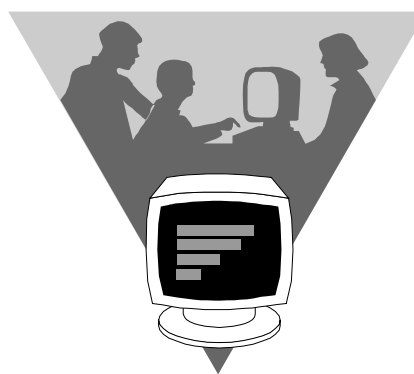
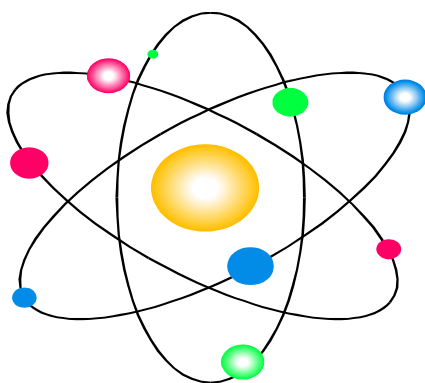


Министерство образования и науки Хабаровского края

Краевое государственное бюджетное образовательное учреждение
дополнительного образования
«Хабаровский краевой центр развития творчества детей и юношества»

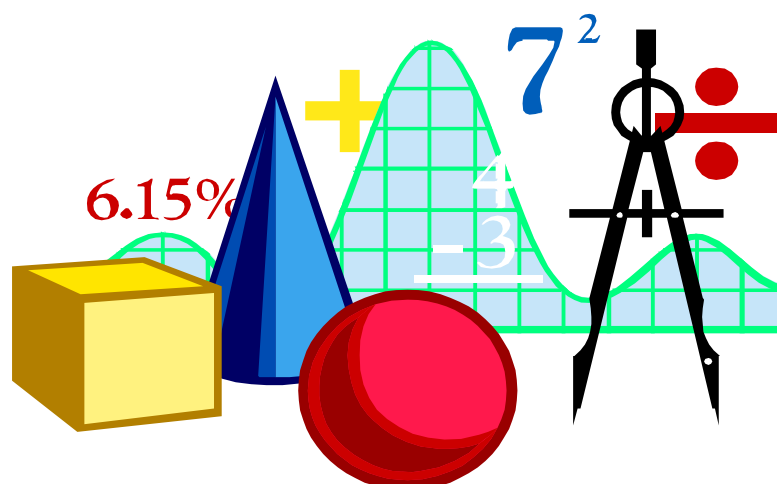
Хабаровская краевая заочная физико-математическая школа



Е.А. Редько, С.В. Летучий, А.В. Крупский

ИНФОРМАТИКА:
методическое пособие
для участников летней (очной) сессии

ВЫПУСК 10



Хабаровск, 2016

ИНФОРМАТИКА: методическое пособие для участников летней (очной) сессии / Е.А. Редько, С.В. Летучий, А.В. Крупский. – Хабаровск: КГБОУ ДО ХКЦРТДиЮ, 2016. – 56 с.

Публикуемые в настоящем сборнике методические материалы разработаны для реализации программы летней (очной) сессии Хабаровской краевой заочной физико-математической школы.

© КГБОУ ДО ХКЦРТДиЮ, 2016

© Е.А. Редько, С.В. Летучий, А.В. Крупский, 2016

СОДЕРЖАНИЕ

Редько Е.А. Работа с элементами графического интерфейса в среде QtCreator	4
Среда визуального программирования Qt	4
Лабораторная работа № 1. Основные понятия визуального программирования в QtCreator.....	6
Лабораторная работа № 2. Работа с кодом программы	10
Лабораторная работа № 3. Менеджеры компоновки	12
Лабораторная работа № 4. Разработка приложения «Арифметика»	17
Варианты темы проектов	19
Летучий С.В. Вычислительные машины изнутри и компьютерные сети.....	20
1. Современная архитектура компьютеров. Машина Фон Неймана	20
2. Архитектурный принцип машины Фон Неймана	21
2.1. Память. Принцип линейности и однородности информации, хранящейся в памяти.....	21
2.2. Принцип неразличимости команд и данных (принцип дуальности информации).....	22
2.3. Шина управления	23
2.4. Принцип автоматической работы	23
2.5. Принцип последовательного выполнения (последовательной работы)	23
Контрольные вопросы и задания по теме «Архитектура»	23
3. Фундамент Internet	24
3.1. Уровни	24
3.2. Обмен данными	25
4. Упаковка.....	26
4.1. Биты, байты и пакеты.....	26
4.2. Инкапсуляция.....	27
4.3. Интерпретация полученных данных	28
5. Шестнадцатеричная система счисления	29
6. Адреса	29
6.1. Физические адреса.....	29
6.2. Выделение IP-адреса	30
Задания для самостоятельной работы по теме «Networksystems».....	31
Крупский А.В. Алгоритмизация и Программирование	32
1. Введение	32
2. Алгоритмы.....	34
3. Среда «Исполнители»	38
4. Примеры решения задач	41

РАБОТА С ЭЛЕМЕНТАМИ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА В СРЕДЕ QT CREATOR

Среда визуального программирования Qt

Qt – кроссплатформенный инструментарий разработки программного обеспечения на языке программирования C++ с пользовательским интерфейсом. Qt впервые вышел в свет в мае 1995 года.

Первыми разработчиками Qt стали Хаавард Норд и Эйрик Чамб-Энг. Они начали разработку в 1991 году, за 3 года до регистрации компании под названием Quasar Technologies. Хаавард проявлял интерес к разработке пользовательских графических интерфейсов на языке C++. В 1990 году Хаавард и Эйрик работали вместе над проектом приложения для баз данных ультразвуковых изображений. Это приложение должно было предоставлять пользовательский графический интерфейс для операционных систем Unix, Macintosh и Windows. Летом этого же года Хаавард предложил создать объектно-ориентированную систему отображения интерфейсов и в 1991 году начал писать графические классы, которые позднее легли в основу Qt. Естественно, проектные решения принимались совместно с Эйриком, которому пришла идея сигналов и слотов в приложении, ставшая главной особенностью Qt.

Концепция сигналов и слотов заключается в том, что компонент (виджет) может посылать сигналы, включающие информацию о событии (например: была нажата кнопка, был введен текст). В свою очередь другие компоненты принимают этот сигнал посредством специализированных функций – слотов. Данная система хорошо подходит для описания пользовательского графического интерфейса.

Преимущества такой системы заключаются в том, что: сигнал можно соединять с неограниченным количеством слотов; слот может принимать сообщения от неограниченного множества сигналов; соединение сигналов и слотов можно производить в любой точке приложения; сигналы и слоты являются механизмами, обеспечивающими связь между объектами. Связь также может выполняться между объектами, которые находятся в различных потоках; при уничтожении объекта происходит автоматическое разъединение всех сигнально-слотовых связей. Это гарантирует, что сигналы не будут отправляться к несуществующим объектам.

Также у данной концепции есть и **недостатки**: сигналы и слоты не являются частью языка C++, поэтому требуются дополнительные действия перед компиляцией; отправка и прием сигналов проходит медленнее, чем обычный вызов функции, применяемый при использовании механизма функций обратного вызова.

Название среды разработки состоит из двух букв Q и t. Буква Q была выбрана в качестве префикса классов, поскольку эта буква имела красивое начертание в шрифте редактора Emacs4, которым пользовался Хаавард, а буква t (toolkit) означала набор инструментариев. Среда Qt лицензируется по лицензиям GNU LGPL5 или GNU GPL6, коммерческая.

Написание исходного кода, редактирование визуальных элементов формы разрабатываемого приложения в Qt осуществляется в приложении среды разработки «QtCreator» (см. Рис. 1).

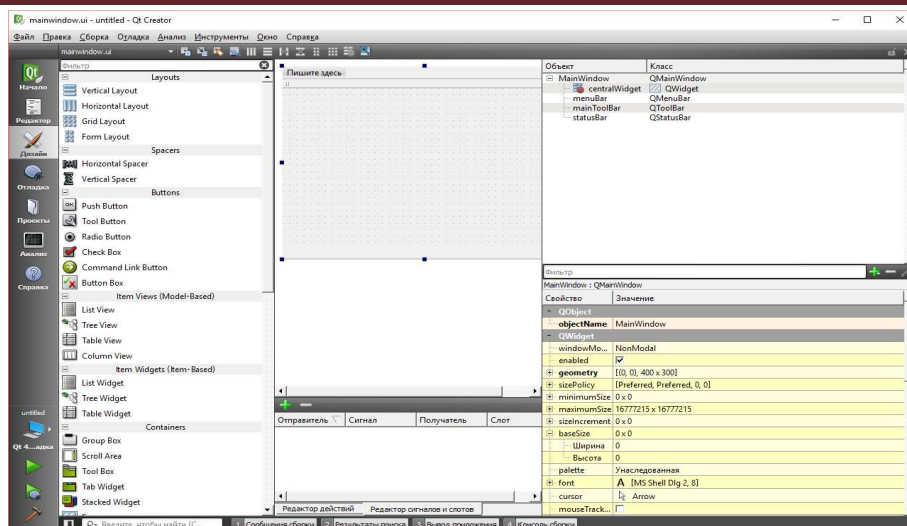


Рис. 1. Окно QtCreator в режиме дизайна

QtCreator включает в себя такие элементы, как:

- **Редактор** – предназначен для редактирования исходного кода разрабатываемого приложения.
- **Дизайнер** – визуальный дизайнер окна формы оконного приложения. В режиме дизайна разработчику становятся доступными *Палитра компонентов* и *Инспектор объектов*.
- **Отладчик** – элемент, позволяющий следить за выполнением программного кода и выявлять ошибки в написании программы.
- **Анализ** – элемент, позволяющий анализировать потребляемые ресурсы приложения и находить неэффективные части исходного кода приложения.

В режиме *Редактора* рассмотрим дерево проекта. Пустой оконный проект состоит из пяти файлов:

- **My_Project.pro** – файл параметров для метаобъектного компилятора среды Qt. В данном файле задаются списки файлов (заголовочные, исходного кода, форм, ресурсных файлов), включенных в проект и настройки компилятора.
- **main.cpp** – главный файл кода приложения, с которого начинается компиляция и выполнения приложения.
- **mainwindow.h** – заголовочный файл формы «MainWindow», содержит объявление класса формы, содержащего основные функции, слоты, сигналы формы.
- **mainwindow.cpp** – файл кода функций, объявленных в классе формы.
- **mainwindow.ui** – файл формата xml, обрабатываемый элементом среды Qt «Дизайн», в окне дизайнера производится редактирование визуальных компонентов приложения.

Разработка внешнего вида (интерфейса) приложения выполняется с помощью виджетов. Этот термин происходит от «windowgadget» и соответствует элементу управления («control») и контейнеру («container») по терминологии Windows.

Каждый виджет может настраиваться в среде QtCreator или вручную посредством изменения его свойств. С помощью свойств можно указать размеры виджетов, их расположение, особенности внешнего вида и др.

Свойства виджетов в QtCreator доступны через окно Инспектора объектов, но их можно изменять и во время работы программы.

В качестве примера рассмотрим следующие свойства:

- **boolvisible** – видимость виджета и, соответственно, всех его подчиненных виджетов; проверка свойства реализуется функцией `boolisVisible()`; а изменение – процедурой `voidsetVisible(boolvisible)`;

– **boolenabled** – способность принимать и обрабатывать сообщения от клавиатуры и мыши: true – способно, false – нет; проверка свойства реализуется функцией `boolisEnabled()`; а изменение – процедурой `voidsetEnabled(boolenabled)`;

– **Qt::WindowModalitywindowModality** – тип окна: `Qt::nonModal` (обычное), `Qt::WindowModal` (модальное); проверка свойства реализуется функцией `Qt::WindowModalitywindowModality ()`; а изменение – процедурой `voidsetWindowModality (Qt::WindowModalitywindowModality)`;

– **QRectgeometry** – размеры и положение виджета относительно родительского окна; размеры задаются прямоугольником типа `QRect` с фиксированным верхним левым углом (свойства `X,Y`), а также шириной и высотой (свойства `width,height`); при изменении размера формы размеры виджетов могут регулироваться компоновщиком в интервале от заданных минимального `minimumSize()` до максимального `maximumSize()`; получение значения осуществляют с помощью функции `QRect&geometry()`, изменение значений процедурами `voidsetGeometry(intx,inty,int w, int h)` или `voidsetGeometry(QRect&)`;

– **QFontfont** – шрифт, которым выполняются надписи в окне;

– **QStringobjectName** – имя объекта (переменной) в программе, устанавливается процедурой `voidsetObjectName()`, читается функцией `objectName()` и используется для задания имени переменной в QtCreator и при отладке программ.

Лабораторная работа № 1. Основные понятия визуального программирования в QtCreator

Цель: Знакомство с базовыми понятиями среды QtCreator, получение навыков создания нового проекта с визуальными компонентами, сохранения, запуска, размещения новых визуальных компонентов на форме.

Что такое виджет? Любой визуальный элемент графического интерфейса пользователя. Этот термин происходит от «windowgadget» и соответствует элементу управления («control») и контейнеру («container») по терминологии Windows.

Примеры виджетов: кнопка (класс `QPushButton`); метка (класс `QLabel`); поле ввода (класс `QLineEdit`); числовое поле-счетчик (класс `QSpinBox`); строка прокрутки (класс `QScrollBar`).

В Qt есть около 50 готовых классов графических элементов, доступных для использования. Родительским классом для всех виджетов является класс `QWidget`: от него наследуются все главные свойства визуальных элементов.

Задание 1. Создание приложения в виде пустого окна

1) Запустите QtCreator. Вид ярлыка программы можно посмотреть на Рис.2.



Рис.2. Ярлык программы QtCreator

2) Создайте проект. Для этого выберите команду (см. Рис. 3) *File (Файл) → New (Новый) → Qt4 GuiApplication (GUI приложение Qt4)*

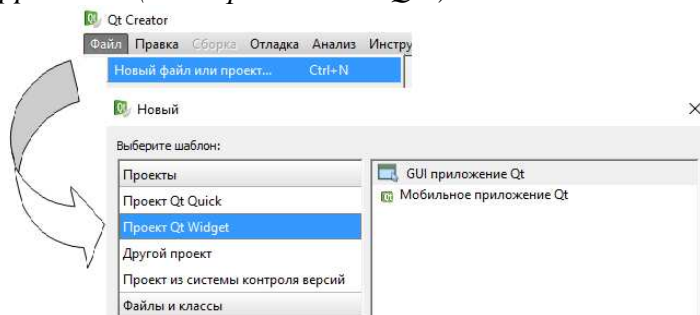


Рис. 3. Вид меню и диалога выбора типа проекта

3) В поле *Name (Название)* напишите имя папки с проектом (см. Рис. 4), пусть будет *first*. Пишите английскими буквами, иначе могут быть проблемы при компиляции.



Рис. 4. Диалог для ввода названия проекта и выбора места размещения

4) В поле *Create in (Создать в)* введите путь к папке, в которой будет создана папка с именем, которое вы ввели в поле *Name*. Внимание! В пути к папке с проектом тоже не должно быть русских букв!

5) В *Class name (Имя класса)* пишем *MyWidget*, в *Base Class (Базовый класс)* выбираем *QWidget* (см. Рис.5).

Информация о классе

Укажите базовую информацию о классах, для которых желаете создать шаблоны файлов исходных текстов.

Имя класса:	MyWidget
Базовый класс:	QWidget
Заголовочный файл:	mywidget.h
Файл исходников:	mywidget.cpp
Создать форму:	<input checked="" type="checkbox"/>
Файл формы:	mywidget.ui

Рис.5. Диалог для именованния класса

6) В результате создания проекта в окне редактора кода среды QtCreator вы увидите иерархическую структуру файлов проекта и текст файла *mywidget.cpp* (см. Рис.6). Сохраните проект, для этого выберите команду *File (Файл) → Save All (Сохранить все)*. После каждого этапа сохраняйтесь.

```

1  #include "mywidget.h"
2  #include "ui_mywidget.h"
3
4  MyWidget::MyWidget (QWidget *parent) :
5      QWidget(parent),
6      ui(new Ui::MyWidget)
7  {
8      ui->setupUi(this);
9  }
10
11 MyWidget::~MyWidget()
12 {
13     delete ui;
14 }
15

```

Рис.6. Текст файла *mywidget.cpp*

7) Скомпилируйте программу, для этого выберите команду *Build (Сборка) → Build All (Собрать все)*, также вместо этого можно нажать на кнопку с молотком в левом нижнем углу окна QtCreator или комбинацию клавиш *CTRL+B*.

8) Запустите программу, для этого выберите команду *Build (Сборка) → Run (Выполнить)*, также вместо этого можно нажать на кнопку с треугольником в левом нижнем углу окна QtCreator или комбинацию клавиш *CTRL+R*.

9) В результате будет запущено пустое окно (см. Рис.7).

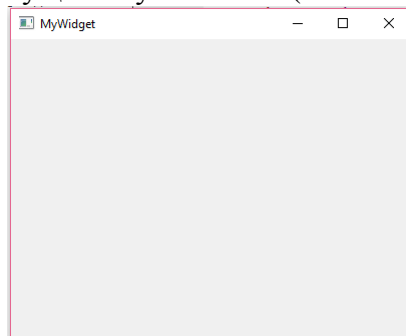
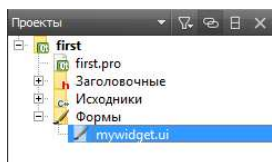


Рис.7. Вид готового окна приложения

Задание 2. Добавление виджетов и редактирование их свойств в инспекторе объектов

Виджеты – визуальные объекты на форме, например кнопки, поля, меню.

В этой задаче, а также во всех последующих задач лабораторной работы № 1, будет использоваться один и тот же проект. Если вы уже успели закрыть проект, то его надо сначала открыть. Чтобы это сделать, выберите команду *File (Файл) → Open (Открыть)*, в появившемся окне откройте файл *.pro вашего проекта.



1) Отредактируем форму, для этого надо сначала перейти в режим редактирования проекта, чтобы это сделать, необходимо воспользоваться окном файловой структуры, в котором откроем файл `mywidget.ui` двойным кликом на имени файла формы (см. Рис. 8).

