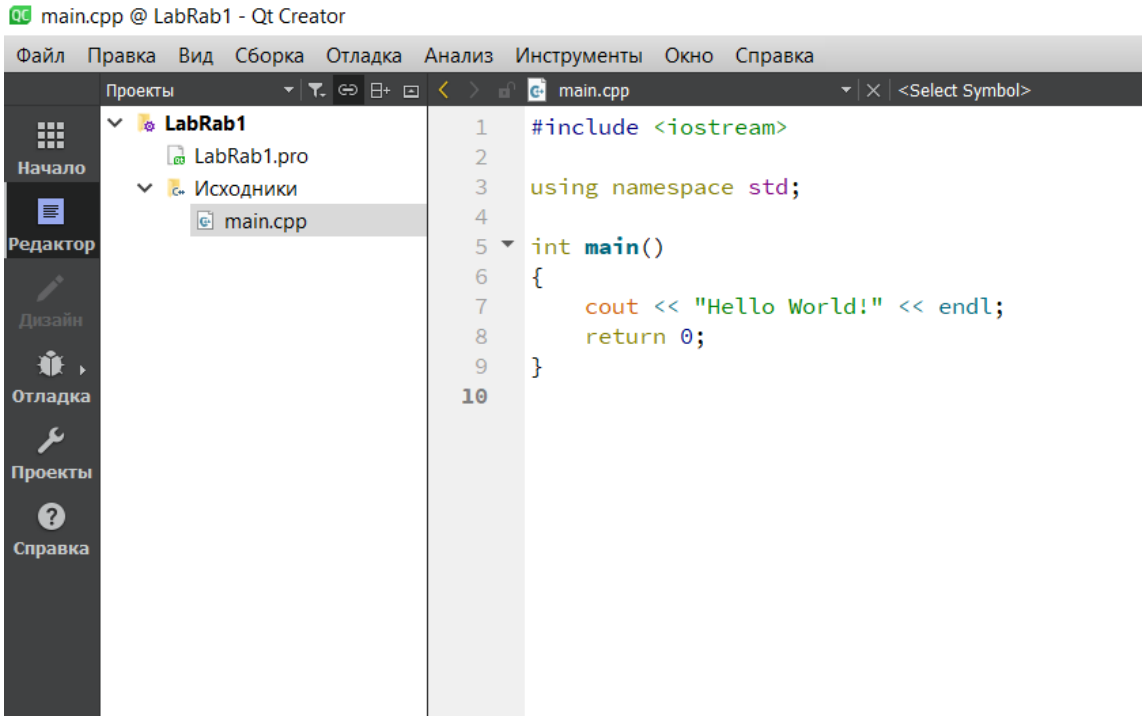

Рисунок 1.1.11 – Шаблон консольного приложения

2) Структура программы на языке C++ следующая:

1. #include <iostream> - подключение заголовочного файла iostream с классами, функциями и переменными для организации ввода-вывода.
2. using namespace std; - подключение пространства имён std.
3. int main() – имя функции. Любая программа на языке C++ состоит из одной или нескольких функций. В написанном шаблоне функция одна – main( ). Функция с именем main обязательно должна быть в любой исполняемой программе.
4. { – начало тела функции.
5. cout – объект для вывода данных на экран.

6. оператор return с аргументом 0 – завершение функции main с кодом 0.

7. } – конец функции main.

3) Вывод данных в C++. Для вывода информации на экран C++ предоставляет множество возможностей. Есть функции, выводящие на экран только строки, только целые или вещественные числа. Функция printf и объект cout могут использоваться для вывода на экран информации любого типа.

Описание функции:

printf(Управляющая строка, <аргумент1, аргумент2, …>);

Управляющая строка записывается в двойных кавычках и содержит информацию двух типов:

- печатаемые символы (константная строка);
- идентификаторы данных (спецификаторы формата)

Функция принимает список аргументов и применяет к каждому спецификатор формата. Количество спецификаторов формата и аргументов должно быть одинаковым.

Основные спецификаторы формата –

%d – целое десятичное число;

%c – один символ;

%s – строка символов;

%e – экспоненциальная запись числа с плавающей точкой;

%f – десятичная запись числа с плавающей точкой;

%u - десятичное число без знака;

%o - целое восьмеричное число без знака;

%x - целое шестнадцатеричное число без знака

Помимо этого, в спецификаторах используются модификаторы, форматирующие выводимую информацию. Рассмотрим применение модификаторов на спецификаторе %f. Аналогично форматируется информация других типов.

Модификатор состоит из двух чисел, разделенных точкой и может иметь лидирующий знак «-». Записывается модификатор после знака «%», обозначающего начало спецификатора. В общем виде модификатор выглядит следующим образом: %m.n спецификатор. Первое число m задает ширину поля вывода для всего значения. Второе число n используется для форматируемого вывода чисел с плавающей точкой и задает количество дробной части числа, выводимых на экран. Отсутствие знака «-» говорит о том, что вывод будет отформатирован по правой границе поля вывода, присутствие – форматирование по левой границе поля вывода.

При записи спецификатора в следующем виде - %10.4f – все выводимое вещественное число запишется в поле из десяти символов. Дробная часть числа будет состоять из 4 знаков.

Описание объекта:

Объект cout используется вместе с оператором вставки << для отображения потока символов. К примеру: «cout << value» или «cout << “String”». Оператор вставки может использоваться множество раз с комбинацией переменных, строк и манипуляторов (endl).

Объект cout можно использовать с другими функциями-членами, к примеру, put(), write() и т.д. Часто используемые функции:

1. cout.put (char value) – отображает символ, сохраненный в value.
2. cout.write (char * str, int n) – отображает первые n символов, прочитанных из str.
3. cout.setf (option) – устанавливает данную опцию. Обычно используются следующие варианты: левый, правый, фиксированный и т. д.
4. cout.unsetf (option) – отменяет данную опцию.
5. cout.precision (int n) – устанавливает десятичную точность равной n при отображении значений с плавающей запятой. То же, что cout << setprecision (n).

4) Ввод данных. Для ввода информации в C++ есть множество функций, но мы рассмотрим функцию (функцию работающую с разнотипными данными) scanf и объект cin.

Описание функции: scanf(спецификатор формата, указатель на переменную).

В функции используются те же спецификаторы формата, что и в функции printf.

Обратите внимание. Имя массива является указателем, поэтому при вводе строк перед именем строки не пишется знак &. При вводе строки с помощью функции scanf строка вводится до первого встреченного пробела. Вся остальная часть строки обрезается.

Описание объекта cin.

Объект cin используется вместе с оператором извлечения (>>) для получения потока символов. Например: «cin >> value;». Оператор извлечения также может использоваться неограниченное количество раз, чтобы принимать несколько входных данных, например: «cin >> var1 >> var2 >> … >> varN;».

Объект cin можно использовать с другими функциями-членами, такими как getline(), read() и т. д. Ниже приведены наиболее часто используемые функции-члены:

1. cin.get (char ch) – читает введенный символ и сохраняет его в ch.
2. cin.getline (char * buffer, int length) – считывает поток символов в строковый буфер, останавливается, когда он прочитал символы длиной 1 или когда он находит символ конца строки (‘\ n’) или конец файла.
3. cin.read (char * buffer, int n) – читает n байтов (или до конца файла) из потока в буфер.
4. cin.ignore (int n) – игнорирует следующие n символов из входного потока.
5. cin.eof() – возвращает ненулевое значение, если достигнут конец файла (eof).

Задание 1.

Необходимо вывести с помощью объектов cin и cout строку в консоли «Привет Мир “Ваше имя”!!!». Примечание: для работы русской локализации необходимо добавить следующие строки:

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


using namespace std;

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
```

Для того чтобы вывести ваше имя, необходимо создать переменную типа String и в неё сначала записать ваше имя.

Также вывести данную строку «Привет Мир “Ваше имя”!!!» уже с помощью функций scanf и printf. Для записи вашего имени в scanf использовать массив char.

Для компиляции вашей программы достаточно нажать на зеленую кнопку в виде стрелки, рисунок 1.1.11.

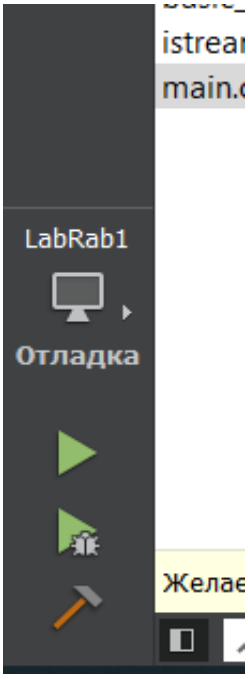

Рисунок 1.1.12 – Кнопка компиляции

1.2 Переменные и типы данных

Переменная (от англ. variable) – поименованная или адресуемая иным способом область памяти, которую можно использовать для доступа к данным.

Тип данных – атрибут, определяющий, какого рода данные могут храниться в объекте: целые числа, символы, данные денежного типа, метки времени и даты, двоичные строки и так далее.

Типы данных в C++

Для представления целых величин в C++ предусмотрены следующие типы данных:

Тип char. Занимает в памяти 1 байт. Используется для представления символов и целых чисел от 0 до 255 (-128 до 127).

Тип int. Занимает в памяти 4 байта. Используется для представления целых чисел в диапазоне -2 147 483 648 до 2 147 483 647.

Тип float. Занимает в памяти 4 байта. Используется для представления чисел с плавающей точкой. от $3.4*10^{-38}$ до $3.4*10^{38}$. Точность вычислений до 7 знаков после запятой.

Тип double. Занимает в памяти 8 байт. Используется для представления чисел с плавающей точкой. от $1.7*10^{-308}$ до $1.7*10^{308}$. Точность вычислений до 15 знаков после запятой.

Тип void – пустой тип. Используется для описания функций

Тип bool. Используется исключительно для хранения результатов логических выражений. У логического выражения может быть один из двух результатов true или false. True – если логическое выражение истинно, false – если логическое выражение ложно.

Полный перечень всех типов данных и их диапазонов можно увидеть на рисунке 1.2.1.

Тип	байт	Диапазон принимаемых значений
целочисленный (логический) тип данных		
bool	1	0 / 255
целочисленный (символьный) тип данных		
char	1	0 / 255
целочисленные типы данных		
short int	2	-32 768 / 32 767
unsigned short int	2	0 / 65 535
int	4	-2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long int	4	-2 147 483 648 / 2 147 483 647
unsigned long int	4	0 / 4 294 967 295
типы данных с плавающей точкой		
float	4	-2 147 483 648.0 / 2 147 483 647.0
long float	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0
double	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0

Рисунок 1.2.1 – Переменные и их диапазон

Код программы для вывода размера и диапазона типа данных bool представлен на рисунке 1.2.2.