Рис. 8. Файловая структура проекта

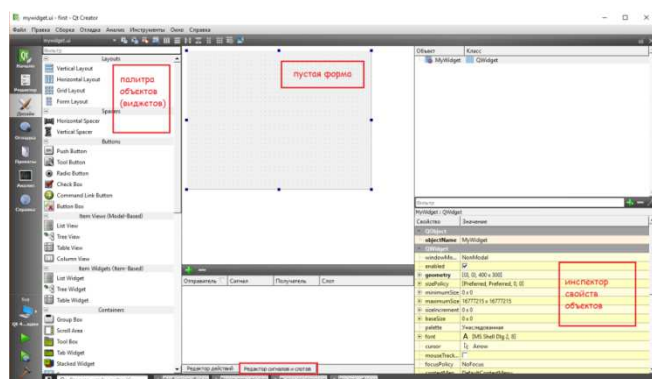


Рис.9. Вид окна QtCreator в режиме конструирования

2) Добавим на форму два поля редактирования (см. Рис. 10). Для этого перетащим с палитры компонентов на форму два Line Edit (виджет Line Edit находится на вкладке InputWidgets)

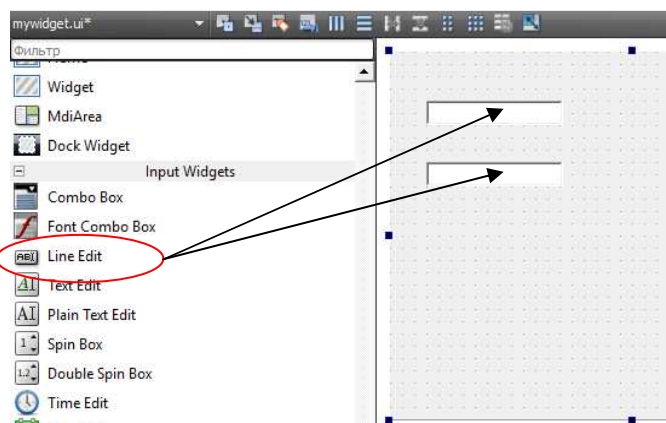


Рис. 10. Размещение визуальных компонентов Line Edit на форме

3) Добавим на форму две кнопки (см. Рис. 11), для этого перетащим с палитры компонентов на форму два Push Button (виджет Push Button находится на вкладке Buttons)

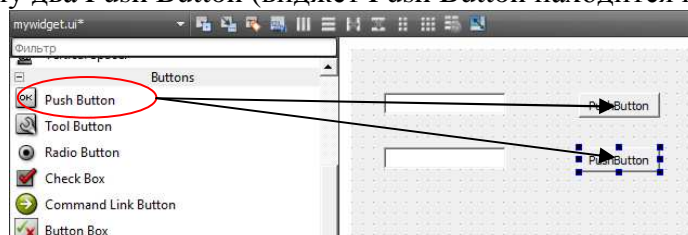


Рис. 11. Размещение визуальных компонентов Push Button на форме

4) Отредактируем имена кнопок и текстовых полей. Для этого выделите кнопку и в инспекторе объектов (см.

5) Рис. 12) отредактируйте свойство `objectName`, пусть оно будет `MyPushButton1`. Таким же образом отредактируйте имя второй кнопки, пусть оно будет `MyPushButton2`. Таким же образом отредактируйте имена Line Edit. Пусть они будут `MyLineEdit1` и `MyLineEdit2`.

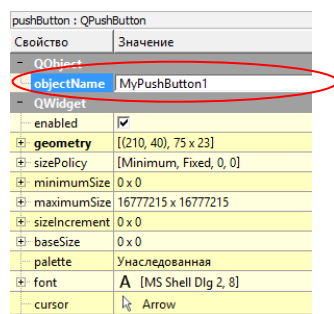


Рис. 12. Инспектор объектов со свойствами

кнопки Push Button

6) Отредактируем надписи на кнопках и полях редактирования. Для этого выделите кнопку `MyPushButton1` и в инспекторе объектов измените свойство `text`, пусть оно будет `Copy`. Таким же образом измените свойство `text` кнопки `MyPushButton2`, пусть оно будет `Clear`. Таким же образом отредактируйте свойство `text` поля `MyLineEdit1`, пусть оно будет `Source`. Таким же образом отредактируйте свойство `text` поля `MyLineEdit2`, пусть оно будет `Destination`.

7) Убедитесь, что все правильно сделали. Посмотрите на форму (см.

8)

9) Рис. 13): на одной кнопке должна быть надпись `Copy`, на другой – `Clear`. В одном поле редактирования должен быть текст `Source`, в другом – `Destination`.

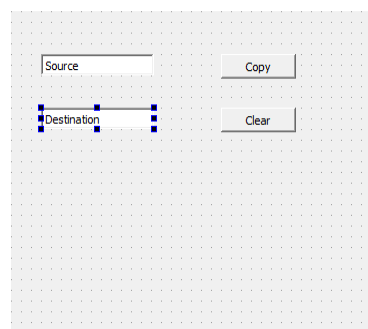


Рис. 13. Вид формы с компонентами после изменения свойства Text

10) Выполняем сохранение проекта (CTRL+SHIFT+S).

11) Компилируем программу (CTRL+B), запускаем проект на исполнение (CTRL+R).

Подведем итоги: Мы научились изменять свойства компонентов при помощи *objectinspector*. Научились переименовывать компоненты (имя – это идентификатор, по которому в дальнейшем к объекту можно обращаться программно) и менять надписи на них (надпись – это визуальное свойство, отвечающее за внешний вид виджета).

Задание для самостоятельной работы. Изучите свойства компонентов и составьте отчет в виде таблицы 1.

Таблица 1. Свойства компонентов

Виджет	Свойство	Тип значения	Назначение (за что отвечает?)
Push Button	Enabled		
	Font		
	Cursor		
Radio Button	Checked		
Text Label	frameShape		
	frameShadow		
	aligment		
	wordWrap		
MyWidget	windowTitle		

Задание 3. Изменение свойств виджетов из кода

1) Добавим на форму метку. Для этого из палитры компонентов перетащим на форму компонент Label (он находится на вкладке DisplayWidgets). Переименуем его, для этого в инспекторе объектов в свойство `objectName` напишем `MyLabel`.

2) Увеличим нашу метку, чтобы в нее вмещался более длинный текст. Для этого схватим ее край мышкой и потянем вправо или влево, чтобы она стала шире.

3) Изменим из кода надпись на MyLabel. Для этого откроем файл mywidget.cpp. Там, в конструкторе класса MyWidget, после строки:

```
ui->setupUi(this);
```

добавим строчку: `ui->MyLabel->setText("My first programm");`

4) После этого файл mywidget.cpp должен выглядеть так:

```
#include "mywidget.h"
```

```
#include "ui_mywidget.h"
```

```
MyWidget::MyWidget(QWidget *parent)
```

```
    : QWidget(parent), ui(new Ui::MyWidget)
```

```
{
```

```
    ui->setupUi(this);
```

```
    ui->MyLabel->setText("My first programm");
```

```
}
```

```
MyWidget::~MyWidget()
```

```
{
```

```
    delete ui;
```

```
}
```

5) Скомпилируйте программу.

6) Запустите программу и проверьте надпись на метке.

Лабораторная работа № 2. Работа с кодом программы

Цель: Изучить систему сигналов-слотов; научиться программно обращаться к свойствам объектов, настраивать «поведение» объектов.

Сигналы и слоты используются для коммуникации между объектами. Механизм сигналов и слотов – главная особенность Qt. Сигнал вырабатывается, когда происходит определенное событие. Слот – это функция, которая вызывается в ответ на определенный сигнал. Виджеты Qt имеют много предопределенных сигналов и слотов, но мы всегда можем сделать дочерний класс и добавить наши сигналы и слоты в нем.

Как связать сигнал со слотом?

При помощи функции `QObject::connect()`. Функция `QObject::connect()` выглядит следующим образом: `QObject::connect (отправитель, SIGNAL (сигнал (список параметров сигнала)), получатель, SLOT (список параметров сигнала))`; где отправитель и получатель являются указателями на объекты и где сигнал и слот являются членами этих объектов. Список параметров сигнала пишется без имен переменных. `SIGNAL()` и `SLOT()` – макросы.

Следует обратить внимание на то, что сигнал и слот должны иметь одинаковые параметры!

Примечание: Если вызов происходит из класса, унаследованного от QObject, тогда QObject:: можно опустить, то есть написать так: connect (отправитель, SIGNAL (сигнал (список параметров сигнала)), получатель, SLOT (список параметров сигнала)).

Задание 1. Настраиваем поведение объектов формы

1) Продолжаем работу в проекте first.

2) Сделаем так, чтобы при нажатии на кнопку с надписью Clear очищались поля ввода. Для этого откроем *Редактор сигналов и слотов* (Signalsandslotseditor), нажмем на «плюс» и в образовавшейся записи, в поле Sender (отправитель сигнала) выберем MyPushButton2. В поле Signal (сигнал) выберем Clicked(). В поле Receiver (получатель сигнала) выберем MyLineEdit1. В поле Slot (слот) выберем Clear(). Еще раз нажмем на «плюс», сделаем так же, как написано выше, только в поле Receiver выберем MyLineEdit2. Сравните свой результат с рис. 1.

Отправитель	Сигнал	Получатель	Слот
MyPushButton2	clicked()	MyLineEdit1	clear()
MyPushButton2	clicked()	MyLineEdit2	clear()

Рис. 14. Редактор сигналов и слотов

3) Скомпилируйте программу (CTRL+B).
 4) Запустите программу (CTRL+R), нажмите на кнопку Clear. Оба поля должны очиститься.

5) Подведем итоги: Мы научились в окне *редактора сигналов и слотов* (Signalsandslotseditor) подключать стандартные сигналы к стандартным слотам. Этим способом можно настроить поведение формы, не написав ни одной строчки кода. Это выгодно отличает Qt от C++ Builder и Delphi. Например, чтобы очищать поля по нажатию кнопки в Delphi или C++ Builder, нам бы пришлось создавать обработчик нажатия кнопки, а в нем писать код, который будет очищать текстовые поля. А в Qt мы можем это сделать, не загромождая «исходник» лишними обработчиками событий.

Задание 2. Создаем обработчики событий

1) Создадим прототип своего слота, для этого откроем файл mywidget.h. В конец класса добавим: publicslots: voidMyEventHandler();

Прототипы слотов надо размещать в разделе *publicslots*, если вам нужен публичный слот, или в разделе *privateslots*, если нужен приватный слот и т.д. После этого файл mywidget.h должен выглядеть так (Рис.):

```

1  #ifndef MYWIDGET_H
2  #define MYWIDGET_H
3
4  #include <QWidget>
5
6  namespace Ui {
7      class MyWidget;
8  }
9
10 class MyWidget : public QWidget
11 {
12     Q_OBJECT
13
14 public:
15     explicit MyWidget(QWidget *parent = 0);
16     ~MyWidget();
17
18 private:
19     Ui::MyWidget *ui;
20
21     public slots: void MyEventHandler();
22 };
23
24 #endif // MYWIDGET_H
25
  
```

Рис. 2. Добавление заголовка функции-слота

Макрос Q_OBJECT необходимо писать в начале определения любого класса, содержащего сигналы или слоты. В наш «исходник» этот макрос уже был добавлен автоматически.

2) Создадим тело слота, для этого откроем файл mywidget.cpp. В его конец добавим:

```

void MyWidget::MyEventHandler()
{
    ui->MyLineEdit2->setText(ui->MyLineEdit1->text());
}
  
```

3) Свяжем сигнал со слотом, для этого в конец конструктора MyWidget (в файле mywidget.cpp) добавим:

```

QObject::connect(ui->MyPushButton1,
    SIGNAL(clicked()), this, SLOT(MyEventHandler()));
  
```

После всех изменений файл `mywidget.cpp` должен выглядеть так (Рис.):

```

1 #include "mywidget.h"
2 #include "ui_mywidget.h"
3
4 MyWidget::MyWidget(QWidget *parent) :
5     QWidget(parent),
6     ui(new Ui::MyWidget)
7 {
8     ui->setupUi(this);
9     ui->MyLabel->setText("My first program!");
10    QObject::connect(ui->MyPushButton1,
11                    SIGNAL(clicked()), this, SLOT(MyEventHandler()));
12
13 }
14
15 MyWidget::~MyWidget()
16 {
17     delete ui;
18 }
19 void MyWidget::MyEventHandler()
20 {
21     ui->MyLineEdit2->setText(ui->MyLineEdit1->text());
22 }
23

```

Рис. 3. Описание тела функции слота и связывание сигнала со слотом

- 4) Компилируем программу (CTRL+B).
- 5) Запускаем программу (CTRL+R), в поле source вводим любой текст и нажимаем кнопку Copy. Текст из поля source должен скопироваться в поле Destination.
- 6) Подведем итоги: Мы научились создавать свой собственный слот и связывать его с сигналом.

Лабораторная работа № 3. Менеджеры компоновки

Цель: Изучить компоненты классов `QHBoxLayout`, `QVBoxLayout` – горизонтальный и вертикальный менеджеры компоновки, отвечающие за размещение и масштабирование подчиненных виджетов в окне приложения.

Общие сведения:

При создании окна приложения на базе любого из классов окон возникает проблема управления расположением окон виджетов в окне приложения. Qt предусматривает два способа решения этой проблемы: задание координат каждого виджета вручную, например посредством метода `setGeometry()`; использование специальных невидимых пользователю менеджеров компоновки.

В первом варианте при изменении размеров окна приложения пересчет геометрических параметров виджетов должен выполняться вручную.

Что такое менеджер компоновки?

Это объект, который устанавливает размер и положение виджетов, располагающихся в зоне его действия. Qt имеет три основных класса менеджеров компоновки:

- `QHBoxLayout` размещает виджеты по горизонтали слева направо (или справа налево, в зависимости от культурных традиций);
- `QVBoxLayout` размещает виджеты по вертикали сверху вниз;
- `QGridLayout` размещает виджеты в ячейках сетки.

Задание 1. Размещение виджетов вручную, без компоновщика.

- 1) Создайте новый проект с названием *primer3_1.pro*.
- 2) В иерархической структуре проекта откройте файл *main.cpp*.
- 3) В разделе «директив препроцессору» подключите библиотеку *qlineedit.h* (см. Рис. 15, строка 3)
- 5) В тексте главной функции после объявления переменной *w* класса *Window* (название класса может отличаться, оно задавалось на этапе создания проекта) динамически

создаем два поля ввода *edit1* и *edit2* и устанавливаем их геометрические параметры вручную (см.

6) Рис. 15, строки 11, 12)

```

main.cpp
1 #include <QtGui/QApplication>
2 #include "window.h"
3 #include "qlineedit.h"
4
5 int main(int argc, char *argv[])
6 {
7     QApplication a(argc, argv);
8     Window w;
9     QLineEdit *edit1=new QLineEdit("Edit1", &w);
10    QLineEdit *edit2=new QLineEdit("Edit2", &w);
11    edit1->setGeometry(20,20,60,60);
12    edit2->setGeometry(120,20,60,60);
13
14    w.show();
15
16    return a.exec();
17 }

```

Рис. 15. Скан файла main.cpp – динамическое создание виджетов и их геометрических параметров

7) Проведите сборку проекта (CTRL+B) и запуск программы (CTRL+R).

8) В результате на экране появляется окно, содержащее оба однострочных редактора (см. Рис. 16, а).

Однако на рисунке 2, б видно, что попытка уменьшения размера окна приводит к нарушению внешнего вида, в результате которого виджеты вообще могут исчезнуть из поля зрения.

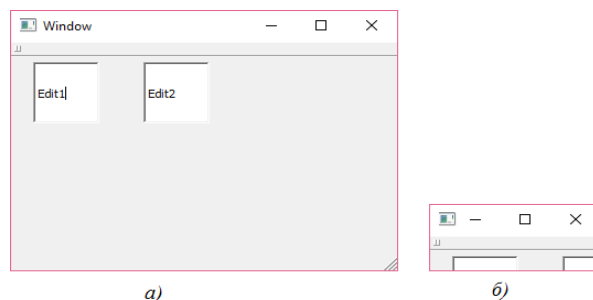


Рис. 16. Вид окна с двумя редакторами QLineEdit в нормальном (а) и свернутом (б) состоянии

Следовательно, при ручной компоновке пришлось бы программировать, как должен изменяться внешний вид окна при изменении его размеров.

В отличие от ручного варианта, при компоновке с использованием менеджеров компоновки осуществляется автоматическая перестройка внешнего вида окна в зависимости от его размеров.

Для управления размещением виджетов на форме QtDesigner использует компоновщики. Для добавления компоновщика компонуемые виджеты должны быть выделены, что можно сделать, щелкая мышкой по виджетам при нажатой клавише CTRL.

Добавление компоновщиков и связывание с ними виджетов осуществляется выбором пунктов меню:

- Tools/Form/Layout Horizontally – компоновать по горизонтали (CTRL+H),
- Tools/Form/Layout Vertically – компоновать по вертикали (CTRL+V),
- Tools/Form/Layout in a Grid – компоновать по сетке (CTRL+G),

или нажатием соответствующих кнопок на панели компонентов дизайнера:



Обратите внимание, что связывание *главного компоновщика с окном* происходит, если выбрать горизонтальную или вертикальную компоновку *при отсутствии выделенных виджетов*.

Задание 2. Знакомство с компоновщиками в режиме конструирования формы

1) Откройте ваш проект *first.pro*.

- 2) Выделите одновременно два поля ввода *LineEdit* (используя зажатую клавишу CTRL):

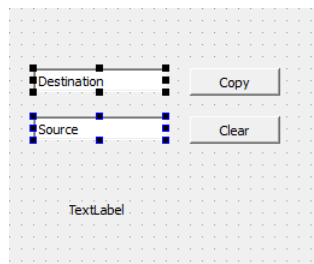


Рис. 17. Одновременное выделение двух виджетов

- 3) Выберите действие «Скомпоновать по горизонтали» в редакторе форм (или CTRL+H):

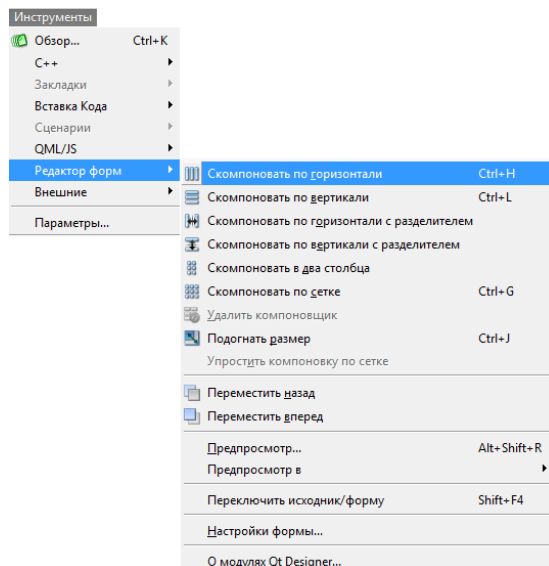


Рис.18. Компоновка виджетов через редактор форм

- 4) Аналогичным образом скомпонуйте две кнопки *Push Button*.
5) Окно приложения (при отсутствии выбранных виджетов) компоуем по вертикали (CTRL+V). Сравните результат (см. Рис.19)

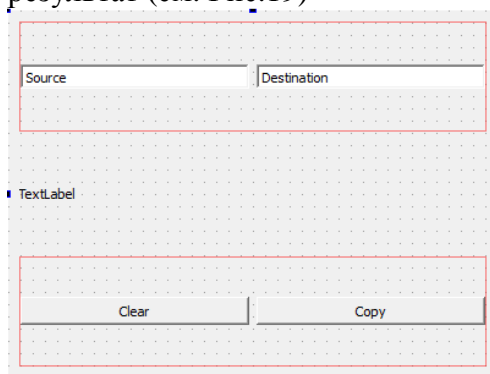


Рис.19. Окно формы, в котором виджеты скомпонованы

- 6) Скомпилируйте проект и запустите его. Подцепите край окна мышкой и измените размер окна. Вы увидите, что размеры кнопок и полей ввода изменяются автоматически.

- 7) Теперь можно объяснить все сделанные действия. Внутри менеджера компоновки виджеты изменяют размер автоматически, но требуется указать главный менеджер компоновки для формы.

Внутри главного менеджера компоновки Вы можете вкладывать другие менеджеры компоновки, комбинировать их, использовать распорки. Вы должны помнить главное: все

дополнительные менеджеры компоновки и распорки должны располагаться внутри *главного* менеджера компоновки.

Задание 3. Создание компоновщиков из кода

Напомним, что в Qt предусмотрены следующие элементы компоновки:

- *QVBoxLayout* – вертикальный компоновщик – управляет расположением виджетов в окне по вертикали;
- *QHBoxLayout* – горизонтальный компоновщик – управляет расположением виджетов в окне по горизонтали;
- *QGridLayout* – табличный компоновщик – управляет расположением виджетов в направляющей двумерной сетке: матрице или таблице.

- 1) Откройте снова проект *primer3_1.pro*.
- 2) В иерархической структуре проекта откройте файл *main.cpp*.
- 3) Объявим переменную *w* класса *QWidget* (центральный виджет – окно) (см. Рис. 20, строка кода 9)
- 4) Подключим библиотеку компоновщиков *qlayout.h* (см. Рис. 20, строка кода 4)
- 5) В тексте главной функции после динамически созданных полей ввода создадим компоновщик *layout* класса *QHBoxLayout* (выравнивание по горизонтали) (см. Рис. 20, строка кода 12).
- 6) Установим значения параметров компоновщика: внешние поля, просвет между виджетами (см. Рис. 20, строки кода 13, 14)
- 7) Свяжем компоновщик *layout* с виджетом окна *w* (см. Рис. 20, строка кода 15)
- 8) Зададим порядок следования элементов внутри компоновщика (см. Рис. 20, строки кода 16, 17)

```

main.cpp main(int, char *[])
1  #include <QtGui/QApplication>
2  #include "window.h"
3  #include "qlineedit.h"
4  #include "qlayout.h"
5
6  int main(int argc, char *argv[])
7  {
8      QApplication a(argc, argv);
9      QWidget w;
10     QLineEdit *edit1=new QLineEdit("Edit1", &w);
11     QLineEdit *edit2=new QLineEdit("Edit2", &w);
12     QHBoxLayout *layout = new QHBoxLayout; // выравнивание по горизонтали
13     layout->setContentsMargins(5, 5, 5, 5); // внешние поля окна
14     layout->setSpacing(5); // просвет между виджетами
15     w.setLayout(layout); // связывание layout с виджетом окна
16     layout->addWidget(edit1); //
17     layout->addWidget(edit2); // задание порядка следования элементов
18
19     w.show();
20
21     return a.exec();
22 }

```

Рис. 20. Скан файла *main.cpp* – динамическое создание горизонтального компоновщика

- 9) В результате работы приложения получаем интерфейс, который при изменении размеров окна сохраняет пропорции (см. Рис. 21).

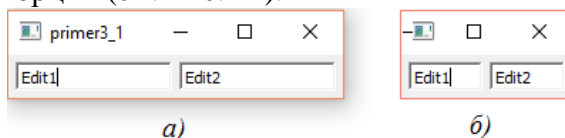


Рис. 21. Внешний вид интерфейса при автоматической компоновке виджетов:
а) в нормальном состоянии; б) в свернутом состоянии

Задание для самостоятельной работы:

1. Изучите элемент вертикальной компоновки.
2. Изучите табличный компоновщик.
3. Изучите действие элемента «пружины».
4. Изучите действие разделителя («ползунка»).

5. Составьте окно с виджетами, скомпонованными различным образом. Объедините все компоновщики с помощью вложения в главный компоновщик окна.

Дополнительная информация:

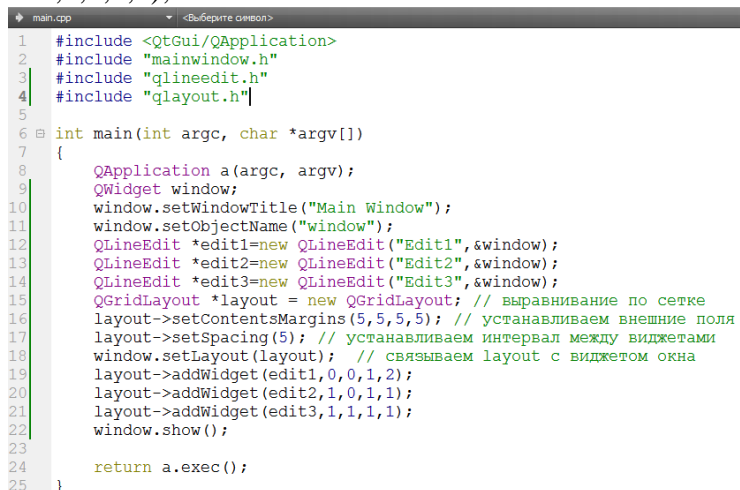
Табличная компоновка предполагает задание координат размещения компонентов с точностью до клетки. При этом допускается размещать виджет в нескольких клетках. Для добавления виджетов в менеджер компоновки используют специальный метод, позволяющий указать область таблицы, которую должен занимать элемент (см. Рис.22).

```
QGridLayout::addWidget(QWidget *widget, // размещаемый виджет
    int fromRow, int fromColumn, // координаты верхней левой ячейки
    int rowSpan, int columnSpan, // количество ячеек по горизонтали и
    // вертикали соответственно
    Qt::Alignment alignment=0); // способ выравнивания
```

Рис.22. Метод добавления виджета в табличный компоновщик

В качестве примера рассмотрим сетку, в которой размещены три поля ввода (см. Рис. 23):

- первое поле занимает две ячейки, начиная с нулевой, в нулевой строке `layout->addWidget(edit1,0,0,1,2);`
- второе поле занимает одну ячейку, начиная с нулевой, в первой строке сетки `layout->addWidget(edit2,1,0,1,1);`
- третье поле ввода занимает одну ячейку, начиная с первой, в первой строке `layout->addWidget(edit3,1,1,1,1);`



```
main.cpp -<Выберите символ>
1 #include <QtGui/QApplication>
2 #include "mainwindow.h"
3 #include "qlineedit.h"
4 #include "qlayout.h"
5
6 int main(int argc, char *argv[])
7 {
8     QApplication a(argc, argv);
9     QWidget window;
10    window.setWindowTitle("Main Window");
11    window.setObjectName("window");
12    QLineEdit *edit1=new QLineEdit("Edit1",&window);
13    QLineEdit *edit2=new QLineEdit("Edit2",&window);
14    QLineEdit *edit3=new QLineEdit("Edit3",&window);
15    QGridLayout *layout = new QGridLayout; // выравнивание по сетке
16    layout->setContentsMargins(5,5,5,5); // устанавливаем внешние поля
17    layout->setSpacing(5); // устанавливаем интервал между виджетами
18    window.setLayout(layout); // связываем layout с виджетом окна
19    layout->addWidget(edit1,0,0,1,2);
20    layout->addWidget(edit2,1,0,1,1);
21    layout->addWidget(edit3,1,1,1,1);
22    window.show();
23
24    return a.exec();
25 }
```

Рис. 23. Пример создания табличного компоновщика и добавления в него виджетов

Для добавления пружины используют метод `addStretch` класса `QBoxLayout`, например `layout->addStretch();`. Эффект использования пружины: размер окон редакторов и расстояние между строчными редакторами минимальны, независимо от размеров окна.

Вместо менеджеров компоновки, которые обычно используют политики пропорционального изменения размеров виджетов при изменении размеров форм, можно использовать разделители `QSplitter`, которые позволяют регулировать размеры виджетов по желанию пользователя.

Разделители бывают вертикальными и горизонтальными. Их применяют в качестве объекта основы окна или его фрагмента (см. Рис.24). В результате форма создается как объект класса `QSplitter`, благодаря чему появляется линия между двумя полями ввода – ползунок, потянув за который, можно изменить соотношение областей, отведенных на каждое поле ввода.


```

1 #include <QtGui/QApplication>
2 #include "mainwindow.h"
3 #include "qsplitter.h"
4 #include "qlineedit.h"
5
6 int main(int argc, char *argv[])
7 {
8     QApplication a(argc, argv);
9     QSplitter splitter(Qt::Horizontal);
10    splitter.setWindowTitle("Main Window");
11    QLineEdit *edit1=new QLineEdit("Edit1",&splitter);
12    QLineEdit *edit2=new QLineEdit("Edit2",&splitter);
13    splitter.show();
14
15    return a.exec();
16 }

```

Рис.24. Создание разделителя splitter (ползунка)

Компоновщики всех рассмотренных типов могут вкладываться один в другой в соответствии с реализуемой схемой окна. Однако при добавлении компоновщика в контейнер другого компоновщика используется не метод *addWidget()*, а метод *addLayout()*, например *layout2->addLayout(layout1)*.

Лабораторная работа № 4. Разработка приложения «Арифметика»

Цель: изучить порядок использования виджетов-переключателей RadioButton, реализующих выбор действия.

1) Выполните проектирование формы в соответствии с образцом (см. Рис. 25). На форме размещены два поля ввода QLineEdit, четыре переключателя RadioButton, кнопка PushButton, метка Label. Для переключателей выполнено вертикальное выравнивание с помощью менеджера компоновки Vertical Layout.

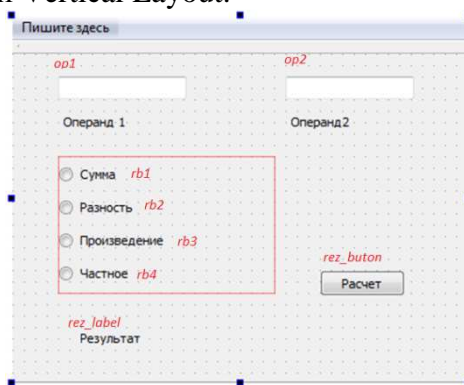


Рис. 25. Форма приложения «Калькулятор»

2) Для виджетов данного приложения необходимо задать следующие имена – свойство name (см. Таблица 2)

Таблица 2. Свойства виджетов для приложения «Калькулятор»

Виджет	Значение свойства name
LineEdit	op1
LineEdit_2	op2
RadioButton	rb1
RadioButton_2	rb2
RadioButton_3	rb3
RadioButton_4	rb4
PushButton	rez_button
Label	rez_label

3) Выполним объектную декомпозицию для данного приложения. Источником сигнала в данном приложении будет кнопка PushButton. По нажатию на кнопку должны происходить следующие действия:

- ✓ считывание данных из полей ввода QLineEdit: op1 и op2;
- ✓ проверка их на допустимые числовые значения;
- ✓ проверка выбранного переключателя RadioButton: rb1->checked, rb2->checked, rb3->checked, rb4->checked;

✓ непосредственно расчет результата и вывод в соответствующую метку rez_label.

4) Прежде чем связывать сигнал со слотом, необходимо написать слот – функцию. Для этого:

4.1. Создаем прототип своего слота. Откроем файл mainwindow.h. В конец класса добавим:

```
publicslots: voidcalc();
```

4.2. Создадим тело слота, для этого откроем файл mainwindow.cpp. В его конец добавим:

```
voidMainWindow::calc()
{
    QStrings1,s2,rez;
    s1=ui->op1->text();
    s2=ui->op2->text();
    boolok1,ok2;
    doublep1,p2,r;
    p1=s1.toDouble(&ok1);
    p2=s2.toDouble(&ok2);
    if(ok1&&ok2)
    {
        if(ui->rb1->isChecked())r=p1+p2;
        if(ui->rb2->isChecked())r=p1-p2;
        if(ui->rb3->isChecked())r=p1*p2;
        if(ui->rb4->isChecked())if(p2!=0)
        r=p1/p2;
    }
    else
    {
        QMessageBox
        msgBox1(QMessageBox::Information,
        "Notcorrect",
        "Divisionbynull!",
        QMessageBox::Ok);
        msgBox1.exec();
    }
    rez.setNum(r);
    ui->rez_label->setText("Rezalt="+rez);
}
else
{
    QMessageBox
    msgBox2(QMessageBox::Information,
    "Notcorrect",
    "Notnumericformat!",
    QMessageBox::Ok);
    msgBox2.exec();
}
}
```

```
mainwindow.cpp
MainWindow::calc()
1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3 #include "qmessagebox.h"
4
5 MainWindow::MainWindow(QWidget *parent) :
6     QMainWindow(parent),
7     ui(new Ui::MainWindow)
8 {
9     ui->setupUi(this);
10    QObject::connect(ui->rez_button,
11        SIGNAL(clicked()), this, SLOT(calc()));
12
13 }
14
15 MainWindow::~MainWindow()
16 {
17     delete ui;
18 }
19 void MainWindow::calc()
20 {
21     QString s1, s2, rez;
22     s1=ui->op1->text();
23     s2=ui->op2->text();
24     bool ok1,ok2;
25     double p1,p2,r;
26     p1=s1.toDouble(&ok1);
27     p2=s2.toDouble(&ok2);
28     if (ok1&&ok2)
29     {
30         if (ui->rb1->isChecked()) r=p1+p2;
31         if (ui->rb2->isChecked()) r=p1-p2;
32         if (ui->rb3->isChecked()) r=p1*p2;
33         {if (ui->rb4->isChecked()) if (p2!=0) r=p1/p2;
34         }
35     }
36     else
37     {
38         QMessageBox msgBox1(QMessageBox::Information,
39         "Not correct",
40         "Division by null!",
41         QMessageBox::Ok);
42         msgBox1.exec();
43     }
44     rez.setNum(r);
45     ui->rez_label->setText("Rezalt = "+rez);
46 }
47 else
48 {
49     QMessageBox msgBox2(QMessageBox::Information,
50     "Not correct",
51     "Not numeric format!",
52     QMessageBox::Ok);
53     msgBox2.exec();
54 }
55 }
```

Рис. 2. Текст файла mainwindow.cpp

4.3. Свяжем созданный слот с сигналом нажатия на кнопку:

```
QObject::connect(ui->rez_button,SIGNAL(clicked()),
this,SLOT(calc()));
```

4.4. В итоге получим следующий файл `mainwindow.cpp` (см. **Ошибка! Неизвестный аргумент ключа.**):

Задание для самостоятельной работы. Составьте новый проект, соответствующий названию, самостоятельно определив количество виджетов и их свойства (учитывайте, что для ввода данных используется `LineEdit`, для вывода результата – `TextLabel`). Установку текстовых значений меток выполните программно.

Расчетные величины вычисляются по нажатию на кнопку. Для этого создайте функцию-слот, которая запускается кнопкой «Вычислить» и выполняет следующие действия:

- производит чтение исходных данных из полей ввода в переменные;
- вычисляет результат решения задачи;
- записывает результат в метку.

Для размещения виджетов используйте менеджеры компоновки.

Варианты тем проектов:

- 1) Проект «Обмен валюты». Необходимо учесть два варианта обмена: покупку валюты и продажу.
- 2) Проект «Решение уравнений». Необходимо предусмотреть решение двух типов уравнений: линейного и квадратичного.
- 3) Проект «Вычисление площади фигуры». Предусмотреть выбор фигур из вариантов: прямоугольник, треугольник или круг.
- 4) Проект «Расчет стоимости путевки». Пользователь имеет возможность выбрать место отдыха.
- 5) Проект «Приветствие». Пользователю предлагается ввести свое имя, после чего программа приветствует его по имени на одном из языков.
- 6) Проект «Вычисление НОД или НОК».
- 7) Проект «Угол часовой стрелки».
- 8) Проект «Сумма цифр трехзначного числа».
- 9) Проект «Объем информации в байтах». Пользователь имеет возможность выполнить сжатие файла.

ВЫЧИСЛИТЕЛЬНЫЕ МАШИНЫ ИЗНУТРИ И КОМПЬЮТЕРНЫЕ СЕТИ

1. Современная архитектура компьютеров. Машина Фон Неймана

В конце 40-х годов прошлого века во многих научных центрах велись разработки электронных вычислительных машин (далее – ЭВМ). Можно отметить первую ламповую ЭВМ ENIAC, построенную в 1945 году сотрудниками Пенсильванского университета США Дж. Эккертом и Дж. Моучли, разработанную в Киеве под руководством академика С.А. Лебедева в 1951 году ЭВМ МЭСМ и другие проекты.



Большой вклад в дело развития вычислительной техники внесли талантливый математик венгерского происхождения **Джон (Янош) Фон Нейман** (John von Neumann).

В 1946 году Фон Нейман (с соавторами) описал в техническом докладе конкретную ЭВМ, обладающую рядом новых особенностей. Со временем стало ясно, что эти особенности желательно включать в архитектуру всех разрабатываемых в то время компьютеров. А вскоре пришло и понимание того, что эти новые свойства вычислительных машин, по сути, описывают архитектуру некоторого абстрактного универсального вычислителя, который сейчас принято называть машиной Фон Неймана. Эта машина является абстрактной моделью ЭВМ, однако, эта абстракция отличается от абстрактных исполнителей алгоритмов (например от хорошо известной машины Тьюринга). Машина Тьюринга может обрабатывать входные данные любого объема, поэтому этот исполнитель алгоритма принципиально нельзя реализовать. Машина Фон Неймана не поддается реализации по другой причине: многие детали в архитектуре этого вычислителя, как он описывается в учебниках, не конкретизированы. Это сделано специально, чтобы не сковывать творческий подход к делу у инженеров разработчиков новых ЭВМ. Можно сказать, что машина Фон Неймана рассматривается не на внутреннем, а только на концептуальном уровне видения архитектуры.

В некотором смысле машина Фон Неймана подобна абстрактным структурам данных. У абстрактных структур данных, например двоичных деревьев, для их использования необходимо предварительно выполнить конкретную реализацию: произвести отображение на структуры данных, на некотором языке программирования, а также реализовать соответствующие операции над этими данными.