```
cout << "     data type      "   << "byte"                    << "      "    << "    max value  " << endl // заголовки столбцов
<< "bool              =  " << sizeof(bool)             << "        " << fixed << setprecision(2) << (pow(2,sizeof(bool) * 8.0) - 1);
```

Рисунок 1.2.2 – Код для вывода размера типа данных

На рисунке 1.2.3 представлен вывод в консоль представленного выше кода.

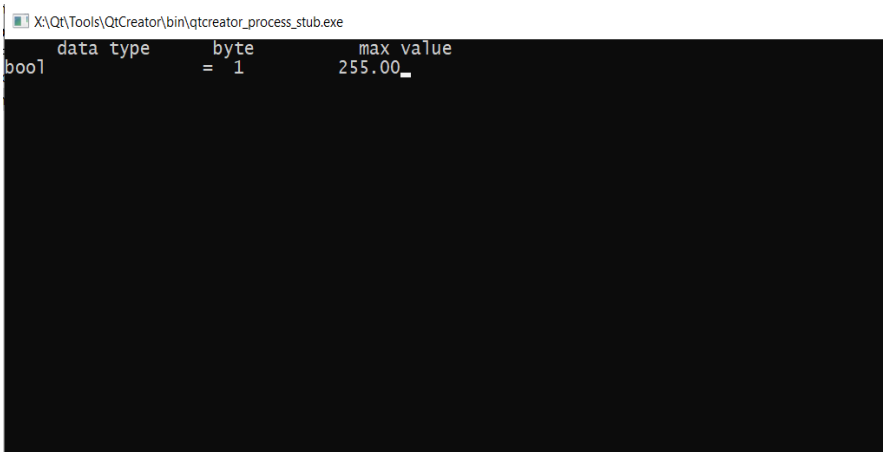

Рисунок 1.2.3 – Вывод размера

Основные операторы

Оператор – это лексема, которая переключает некоторые вычисления, когда применяется к переменной или к другому объекту в выражении. Язык С++ представляет большой набор операторов арифметических и логических операторов.

Таблица 1.2.1 – Унарные операторы языка С++

Код оператора	Название	Результат операции
&	адресный оператор	выражение *&x* - адрес переменной *x*
+	унарный плюс	+5 – положительная константа
-	унарный минус	-4 – отрицательная константа, -*x* – значение переменной *x* с обратным знаком
!	логическое отрицание	*!x* принимает значение 0 (лжи), если x имеет ненулевое (истинное) значение и наоборот
++	префиксное/ постфиксное увеличение	*int x = 5; ++x;* увеличит *x* на единицу; *int x = 5; x++;* увеличит *x* на единицу
--	префиксное/ постфиксное уменьшение	*int x = 5; --x;* уменьшит x на единицу; int x = 5; x--; увеличит x на единицу

Таблица 1.2.2 – Бинарные операторы языка С++

Код оператора	Название	Результат операции
Аддитивные операторы		

+	бинарный плюс	вычисление суммы, например:
		int x = 2,
		y = 1;
		z = x+y;

-	Бинарный минус	вычисление разности, например:
		int x = 2,
		y = 1;
		z = x-y;

Мультипликативные операторы		

*	умножение	вычисление произведения, например:
		int x = 2,
		y = 1;
		z = x*y;

/	деление	вычисление частного, например:
		int x = 12,
		y = 2;
		z = x/y;

%	остаток	вычисление остатка от деления, например:
		int x = 12,
		y = 7;
		z = x%y;

Логические операторы		
&&	логическое AND (И)	проверка условий, связанных логическим И
||	логическое OR (ИЛИ)	проверка условий, связанных логическим ИЛИ
Операторы присваивания		
=	присваивание	присвоить переменной заданное значение или значение другой переменной

Продолжение таблицы 1.2.2

Операторы отношения		
<	меньше чем	x<y, x меньше y
>	больше чем	x>y, x больше y
<=	меньше чем или равно	x<=y, x меньше или равно y
>=	больше чем или равно	x>=y, x больше или равно y
Операторы эквивалентности		
= =	равно	x= =y, x равно y
!=	не равно	x!=y, x не равно y
,	оператор перечисления	выполнить разделенные оператором действия слева направо, например
		y+=5,x-=4,y+=x;

Библиотека математических функций math.h

C++ поддерживает множество математических функций, прототипы которых описаны в файле *math.h.* Познакомимся с некоторыми из них.

abs(int x) возвращает модуль целого числа *x*.

acos(double x) возвращает арккосинус числа *x* в радианах. *asin(double x)* возвращает арксинус числа *x* в радианах.

atan(double x) возвращает арктангенс числа *x* в радианах.

*atof(char *s, double x)* преобразует строку *s* в вещественное число *x. cos(double x)* возвращает косинус числа

x (*x* задано в радианах)

ceil(double x) округляет число *x* в большую сторону *exp(double x)* возвращает экспоненту числа *x.*

fabs(double x) возвращает модуль вещественного числа *x. sin(double x)* возвращает синус числа *x* (*x* задано в радианах).

sqrt(double x) возвращает квадратный корень числа *x. tan(double x)* возвращает тангенс числа *x* (*x* задано в радианах).

floor(double x) округляет число *x* в меньшую сторону

fmod(double x, double y) возвращает остаток от деления числа *x* на число *y. hipot(double x, double y)* возвращает квадрат суммы числа *x* и числа *y.*

log(double x) возвращает натуральный логарифм числа *x.*

log10(double x) возвращает десятичный логарифм числа *x. modf(double x,double& y)* возвращает дробную часть числа *x,* по адресу *y* записывается целая часть исходного числа *x. pow(double x, double y)* возвращает *x* в степени *y.*

Для использования всех вышеперечисленных функций подключите библиотеку *math.h:*

#include< math.h>

Задание 2.

Написать программу, вычисляющую значение по индивидуальному варианту. На экран вывести найденное значение.

Примечание: при использовании математических функций необходимо подключить библиотеку math.h. (<cmath>)

Тригонометрические функции на вход получают значения в радианах.

Рекомендуется при вводе задавать значения в градусах, а затем переводить в радианы по формуле: *rad = M_PI*grad/180*, где *M_PI* – стандартная константа, описанная в библиотеке *math.h (cmath), grad* – значение угла в градусах.

В1. Даны x, y. Вычислить a, если $a = 1 + |y + sin(x)|$. Числа вводить с клавиатуры.

В2. Даны x, y, z. Вычислить a, если $a = 1 + x|y - tg(z)|$. Числа вводить с клавиатуры.

В3. Даны x, y. Вычислить a, если $a = |x - 1| - |y|$. Числа вводить с клавиатуры.

В4. Даны x, y. Вычислить a, если $a = \frac{x}{2}(1 + tg(y))$. Числа вводить с клавиатуры.

В5. Дано x. Вычислить a, если $a = \frac{1}{2} + sin(x)$. Числа вводить с клавиатуры.

В6. Даны x, y. Вычислить a, если $a = 2cos(x - \frac{y}{6})$. Числа вводить с клавиатуры.

В7. Дано x. Вычислить a, если $a = cos(tg(x) - 1)$. Числа вводить с клавиатуры.

В8. Даны x, y. Вычислить a, если $a = (1 + y)\left(2x + \sqrt{y} - (x + y)\right)$. Числа вводить с клавиатуры.

В9. Даны x, y, z. Вычислить a, если $a = x(tg(y) + z)$. Числа вводить с клавиатуры.

В10. Даны x, y, z. Вычислить a, если $a = \sqrt{x - \frac{y}{\sqrt{z}}}$. Числа вводить с клавиатуры.

Контрольные вопросы

1. Какая функция используется в Си для ввода информации?
2. Какая функция используется в Си для вывода информации?
3. Какой тип данных Си соответствует спецификатору «%d»?
4. Какой тип данных Си соответствует спецификатору «%f»?
5. Дайте определение понятия «переменная»
6. Дайте определение понятия «идентификатор»
7. Сколько переменных требуется описать в программе, если необходимо решить следующую задачу – «С клавиатуры вводятся три числа, необходимо вывести на экран значение минимального из этих трех чисел»?
8. Переменная j описана в программе следующим образом: int j. Запишите функцию scanf для считывания значения в переменную j.

Содержание отчета

1. Цель работы;
2. Подробное описание всех этапов проделанной работы;
3. Анализ проделанной работы;
4. Листинг программы;
5. Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №2
«Условия и циклы»

Цель работы: ознакомиться с условными операторами, а также с операторами, реализующими циклический процесс в языке C++.

2.1 Условный оператор if

Для организации вычислений в зависимости от какого-либо условия в C++ предусмотрен условный оператор ***if***, который в общем виде записывается следующим образом:

if (условие) оператор_1; else оператор_2;

где условие – это логическое (или целое) выражение, переменная или константа, оператор_1 и оператор_2 – любой оператор языка C(C++) .

Сначала вычисляется значение выражения, указанного в скобках. Если оно имеет истинное значение (true), то выполняется оператор_1. В противном случае, имеет значение ложь (false), выполняется оператор_2.

Внимание! Не путайте знак проверки равенства == и оператор присваивания =.

Внимание! Если в задаче требуется, чтобы в зависимости от значения условия выполнялся не один оператор, а несколько, их необходимо заключать в фигурные скобки, как составной оператор. В этом случае компилятор воспримет группу операторов как один:

```
if ( условие )
{оператор_1; оператор_2; ... }
else
{оператор_3; оператор_4; ... }
```

Альтернативная ветвь else в условном операторе может отсутствовать, если в ней нет необходимости.

Условные операторы могут быть вложены друг в друга. При вложениях условных операторов всегда действует правило: альтернатива else считается принадлежащей ближайшему if. Например, в записи

if (условие_1) if (условие_2) оператор_А; else оператор_Б;

оператор_Б относится к условию_2, а в конструкции

```
if (условие_1) { if (условие_2) оператор_А; }
else оператор_Б;
```

он принадлежит оператору if с условием_1.

Рассмотрим пример:

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

int main()
{
    setlocale(0, "");
    double num;

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

    if (num < 10) // Если введенное число меньше 10.
        cout << "Это число меньше 10." << endl;
    else if (num == 10)
        cout << "Это число равно 10." << endl;
    else  // иначе
        cout << "Это число больше 10." << endl;

    return 0;
}
```

2.2 Оператор switch

Он необходим в тех случаях, когда в зависимости от значений переменной надо выполнить те или иные операторы:

switch (выражение)
{
case значение_1: операторы_1; break;
case значение_2: операторы_2; break;
case значение_3: операторы_3; break;
…
case значение_n: операторы_n; break;
default: операторы; break;
}

Оператор работает следующем образом. Вычисляется значение выражения. Затем выполняются операторы, помеченные значением, совпадающим со значением выражения. То есть если, выражение принимает значение_1, то выполняются операторы_1 и т.д. Если выражение не принимает ни одного из значений, то выполняются операторы, расположенные после слова default.

Ветвь default может отсутствовать, тогда оператор имеет вид:

switch (выражение)
{
case значение_1: операторы_1; break;
case значение_2: операторы_2; break;
case значение_3: операторы_3; break;
…
case значение_n: операторы_n; break;
}

Оператор break необходим для того, чтобы осуществить выход из оператора switch. Если он не указан, то будут выполняться следующие операторы из списка, несмотря на то что значение, которым они помечены, не совпадает со значением выражения.

Рассмотрим применение оператора варианта. Пусть необходимо вывести на печать название дня недели, соответствующее заданному числу, при условии, что в месяце 31 день и 1-е число – понедельник.

Для решения задачи воспользуемся операцией %, позволяющей вычислить остаток от деления двух чисел, и условием, что 1-е число – понедельник. Если в результате остаток от деления заданного числа на семь будет равен единице, то это понедельник, двойке – вторник, тройке – среда и так далее. Следовательно, при построении алгоритма необходимо использовать семь условных операторов, рисунок 2.2.1.

```
#include <iostream>
using namespace std;
int main ( )
{
    unsigned int D,R;
    cout<<" \ n D = "; cin >>D;
    if (D<32) //Проверка введённого значения.
    {
        R=D%7;
        switch (R)
        {
            case 1 : cout<<" Понедельник \n "; break;
            case 2 : cout<<" Вторник \n "; break;
            case 3 : cout<<" Среда \n "; break;
            case 4 : cout<<" Четверг \n "; break;
            case 5 : cout<<" Пятница \n "; break;
            case 6 : cout<<" Суббота \n "; break;
            case 0 : cout<<" Воскресенье \n "; break;
        }
    }
    //Сообщение об ошибке в случае некорректного ввода.
    else cout<<" ОШИБКА! \n ";
    return 0;
}
```

Рисунок 2.2.1 – Пример оператора Switch

2.3 Циклы

Цикл – это повторение одного и того же участка кода в программе. Последовательность действий, которые повторяются, называют телом цикла. Один проход цикла – это шаг или итерация. Переменные, изменяющиеся внутри цикла и влияющие на его окончание, называются параметрами цикла.

В C++ предусмотрены три оператора, реализующих циклический процесс: while, do while и for. Рассмотрим каждый из них.

2.3.1 Цикл while

Цикл while выполняет некоторый код, пока его условие истинно, то есть возвращает true. Он имеет следующее определение:

while(условие)
{ // выполняемые действия }

После ключевого слова while в скобках идет условное выражение, которое возвращает true или false. Затем в фигурных скобках идет набор инструкций, которые составляют тело

цикла. И пока условие возвращает true, будут выполняться инструкции в теле цикла. Когда мы не знаем, сколько итераций должен произвести цикл, используем while или do...while.

Ниже приведен исходный код программы, считающей сумму всех целых чисел от 1 до 1000.

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

int main()
{
   int i = 0; // инициализируем счетчик цикла.
   int sum = 0; // инициализируем счетчик суммы.
   while (i < 1000)
   {
      i++;
      sum += i;
   }
   cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
   return 0;
}
```

2.3.2 Цикл do while

В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, то есть не равно 0, то цикл повторяется. Формальное определение цикла:

```
do
{  инструкции }
while(условие);
```

Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла do while.

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

int main ()
{
   int i = 0; // инициализируем счетчик цикла.
   int sum = 0; // инициализируем счетчик суммы.
   do {// выполняем цикл.
      i++;
      sum += i;
   } while (i < 1000); // пока выполняется условие.
   cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
   return 0;
}
```

2.3.3 Цикл for

Цикл for имеет следующее определение:

```
for (выражение_1; выражение_2; выражение_3)
{
   // тело цикла
}
```

выражение_1 выполняется один раз при начале выполнения цикла и представляет установку начальных условий, как правило, это инициализация счетчиков – специальных переменных, которые используются для контроля за циклом.

выражение_2 представляет условие, при соблюдении которого выполняется цикл.

выражение_3 задает изменение параметров цикла, часто здесь происходит увеличение счетчиков цикла на единицу.

Решение задачи на поиск суммы чисел от 1 до 1000, с применением цикла for.

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

int main()
{
   int i; // счетчик цикла
   int sum = 0; // сумма чисел от 1 до 1000.
   for (i = 1; i <= 1000; i++) // задаем начальное значение 1, конечное 1000 и задаем шаг цикла - 1.
   {
      sum = sum + i;
   }
   cout << "Сумма чисел от 1 до 1000 = " << sum << endl;
   return 0;
}
```

Задание

1. Дано натуральное число. Найдите количество четных цифр.
2. Найдите n-ое число Фибоначчи.
3. Найдите четырехзначные числа, сумма цифр которых равна 15.
4. Найдите все делители данного натурального числа.
5. Найдите количество целых чисел от a до b включительно, которые делятся на 12.
6. Вывести на экран числа от 1000 до 9999 такие, что все цифры различны.
7. Вывести на экран числа от 1000 до 9999 такие, что среди цифр нет цифр 5 и цифры 6.
8. Найдите сумму 1+1/2+1/3+…+1/n.
9. Вывести все пятизначные числа, которые делятся на 2, у которых средняя цифра нечетная, и сумма всех цифр делится на 4.
10. Для данного натурального числа найдите число, цифры которого записаны в обратном порядке.

Контрольные вопросы

1. Как работает условный оператор if ?
2. Как работает оператор варианта switch ?
3. Опишите синтаксис конструкции if else.
4. Операторы цикла в C++ .
5. Что такое цикл?

6. Чем отличаются for и while?

7. Что такое вложенный цикл?

8. Какое значение примет переменная x после выполнения следующего фрагмента программы:

```
…
int x = 10;
int k = 12,
z = 74;
if (k<z) x = 1; else x = 0;
…
```

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №3
«Массивы и указатели»

Цель работы: ознакомиться с работой массивов и указателей в языке C++.

3.1 Массивы

Массив – совокупный тип данных, который позволяет получить доступ ко всем переменным одного и того же типа данных через использование одного идентификатора.

Для того чтобы использовать массив, его надо сначала объявить. Для этого нужно использовать конструкцию:

<тип> <имя массива> [<кол-во элементов >];

Например:

```
int arr[50];
```

Этим кодом мы создали массив типа int с именем arr в котором может храниться до 50-ти элементов.

Способ указания значения элементам массива при его инициализации:

```
int arr[] = {0, 1, 2, 3, 4, 5}; // массив будет иметь 6 элементов (от 0 до 5)
int mas[100] = {0}; // все 100 элементов будут иметь значение 0
```

То есть, для того чтобы задать значения, нужно сразу после объявления массива указать через равно в фигурных скобках требуемые значения.

Внимание! В C++ нумерация элементов массива идет с нуля. Таким образом второй элемент будет иметь индекс 1, а десятый – 9.

Двумерный массив (матрицу) можно объявить так:

тип имя_переменной[n][m];

где n - количество строк (от 0 до (n-1)), m - количество столбцов (от 0 до (m-1)).

Например, *int a[3][4]*;

```
int a[3][4] = {{4, 7, 8, 2}, {9, 66, -1, 0}, {7, 5, -5, 0}}; //инициализация двумерного массива
```

Обращаются к элементу матрицы указывая последовательно в квадратных скобках, соответствующие индексы. Например элемент матрицы *a* находящийся в первой строке и втором столбце будет объявлен как: *a[1][2]*.

3.2 Указатели

Указатели представляют собой объекты, значения которых служат адреса других объектов (переменных, констант, указателей) или функций.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *. Например, определим указатель на объект типа int:

```
int *p;
```

Здесь указатель не ссылается ни на какой объект. Теперь присвоим указателю адрес переменной:

```
int x = 10; // определяем переменную
int *p; // определяем указатель
p = &x; // указатель получает адрес переменной
```

Для получения адреса переменной применяется операция **&.**

Важно! У переменной и указателя, который указывает на ее адрес, должно быть соответствие по типу.

Например:

```
#include <iostream>

int main()
{
    int x = 10;
    int *p;
    p = &x;
    std::cout << "Address = " << p << std::endl;
    std::cout << "Value = " << *p << std::endl;
    return 0;
}
```

В консоли мы получим

```
Address = 0x60fe98
Value = 10
```

Динамический массив – это массив, в котором количество элементов и выделенный на него объем памяти может меняться как при инициализации, так и при работе с ним.

Для выделения памяти под динамический массив используется оператор ***new***:

```
int *numbers = new int[4]; // динамический массив из 4 чисел
```

В данном случае определяется массив из четырех элементов типа int, но каждый из них имеет неопределенное значение. Инициализируем массив значениями:

```
int *n1 = new int[4];   // каждый элемент имеет неопределенное значение
int *n2 = new int[4](); // каждый элемент имеет значение по умолчанию - 0
int *n3 = new int[4]{ 1, 2, 3, 4 }; // массив состоит из чисел 1, 2, 3, 4
```

В последнем случае следует учитывать, что если значений в фигурных скобках больше чем длина массива, то оператор *new* не сможет создать массив. Если переданных значений, наоборот, меньше, то элементы, для которых не указаны значения, инициализируются значением по умолчанию.

После создания динамического массива мы сможем с ним работать по полученному указателю, получать и изменять его элементы:

```
int n = 5;  // размер массива
int *p = new int[n]{ 1, 2, 3, 4, 5 };
for (int *q = p; q != p + n; q++)
{
  std::cout << *q << "\t";
}
```

Для удаления динамического массива и освобождения его памяти применяется специальная форма оператора ***delete***:

delete [] указатель_на_динамический_массив;

Например:

```
#include <iostream>

int main()
{
    int n = 5;  // размер массива
    int *p = new int[n]{ 1, 2, 3, 4, 5 }; // массив состоит из чисел 1, 2, 3, 4
    for (int *q = p; q != p + n; q++)
    {
        std::cout << *q << "\t";
    }

    std::cout << std::endl;

    delete [] p;

    return 0;
}
```

Вектор – это структура данных, которая является моделью динамического массива.

Для использования векторов в своей программе, необходимо подключить заголовочный файл <*vector*>.

Чтобы объявить вектор, нужно пользоваться конструкцией:

vector < тип данных > <имя вектора>;

Вектор можно объявить следующим образом:

std::vector<int> myVector; // мы создали пустой вектор типа int

myVector.reserve(10); // тут мы зарезервировали память под 10 элементов типа int

Пример:

```
#include <iostream>
#include <vector> // подключаем модель Векторов
using namespace std;
int main()
{
    vector<int> myVector(10);   // объявляем вектор
    //размером в 10 элементов и инициализируем их нулями
    // вывод элементов вектора на экран
    for(int i = 0; i < myVector.size(); i++)
        cout << myVector[i] << ' ';
    return 0;
}
```

Компилятор выведет:

```
0 0 0 0 0 0 0 0 0 0

...Program finished with exit code 0
Press ENTER to exit console.
```

Методы класса *vector*:

- size() - определить размер вектора;
- max_size() - максимально допустимый размер массива;
- capacity() - определить размер массива с учетом зарезервированной памяти;
- empty() - определить, пустой ли вектор;
- reserve() - зарезервировать память для дополнительных элементов массива;
- resize() - изменить размер массива;
- push_back() - добавить элемент в конец вектора;
- pop_back() - удалить последний элемент вектора;

– clear() - удаляет из массива все элементы;
– swap() - обмен местами двух векторов;
– присваивание массивов. Перегруженный оператор =;
– erase() - удалить один или несколько элементов заданного диапазона;
– insert() - вставляет элемент или группу элементов в вектор;
– at() - получить элемент вектора по его позиции (индексу);
– front() - возвращает ссылку на первый элемент вектора;
– back() - возвращает ссылку на последний элемент вектора;
– data() - получить указатель на вектор;
– begin() - вернуть итератор, указывающий на первый элемент вектора;
– end() - вернуть итератор, указывающий на последний элемент массива.

Задание

1. Дана матрица. Вывести на экран все четные строки, то есть с четными номерами.
2. Дана матрица. Вывести на экран все нечетные столбцы, у которых первый элемент больше последнего.
3. Дан двухмерный массив 5×5. Найти сумму модулей отрицательных нечетных элементов.
4. Дан двухмерный массив n×m элементов. Определить, сколько раз встречается число 7 среди элементов массива.
5. Дана квадратная матрица. Вывести на экран элементы, стоящие на диагонали.
6. Дана матрица. Вывести k-ю строку и p-й столбец матрицы.
7. Дана матрица размера m x n. Вывести ее элементы в следующем порядке: первая строка справа налево, вторая строка слева направо, третья строка справа налево и так далее.
8. Создать квадратную матрицу, на диагонали которой находятся тройки, выше диагонали находятся двойки, остальные элементы равна единице.
9. Сформировать матрицу n x m, состоящую из нулей и единиц, причем единицы находятся только в угловых клетках.
10. Заполнить матрицу так, чтобы сумма элементов в каждой строке была равна номеру этой строки.

Контрольные вопросы

1. Что такое массив?
2. Как объявить двумерный массив?
3. Что такое динамический массив? Как его инициализировать?
4. Как осуществляется доступ к отдельному элементу одномерного и двумерного массива?
5. Каким образом выводятся элементы массива на экран?
6. Приведите пример фрагмента программы, который выводит на экран двумерный массив в виде матрицы.
7. Что такое вектор?

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Вывод по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №4
«Файловый ввод-вывод»

Цель работы: научиться работать с файловым вводом-выводом языка C++.

Для работы с файлами необходимо подключить заголовочный файл **<fstream>**. В <fstream> определены несколько классов и подключены заголовочные файлы <ifstream> – файловый ввод и <ofstream> – файловый вывод.

Класс ***ifstream*** предназначен для ввода файлового потока. Основные методы данного класса описаны в таблице 4.1.

Таблица 4.1 – Основные методы класса ifstream

Метод	Описание
open	Открывает файл для чтения
get	Читает один или более символов из файла
getline	Читает символьную строку из текстового файла или данные из бинарного файла до определенного ограничителя
read	Считывает заданное число байт из файла в память
eof	Возвращает ненулевое значение (true), когда указатель потока достигает конца файла
close	Закрывает файл

Класс ***ofstream*** предназначен для вывода данных из файлового потока. Основные методы данного класса описаны в таблице 4.2.

Таблица 4.2 – Основные методы класса ofstream

Метод	Описание
open	Открывает файл для записи
put	Записывает одиночный символ в файл
write	Записывает заданное число байт из памяти в файл
close	Закрывает файл

Рассмотрим пример.

```
#include "stdafx.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[])
{
    ofstream fout("cppstudio.txt"); // создаём объект класса ofstream для записи
    //и связываем его с файлом cppstudio.txt
    fout << "Работа с файлами в C++"; // запись строки в файл
    fout.close(); // закрываем файл
    system("pause");
    return 0;
}
```

В C++ существует функция – *is_open()*, которая возвращает целые значения: 1 – если файл был успешно открыт, 0 – если файл не открылся.

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе *ios_base* предусмотрены константы, которые определяют режим открытия файлов, таблица 4.3.

Таблица 4.3 – Константы для открытия файлов

Константа	Описание
ios_base::in	открыть файл для чтения
ios_base::out	открыть файл для записи
ios_base::ate	при открытии переместить указатель в конец файла
ios_base::app	открыть файл для записи в конец файла
ios_base::trunc	удалить содержимое файла, если он существует
ios_base::binary	открытие файла в двоичном режиме

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции *open()*.