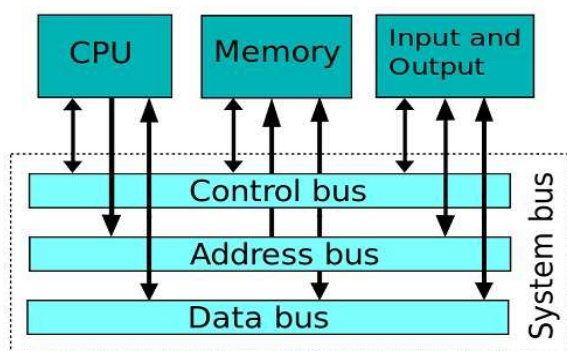


Рис. 1. Архитектурный принцип машины фон Неймана. 1. CPU (ЦПУ) 2. ОЗУ (RAM, Memory) 3. УВВ (IAO Input and Output) 4. Controlbus (Шина управления) 5. Addressbus (Шина адреса) 6. Databus (Шина данных) 6. Systembus (Системная шина – все три шины в совокупности).

Можно сказать, что в машине Фон Неймана зафиксированы те прогрессивные особенности архитектуры, которые в той или иной степени должны были быть присущи всем компьютерам того времени. Разумеется, практически все современные ЭВМ по своей

архитектуре в значительной степени отличаются от машины Фон Неймана, однако эти отличия удобно изучать именно как отличия, проводя сравнения и сопоставления с машиной

Фон Неймана. При рассмотрении данной машины тоже часто будет обращать внимание на отличия архитектуры машины Фон Неймана от современных ЭВМ. Основополагающие свойства архитектуры машины Фон Неймана будут сформулированы в виде принципов Фон Неймана. Эти принципы многие годы определяли основные черты архитектуры нескольких первых поколений ЭВМ.

На рис. 1 приведена схема машины Фон Неймана: толстыми (двойными) стрелками показаны *потоки команд и данных*, а тонкими – *передача между отдельными устройствами компьютера управляющих и информационных сигналов*. Выполнение каждой команды приводит к выработке последовательности управляющих сигналов, которые заставляют узлы компьютера совершать те или иные действия. С помощью же информационных сигналов одни узлы компьютера сообщают другим узлам о том, что они успешно выполнили действия, предписанные управляющими сигналами, либо зафиксировали ошибки в своей работе.

Как видно из приведенного рисунка, машина Фон Неймана состоит из *памяти, устройств ввода/вывода и центрального процессора (ЦП CPU)*. Заметим, что начальный период развития вычислительной техники был представлен просто процессорами, однако позже, когда в составе ЭВМ, кроме основного, появились и другие процессоры, этот основной процессор стали называть центральным процессором (CPU – centralprocessingunit).

Давайте последовательно рассмотрим все узлы машины Фон Неймана и выполняемые ими функции, при этом сформулируем основные принципы архитектуры Фон Неймана.

2. Архитектурный принцип машины Фон Неймана

2.1. Память.

Принцип линейности и однородности информации, хранящейся в памяти

Память машины Фон Неймана – это линейная (упорядоченная) и однородная последовательность некоторых элементов, называемых ячейками. В любую ячейку памяти другие устройства машины (по толстым стрелкам на схеме рис. 1) могут записать и считать информацию, причем время чтения из любой ячейки одинаково для всех ячеек памяти. Время записи в любую ячейку тоже одинаково (это и есть принцип однородности памяти). Разумеется, время чтения из ячейки памяти может не совпадать со временем записи в нее. Такая память в современных компьютерах называется *памятью с произвольным доступом (RandomAccessMemory – RAM)*. На практике современные ЭВМ могут иметь участки памяти разных видов. Например, некоторые области памяти поддерживают только *чтение информации* (на английском эта память называется *ReadOnlyMemory – ROM*), данные в такую память записываются один раз при изготовлении этой памяти, они сохраняются при отключении электрического напряжения. Ячейки памяти в машине Фон Неймана нумеруются от нуля до некоторого положительного числа N (это и означает, что память линейная), причем число N в «настоящих» ЭВМ часто является степенью двойки, минус единица. Адресом ячейки называется ее номер. Каждая ячейка состоит из более мелких частей, именуемых разрядами и нумеруемых также от нуля и до определенного числа.

Количество разрядов в ячейке обозначает разрядность памяти. Каждый разряд может хранить одно число в некоторой системе счисления. В большинстве ЭВМ используется двоичная система счисления, т.к. это более выгодно с точки зрения инженерной реализации. В этом случае каждый разряд хранит одну двоичную цифру или один бит информации. Восемь последовательных бит составляют один байт. Сам Фон Нейман тоже был сторонником использования двоичной системы счисления, что позволяло хорошо описывать архитектуру узлов ЭВМ с помощью логических (булевых) выражений.

Содержимое ячейки называется машинным словом. С точки зрения архитектуры, *машинное слово* – это минимальный объем данных, которым могут обмениваться между

собой различные узлы машины по толстым стрелкам на схеме (не надо, однако, забывать о передаче сигналов по тонким стрелкам). Из каждой ячейки памяти можно считать копию машинного слова и передать ее в другое устройство компьютера, при этом оригинал не меняется.

При записи в память старое содержимое ячейки пропадает и заменяется новым машинным словом. Дело в том, что в такой памяти (она называется *динамической памятью (Dynamic RAM – DRAM)*, ее использование экономически более выгодно, она дешевле) при чтении оригинал разрушается, и его приходится каждый раз восстанавливать после чтения данных. Кроме того, хранимые в динамической памяти данные разрушаются и сами по себе с течением времени («микроконденсаторы» КМОП структуры, ячеек теряют электрический заряд), поэтому приходится часто (через каждый такт процессора) восстанавливать (регенерировать) содержимое этой памяти.

Важный параметр – стоимость. Для основной памяти ЭВМ пока достаточно знать, что чем быстрее такая память, тем она, естественно, дороже. Конкретные значения стоимости памяти меняются весьма быстро с развитием вычислительной техники, сейчас примерная стоимость оперативной памяти составляет несколько центов за мегабайт.

2.2. Принцип неразличимости команд и данных (принцип дуальности информации)

С точки зрения программиста, машинное слово представляет собой либо *команду*, либо подлежащие обработке *данные*. Этот принцип Фона Неймана заключается в том, что данные и команды неотличимы друг от друга: в памяти и те, и другие представляются некоторым набором разрядов на шине данных, причем по внешнему виду машинного слова нельзя определить, что оно собой представляет: команду или данные.

Из неразличимости команд и данных вытекает следствие – принцип однородности (дуальности) информации на шине данных. Этот принцип является очень важным, его суть состоит в том, что программа хранится в памяти вместе с данными, которые она обрабатывает. Чтобы понять важность этого принципа, рассмотрим, как программы вообще появляются в памяти машины. Понятно, что, во-первых, команды программы наравне с данными могут вводиться в память «из внешнего мира» с помощью устройства ввода (этот способ был основным в первых компьютерах). А теперь надо вспомнить, что на вход алгоритма можно, вообще говоря, в качестве входных данных подавать запись некоторого другого алгоритма (в частности свою собственную запись). Остается сделать последний шаг в этих рассуждениях и понять, что выходными данными алгоритма тоже может быть запись некоторого другого алгоритма. Таким образом, одна программа может в качестве результата своей работы поместить в память компьютера другую программу. Как вы уже вероятно знаете, именно так и работают компиляторы языков высокого уровня и различных микроконтроллеров, переводящие (транслирующие) программы с одного языка на другой.

Следствием принципа однородности (дуальности) информации на шине данных является то, что программа может изменяться во время счета самой этой программы. В частности, такая программа может самомодифицироваться, то есть изменять себя, во время выполнения (звучит как *runtime*). В настоящее время самомодифицирующиеся программы применяются крайне редко, в то время как на первых ЭВМ использование самомодифицирующихся программ часто было единственным способом реализации некоторых алгоритмов на языке машины.

Заметим также, что, когда Фон Нейман (с соавторами) писал техническое задание на свою машину, многие из прежних ЭВМ хранили программу в памяти одного вида, а данные – в памяти другого вида, поэтому этот принцип являлся в то время революционным. В современных ЭВМ и программы, и данные, как правило, хранятся в одной и той же памяти.

2.3. Шина управления

Как ясно из самого названия, устройство управления (**ШУ – controlbus**) управляет всеми остальными устройствами ЭВМ. Оно осуществляет это путем посылки управляющих сигналов, подчиняясь которым, остальные устройства производят определенные действия, предписанные этими сигналами. Следует обратить внимание, что это устройство является единственным, от него на рис. 1 отходят тонкие стрелки управляющих сигналов ко всем другим устройствам. Остальные устройства на этой схеме могут «командовать» только памятью, делая ей запросы на чтение и запись машинных слов, выставляя соответствующие значения адреса на адресную шину и получая ответ на шину данных.

2.4. Принцип автоматической работы

Этот принцип для цифровых вычислительных машин еще называют принципом программного (или микропрограммного) управления. Машина, выполняя записанную в ее памяти программу, функционирует автоматически, без участия человека, если только такое участие не предусмотрено в самой программе, например при вводе данных. Пример устройства, которое может выполнять команды, как ЭВМ, но не в автоматическом режиме, – обычный (непрограммируемый) калькулятор. Последовательность команд для калькулятора задает сам человек. **Программа** – непустое множество записанных в памяти (не обязательно последовательно) машинных команд, задающих действия, описывающих шаги работы алгоритма. Команды программы обрабатывают хранимые в памяти компьютера данные.

Таким образом, **программа** – это запись алгоритма на языке машины. **Язык машины** – набор всех возможных операций, выполняемых командами, и допустимые форматы этих команд. Каждая команда однозначно определяется своим кодом операции (в простейшем случае это просто номер команды).

2.5. Принцип последовательного выполнения (последовательной работы)

Устройство управления выполняет некоторую команду от начала до конца, а затем по определенному правилу выбирает следующую команду для выполнения, затем следующую и т.д. При этом либо каждая команда сама явно указывает на команду, которая будет выполняться за ней (такие команды называются командами перехода), либо следующей будет выполняться команда из ячейки, расположенной в памяти непосредственно вслед за той ячейкой, в которой хранится только что выполненная команда. Этот процесс продолжается, пока не будет выполнена специальная команда останова, либо при выполнении очередной команды не возникнет аварийная ситуация (например деление на ноль). **Аварийная ситуация** – это аналог безрезультативного останова алгоритма, например, для машины Тьюринга это чтения из клетки ленты символа, которого нет в заголовке ни одной колонки таблицы. Хотя в архитектуре Тьюринга есть существенный ряд отличий от архитектуры Фон Неймана.

Контрольные вопросы и задания к теме «Архитектура»

1. Почему машина Фон Неймана является абстрактной ЭВМ?
2. В чем заключается принцип линейности и однородности памяти?
3. Объясните разницу между понятиями «ячейка», «адрес ячейки» и «машинное слово».
4. Чем отличаются статическая и динамическая память компьютера? Какие еще виды памяти машины существуют?
5. Сформулируйте принцип неразличимости команд и данных.

6. Что такое язык машины применительно к архитектуре Фон Неймана?
7. Чем отличается регистровая и основная память компьютера?
8. Что называется «позиционной системой счисления», и какие системы счисления использует машина?
9. В чем заключается принцип однородности информации?
10. Переведите десятичное число 4000 в двоичный формат и проведите проверку правильности перевода.
11. Переведите десятичное число 1020 в двоичную, шестнадцатеричную, восьмеричную систему счисления с проверкой правильности перевода во все системы.
12. Переведите десятичное число 2080 в шестнадцатеричную систему счисления путем деления на основание системы счисления 16-ть и проверьте правильность перевода.
13. Переведите десятичное число 1450 в восьмеричную систему счисления путем деления на основание системы счисления 8-мь и проверьте правильность перевода.
14. Переведите шестнадцатеричное число FC34 в десятичное, двоичное, восьмеричное и проверьте правильность перевода.
15. Переведите восьмеричное число 3751 в десятичное, двоичное, шестнадцатеричное и проверьте правильность перевода.

3. Фундамент Internet

Тема протокола *IP (Internet Protocol – протокол Internet)* чрезвычайно обширна и может отпугнуть начинающих, если не представить ее простым, понятным языком, постепенно объясняя неизвестные аббревиатуры, понятия и принципы работы. Для рассматриваемого здесь набора протоколов чаще используют аббревиатуру *TCP/IP (Transmission Control Protocol / Internet Protocol – протокол управления передачей / протокол Internet)*. С помощью этого набора протоколов осуществляется взаимодействие между компьютерами по глобальной сети *Internet*. Существуют и другие протоколы для обмена данными по сети, например протокол *AppleTalk* для компьютеров *Apple*. Как правило, такие протоколы применяются в корпоративных локальных сетях (*intranet-сеть*). Большинство соединений в *Internet* осуществляется с использованием стека протоколов *TCP/IP*, который признан стандартом для взаимодействия между компьютерами в глобальной сети.

3.1. Уровни

На рис. 2 изображена логическая схема взаимодействия двух удаленных компьютеров согласно модели *TCP/IP*. Итак, допустим, нам нужно загрузить *Web-страницу* на компьютер, которому соответствует элемент блок-схемы, обозначенный как *Web-браузер* (см. рис. 2). Прежде чем отправить запрос на получение данных *Web-страницы*, *Web-серверу* на стороне отправителя этот запрос следует упаковать. Данные передаются на самый низкий уровень стека, с прохождением упаковки на каждом из промежуточных уровней. При этом на каждом уровне к сообщению добавляется определенная информация. Затем полученный пакет пересылается по *Internet*. На стороне компьютера адресата сообщение распаковывается в обратном порядке, проходя через все уровни к самому верхнему. Каждый уровень использует предназначенную для него информацию. Оставшаяся часть сообщения передается на более высокий уровень вплоть до уровня приложений (элемент схемы, обозначенный как *Web-сервер*).

Кратко рассмотрим каждый из уровней модели *TCP/IP*.

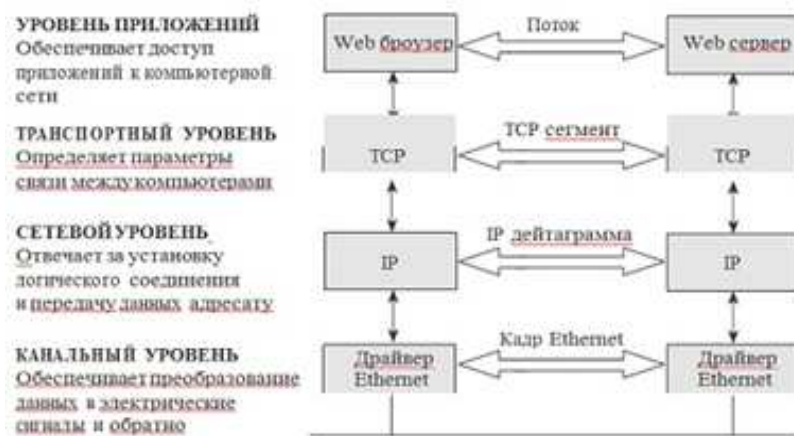


Рис. 2. Модель стека протоколов TCP/IP

➤ **Уровень приложений (application layer)** является высшим уровнем модели TCP/IP. На этом уровне реализуется доступ приложений (в нашем примере *Web-браузера* и *Web-сервера*) к компьютерной сети.

➤ **Транспортный уровень (transport layer)** расположен ниже уровня приложений. На этом уровне устанавливаются многие параметры взаимодействия двух компьютеров и обеспечивается надежная работа других, по своей сути ненадежных, уровней. Данный уровень служит посредником между уровнем приложений и нижними уровнями, ориентированными на передачу данных по сети.

Мы рассмотрим два протокола транспортного уровня: *TCP*, который гарантирует надежную доставку сообщений, и *UDP (User Datagram Protocol – протокол доставки пользовательских дейтаграмм)*, который такой надежной доставки не гарантирует. В нашем примере требовалось использование *TCP*, так как потеря данных недопустима.

➤ **Сетевой уровень (network layer)** отвечает за пересылку данных с одного компьютера на другой (в нашем случае запрос пересылается на *Web-сервер*), нередко только через один транзитный участок или переход (*hop*). Переходом мы будем называть участок сети между компьютером и маршрутизатором или между двумя маршрутизаторами на пути доставки пакета к адресату.

➤ **Канальный уровень (link layer)** является самым низким в иерархии стека TCP/IP. На этом уровне обеспечивается взаимодействие с физической средой передачи данных. В нашем случае двоичные данные преобразовываются в электрические сигналы, так как физической средой передачи является *Ethernet*. Для получения и отправки данных используется определенный интерфейс.

3.2. Обмен данными

Еще раз обратимся к рис. 2. Теоретически процесс передачи данных описывается следующим образом. Запрос на получение *Web-страницы* проходит через уровни компьютера отправителя (которые часто называют стеком TCP/IP) «сверху вниз». Сообщение направляется компьютеру-адресату, где оно проходит обратное преобразование по стеку TCP/IP «снизу вверх». На рис. 2 вертикальные стрелки между уровнями обозначают поток данных на локальном компьютере. Горизонтальные стрелки указывают на то, что каждый уровень передает определенную информацию (упаковывает сообщение) соответствующему уровню на удаленном компьютере. Несмотря на то, что два компьютера не взаимодействуют непосредственно между собой, применение стека TCP/IP создает у пользователя такое впечатление.

Поэтому повторим основные моменты и закрепим терминологию. Термин «стек TCP/IP» используется для описания многоуровневой модели обработки запросов и ответов. **Инкапсуляцией** называется упаковка сообщения одного протокола в сообщение другого и

добавление к нему определенной информации (например идентифицирующих заголовков), предназначенной для соответствующего уровня на удаленном компьютере. Каждый уровень на компьютере-отправителе добавляет к сообщению собственный заголовок, и на компьютере-получателе в первую очередь учитывается заголовок пакета для соответствующего уровня. Полученное сообщение проходит обратный процесс распаковки с удалением заголовков на каждом из уровней до тех пор, пока, наконец, не будет достигнут высокий уровень. При ответе на запрос процесс повторяется в обратном порядке, начиная с упаковки ответного сообщения на хосте *Web-сервера* и заканчивая его распаковкой и передачей уровню приложений, поддерживающему *Web-браузер* на компьютере-получателе.

4. Упаковка

Данные, обмен которыми осуществляется между двумя хостами, должны быть сохранены в каком-то формате, стандартном для каждого уровня стека *TCP/IP*. *Хост (host)* – это общий термин, которым можно назвать рабочую станцию, маршрутизатор, *Web-сервер* и т.д. Общей особенностью хостов является наличие соединения с сетью, по которой можно обмениваться данными. В общем случае все упакованные данные называются *пакетом (package)*. Проблемы с терминологией возникают из-за того, что на каждом уровне стека *TCP/IP* при взаимодействии двух удаленных приложений под этим пакетом понимается различная информация (учитывая добавленные служебные заголовки). Далее мы рассмотрим базовые понятия, касающиеся упаковки данных, а именно: бит, байт, пакет, инкапсуляция и интерпретация данных.

4.1. Биты, байты и пакеты

Наименьшей единицей информации принято считать бит. Значением бита может быть 0 или 1, поэтому бит часто называют *двоичной цифрой (binary)*. Ясно, что в одном бите нельзя передать достаточный объем информации, поэтому их группируют по восемь. Восемь битов составляют один байт. Минимальный объем информации, пусть даже увеличенный в восемь раз, все равно остается недостаточно большим, но один байт способен хранить значение стандартного символа *ASCII*, например буквы, знака препинания или целого числа до 255 (28–1).

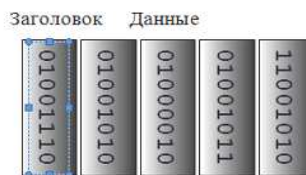
27	26	25	24	23	22	21	20
1	1	1	1	1	1	1	1
128	64	32	16	8	4	2	1

Рис. 3. Схема байта

На рис. 3 показана схема байта. Для битов используется *двоичный код – набор нулей и единиц*. Каждый бит можно представить некоторой степенью основы двоичной системы счисления – числа 2. Значение байта составляет диапазон степеней числа 2 от 20 до 27. Это пояснить довольно просто: если значением всех битов одного байта является 0, то и значение байта равно 0, если же значением всех битов одного байта является 1, то, сложив все значения степеней битов, начиная с наименьшего ($2^0=1$), получим $1+2+4+8+16+32+64+128=255$ – максимальное значение одного байта. Проанализируем смысл этого значения позже, при обсуждении *IP-адресов*. Только что мы выполнили преобразование двоичного значения в десятичное. Для преобразования байта данных из двоичного вида в десятичный достаточно представить его в виде степеней числа 2 и простым сложением полученных значений каждого бита получить искомое десятичное значение.

Для передачи по сети несколько байтов объединяются в один пакет. На рис. 4 показана истинная ситуация при передаче данных по сети: передача любого количества полезных данных обеспечивается за счет добавления определенного объема служебной информации. Требуется некоторые действия для упаковки данных перед отправкой по сети

и их последующей распаковки на стороне адресата (и, конечно, для подтверждения достоверности сообщения). Для проверки целостности переданного пакета предназначено специальное поле **CRC** (*CyclicRedundancyCheck* – **циклическая проверка четности с избыточностью**), значение которого часто называют **контрольной суммой**.



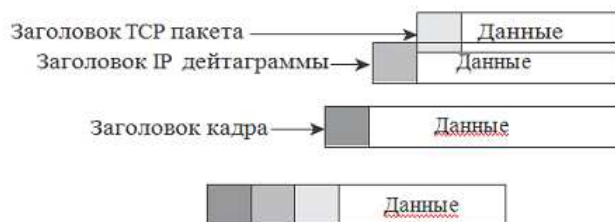
Заголовок содержит информацию об адресате и отправителе, а также о типе передаваемой информации, что напоминает обычный почтовый конверт

Рис. 4. Пакет данных

Как и почтовый конверт, **IP-пакет** должен нести информацию об адресате и отправителе (см. рис. 4). В сетях, по крайней мере в сетях *Ethernet*, аналогом домашнего адреса можно считать **MAC-адрес** (*MediaAccessController* – **контроллер доступа к среде**) вашего сетевого адаптера. Этот аппаратный адрес присваивается производителем сетевого оборудования. MAC-адрес представляет собой 48-битное число, т.е. может быть достаточно большим (248–1). О том, чем различаются **IP-** и **MAC-адреса**, рассказано в разделе «Адреса» этой главы. Для создания **кадра** (*frame*) к нему должна быть добавлена информация заголовков каждого из уровней **TCP/IP**. В последнюю очередь к кадру добавляется информация физического уровня, и он передается в линию связи с помощью **сетевого адаптера** (*NIC* – *networkinterfacecard*). Заголовок кадра имеет размер 14 байт и содержит поля для хранения **MAC-адресов** отправителя и адресата, служебную информацию кадра (ее размер может изменяться) и 4-байтовую завершающую часть (окончание) для передачи кода **CRC**.

4.2. Инкапсуляция

Рассмотрим схему многоуровневой упаковки сообщения (рис. 5). Теоретически различные уровни стека протоколов одного компьютера «обращаются» к соответствующим уровням на другом компьютере. Уровни расположены один над другим, что позволяет говорить о «стеке протоколов **TCP/IP**». На каждом уровне пакет состоит из собственного заголовка и самих данных, которые часто называют **полезной нагрузкой** (*payload*). Смысл всего процесса инкапсуляции заключается в передаче на другой компьютер какой-либо информации, но на пути к адресату на каждом из уровней заголовки добавляются к уже существующей информации. Заголовки вышележащего уровня считаются данными для более низкого уровня.



При прохождении пакета вниз по стеку на каждом уровне добавляются свои заголовки

Рис. 5. Добавление заголовков

Предположим, нам нужно послать сообщение или передать какую-либо информацию на удаленный компьютер. Сначала эта информация формируется в единое целое с помощью программы типа *telnet* или программы для работы с электронной почтой. **TCP-пакет**, который называют **TCP-сегментом** (*TCP segment*), состоит из **TCP-заголовка** и **данных**. **UDP-пакет** называют **дейтаграммой**, что часто приводит к путанице, так как **дейтаграммой** называется и пакет, формирующийся на сетевом уровне (*протокол IP*). **TCP-**

сегмент передается вниз по стеку протоколов на уровень *протокола IP*. На сетевом уровне в начало *TCP-сегмента* добавляется информация *заголовка IP*, и таким образом формируется *IP-дейтаграмма*. В действительности и заголовок, и *данные TCP* на уровне *протокола IP* рассматриваются как единый блок данных. *IP-дейтаграмма* передается на канальный уровень *стека TCP/IP*, где превращается в кадр: в начало *IP-дейтаграммы* добавляется заголовок кадра, содержащий служебную информацию для доставки этого кадра по физической среде передачи, например по *сети Ethernet*. Когда пакет достигает компьютера-адресата, весь описанный процесс повторяется с точностью до наоборот: при прохождении сообщения снизу вверх по *стеку TCP/IP* на каждом из уровней удаляется соответствующий заголовок. С помощью заголовков осуществляется обмен информацией между аналогичными уровнями *стека TCP/IP* взаимодействующих компьютеров.

4.3. Интерпретация полученных данных

При прохождении сообщения по стеку протоколов его информация представляет собой набор нулей и единиц. Как же понять, что скрывается за этой последовательностью? Представим себе, например, заголовок *IP-дейтаграммы*. Как узнать, какой протокол был использован на более высоком уровне? Безусловно, эти сведения позволяют понять принципы работы протокола. В данном случае под термином «*протокол*» понимается набор согласованных правил или форматов обмена сообщениями. В каждом протоколе (например IP, TCP, UDP или ICMP) используется собственный вариант схемы обмена данными и соответствующий формат этих данных.

Рассмотрим заголовок *IP-дейтаграммы* (рис. 6). Для каждого поля заголовка зарезервировано определенное количество битов. В поле «Протокол» сохраняется информация о протоколе более высокого уровня. Каждая строка *IP-заголовка* содержит 32 бит (с 0 по 31 включительно), что равняется четырем байтам. Счет с нуля несколько усложняет задачу определения места нужного байта или бита. В первой строке отображены байты с 0 по 3, во второй – с 4 по 7, а в третьей – с 8 по 11. Обратите внимание на то, что выделенное поле «Протокол» находится в третьей строке. Длина предшествующего поля *TTL* составляет 1 байт. Этот байт является восьмым. Следовательно, поле «Протокол», длина которого тоже 1 байт, является 9-м байтом. Это значит, что для определения протокола, использованного на более высоком уровне, необходимо исследовать этот 9-й (по сути 10-й, ведь счет начинается с 0) байт. То есть весь секрет в том, что надо знать, в каком месте пакета определенного уровня следует искать необходимую информацию; расположение конкретного поля можно найти по известному смещению.

0	15	31
Версия	Длина заголовка	Тип обслуживания
Идентификатор		Общая длина
TTL (время жизни)	Протокол	Смещение фрагмента
Контрольная сумма заголовка		
IP адрес отправителя		
IP адрес получателя		

Заголовок IP дейтаграммы без параметров общей длиной 20 байт

Рис. 6. Поля IP-дейтаграммы

Итак, мы определили место расположения поля «Протокол». Что же оно собой представляет и для каких целей служит? Значение этого поля указывает на то, какой протокол использовался для инкапсуляции данных на более высоком уровне. Предположим, что значение байта этого поля равно 17. На самом деле в поле хранится шестнадцатеричное значение, например 11, которое соответствует десятичному значению 17. Это означает, что на транспортном уровне был использован протокол *UDP*. Значение 6 свидетельствует, что встроенным пакетом является *TCP*-пакет, а значение 1 соответствует *ICMP*-пакету.

5. Шестнадцатеричная система счисления

Основой двоичной системы счисления является число 2, а двоичный код состоит из нулей и единиц. Именно двоичный код используется при работе всех компьютеров. Так зачем же нужно создавать дополнительные сложности и вводить новую систему счисления, основой которой является число 16? Проблема заключается в том, что при использовании двоичной системы счисления для записи большого числа приходится использовать значительное количество битов, и числа быстро становятся слишком громоздкими. **Шестнадцатеричная система** позволяет представить двоичные числа в более кратком виде. Один шестнадцатеричный символ позволяет заменить четыре двоичных разряда ($2^4=16$). Рассмотрим, например, поле заголовка протокола *IP* размером в 8 бит. Его значение можно преобразовать в два шестнадцатеричных символа. Как уже указывалось, двоичное число 17 соответствует использованному протоколу *UDP*. Как же получить из десятичного 17 шестнадцатеричное 11?

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	1	0	0	0	1

Здесь показаны степени числа 2, составляющие десятичное число 17 (для получения этого числа нужно сложить $2^4=16$ и $2^0=1$). Эти биты разделены на два шестнадцатеричных числа, по 4 бит каждый. Крайние слева четыре бита (или шестнадцатеричный символ), которые называют еще **битами старшего разряда (mostsignificant)**, имеют значение 0001. Аналогично четыре крайних справа бита, которые называют **битами младшего разряда (leastsignificant)**, тоже имеют значение 0001. Каждый шестнадцатеричный символ может принимать значения от 0 до 15. Каждый значащий разряд шестнадцатеричного числа в 16 раз больше предыдущего разряда.

Поэтому при делении десятичного числа (17) на 16 число, получившееся в остатке (1), является младшей шестнадцатеричной цифрой. Продолжим деление, записывая остаток слева от первой цифры, и получим шестнадцатеричное число 11. Для того чтобы шестнадцатеричные значения можно было отличить от десятичных, к ним обычно добавляют приставку 0x, например 0x11.

6. Адреса

Скорее всего, вам знаком термин «IP-адрес». Но что он действительно означает и для чего предназначен? Как именно один хост направляет информацию другому? Ответы на эти и другие вопросы можно найти в этом разделе.

6.1. Физические адреса

Можно искать физические *MAC-адреса* отправителя и получателя в заголовке *IP-пакета* до посещения, но так ничего и не обнаружить. *MAC-адреса* не имеют никакого отношения к *протоколу IP*, в котором используются логические адреса. В некоторых случаях *MAC-адреса* могут вообще не существовать.

MAC-адреса используются для идентификации *сетевого адаптера Ethernet* в сети. Сетевой адаптер сам по себе никак не зависит от существования *IP*, заголовков *IP-пакетов* или *логических IP-адресов*. Итак, мы столкнулись с проблемой несоответствия двух понятий. Очевидно, что для реализации взаимодействия двух компьютеров должен существовать способ установки соответствия между логическими *IP-адресами* и *физическими MAC-адресами*. Знаете ли вы *IP-адрес* вашего настольного компьютера? Все дело в том, что на сегодняшний день большинство наших компьютеров не имеет постоянного *IP-адреса*. Получить и зарезервировать определенный *IP-адрес* (или *диапазон адресов*) очень дорого.

Большинству компьютеров *IP-адреса* выделяются на время одного сеанса работы в сети, для чего провайдер услуг *Internet (ISP)* использует специальное программное обеспечение, например протокол *DHCP (DynamicHostConfigurationProtocol – протокол динамической конфигурации хостов)*.

6.2. Выделение IP-адреса

Протокол DHCP позволяет осуществлять динамическое назначение компьютерам *IP-адресов* и отказаться от трудоемкого процесса управления закрепленными за каждым хостом статическими *IP-адресами*. С помощью *DHCP* осуществляется централизованное и автоматическое назначение временных *IP-адресов*, что значительно повышает эффективность управления крупными сетями. Это очень удобно для администратора сети, но усложняет работу администратора системы безопасности (например при поиске компьютеров современными *IP-адресами*, которые *являются источником потенциально опасных действий*).

Каково же общее количество возможных IP-адресов? Их точное число – 232 (так как адрес состоит из 32 бит), т.е. более 4 млрд. Однако не все IP-адреса доступны, существуют зарезервированные диапазоны адресов. Во время стремительного роста популярности *Internet* пришлось с грустью констатировать факт: количество свободных IP-адресов быстро уменьшается. Встал вопрос: как же выйти из создавшегося положения?

Первый способ заключается в использовании *DHCP* отдельными сетевыми узлами, которые выделяют IP-адреса во временное пользование. Это позволяет снизить общее количество необходимых одновременно IP-адресов. Альтернативным способом является применение *зарезервированных адресов для частных сетей*. Организация *IANA (InternetAssignedNumbersAuthority – полномочный комитет по надзору за присвоением номеров Internet)* зарезервировала некоторые диапазоны адресов для использования только во внутренних локальных сетях. Например, *IP-адреса*, начинающиеся с 192.168 и 172.16, могут использоваться только для обмена информацией между хостами в рамках локальной сети. Такой трафик не должен проходить через внешний шлюз конкретного узла. Этот метод позволяет сэкономить *IP-адреса* для отдельного узла *Internet* и использовать для его внутренних целей указанные адреса сетей класса B.

Для преобразования физических *MAC-адресов* в логические *IP-адреса* предназначен протокол *ARP (AddressResolutionProtocol – протокол преобразования адресов)*. *ARP* как таковой не является протоколом *Internet*. Он служит для отправки *кадра Ethernet (запроса)* всем компьютерам данного сегмента сети. Такой запрос называют *широковещательным*. Широковещательное сообщение отправляется всем компьютерам сегмента или всей сети. Стоит подчеркнуть, что *протокол ARP* предназначен для опроса только хостов локальной сети и не может применяться для обмена данными между хостами разных сетей.

Компьютер-отправитель посылает широковещательный *ARP-запрос* всем компьютерам локальной сети о наличии у них *MAC-адреса*, соответствующего определенному *IP-адресу*. Компьютер, которому предназначено сообщение, в ответ на запрос отправляет свой *MAC-адрес*. В результате такого обмена информацией и запрашивающий, и ответивший, а также все другие компьютеры, подключенные к сети, заносят новую запись в свои *ARP-таблицы* соответствий физических и логических адресов. Создание такой таблицы позволяет сократить количество новых *ARP-запросов*. В конечном итоге все компьютеры сегмента локальной сети будут взаимодействовать с помощью *MAC-*, а не *IP-адресов*. При транзакции по стеку *TCP/IP* обмен информацией может осуществляться между соответствующими уровнями стека, но при действительной доставке данных используются именно *MAC-адреса* двух хостов. Почему же *MAC-адреса* столь объемны? Ведь 48 бит дают огромное число возможных значений. Такой размер был выбран для получения абсолютно уникальных и «вечных» адресов. Эта фраза хороша только на слух, и

уже сейчас планируется расширить длину *MAC-адреса* до 128 бит с целью удовлетворения существующих требований указания кода производителя в *MAC-адресах* сетевых адаптеров.

Задания для самостоятельной работы по теме «Networksystems»

1. Кратко опишите основную особенность построения сети Internet и ее внутреннюю структуру.
2. Что является минимальной единицей локальной сети?
3. Что такое локальная сеть и в чем ее отличие от глобальной сети?
4. Перечислите все известные вам протоколы сети Internet и по возможности их назначение.
5. По имеющимся частям IP-адреса и их буквенному соответствию восстановите правильную последовательность следования декад: А 64 Б 3.13 В 3.133 Г 20.
6. Составьте правильную последовательность URL-адреса по имеющимся его фрагментам: А .netBftpС :// DhttpE / F .orgGtxt
7. Если маска подсети 255.255.255.224 и IP-адрес компьютера в сети 162.198.0.157, то порядковый номер компьютера в сети равен...
8. По заданным IP-адресу узла и маске определите адрес сети. IP-адрес узла: 142.9.199.145 Маска: 255.255.192.0. При записи ответа выберите из приведенных в таблице чисел четыре элемента IP-адреса и запишите в нужном порядке соответствующие им буквы, без использования точек.

A	B	C	D	E	F	G	H
0	9	16	64	128	142	192	224

9. Адрес сети получается в результате применения поразрядной конъюнкции к заданному IP-адресу узла и маске. Обычно маска записывается по тем же правилам, что и IP-адрес: в виде четырех байтов, причем каждый байт записывается в виде десятичного числа. Например, пусть IP-адрес узла равен 231.32.255.131, а маска равна 255.255.240.0. Тогда адрес сети равен 231.32.240.0. Для узла с IP-адресом 235.116.177.140 адрес сети равен 235.116.160.0. Чему равен третий слева байт маски? Ответ запишите в виде десятичного числа.

10. Как называется и для чего используется IP-адрес 127.0.0.1 в локальной сети?

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

1. Введение

Программы составляет программист. Рассмотрим, что же необходимо программисту для создания программы (рис.1).

Прежде чем создать программу, программист придумывает **алгоритм**.

Затем выбирает **исполнителя** своего алгоритма.

И в соответствии с **языком описания алгоритма**, понятному исполнителю, составляет **программу**.

Таким образом, мы наметили путь от алгоритма к программе.



Рис.1

Понятие алгоритма

Прежде чем что-нибудь сделать, надо составить **план**.

И в жизни мы все время составляем планы наших действий. Например, утром большинство из нас действует по такому плану: встать → одеться → умыться → позавтракать → выйти из дома в школу или на работу.

Такой план действий называют **алгоритмом**.

Алгоритм – это последовательность или план действий, которые нужно выполнить для решения определенной задачи.

Алгоритмизация – это техника разработки (составления) алгоритма для решения задач на ЭВМ.

Алгоритм состоит из отдельных шагов – команд. Для каждого шага алгоритма можно и нужно составить более **подробный план**.