```
ofstream fout("cppstudio.txt", ios_base::app); // открываем файл
//для добавления информации к концу файла
fout.open("cppstudio.txt", ios_base::app); // открываем файл
//для добавления информации к концу файла
```

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или |, например: *ios_base::out* | *ios_base::trunc* – открытие файла для записи, предварительно очистив его.

Задание

1. Дан текстовый файл, содержащий целые числа. Удалить из него все четные числа.
2. В данном текстовом файле удалить все слова, которые содержат хотя бы одну цифру.
3. Дан текстовый файл. Создать новый файл, каждая строка которого получается из соответствующей строки исходного файла перестановкой слов в обратном порядке.

4. Создать и заполнить файл случайными целыми значениями. Выполнить сортировку содержимого файла по возрастанию.

5. В файле, содержащем фамилии студентов и их оценки, изменить на прописные буквы фамилии тех студентов, которые имеют средний балл за национальной шкалой более «4».

6. Из текстового файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только четное количество таких слов.

7. Текстовый файл содержит записи о телефонах и их владельцах. Переписать в другой файл телефоны тех владельцев, фамилии которых начинаются с букв К и С.

8. В файле содержится совокупность текстовых строк. Изменить первую букву каждого слова на заглавную.

9. В файле содержится текстовая строка. Определить частоту повторяемости каждой буквы в тексте и вывести ее.

Контрольные вопросы

1. Какими способами можно открыть текстовый файл
2. Опишите основные режимы открытия документа, доступные в функции open().
3. Что такое ofstream?
4. Какой есть аналог ofstream?
5. С какими типами файлов может работать библиотека fstream?

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №5
«Основы ООП»

Цель работы: получить навыки создания простейших классов и понимания основ ООП.

5.1 Объектно ориентированное программирование

Объектно-ориентированное программирование (ООП) – это подход, при котором вся программа рассматривается как набор взаимодействующих друг с другом объектов.

Объектный подход родился как важный шаг на пути качественного написания больших программ. В нём предлагается разделять программу на самостоятельные части – объекты, наделенные собственными свойствами, текущим состоянием, и умеющие взаимодействовать друг с другом и с окружающей средой – примерно так, как это происходит у объектов реального мира.

Объект состоит из следующих трёх частей (рисунок 5.1):

- имя объекта;
- состояние (переменные состояния);
- методы (операции).

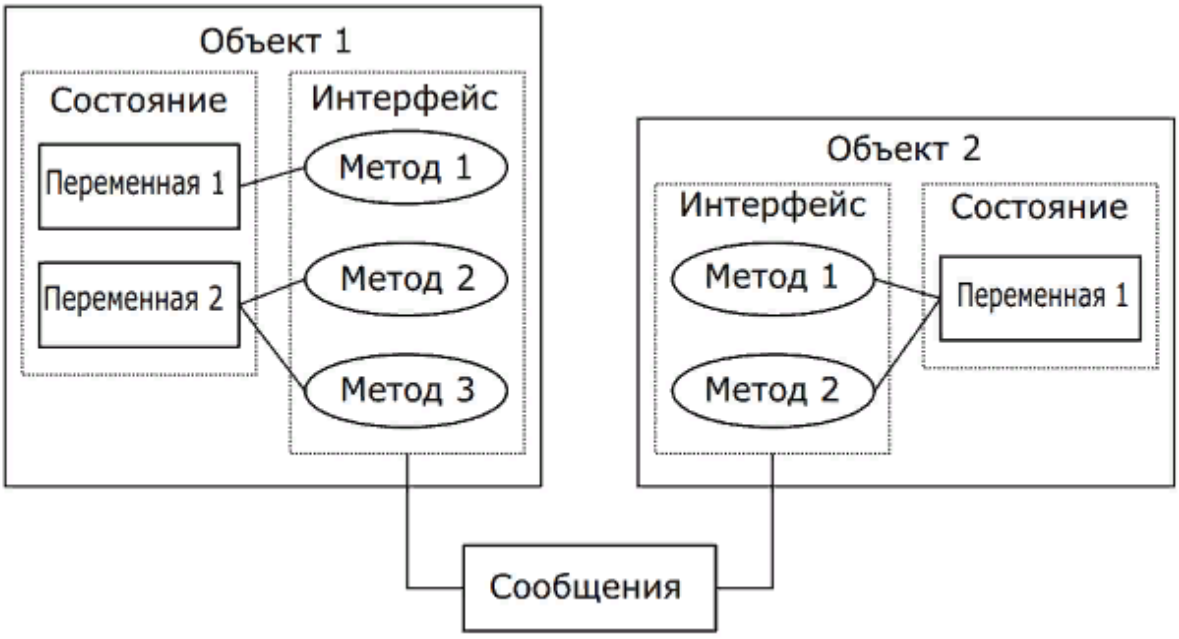

Рисунок 5.1 – Основные три части объекта

Интерфейс объекта с окружающей средой (пользователем, операционной системой и т. д.) полностью осуществляется методами: к состоянию объекта нет другого доступа извне, кроме как через его методы.

Закрытость внутреннего состояния объекта от окружающей среды известна также как свойство **инкапсуляции**.

Во многих языках частью инкапсуляции является сокрытие данных. Для этого существуют **модификаторы доступа**:

public – к атрибуту может получить доступ любой желающий;

private – к атрибуту могут обращаться только методы данного класса;

protected – то же, что и private, только доступ получают и наследники класса в том числе.

Однотипные объекты образуют **класс**. Под однотипными объектами понимаются такие объекты, у которых одинаковый набор методов и переменных состояния. При этом объекты, принадлежащие одному классу, имеют разные имена и, вероятно, разные значения переменных состояния. Членами класса будут переменные и функции, объявленные внутри класса. Функции-члены класса называют **методами** этого класса, а переменные-члены класса называют **свойствами** класса.

Класс описывается так:

```
class имя_класса
{
    закрытые члены класса
    public :
    открытые члены класса
};
```

По умолчанию все функции и переменные, объявленные в классе, являются закрытыми, т. е. имеют модификатор доступа *private*.

В реальном мире из родственных по смыслу сущностей часто можно составить иерархию "от общего к частному". Такие отношения в ООП называются **наследованием**. Из двух классов, находящихся в отношении наследования, более общий класс называется базовым или родительским классом, а класс, представляющий собой более частный случай, называется дочерним или производным классом. Производный класс может заимствовать атрибуты (свойства и методы) базового класса.

Следующее основополагающее качество: **полиморфизм** - объекты могут вести себя по-разному в зависимости от ситуации. Одно из основных проявлений полиморфного поведения – перегрузка функций. Объект может содержать в себе несколько методов с одинаковыми именами, принимающих разные наборы параметров. В результате, передавая объекту данные, можно обращаться к одному и тому же имени метода, не заботясь о типе, в котором представлены данные. Правильно сконструированный объект автоматически выполнит наиболее подходящий метод из группы.

Инкапсуляция, наследование и полиморфизм являются тремя основополагающими принципами ООП.

Бывает ситуация, когда требуется за один раз присвоить значения переменным-членам класса (всем или большинству): это момент создания объекта, т.е. переменной-экземпляра класса. C++ позволяет создать специальный метод, который будет автоматически вызываться для инициализации переменных-членов объекта при его создании. Такой метод называется **конструктором**. Конструктор может принимать любые аргументы, как и любой другой метод.

Если конструктор класса не был определен, то компилятор самостоятельно создает **конструктор по умолчанию** (пустой и без входных параметров).

Конструктор может вызываться явно или неявно. Компилятор сам вызывает конструктор в том месте программы, где создается объект класса.

У описания конструкторов в языке C++ есть следующие особенности:

– имя конструктора в C++ совпадает с именем класса;

– конструктор не возвращает никакое значение, но при описании конструктора не используется и ключевое слово void.

Функция обратная конструктору - **деструктор**. Эта функция вызывается при удалении объекта.

В C++ деструкторы имеют имена, состоящие из имени класса с префиксом - тильдой: " ~*имя_класса* ". Как и конструктор, деструктор не возвращает никакое значение, но в отличие от конструктора он не может быть вызван явно.

Пример:

```
#include <iostream>
#include <math.h>
using namespace std;
class spatial_vector
{
        double x, y, z;
public :
        spatial_vector ( );
        ~ spatial_vector ( ) { cout << "Работа деструктора\n "; }
        double abs ( ) { return sqrt ( x*x + y*y + z*z ); }
};
spatial_vector::spatial_vector ( )
{
        //конструктор класса vector
        x=y=z =0;
        cout << "Работа конструктора\n ";
}
main ( )
{
        spatial_vector a;//создаётся объект a с нулевыми значениями
        cout << a.abs ( ) << endl;
}
```

5.2 Ключевое слово «this»

Указатель *this* – это указатель на адрес объекта класса, при этом он является скрытым первым параметром любого метода класса (кроме статических методов), а типом указателя выступает имя класса.

Каждый метод класса неявно содержит в качестве поля данных указатель:

*имя_класса *this;*

При вызове метода ему передается неявный аргумент, содержащий адрес объекта, для которого эта функция вызывается:

```
class example {
  int m;
public:
  int readm() { return m; }  // return this->m
};
void f() {
  example aa, bb;
  int a = aa.readm();  // this указывает на aa
  int b = bb.readm(); // this указывает на bb
}
```

В первом случае функции *readm()* неявно передается указатель на объект *aa*, а во втором случае – *bb*.

Использование this необходимо в функциях, которые непосредственно работают с указателем на объект:

this – указатель на объект (адрес объекта)

**this* – разыменованый указатель (сам объект)

Указатель this удобно использовать в конструкторах, когда имена передаваемых параметров совпадают с именами полей класса:

```
class date
{
  int day, month, year;
public:
  date(int day, int month, int year)
  {
    this->day = day; //поле класса = аргумент
    this->month = month;
    this->year = year;
  }
};
```

В случае если аргументы и поля класса имеют разные имена, указатель *this* при обращении к полям класса может быть опущен. Но в приведенном примере использование *this* указывает, что мы ходим обратиться именно к полю класса.

5.2 Ключевое слово «static»

В C++ предусмотрен способ совместного использования элемента данных несколькими объектами – **статические члены класса**. Типичное применение этого механизма – быстрый и эффективный обмен информацией между однотипными объектами за счет общей переменной.

Чтобы объявить статический элемент класса, перед ним необходимо указать ключевое слово ***static***.

Рассмотрим пример. Создадим класс *point* и добавим статическое свойство *count* – счётчик, указывающий, сколько экземпляров класса существует в памяти в настоящий момент.

```
#include<iostream>
using namespace std;
class point
{
        int x, y;
        static int count;
public :
        point ( ) { cout << "Создаётся точка с номером " << ++count << endl; }
        ~ point ( ) { cout << "Разрушается точка с номером " << count-- << endl; }
};
int point::count;
main ( ) {
        point a,b,c;
}
```

Статическая переменная-член класса должна быть также объявлена в программе в качестве глобальной переменной с указанием её принадлежности классу (см. в примере строку перед описанием функции *main*). В результате программа сначала создаст, а потом разрушит три объекта класса *point*:

```
Создаётся точка с номером 1
Создаётся точка с номером 2
Создаётся точка с номером 3
Разрушается точка с номером 3
Разрушается точка с номером 2
Разрушается точка с номером 1
```

5.3 Методы

Метод класса также можно объявить **статическим**. Статическую функцию-член вы можете использовать без создания объекта класса. Доступ к статическим функциям осуществляется с использованием имени класса и оператора разрешения области видимости (::). При использовании статической функции-члена есть ограничения, такие как:

- внутри функции обращаться можно только к статическим членам данных, другим статическим функциям-членам и любым другим функциям извне класса;
- статические функции-члены имеют область видимости класса, в котором они находятся;
- вы не имеете доступа к указателю *this* класса, потому что мы не создаем никакого объекта для вызова этой функции.

Рассмотрим пример:

```
#include <iostream>

class A {
public:
  A() { std::cout << "Constructor A" << std::endl; }
  ~A() { std::cout << "Destructor A" << std::endl; }

  static void foo() { // строка 8
    std::cout << "static foo()" << std::endl;
  }
};

int main() {
  A::foo(); // строка 14
  return 0;
}
```

В классе A в строке 8 у нас есть статическая функция-член *foo()*. В строке 14, мы вызываем функцию используя имя класса и оператор разрешения области видимости и получаем следующий результат программы:

```
static foo()
```

Если бы метод *foo()* был бы нестатическим, то компилятор выдал бы ошибку на выражение в строке 14, т.к. нужно создать объект для того, чтобы получить доступ к его нестатическим методам.

Задание

1. Создать класс, содержащий динамический массив и количество элементов в нем. Добавить конструктор, который выделяет память под заданное количество элементов, и деструктор. Добавить методы, позволяющие заполнять массив случайными числами, переставлять в данном массиве элементы в случайном порядке, находить количество различных элементов в массиве, выводить массив на экран.

2. Составить описание класса для определения одномерных массивов строк фиксированной длины. Предусмотреть контроль выхода за пределы массива, возможность обращения к отдельным строкам массива по индексам, выполнения операций поэлементного сцепления двух массивов с образованием нового массива, слияния двух массивов с исключением повторяющихся элементов, а также вывод на экран элемента массива по заданному индексу и всего массива.

3. Описать класс «домашняя библиотека». Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

4. Создать класс для хранения комплексных чисел. Реализовать операции над комплексными числами: сложение, вычитание, умножение, деление, сопряжение, возведение в степень, извлечение корня.

5. Описать класс, представляющий треугольник. Предусмотреть методы для создания объектов, вычисления площади, периметра и точки пересечения медиан. Описать свойства для получения состояния объекта.

Контрольные вопросы

1. Основные модификаторы доступа к данным.
2. Полиморфизм, наследование, инкапсуляция.
3. Для чего нужен конструктор?
4. Зачем нужен указатель *this*?
5. Статические члены и методы класса.

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №6
«Перегрузка функций, членов класса»

Цель работы: ознакомиться с перегрузкой методов и операторов в языке C++.

Перегрузка – возможность создавать функции (например, методы класса) с одинаковыми именами и разными наборами параметров, вызываемые в разных ситуациях для решения однотипных задач (одно из ключевых проявлений полиморфизма в C++). Компилятор различает их благодаря тому, что они *имеют разный набор параметров*. Списки параметров перегруженных функций должны отличаться по следующим признакам:

– количеством параметров;

– если количество параметров одинаковое, то по типам параметров.

В точки вызова компилятор анализирует типы аргументов и определяет, какая конкретно функция должна быть вызвана.

Перегруженные функции не могут иметь разные типы возвращаемого значения, спецификатор исключений или спецификатор удаленной функции (=delete) при одинаковых параметрах.

```
void Foo();
char Foo(); // ошибка
void Foo(int x);
void Foo(int x) noexcept;  // ошибка
void Foo(double x);
void Foo(double) = delete; // ошибка
```

Синтаксис перегрузки операторов очень похож на определение функции с именем *operator@*, где @ – это идентификатор оператора (например +, -, <<, >>). Рассмотрим пример:

```
class Integer
{
private:
    int value;
public:
    Integer(int i): value(i)
    {}
    const Integer operator+(const Integer& rv) const {
        return (value + rv.value);
    }
};
```

Различают три основных вида операторов: унарные, бинарные и *n*-арные (*n*>2).

Унарные операторы – это операторы, которые для вычислений требуют одного операнда, который может размещаться справа или слева от самого оператора. Примеры унарных операторов:

i++

--a

-8

Бинарные операторы – это операторы, которые для вычисления требуют двух операндов. Фрагменты выражений с бинарными операторами +, –, %, * :

a+b

f1-f2

c%d

x1*x2

n-арные операторы для вычислений требуют более двух операндов. В языке C++ есть тернарная операция **?:**, которая для своей работы требует три операнда.

Напишем класс *simple_fraction*, который будет описывать простую дробь с целыми числителем и знаменателем. И определим операторы сложения, вычитания, умножения и деления для этого класса.

```
class simple_fraction
{
public:
   simple_fraction(int numerator, int denominator)
   {
      if (denominator == 0) // Ошибка деления на ноль
         throw std::runtime_error("zero division error");
      this->numerator = numerator;
      this->denominator = denominator;
   }

   // Определение основных математических операций для простой дроби
   double operator+ (int val) { return number() + val; } // Сложение
   double operator- (int val) { return number() - val; } // Вычитание
   double operator* (int val) { return number() * val; } // Умножение
   double operator/ (int val) // Деление
   {
      if (val == 0) {
         throw std::runtime_error("zero division error");
      }
      return number() / val;
   }

   // Получение значения дроби в виде обычного double-числа
   double number() { return numerator / (double) denominator; }
private:
   int numerator; // Числитель
   int denominator; // Знаменатель
};
```

Пример использования класса *simple_fraction*:

```
// Простая дробь 2/3
simple_fraction fr(2, 3);

double sum = fr + 10; // сумма
double diff = fr - 10; // разность
double factor = fr * 10; // произведение
double div = fr / 10; // частное
```

Можно перегрузить оператор сложения для двух простых дробей, который будет возвращать новую простую дробь. Тогда, нам придется привести дроби к общему знаменателю и вернуть другую простую дробь.

На использование перегруженных операторов накладываются следующие ограничения:

– при перегрузке оператора нельзя изменить приоритет этого оператора;

– нельзя изменить количество операндов оператора. Однако, в коде операторной функции можно один из параметров (операндов) не использовать;

– нельзя перегружать операторы *::*, *.*, ***, *?:*;

– нельзя вызвать операторную функцию с аргументами по умолчанию. Исключение – операторная функция вызова функции *operator()()*.

Задание

1. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками: >> перевертывание строки (запись символов в обратном порядке); ++ нахождение наименьшего слова в строке.

2. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками: ++ преобразование символов строки в прописные (заглавные) символы; -- нахождение самого короткого слова в строке.

3. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками: + конкатенация двух строк; ++ преобразование символов строки в строчные (маленькие) символы.

4. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками: - удаление одной строки из другой (если одна строка является подстрокой другой); -- преобразование символов строки в строчные (маленькие) символы.

5. Определить класс список элементов. В определение класса включить два конструктора для определения списка по его размеру и путем копирования другого списка. Определить операции над списком: ++ сортировка списка по возрастанию; -- расположение элементов списка в обратном порядке.

Контрольные вопросы

1. Что такое унарные и бинарные операторы?
2. Для чего нужна перегрузка методов и операторов?
3. Какие функции называются «перегруженными»?
4. Что означает термин «перегрузка»?
5. По каким признакам отличаются перегруженные функции? Пример.
6. Могут ли считаться перегруженными функции, которые имеют одинаковые имена, одинаковое количество и типы параметров, но которые возвращают значение разных типов?
7. Для чего используется перегрузка конструкторов класса? Преимущества перегрузки конструкторов класса.
8. Каким образом осуществляется доступ к перегруженной функции с помощью указателя на функцию? Пример.

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №7
«Наследование. Полиморфизм»

Цель работы: ознакомиться с абстрактными классами и принципами наследования.

Наследование представляет один из ключевых аспектов ООП, который позволяет наследовать функциональность одного класса в другом.

Использование наследования означает, что нам не нужно переопределять информацию из родительских классов в дочерних. Мы автоматически получаем методы и переменные-члены суперкласса через наследование, а затем просто добавляем специфичные методы или переменные-члены, которые хотим. Это не только экономит время и усилия, но также является очень эффективным: если мы когда-либо обновим или изменим базовый класс (например, добавим новые функции или исправим ошибку), то все наши производные классы автоматически унаследуют эти изменения!

В таблице 7.1 можно увидеть какими доступами можно пользоваться при наследовании.

Таблица 7.1 – Доступы наследования

	Модификатор наследования		
Модификатор доступа	**public**	**private**	**protected**
public	public	public	protected
private	Нет доступа	Нет доступа	Нет доступа
protected	protected	protected	protected

Чтобы наследовать класс нужно использовать конструкцию:
class <имя потомка> : <модификатор наследования> <имя родительского класса>{};

В *<модификатор наследования>* можно задать какими модификаторами доступа родительского класса можно будет пользоваться в дочернем.

Рассмотрим пример:

```
class Animals {
  public:
    int counter;  // общее кол животных
  protected:
    int zebras;
    int bears;
    int dogs;

    // функция вычисление общего количества животных
    count_animals() {
      counter = dogs + bears + zebras;
    }
    set_dogs(int count_of_dogs) {
      dogs = count_of_dogs;
    }
};

class Dog : public Animals {
  public:
    int count_dogs() {
      return dogs;  // использовали переменную dog
    }
};
```

Наследованные конструкторы будут вызываться в порядке их наследования. Для наследования конструктора нужно использовать следующую конструкцию:

<имя класса> (<имя переменных конструктора>) : <родительский класс> (<переменные конструктора>) { <тело> };

В начале указываем имя дочернего класса – *<имя класса>*. Далее *<имя переменных конструктора>* указываем столько имен переменных сколько требует этого родительский конструктор, дальше передаем базовому конструктору в скобках (*<переменные конструктора>*) объявленные переменные. *<тело>* – это тело конструктора.

Но чтобы все это работало, в конструкторе базового класса к переменным нужно обращаться через this->.

Пример:

```
class Animal {
  public:
    animal () {
      cout << "Создан класс без первичных объвлений";
    }
    animal (int counter) {
      this->count_of_animal = counter;
    }
  protected:
    int counter;
    int count_of_animal;
};

class Dog : Animal {
  public:
    dog () : animal () {} // перегрузка
    dog (int counter) :   // конструкторов
    animal (counter) {}

    get_count_animal() {
      return count_of_animal;
    }
};
```

Наследованные деструкторы вызываются наоборот по сравнению с вызывом конструктора: сначала вызывается деструктор класса, затем деструкторы элементов класса, а потом деструктор базового класса.

Когда наследование от класса нежелательно, то помощью спецификатора ***final*** можно запретить наследование:

class User final
{};

Наследование методов. Класс потомок наследует все методы базового класса, кроме конструкторов, деструктора и операции присваивания. Не наследуются ни дружественные функции, ни дружественные отношения классов.

Если имя метода в наследнике совпадает с именем метода базового класса, то метод производного класса скрывает все методы базового класса с таким именем.

Виртуальные методы. В C++ реализован механизм позднего связывания, когда разрешение ссылок на функцию происходит на этапе выполнения программы. Этот механизм реализован с помощью **виртуальных методов**. Для определения виртуального метода используется спецификатор ***virtual***:

virtual void draw(int x, int y, int position);

Иногда, когда функция объявляется в базовом классе, она не выполняет никаких значимых действий, поскольку часто базовый класс не определяет законченный тип, а нужен чтобы построить иерархию.

Методы, которые нужны только для того, чтобы их обязательно переопределили в производных классах, называются ч**исто виртуальными методами**. Такие методы не определяются в базовом классе. У них нет тела, а есть только декларации об их существовании.

Чисто виртуальная функция выглядит в описании класса следующим образом:

virtual тип имя_функции (список параметров) = 0;

Как можно заметить, функцию делает чисто виртуальной приравнивание её описания к нулю.

Класс, содержащий хотя бы один чисто виртуальный метод, называется **абстрактным классом**. Поскольку у чисто виртуального метода нет тела, то создать объект абстрактного класса невозможно.

Правила использования виртуальных методов:

– если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным;

– виртуальные методы наследуются;

– если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса с помощью операции доступа к области видимости;

– виртуальный метод не может объявляться с модификатором *static*, но может быть объявлен как дружественная функция;

– если производный класс содержит виртуальные методы, они должны быть определены в базовом классе хотя бы как чисто виртуальные.

Переопределение методов. Переопределение означает, что вы создали иерархию классов, у которой в базовом классе есть виртуальная функция и вы можете переопределить её в производном классе.

Рассмотрим пример:

```
class Base {
public:
  // реализует некоторый алгоритм
  void doSomething() {
    prepare(); // сперва выполнить подготовку

    // основная часть алгоритма не приведена для краткости
  }
private:
  virtual void prepare() {}
};

class Derived : public Base {
private:
  virtual void prepare() {
    // выполнить подготовку
    std::cout << "hello world :)\n";
  }
};

Derived obj;
obj.doSomething();
```

В последнем примере, при вызове функции *doSomething* будет работать функция *prepare* из производного класса *Derived*.

Задание

1. Разработать программу с использованием наследования классов, реализующую классы: графический объект; круг; квадрат. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его размер и координаты.
2. Разработать программу с использованием наследования классов, реализующую классы: железнодорожный вагон; вагон для перевозки автомобилей; цистерна. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его вес и количество единиц товара в вагоне.
3. Разработать программу с использованием наследования классов, реализующую классы: массив; стек; очередь. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран количество элементов и добавьте элемент.
4. Разработать программу с использованием наследования классов, реализующую классы: воин; пехотинец(винтовка); матрос(кортик). Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его возраст и вид оружия.
5. Разработать программу с использованием наследования классов, реализующую классы: точка; линия; круг. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран координаты и размер.
6. Разработать программу с использованием наследования классов, реализующую классы: работник больницы; медсестра; хирург. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран возраст и название должности.
7. Разработать программу с использованием наследования классов, реализующую классы: точка; квадрат пирамида. Используя виртуальные функции, не зная с объектом какого класса вы работаете, выведите на экран его размер и координаты.

Контрольные вопросы

1. Как наследуются конструкторы и деструкторы?
2. Что такое чисто виртуальный метод?
3. Что такое абстрактный класс?

Содержание отчета

1) Цель работы;
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

ЛАБОРАТОРНАЯ РАБОТА №8
«Обработчики исключений»

Цель работы: ознакомиться с обработчиками исключений в языке C++.

Исключительная ситуация (англ. "exception") или **исключение** – это что-то особенное, случившееся в работе программы, это какая-то возникшая ошибка, но не обязательно: например, это может быть нестандартное стечение обстоятельств или просто ситуация, требующая нетиповой обработки.

Если в программе (или в библиотечной функции, которую использует программа) возникает какая-то неразрешимая ситуация, то генерируется исключение. Это означает, что вместо нормального продолжения программы управление передаётся на другую ветку алгоритма, специально предназначенную для обработки такой исключительной ситуации. Эта другая ветвь программы – обработчик исключения – может просто прервать программу, а может что-то изменить, чтобы программа могла продолжить выполнение. Причём, даже если исключение возникает в библиотечной функции, обработчик исключения всё равно находится в программе, использующей эту библиотеку – в той части кода, которая непосредственно вызвала конкретную нестандартную ситуацию и потому лучше может справиться с её обработкой.

Типы ошибок, которые могут возникать в программах

В программах на C++ могут возникать ошибки. Различают три типа ошибок, которые могут возникать в программах:

- **синтаксические**. Это ошибки в синтаксисе языка C++. Они могут встречаться в именах операторов, функций, разделителей и т.д. В этом случае компилятор определяет наличие синтаксической ошибки и выдает соответствующее сообщение. В результате исполняющий (*.exe) файл не создается, и программа не выполняется;
- **логические**. Это ошибки программиста, которые сложно обнаружить на этапе разработки программы. Эти ошибки обнаруживаются на этапе выполнения во время тестирования работы программы. Логические ошибки можно обнаружить только по результатам работы программы. Примером логических ошибок может быть неправильная работа с указателями в случаях выделения/освобождения памяти;
- **ошибки времени выполнения**. Такие ошибки возникают во время работы программы. Ошибки времени выполнения могут быть логическими ошибками программиста, ошибками внешних событий (например, нехватка оперативной памяти), неверным вводом данных пользователем и т.п. В результате возникновения ошибки времени выполнения, программа приостанавливает свою работу. Поэтому, важно перехватить эту ошибку и правильно обработать ее для того, чтобы программа продолжила свою работу без остановки.

Примеры действий в программе, которые могут привести к возникновению исключительных ситуаций:

1. деление на нуль;
2. нехватка оперативной памяти при использовании оператора new для ее выделения (или другой функции);
3. доступ к элементу массива за его пределами (ошибочный индекс);
4. переполнение значения для некоторого типа;
5. взятие корня из отрицательного числа;
6. другие ситуации.

Чтобы исключение, сгенерированное в одном блоке кода, могло найти нужный обработчик, находящийся в другом блоке, при генерации исключения выбрасывается

индикатор исключения. Индикатором служит объект или переменная некоторого конкретного типа. При возникновении исключения будет выбран тот обработчик, в описании которого указан тот же тип ожидаемого индикатора. Обработчики различают исключения по типам данных индикатора, и поэтому в наиболее распространённом случае в качестве индикатора указывают объект некоторого класса, специально предусмотренного для этой цели.

Чтобы использовать исключения, нужно поместить вызов кода, в котором исключение может возникнуть, в специальный блок try{}. Следом за этим блоком должен следовать блок *catch*(){}, внутрь которого помещают код, обрабатывающий исключительную ситуацию. Например:

```
f ( )
{
    //генерируем исключение, если возникла соответствующая ситуация
    if (.... ) throw индикатор;
.....
}
.....
try
{
    //вызываем код, который может сгенерировать исключение:
    f ( );
}
catch (индикатор)
{
    //обрабатываем исключение
.....
}
```

Обработчик исключения всегда следует сразу за блоком *try*{}. В круглых скобках после *catch* указывается индикатор исключения, которое этот блок обрабатывает. Чтобы сгенерировать исключение, нужно использовать специальную конструкцию throw, после которой указывается индикатор.

В следующем примере объявим два пустых класса для использования в качестве индикаторов: класс *unknown_exception*, означающий получение неизвестного ответа от пользователя, и класс *abort_exception*, сигнализирующий о необходимости немедленного выхода из программы. Сама программа задаёт пользователю вопрос о выполнении последовательно 100 неких абстрактных пронумерованных команд. Диалог реализуется функцией *confirm*(), спрашивающей у пользователя подтверждение на выполнение команды с заданным номером и анализирующей ответ ("y"– подтверждение, "n" – отказ, "a" – немедленный выход):

```
#include <iostream>
#include <math.h>
using namespace std;
class unknown_except ion { };
class abort_ exception { };
bool confirm ( int i )
{
char c;
cout << "Подтвердите команду " << i << " ( y / n / a/да/нет/выход) : ";
cin >> c;
cin.ignore ( ); //очищаем буфер если введены лишние символы
switch ( c ) {
case " y " : return true;
case " n " : return false;
```

```
case " a " : throw abort_exception ( );
default : throw unknown_exception ( );
}
}
main ( )
{
cout << "Демонстрация диалога подтверждения при выполнении"<<" 100 команд\n ";
for ( int i =1; i <=100; i++) {
try{
        if ( confirm ( i ) ) cout << "КОМАНДА "<< i <<" ВЫПОЛНЕНА\n ";
        else cout << "КОМАНДА " << i << " ОТМЕНЕНА\n ";
}
catch ( unknown_exception ) {
cout << "Неизвестный ответ пользователя\n ";
i --; // возвращаемся к предыдущей команде
}
catch ( abort_exception ) {
cout << "Выполняется немедленный выход из программы\n ";
return 0;
}
cout << "Продолжение демонстрации диалога\n ";
}
}
```

Из примера видно, что обработчики исключений должны следовать друг за другом каждый в своём блоке *catch*(). После того, как отработает один из обработчиков, управление передастся на код, следующий за последним блоком *catch*() в данной цепочке.

Обратите внимание, что в блоке *catch*() указан в качестве параметра только тип данных – класс-индикатор. Это допустимо с учётом того, что обработчик исключения не собирается извлекать никаких данных из переданного индикатора, да и сами классы-индикаторы, созданные в программе, являются пустыми и используются только для того, чтобы различать исключительные ситуации.

Классы-индикаторы исключения могут принадлежать к общей иерархии наследования, т.е. быть в отношениях "родитель-потомок". При этом обработчики индикаторов-родительских классов могут перехватывать исключения с индикаторами-потомками (можно считать такое поведение проявлением полиморфизма). Поэтому родительские обработчики нужно обязательно указывать после дочерних в цепочке блоков *catch* – иначе дочерний обработчик никогда не получит управление. В самом конце цепочки можно указать *catch*, у которого в круглых скобках вместо индикатора три точки. Такой блок будет перехватывать абсолютно любые исключения:

```
class general_error { };
class out_of_range : public general_error { };
.............. .
try {.............}
catch ( out_of_range )
{ cout << "Выход индекса за границу массива\n "; }
catch ( general_error )
{ cout << "Общий сбой в работе программы\n "; }
catch (...) { cout << "Неизвестная ошибка\n "; }
```

В приведённом примере объявлено два различных класса- индикатора, один базовый, для исключений общего типа, и один производный от него, для исключительной ситуации типа "недопустимый индекс при обращении к в массиву". Если бы порядок следования

обработчиков был другим, обработчик индикатора *out_of_range* никогда не смог бы активироваться.

Если обработчик перехватил исключение, но обнаружил, что не сможет справиться с его обработкой, он может вызвать *throw* без аргументов: это передаст исключение дальше по цепочке уровней вложенности, на случай если на более высоком уровне есть обработчик, способный так или иначе решить возникшую ситуацию.

Если некоторая функция содержит инструкции *throw*, в её заголовке можно явно прописать, какие исключения она может генерировать:

void f() throw (x,y,z);

Функция *f*() может генерировать исключения с классами- индикаторами *x, y, z* (и производными от них). Если заголовок описан как "*void f() throw ()*" – функция не генерирует исключений вообще. Если в заголовке функции ничего не указано, она может генерировать любые исключения (без этого последнего правила не смогли бы работать программы, не использующие обработку исключений).

Если функция попытается сгенерировать исключение, отсутствующее в её списке – вызовется специальная функция *void unexpected*(), которая в свою очередь вызовет фунцкию *void terminate*(), а та вызовет функцию *abort*(), аварийно завершающую программу. Программист имеет возможность заменить функцию *unexpected*() или функцию *terminate*() – или сразу обе – на свои собственные, изменив таким образом обработку неспецифицированных исключений. Для такой замены нужно вызвать специальные библиотечные функции *set_unexpected*() и *set_terminate*(), передав им адреса новых функций в качестве аргументов.

Задание

1. Переопределите оператор ++ для указателя на массив целых, обработайте ошибку выхода за границы массива.
2. Опишите функцию, возвращающую день недели по дню и месяцу, обработайте ошибки неверного дня или месяца.
3. Опишите функцию анализа номера телефона, обработайте ошибку задания номера в неверном формате (допустимый формат - +7(095)555-44-33).
4. Опишите оператор [] для списка элементов, обработайте ошибку выхода за границы массива.

Контрольные вопросы

1. Что такое исключительная ситуация?
2. Какие типы ошибок могут возникать в программе?
3. Какую цель преследует использование в программе обработки исключений?
4. Как оформляется блок обработки исключений?
5. Что такое обработчики исключений?

Содержание отчета

1) Цель работы
2) Подробное описание всех этапов проделанной работы;
3) Анализ проделанной работы;
4) Листинг программы;
5) Выводы по данной лабораторной работе.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Kvodo Computing Science & Discrete Match [Электронный ресурс]. – Режим доступа: https://kvodo.ru/uslovnyie-operatoryi-if-i-switch.html.
2. Метанит [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/2.13.php.
3. Code-Live [Электронный ресурс]. – Режим доступа: https://code-live.ru/post/cpp-loops/ .
4. Ravesli [Электронный ресурс]. – Режим доступа: https://ravesli.com/urok-74-massivy-chast-1/ .
5. Метанит [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/4.1.php .
6. Метанит [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/4.12.php .
7. CppStudio [Электронный ресурс]. – Режим доступа: http://cppstudio.com/post/446/ .
8. CppStudio [Электронный ресурс]. – Режим доступа: http://cppstudio.com/post/8453/ .
9. CodeLessons [Электронный ресурс]. – Режим доступа: https://codelessons.ru/cplusplus/nasledovanie-klassov-v-c-chto-eto-i-kak-on-rabotaet.html .
10. BestProg [Электронный ресурс]. – Режим доступа: https://www.bestprog.net/ru/2019/09/14/c-the-concept-of-an-exceptional-situation-instruction-try-catch-operator-throw-examples-of-using-ru/ .