Остановиться надо на таком плане, в котором исполнителю будет понятно, как выполнить каждый шаг.

Пример алгоритма кипячения чайника:

1. Проверить, есть ли в чайнике вода, если есть, то перейти к действию 2. Если нет, то налить воду в чайник и перейти к действию 2.
2. Зажечь плиту и перейти к действию 3.
3. Поставить чайник на плиту и перейти к действию 4.
4. Если чайник со свистком, ждать, пока он не засвистит, и перейти к действию 5. Если чайник без свистка, ждать до тех пор, пока из носика не пойдет пар, и перейти к действию 5.
5. Выключить плиту.

Понятие исполнителя алгоритма

Исполнитель – это тот, кто умеет понимать и выполнять некоторые команды.

Пример исполнителя алгоритма «Папа» (рис. 2).

ИСПОЛНИТЕЛЬ АЛГОРИТМА



Рис. 2

ФОРМАЛЬНЫЕ И НЕФОРМАЛЬНЫЕ ИСПОЛНИТЕЛИ



Рис.3

Классификация исполнителей: неформальный и формальный (рис. 3). В отличие от устройств (роботов и компьютеров), человек является **неформальным** исполнителем алгоритма. Что это означает? Во-первых, человек не выполняет алгоритм бездумно и формально. Соображения здравого смысла часто берут верх, какими бы строгими ни были инструкции. Во-вторых, человек, даже если он еще мал, обладает определенным жизненным опытом. И чем больше этот опыт, тем менее подробным может быть алгоритм. В-третьих, неформальному исполнителю известна **конечная цель**, и он к ней стремится. **Формальный** исполнитель не обладает ни жизненным опытом, ни здравым смыслом, не стремится к конечному результату и **все инструкции выполняет буквально**.

Система команд исполнителя

Команда – это указание исполнителю выполнить конкретное действие.

Для решения большинства задач недостаточно отдать одну команду исполнителю, надо составить для него алгоритм – план действий, состоящий из команд, которые ему понятны (входят в его **систему команд исполнителя**).

Система команд исполнителя (СКИ) – набор команд, понятных исполнителю. Исполнитель может выполнить только те команды, которые входят в его СКИ.

Исполнителями могут быть люди: ученик, рабочий, учитель; животные: дрессированные собака, кошка; машины: станки, роботы, компьютеры.

Язык описания алгоритма и программа

Исполнитель сможет выполнить алгоритм, если он ему **известен**, т.е. если алгоритм ему сообщили.

Для людей важнейшим способом общения является **язык**.

Для формального исполнителя язык общения не может быть многозначным. Для таких исполнителей разрабатывают и используют специальные **искусственные языки**, не допускающие различные толкования отдельных слов и выражений.

Исполнителю нужно сообщить четкий **алгоритм**, в соответствии с которым он будет действовать.

Программа – это алгоритм, записанный на языке исполнителя.

Программирование – это процесс перевода алгоритма на язык определенного исполнителя.

Так как **разные исполнители** владеют разными языками, а один и тот же алгоритм могут выполнять разные исполнители, то на основе одного алгоритма могут быть написаны **разные программы**.

В настоящее время существует **множество языков программирования**: Си, Паскаль, Делфи, Си++, Визуал Бейсик, Джава и другие.

Чтобы **научиться** писать программы на том или ином языке, нужно изучить **алфавит, словарь и грамматические правила**, по которым строятся предложения в этом языке. При

этом **не допускаются никакие отклонения** от правил написания слов и предложений. Иначе исполнитель просто **откажется** выполнять программу и выдаст ошибку.

Практическое задание

1) Придумайте и составьте словесный алгоритм из жизни (например, алгоритм уборки комнаты; алгоритм проведения выходного дня; алгоритм собирания ягод).

2) Ответьте на *контрольные вопросы*:

1. Кто составляет программы?
2. Опишите путь от алгоритма к программе.
3. Что такое «алгоритм»?
4. Что такое «исполнитель алгоритма»?
5. Приведите примеры формальных исполнителей алгоритма.
6. Приведите примеры неформальных исполнителей алгоритма.
7. Что такое «команда»?
8. Что такое «СКИ»?
9. Что такое «язык описания алгоритма»?
10. Приведите примеры языков программирования.
11. Что такое «программа»?
12. Что такое «программирование»?

2. Алгоритмы

Виды алгоритмов

Следуя определенным правилам, алгоритмы делят на следующие виды: линейные; условные; циклические; смешанные.

Линейный алгоритм

В линейном алгоритме команды выполняются последовательно, одна за другой.

Примером линейного алгоритма может служить алгоритм заваривания чая:

вскипятить воду
сполоснуть заварочный чайник горячей водой
насыпать заварку
залить заварку кипятком
закрыть чайник чем-нибудь теплым
подождать 5 минут
теперь можно пить чай

Условный алгоритм

Условные алгоритмы **всегда** содержат какое-либо **условие**, от выполнения которого зависят дальнейшие действия алгоритма. Условный алгоритм легко **определить** по ключевым словам: **если, то, иначе**.

Примером условного алгоритма может служить алгоритм перехода улицы:

подойти к пешеходному переходу
если есть светофор, то
ждать зеленого света
перейти улицу
иначе
ждать, пока слева не будет машин
перейти улицу до середины
ждать, пока справа не будет машин
перейти вторую половину улицы

Циклический алгоритм

Циклические алгоритмы основаны на каких-либо **повторяющихся действиях**. В циклическом алгоритме некоторые действия повторяются несколько раз. Существуют два вида циклических алгоритмов. В одном из них мы знаем заранее, сколько раз надо сделать

эти действия, в другом мы должны остановиться лишь тогда, когда выполнится некоторое условие.

Примером цикла первого типа является наша жизнь в рабочие дни (с понедельника по субботу): мы выполняем 6 раз почти одни и те же действия.

```
/* Число шагов известно */
повторить 6 раз
    проснуться
    встать
    позавтракать
    пойти в школу
    вернуться домой
    пообедать
    сделать уроки
    поиграть в футбол
    лечь спать
/* программа на воскресенье */
спать
...
```

Пример цикла второго типа – алгоритм распилки бревна: мы не можем заранее сказать, сколько раз нам надо провести пилой от себя и на себя, так как это зависит от плотности дерева, качества пилы и наших усилий. Однако мы точно знаем, что надо закончить работу, когда очередное отпиленное полено упадет на землю.

```
/* Число шагов неизвестно,
но ограничено условием
*/
положить бревно на козлы
наметить место распила
пока полено не отвалится
    пилить от себя
    пилить на себя
положить полено в поленницу
```

Свойства алгоритмов

К алгоритмам предъявляют специальные **требования**, которые необходимо соблюдать при их составлении.

Эти требования называют **свойствами** алгоритмов: понятность, дискретность, определенность, результативность, массовость.

Рассмотрим по порядку все свойства алгоритмов

Понятность алгоритма. Это свойство означает, что исполнитель алгоритма должен **понимать**, как его выполнять. То есть, имея алгоритм и какие-то исходные данные, исполнитель **должен знать**, как надо действовать для выполнения этого алгоритма. Машина – **формальный** исполнитель алгоритма. И она выполняет лишь те команды, которые ты даешь ей, нажимая на кнопки.

Дискретность алгоритма. Дискретность, или пошаговость, алгоритма означает, что алгоритм должен состоять из последовательного выполнения простых **отдельных шагов**. Свойство дискретности будет нарушено в том случае, если алгоритм будет представлен не в виде четких отдельных шагов, а в виде какого-либо повествовательного рассказа.

Определенность алгоритма. Это свойство означает, что каждый шаг алгоритма должен быть четким, однозначным и не оставлять исполнителю возможность самому принимать какие-то решения.

Результативность алгоритма. Результативность, или конечность, алгоритма состоит в том, что алгоритм должен приводить к нужному результату.

Массовость алгоритма. Свойство массовости означает, что алгоритм решения задачи разрабатывается в общем виде, то есть он должен быть применим к похожим задачам, различающимся лишь исходными данными.

Способы представления алгоритмов

Один и тот же алгоритм можно представить в одной из **трех форм**: словесная, графическая, программная

Словесная форма представления алгоритма отображает алгоритм в виде последовательности действий на обычном языке, например на русском.

Пример: Алгоритм нахождения среднего арифметического трех чисел: 1) сложить два числа; 2) к полученной сумме прибавить третье число; 3) разделить сумму чисел на 3.

Графический способ представления алгоритмов изображается в виде последовательности связанных между собой **блоков**, каждый из которых соответствует выполнению определенного действия. Такое графическое представление называют **схемой алгоритма**, или **блок-схемой**.

Пример блок-схемы алгоритма нахождения среднего арифметического трех чисел (рис. 4):

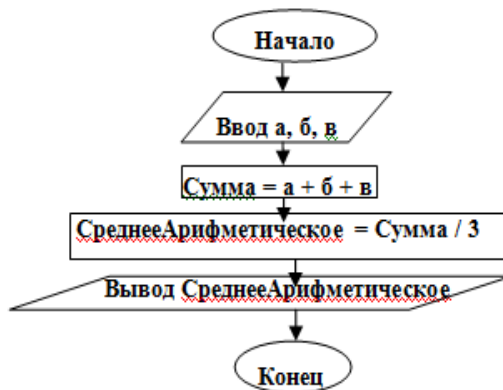


Рис. 4. Пример блок-схемы

Программный способ представления алгоритмов – это и есть самая настоящая программа, написанная на языке программирования.

Пример программы вычисления среднего арифметического трех чисел (рис. 5):

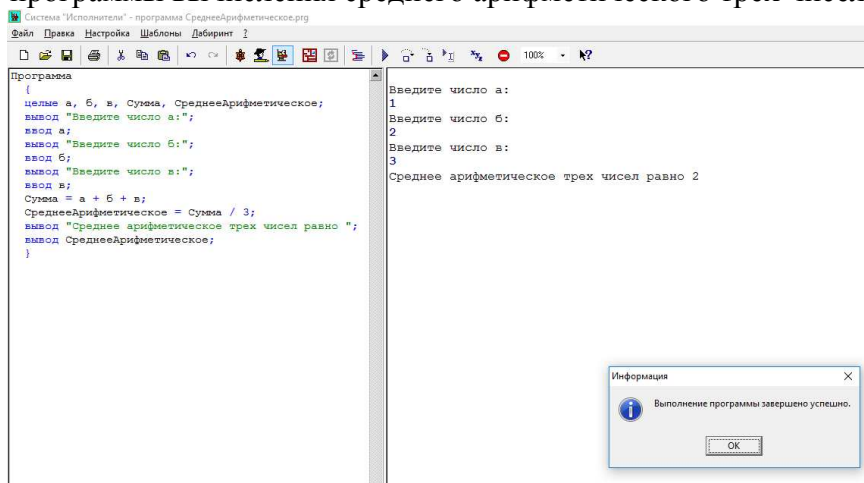


Рис.5. Фрагмент окна программы «Исполнители»

Блок-схемы алгоритмов

Блок-схема представляет собой алгоритм в виде последовательно соединенных блоков. Для того чтобы все программисты понимали, что за алгоритм представлен в виде блок-схемы, существуют **специальные правила** записи блок-схем алгоритмов.

В блок-схеме каждому типу действия соответствует своя **геометрическая фигура**.

Рассмотрим основные блоки блок-схем.

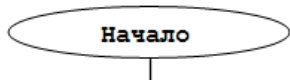


Рис. 6

Блок начала алгоритма **всегда** имеет один выход и не имеет входов.

Блок ввода-вывода информации отображается в виде параллелограмма (рис. 7).

Блок начала алгоритма изображается в виде овала, внутри которого пишут слово **НАЧАЛО** (рис. 6).

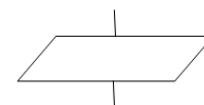


Рис. 7

Этот блок используют в том случае, когда необходимо ввести информацию в компьютер с клавиатуры (рис. 8) или вывести на экран монитора (рис. 9). Внутри блока

записывают то, что необходимо ввести или вывести. Этот блок имеет один вход и один выход.

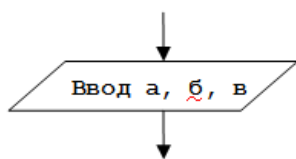


Рис. 8 Ввод информации с клавиатуры

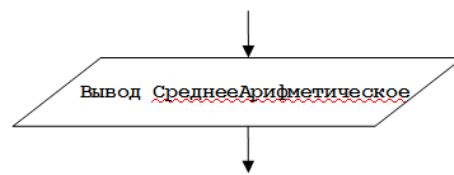


Рис. 9. Вывод информации на экран монитора

Блок действия отображается в виде прямоугольника (рис. 10). В этом блоке записываются действия, выполняющиеся в алгоритме. Обычно это выполнение каких-либо операций. Этот блок также имеет один вход и один выход.

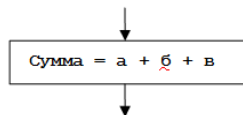


Рис. 10. Блок действия

Блок условия отображается в виде ромба (рис. 11). Внутри ромба записывают условие, которое нужно проверить. Этот блок имеет один вход и два выхода.

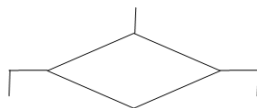


Рис.11. Блок условия

Над линиями выходов пишут слово «да», «нет» или ставят «+», «-» (рис. 12).

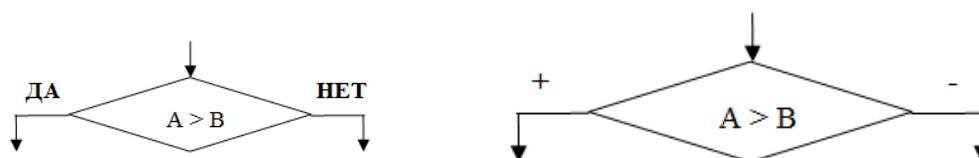


Рис.12

Таким образом, **если условие внутри ромба выполняется**, алгоритм следует дальше по ветке, соответствующей слову «да» или знаку «+».

А в том случае, **если условие не выполняется**, алгоритм продолжает выполняться по ветке «нет» или знаку «-».

Блок конца алгоритма изображается в виде овала, внутри которого пишут слово **КОНЕЦ** (рис. 13).

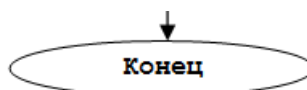


Рис.13. Блок конца

Это всегда завершает алгоритм и имеет один вход и не имеет выходов.

Примеры блок-схемы

Пример № 1. Построить блок-схему алгоритма вычисления выражения: $10+2*8-4=$

Решение. Словесное описание алгоритма:

1. Вычислить произведение $2*8$.
2. Вычислить сумму произведения $2*8$ и 10 ($16+10$).
3. Вычислить разность полученной суммы и 4 ($16-4$).
4. Вывести ответ на экран.

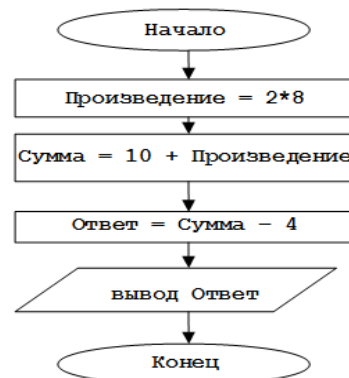


Рис. 14

Пример № 2. Ввести два натуральных числа А и В. Определить наибольшее число.

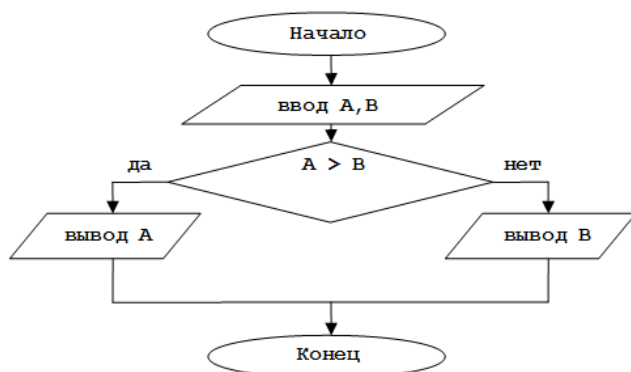


Рис. 15

Решение.

Словесное описание алгоритма:

1. Ввести числа А и В.
2. Если $A > B$, то вывести на экран число А.
3. Иначе вывести на экран число В.

Практическое задание

- 1) Составьте блок-схему задачи вычисления произведения трех чисел.
- 2) Ответьте на *контрольные вопросы*:
 1. Какие виды алгоритмов вы знаете?
 2. Линейный алгоритм. Приведите пример алгоритма.
 3. Условный алгоритм. Приведите пример алгоритма.
 4. Циклический алгоритм. Приведите пример алгоритма.
 5. Способы представления алгоритма.
 6. Блок-схема алгоритма.
 7. Основные блоки блок-схем и их назначение.

3. Среда «Исполнители»**Знакомство со средой «Исполнители»**

Среда имеет название **Исполнители**, так как в ней можно работать с тремя исполнителями: Роботом, Чертежником и Черепахой.

Каждый исполнитель имеет свою **систему команд** и выполняет программу, которая вводится в **текстовом редакторе**. При этом все действия исполнителя отображаются на экране.

Рассмотрим **интерфейс** системы «Исполнители» (рис. 16).

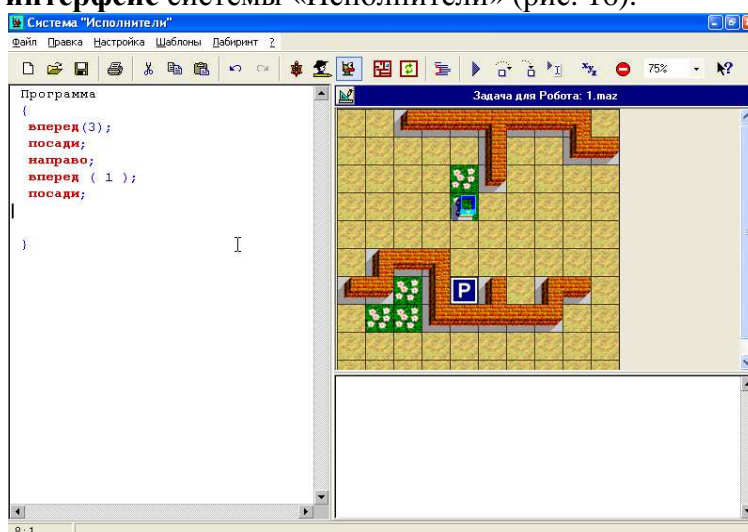


Рис. 16. Интерфейс системы «Исполнители»

Верхняя часть содержит меню и панель инструментов (рис. 17).

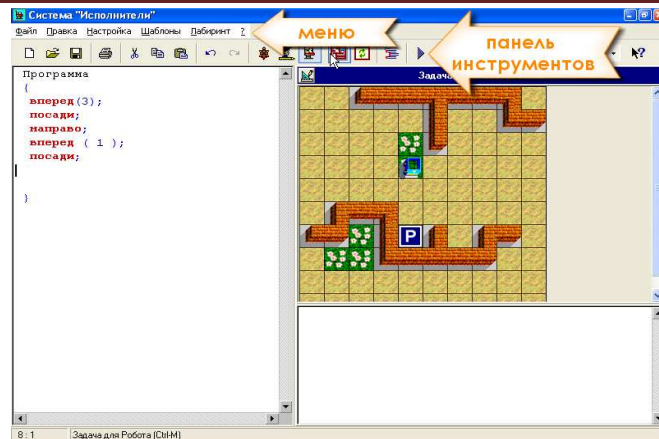


Рис. 17. Меню и панель инструментов
Основная часть окна разделена на три области:

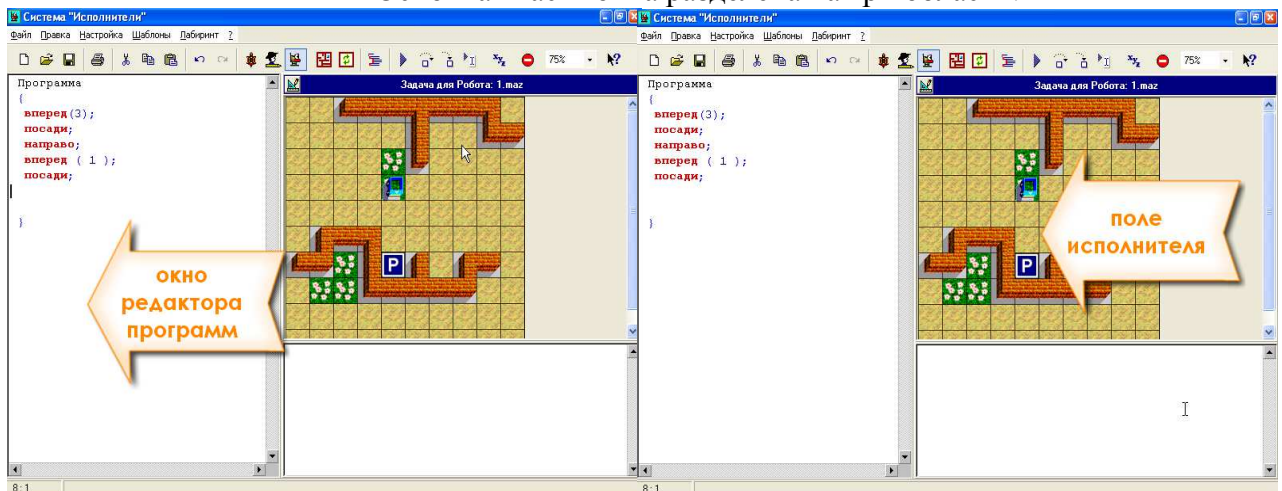


Рис. 18. Окно редактора программ

Рис. 19. Поле исполнителя



Рис. 20. Консольное окно

В **окне редактора** набирается текст программы для исполнителя на специальном языке программирования.

После этого надо запустить программу на выполнение. Для этого нужно нажать клавишу **F9** либо кнопку с треугольной стрелкой на панели инструментов (рис. 21).

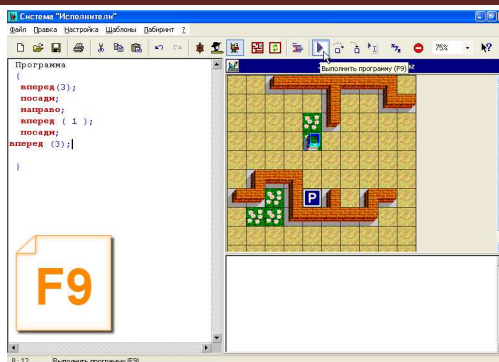


Рис. 21. Запуск программы на исполнение

Когда исполнитель выполняет программу, его действия отображаются на поле исполнителя.

Каждый исполнитель может делать что-то свое. Черепашка и Чертежник могут рисовать на белом листе, а Робот выполняет специальное задание по озеленению местности.

Помощь по работе системы «Исполнители» вызывается по кнопке **F1** (рис. 22).

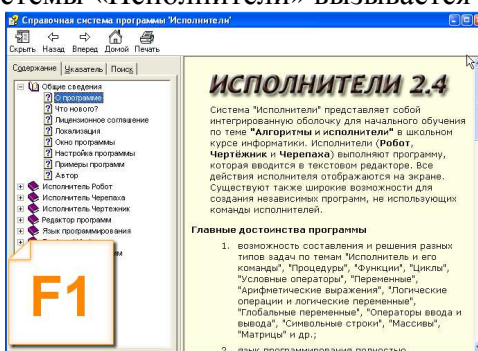


Рис. 22. Справочная система программы «Исполнители»

Если необходимо узнать о конкретном элементе главного окна, нужно щелкнуть по кнопке с вопросительным знаком или выбрать пункт меню «?» (рис. 23). При этом курсор изменит форму на вопрос, и, щелкнув на каком-то элементе, можно получить справку о нем.

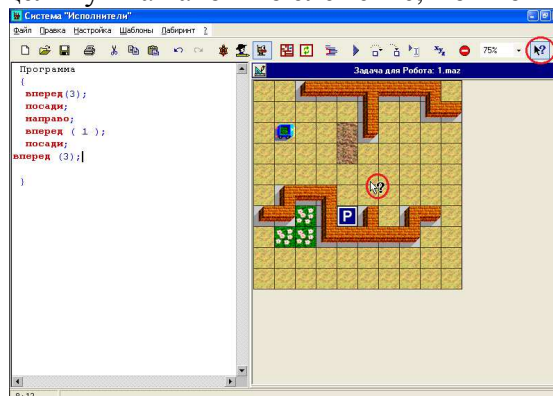


Рис.23. Справка по выбранному элементу

Окно текстовой консоли предназначено для ввода данных с клавиатуры и вывода текстовой информации.

Запустить систему «Исполнители» можно, выполнив двойной щелчок мышью на значке файла (рис. 24).

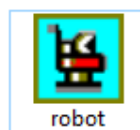


Рис.24. Значок файла системы «Исполнители»

Контрольные вопросы

1. Почему программа называется «система «Исполнители»?»
2. Опишите интерфейс системы «Исполнители»?
3. Как запустить программу в системе «Исполнители»?
4. Как вызвать справку программы?
5. Как получить справку по отдельным элементам интерфейса программы?
6. Назначение окна текстовой консоли.

4. Примеры решения задач**Задача № 1. Получить реверсную запись трехзначного числа****Формулировка.**

Сформировать число, представляющее собой реверсную (обратную в порядке следования разрядов) запись заданного трехзначного числа. Например, для числа 341 таким будет 143. Давайте разберемся с условием. В нашем случае с клавиатуры вводится некоторое трехзначное число (трехзначными называются числа, в записи которых три разряда (то есть три цифры), например: 115, 263, 749 и т.д.).

Нам необходимо получить в некоторой переменной число, которое будет представлять собой реверсную запись введенного числа. Другими словами, нам нужно перевернуть введенное число «задом наперед», представить результат в некоторой переменной и вывести его на экран.

Решение.

Определимся с выбором переменных и их количеством. Ясно, что одна переменная нужна для записи введенного числа с клавиатуры, мы обозначим ее как **n**.

Так как нам нужно переставить разряды числа **n** в некотором порядке, следует для каждого из них также предусмотреть отдельные переменные. Обозначим их как **a** (для разряда единиц), **b** (для разряда десятков) и **c** (для разряда сотен).

Теперь можно начать запись самого алгоритма. Будем разбирать его поэтапно:

1) Вводим число **n**.

2) Работаем с разрядами числа **n**. Как известно, последний разряд любого числа в десятичной системе счисления – это остаток от деления этого числа на 10. В терминах языка **системы «Исполнители»** это означает, что для получения разряда единиц нам необходимо присвоить переменной **a** остаток от деления числа **n** на 10. Этому шагу соответствует следующий оператор:

$$a = n \% 10;$$

Получив разряд единиц, мы должны отбросить его, чтобы иметь возможность продолжить работу с разрядом десятков. Для этого разделим число **n** на 10. В терминах **системы «Исполнители»**, опять же, это означает: присвоить переменной **n** результат от деления без остатка числа **n** на 10. Это мы сделаем с помощью оператора

$$n = n / 10;$$

3) Очевидно, что после выполнения **п. 2** в переменной **n** будет храниться двухзначное число, состоящее из разряда сотен и разряда десятков исходного. Теперь, выполнив те же самые действия еще раз, мы получим разряд десятков исходного числа, но его уже нужно присваивать переменной **b**.

4) В результате в переменной **n** будет храниться однозначное число – разряд сотен исходного числа. Мы можем без дополнительных действий присвоить его переменной **c**.

5) Все полученные в переменных числа – однозначные. Теперь переменная **n** нам больше не нужна, и в ней нужно сформировать число-результат, в котором **a** будет

находиться в разряде сотен, **b** – десятков, **c** – единиц. Легко понять, что для этого нам следует умножить **a** на 100, прибавить к полученному числу **b**, умноженное на 10 и **c** без изменения, и весь этот результат присвоить переменной **c**. Это можно записать так:

$$n = 100 * a + 10 * b + c;$$

б) Далее остается только вывести полученное число на экран.

Код:

```
Программа
{
int n, a, b, c;
print "Введите трехзначное число:";
inputn;
a = n % 10;
n = n / 10;
b = n % 10;
n = n / 10;
c = n;
n = 100 * a + 10 * b + c;
print "Реверсная запись введенного числа: ";
print n;
}
```

Задача № 2. Посчитать количество единичных битов числа

Формулировка.

Дано натуральное число меньше 16. Посчитать количество его единичных битов. Например, если дано число 9, запись которого в двоичной системе счисления равна 1001_2 (подстрочная цифра 2 справа от числа означает, что оно записано в двоичной системе счисления), то количество его единичных битов равно 2.

Решение.

Нам необходима переменная для ввода с клавиатуры. Обозначим ее как **n**. Так как мы должны накапливать количество найденных битов, возникает потребность в еще одной переменной. Обозначим ее как **count** («count» в переводе с англ. означает «считать», «подсчет» и т.д.).

Переменные возьмем типа **int** (они могут принимать значения от 0 до 65535), и пусть в данном случае такой объем избыточен, но это не принципиально важно. Как же сосчитать количество битов во введенном числе? Ведь число же вводится в десятичной системе счисления, и его нужно переводить в двоичную?

На самом деле все гораздо проще. Здесь нам поможет одно интересное правило:

Остаток от деления любого десятичного числа **x** на число **p** дает нам разряд единиц числа **x** (его крайний разряд справа) в системе счисления с основанием **p**.

То есть, деля некоторое десятичное число, например, на 10, в остатке мы получаем разряд единиц этого числа в системе счисления с основанием 10. Возьмем, например, число 3468. Остаток от деления его на 10 равен 8, то есть разряду единиц этого числа.

Понятно, что такие же правила господствуют и в арифметике, в других системах счисления, в том числе в двоичной системе.

Предлагаю поэкспериментировать: запишите на бумаге десятичное число, затем, используя любой калькулятор с функцией перевода из одной системы счисления в другую, переведите это число в двоичную систему счисления и также запишите результат. Затем разделите исходное число на 2 и снова переведите в двоичную систему. Как оно изменилось в результате? Вполне очевидно, что у него пропал крайний разряд справа, или, как мы уже

говорили ранее, разряд единиц. Но как это использовать для решения задачи? Воспользуемся тем, что в двоичной записи числа нет цифр, кроме 0 и 1.

Легко убедиться в том, что сложив все разряды двоичного числа, мы получаем как раз таки количество его единичных битов. Это значит, что вместо проверки значений разрядов двоичного представления числа мы можем прибавлять к счетчику сами эти разряды: если в разряде был 0, значение счетчика не изменится, а если 1, то увеличится на единицу.

Теперь, резюмируя вышеприведенный итог, можно поэтапно сформировать сам алгоритм:

1) Вводим число **n**.

2) Обнуляем счетчик разрядов **count**. Это делается потому, что значения всех переменных при запуске программы считаются неопределенными, и хотя в большинстве компиляторов системы «Исполнители» они обнуляются при запуске, все же считается признаком «хорошего тона» в программировании обнулить значение переменной, которая будет изменяться в процессе работы без предварительного присваивания ей какого-либо значения.

3) Прибавляем к **count** разряд единиц в двоичной записи числа **n**, то есть остаток от деления **n** на 2:

```
count = count + n % 2;
```

Строго говоря, мы могли бы не прибавлять предыдущее значение переменной **count** к остатку от деления, так как оно все равно было нулевым. Но мы поступили так для того, чтобы сделать код более однородным, далее это будет видно.

Учтя разряд единиц в двоичной записи **n**, мы должны отбросить его, чтобы исследовать число далее. Для этого разделим **n** на 2. На языке системы «Исполнители» это будет выглядеть так:

```
n = n / 2;
```

4) Теперь нам нужно еще два раза повторить **п. 3**, после чего останется единственный двоичный разряд числа **n**, который можно просто прибавить к счетчику без каких-либо дополнений:

```
count = count + n;
```

5) В результате в переменной **count** будет храниться количество единичных разрядов в двоичной записи исходного числа. Осталось лишь вывести ее на экран.

Код:

```
Программа
{
int n, count;
print "Введите число в диапазоне от 0 до 15:";
input n;
count = 0;
count = count + n % 2;
n = n / 2;
count = count + n % 2;
n = n / 2;
count = count + n % 2;
n = n / 2;
count = count + n;
print "Количество единиц в введенном числе:";
print count;
}
```

Программа работает правильно на всех вариантах правильных исходных данных, в чем несложно убедиться с помощью простой проверки.

Задача № 3. Проверить, является ли четырехзначное число палиндромом**Формулировка.**

Дано четырехзначное число. Проверить, является ли оно палиндромом.

Примечание: **Палиндромом** называется число, слово или текст, одинаково читающиеся слева направо и справа налево. Например, в нашем случае это числа 1441, 5555, 7117 и т.д. Примеры других чисел-палиндромов произвольной десятичной разрядности, не относящиеся к решаемой задаче: 3, 787, 11, 91519 и т.д.

Решение.

Для ввода числа с клавиатуры будем использовать переменную **n**. Вводимое число принадлежит множеству натуральных чисел и четырехзначно, поэтому оно заведомо больше 255, так что будем использовать тип **int**.

Какими же свойствами обладают числа-палиндромы? Из указанных примеров легко увидеть, что в силу своей одинаковой «читаемости» с двух сторон в них равны первый и последний разряд, второй и предпоследний и т. д. вплоть до середины.

Причем если в числе нечетное количество разрядов, то серединную цифру можно не учитывать при проверке, так как при выполнении названного правила число является палиндромом вне зависимости от ее значения.

В нашей же задаче все даже несколько проще, так как на вход подается четырехзначное число. А это означает, что для решения задачи нам нужно лишь сравнить первую цифру числа с четвертой и вторую цифру с третьей.

Если выполняются оба эти равенства, то число – палиндром.

Остается только получить соответствующие разряды числа в отдельных переменных, а затем, используя условный оператор, проверить выполнение обоих равенств с помощью булевского (логического) выражения.

Однако не стоит спешить с решением. Может быть, мы сможем упростить выведенную схему?

Возьмем, например, уже упомянутое выше число 1441. Что будет, если разделить его на два числа двузначных числа, первое из которых будет содержать разряд тысяч и сотен исходного, а второе – разряд десятков и единиц исходного. Мы получим числа 14 и 41.

Теперь, если второе число заменить на его реверсную запись (это мы делали в задаче 1), мы получим два равных числа 14 и 14! Это преобразование вполне очевидно, так в силу того, что палиндром читается одинаково в обоих направлениях, он состоит из дважды повторяющейся комбинации цифр, и одна из копий просто повернута задом наперед.

Отсюда вывод: нужно разбить исходное число на два двузначных, одно из них реверсировать, а затем выполнить сравнение полученных чисел с помощью условного оператора **if**. Кстати, для получения реверсной записи второй половины числа нам необходимо завести еще две переменные для сохранения используемых разрядов. Обозначим их как **a** и **b**, и будут они типа **int**.

Теперь опишем сам алгоритм:

1) Вводим число **n**.

2) Присваиваем разряд единиц числа **n** переменной **a**, затем отбрасываем его.

После присваиваем разряд десятков **n** переменной **b** и также отбрасываем его:

```
a = n % 10;  
n = n / 10;  
b = n % 10;  
n = n / 10;
```

3) Присваиваем переменной **a** число, представляющее собой реверсную запись хранящейся в переменных **a** и **b** второй части исходного числа **n** по уже известной формуле:

```
a = 10 * a + b;
```

4) Теперь мы можем использовать проверку булевского выражения равенства полученных чисел **n** и **a** с помощью оператора **if** и организовать вывод ответа с помощью ветвлений: `if (n == a) print «Да» else print «Нет»;`

Так как в условии задачи явно не сказано, в какой форме необходимо выводить ответ, мы будем считать логичным вывести его на интуитивно понятном пользователю уровне, доступном в средствах самого языка системы «Исполнители».

Напомним, что с помощью оператора **print** можно выводить результат выражения булевского типа, причем при истинности этого выражения будет выведено слово «Истина», при ложности – слово «Ложь».

Тогда предыдущая конструкция с **if** может быть заменена на `Print (n == a);`

Код:

```
Программа
{
intn, a, b;
print "Введите четырехзначное число:";
input n;
a = n % 10;
n = n / 10;
b = n % 10;
n = n / 10;
a = 10 * a + b;
print "Введенное число является палиндромом? ";
print (n == a);
}
```

Задача № 4. Найти наибольший нетривиальный делитель натурального числа

Формулировка.

Дано натуральное число. Найти его наибольший нетривиальный делитель или вывести единицу, если такового нет.

Примечание 1: Делителем натурального числа **a** называется натуральное число **b**, на которое **a** делится без остатка. То есть выражение «**b** – делитель **a**» означает: $a / b = k$, причем **k** – натуральное число.

Примечание: нетривиальным делителем называется делитель, который отличен от 1 и от самого числа (так как на единицу и само на себя делится любое натуральное число).

Решение.

Пусть ввод с клавиатуры осуществляется в переменную **n**. Попробуем решить задачу перебором чисел. Для этого возьмем число, на единицу меньше **n**, и проверим, делится ли **n** на него. Если да, то выводим результат и выходим из цикла с помощью оператора **break**. Если нет, то снова уменьшаем число на 1 и продолжаем проверку.

Если у числа нет нетривиальных делителей, то на каком-то шаге проверка дойдет до единицы, на которую число гарантированно поделится, после чего будет выдан соответствующий условию ответ.

Хотя, если говорить точнее, следовало бы начать проверку с числа, равного $n / 2$ (чтобы отбросить дробную часть при делении, если **n** нечетно), так как ни одно натуральное число не имеет делителей больших, чем половина этого числа.

В противном случае частное от деления должно быть натуральным числом между 1 и 2, которого просто не существует.

Алгоритм на естественном языке:

- 1) Ввод **n**.
- 2) Запуск цикла, при котором **i** изменяется от $n / 2$ до 1. В цикле:

1. Если n делится на i (то есть остаток от деления числа n на i равен 0), то выводим i на экран и выходим из цикла с помощью **break**.

Код:

```

Программа
{
inti, n;
print "Введите натуральное число:";
input n;
for (i = n / 2; i >= 1; i = i - 1)
{
if (n % i == 0)
{
print "Наибольший нетривиальный делитель введенного числа:";
print i;
break;
}
}
}

```

Задача № 5. Вывести на экран все простые числа до заданного

Формулировка.

Дано натуральное число. Вывести на экран все простые числа до заданного включительно.

Решение.

В решении данной задачи используется решение предыдущей.

Нам необходимо произвести тест простоты числа (обозначим его как **код 1**):

```

count = 0;
for (i = 1; i <= n; i = i + 1)
if (n % i == 0) count = count + 1;
print (count == 2);

```

Здесь n – проверяемое на простоту число. Напомним, что данный фрагмент кода в цикле проверяет, делится ли n на каждое i в отрезке от 1 до самого n , и если n делится на i , то увеличивает счетчик **count** на 1. Когда цикл заканчивается, на экран выводится результат проверки равенства счетчика числу 2.

В нашем же случае нужно провести проверку на простоту всех натуральных чисел от 1 до заданного числа (обозначим его как n).

Следовательно, мы должны поместить **код 1** в цикл по всем k от 1 до n .

Также в связи с этим необходимо заменить в **коде 1** идентификатор n на k , так как в данном решении проверяются на простоту все числа k .

Кроме того, теперь вместо вывода ответа о подтверждении/опровержении простоты k необходимо вывести само это число в случае простоты:

```

if (count == 2) print (k + " ");

```

Обобщая вышесказанное, приведем алгоритм на естественном языке:

- 1) Ввод n .
- 2) Запуск цикла, при котором k изменяется от 1 до n . В цикле:
 1. Обнуление переменной **count**.
 2. Запуск цикла, при котором i изменяется от 1 до k . В цикле:
 - a. Если k делится на i , то увеличиваем значение переменной **count** на 1.
 3. Если **count** = 2, выводим на экран число k и символ пробела.

Код:

```

Программа
{
inti, k, n, count;
print "Введите крайнюю верхнюю границу диапазона чисел:";
inputn;
println "Простые числа в диапазоне от 0 до заданного:";
for (k = 1; k <= n; k = k + 1)
{
count = 0;
for (i = 1; i <= k; i = i + 1)
if (k % i == 0)
count = count + 1;
if (count == 2)
println k;
}
}

```

Вычислительная сложность.

В данной задаче мы впервые столкнулись с вложенным циклом (когда один цикл запускается внутри другого). Это означает, что наш алгоритм имеет нелинейную сложность (при которой количество выполненных операций равно размерности исходных данных (в нашем случае это n – количество чисел) плюс некоторое количество обязательных операторов).

Давайте посчитаем количество выполненных операций в частном случае.

Итак, пусть необходимо вывести все натуральные простые числа до числа 5. Очевидно, что проверка числа 1 пройдет в 1 + 2 шага, числа 2 – в 2 + 2 шага, числа 3 – в 3 + 2 шага и т.д. (прибавленная двойка к каждому числу – два обязательных оператора вне внутреннего цикла), так как мы проверяем делители во всем отрезке от 1 до проверяемого числа.

В итоге количество проведенных операций (выполненных операторов на низшем уровне) представляет собой сумму: 3 + 4 + 5 + 6 + 7 (также опущен обязательный оператор ввода вне главного (внешнего) цикла).

Очевидно, что при выводе всех простых чисел от 1 до n приведенная ранее сумма будет такой:

$$1 + 3 + 5 + 6 + \dots + (n - 1) + n + (n + 1) + (n + 2),$$

где вместо многоточия нужно дописать все недостающие члены суммы. Очевидно, что это сумма первых $(n + 2)$ членов арифметической прогрессии с вычтенным из нее числом 2.

Как известно, сумма k первых членов арифметической прогрессии выражена формулой:

$$S_k = \frac{a_1 + a_k}{2} k,$$

где a_1 – первый член прогрессии, a_k – последний.

Подставляя в эту формулу наши исходные данные и учитывая недостающее до полной суммы число 2, получаем следующее выражение:

$$S_n = \frac{1 + (n + 2)}{2} (n + 2) - 2 = \frac{(n + 2) + (n + 2)^2}{2} - \frac{4}{2} = \frac{n + 2 + n^2 + 4n + 4 - 4}{2} = \frac{n^2 + 5n + 2}{2} = \frac{1}{2} n^2 + \frac{5}{2} n + 1$$

Чтобы найти асимптотическую сложность алгоритма, отбросим коэффициенты при переменных и слагаемые с низшими степенями (оставив, соответственно, слагаемое с самой высокой степенью). При этом у нас остается член n^2 , значит, асимптотическая сложность алгоритма – $O(n^2)$.

Конечно, в дальнейшем мы не будем так подробно находить асимптотическую сложность алгоритмов, а тем более вычислять количество требуемых операций, что интересно только теоретически.

На самом деле, конечно, нас интересует лишь порядок роста времени работы алгоритма (количества необходимых операций), который можно выявить из анализа вложенности циклов и некоторых других характеристик.

Задача № 6. Проверить, является ли заданное натуральное число совершенным

Формулировка.

Дано натуральное число. Проверить, является ли оно совершенным.

Примечание: совершенным числом называется натуральное число, равное сумме всех своих собственных делителей (то есть натуральных делителей, отличных от самого числа).

Например, 6 – совершенное число, оно имеет три собственных делителя: 1, 2, 3, и их сумма равна $1 + 2 + 3 = 6$.

Решение.

Код основной части (назовем его **кодом 1**):

```
count = 0;
for (i = 1; i <= n; i = i + 1)
    if (n % i == 0) count = count + 1;
```

Как видно, в этом цикле проверяется делимость числа **n** на все числа от 1 до **n**. Чтобы переделать этот код под текущую задачу, нужно вместо инкрементации (увеличения значения) переменной-счетчика прибавлять числовые значения самих делителей к некоторой переменной для хранения суммы (обычно ее мнемонически называют **sum**, что в пер. с англ. означает «сумма»). В связи с этим оператор `if (n % i == 0) count = count + 1;` в **коде 1** теперь уже будет выглядеть так: `if (n % i == 0) sum = sum + i;`

Кроме того, чтобы не учитывалось само число **n** при суммировании его делителей (насколько мы помним, этот делитель не учитывается в рамках определения совершенного числа), цикл должен продолжаться не до **n**, а до **n – 1**.

Правда, если говорить точнее, то цикл следовало бы проводить до **ndiv 2**, так как любое число **n** не может иметь больших делителей, иначе частное от деления должно быть несуществующим натуральным числом между 1 и 2.

Единственное, что останется теперь сделать, – это вывести ответ, сравнив число **n** с суммой его делителей **sum** как результат булевского выражения через **print**:

```
print (n == sum);
```

Код:

```
Программа
{
    inti, n, sum;
    print "Введите натуральное число:";
    inputn;
    sum = 0;
    for (i = 1; i <= n / 2; i = i + 1)
        if (n % i == 0) sum = sum + i;
    print "Является введенное число совершенным? ";
    print (n == sum);
}
```

Задача № 7. Проверить монотонность последовательности цифр числа

Формулировка.

Дано натуральное число n . Проверить, представляют ли цифры его восьмеричной записи строго монотонную последовательность. При этом последовательность из одной цифры считать строго монотонной.

Примечание: В математике строго возрастающие и строго убывающие последовательности называются строго монотонными. В строго возрастающей последовательности каждый следующий член **больше** предыдущего. Например: 1, 3, 4, 7, 11, 18. В строго убывающей последовательности каждый следующий член **меньше** предыдущего. Например: 9, 8, 5, 1.

Решение.

Здесь нам нужно будет последовательно получить разряды восьмеричной записи числа, двигаясь по записи числа справа налево.

Как мы уже знаем, последний разряд числа в восьмеричной системе счисления есть остаток от деления этого числа на 8. Попытаемся определить несколько общих свойств строго возрастающих (обозначим пример как 1) и строго убывающих (обозначим как 2) последовательностей (для наглядности будем сразу брать восьмеричные последовательности):

1) 3, 4, 5, 8, 9, 11.

2) 8, 7, 3, 2, 0.

Для начала введем в рассмотрение некоторую формулу, обозначим ее как (I):

$$\delta_i = a_i - a_{i+1},$$

где a_i – член заданной последовательности с индексом i . Нетрудно понять, что эта формула определяет разность между двумя соседними элементами: например, если $i = 5$ (то есть мы рассматриваем пятую разность), формула будет выглядеть так: $\delta_5 = a_5 - a_6$.

При этом стоит учитывать множество всех значений, которые может принимать i . Например, для последовательности (1) i может принимать значения от 1 до 5 включительно, для последовательности (2) – от 1 до 4 включительно.

Легко проверить, что для всех других i формула (I) не имеет смысла, так как в ней должны участвовать несуществующие члены последовательности.

Найдем все δ_i для последовательности (1): $\delta_1 = 3 - 4 = -1$, $\delta_2 = 4 - 5 = -1$, $\delta_3 = 5 - 8 = -3$, $\delta_4 = 8 - 9 = -1$, $\delta_5 = 9 - 11 = -2$.

Как видим, они все отрицательны. Нетрудно догадаться, что это свойство сохраняется для всех строго возрастающих последовательностей.

Выпишем все δ_i для последовательности (2), не расписывая при этом саму формулу: 1, 4, 1, 2. Видим, что все они положительны.

Кстати, весьма примечательно, что в математическом анализе определение монотонной функции дается в терминах, подобных используемым в нашей формуле (I), которая рассматривается там несколько более гибко, а δ_i при этом называется *приращением* и имеет несколько иное обозначение.

Можно обобщить сказанное тем, что последовательность *приращений* показывает, на какую величину *уменьшается* каждый член исследуемой последовательности чисел, начиная с первого.

Понятно, что если каждый член числовой последовательности уменьшается на положительную величину, то эта последовательность строго убывает и т.д.

Из всех этих рассуждений делаем вывод о том, что числовая последовательность является строго монотонной (то есть строго возрастающей или строго убывающей) тогда и только тогда, когда δ_i имеют один и тот же знак для всех i .

Таким образом, мы вывели понятие, которое можно проверить с помощью последовательности однотипных действий, то есть циклической обработки.

Теперь нам необходимо попробовать унифицировать проверку знакопостоянства всех δ_i для последовательностей обоих видов. Для этого рассмотрим произведение каких-либо двух δ_i в последовательностях (1) и (2).

Примечательно, что оно положительно как произведение чисел одного знака. Таким образом, мы можем в любой последовательности взять $delta_1$ и получить произведения его со всеми остальными $delta$ (обозначим эту формулу как (II)):

$$p = delta_1 * delta_i$$

Если все p положительны, то последовательность строго монотонна, а если же возникает хотя бы одно отрицательное произведение p , то условие монотонности нарушено.

Теперь перенесем эти рассуждения из математики в программирование и конкретизируем их на последовательность цифр восьмеричной записи числа.

Обозначим идентификатором **delta** результат вычисления формулы (I) для двух текущих соседних членов последовательности.

Каким же образом двигаться по разрядам числа **n** и обрабатывать все **delta**? Сначала мы можем получить последнюю цифру числа **n** (назовем ее **b**), отбросить ее и получить предпоследнюю цифру **n** (назовем ее **a**), отбросить ее тоже, а затем вычислить **delta = a – b**.

Кстати, отметим, что при таком подходе мы будем для всех i находить $delta_i$, двигаясь по ним справа налево.

Начальному фрагменту этих действий соответствует следующий код:

```
b = n % 8;
n = n / 8;
a = n % 8;
n = n / 8;
delta = a - b;
```

Теперь мы можем войти в цикл с предусловием $n <> 0$. В каждом шаге цикла мы должны присвоить переменной **b** число **a**, затем считать следующий разряд в **a** и отбросить этот разряд в **n**.

Таким способом мы «сдвигаем» текущую пару: например, на первом шаге в примере (2) мы до входа в цикл использовали бы цифры 2 (в переменной **a**) и 0 (в переменной **b**), затем при входе в цикл скопировали бы 2 в **b** и 3 в **a** – таким образом, все было бы готово для исследования знака по произведению. В связи с этим основной цикл будет выглядеть так:

```
while (n != 0)
{
    b = a;
    a = n % 8;
    n = n / 8;
    ...
}
```

На месте многоточия и будет проверка знакопостоянства произведений. Воспользовавшись формулой (II), мы заменим $delta_i$ в качестве текущего на саму разность **a – b**, чтобы не задействовать дополнительную переменную. В итоге, если теперь **delta * (a – b) <= 0**, то выходим из цикла:

```
if (delta * (a - b) <= 0) break;
```

Теперь рассмотрим развитие событий по завершении цикла:

1) Если произойдет выход по завершении цикла, то есть «закончится» число в связи с превращением его в 0 на некотором шаге, то значения **a**, **b** и **delta** будут содержать значения, подтверждающие строгую монотонность последовательности, что можно проверить с помощью вывода значения булевского выражения на экран.

2) Если в теле цикла произошел выход через условный оператор, то эти переменные будут содержать значения, с помощью которых выявлено условие нарушения строгой монотонности. Это значит, что на выходе из цикла ответ можно выводить в формате:

```
println (delta * (a - b) > 0);
```

Код:

```

Программа
{
int n, a, b, delta;
print "Введите число:";
inputn;
b = n % 8;
n = n / 8;
a = n % 8;
n = n / 8;
delta = a - b;
while (n <> 0)
{
b = a;
a = n % 8;
n = n / 8;
if (delta * (a - b) <= 0)
break;
}
print "Монотонность числа? ";
print (delta * (a - b) > 0);
}

```

Кстати, что будет при вводе числа **n**, меньшего 8, ведь его восьмеричная запись однозначна? Хотя наша цепочка делений еще до входа в цикл требует двух цифр в записи числа.

Посмотрим, как поведет себя программа при вводе числа **n** из единственной цифры **k**: сначала **k** идет в **b** (строка 9), затем отбрасывается в **n** (строка 10), которое теперь равно 0, затем в **a** идет число 0, так как $0 \bmod 8 = 0$ (строка 11).

При этом строка 12 уже ничего не дает, так как **n**, которое сейчас равно нулю, присваивается значение $0 \operatorname{div} 8 = 0$.

Далее вычисляется $\mathit{delta} = -k$ (строка 13), затем игнорируется цикл, так как $\mathit{n} = 0$ (строки 14–19), затем выводится на экран значение выражения $-k * -k > 0$ (строка 20), которое истинно для любого действительного **k** кроме нуля, который и не входит в условие нашей задачи, так как **n** натуральное.

Как видим, вырожденный случай прошел обработку «сам собой», и для него не пришлось разветвлять программу, что выражает несомненный плюс.

Задача № 8. Получить каноническое разложение числа на простые множители

Формулировка.

Дано натуральное число **n** ($\mathit{n} > 1$). Получить его каноническое разложение на простые множители, то есть представить в виде произведения простых множителей. При этом в разложении допустимо указывать множитель 1. Например, $264 = 2 * 2 * 2 * 3 * 11$ (программе допустимо выдать ответ $264 = 1 * 2 * 2 * 2 * 3 * 11$).

Решение.

Данная задача имеет достаточно красивое решение.

Из *основной теоремы арифметики* известно, что для любого натурального числа больше 1 существует его каноническое разложение на простые множители, причем это разложение единственно с точностью до порядка следования множителей.

Например, $12 = 2 * 2 * 3$ и $12 = 3 * 2 * 2$ – это одинаковые разложения. Рассмотрим каноническую форму любого числа на конкретном примере.

Например, $264 = 2 * 2 * 2 * 3 * 11$.

Каким образом можно выявить эту структуру?

Чтобы ответить на этот вопрос, вспомним изложенные в любом школьном курсе алгебры правила деления одночленов, представив, что числа в каноническом разложении являются переменными. Как известно, если разделить выражение на переменную в некоторой степени, содержащуюся в этом выражении в той же степени, оно вычеркивается в ее записи.

То есть, если мы разделим 264 на 2, то в его каноническом разложении уйдет одна двойка. Затем мы можем проверить, делится ли снова получившееся частное на 2. Ответ будет положительным, но третий раз деление даст остаток.

Тогда нужно брать для рассмотрения следующее натуральное число 3 – на него частное разделится один раз. В итоге, проходя числовую прямую в положительном направлении, мы дойдем до числа 11, и после деления на 11 n станет равно 1, что будет говорить о необходимости закончить процедуру.

Почему при таком «вычеркивании» найденных сомножителей мы не получим делимостей на составные числа?

На самом деле, здесь все просто: любое составное число является произведением простых сомножителей, меньших его.

В итоге получается, что мы вычеркнем из n все сомножители любого составного числа, пока дойдем до него самого в цепочке делений.

Например, при таком переборе n никогда не разделится на 4, так как «по пути» к этому числу мы вычеркнем из n все сомножители-двойки.

Алгоритм на естественном языке:

- 1) Ввод n .
- 2) Присвоение переменной p числа 2.
- 3) Вывод числа n , знака равенства и единицы для оформления разложения.
- 4) Запуск цикла с предусловием $n <> 1$. В цикле:
 1. Если $n \% p = 0$, то вывести на экран знак умножения и переменную p , затем разделить n на p , иначе увеличить значение i на 1.

Код:

```
Программа
{
  int n, p;
  print "Введите число:";
  input n;
  p = 2;
  print n;
  print " = 1";
  while (n != 1)
  {
    if ((n % p) == 0)
    {
      print " * ";
      print p;
      n = n / p;
    }
    else
      p = p + 1;
  }
}
```

Задача № 9. Сформировать число из двух заданных чередованием разрядов**Формулировка.**

Даны два натуральных числа одинаковой десятичной разрядности. Сформировать из них третье число так, чтобы цифры первого числа стояли на нечетных местах третьего, а цифры второго – на четных. При этом порядки следования цифр сохраняются. Например, при вводе 1234 и 5678 программа должна выдать ответ 15263748 (для наглядности разряды обоих чисел выделены разными цветами).

Решение.

Так как у чисел (обозначим их **a** и **b**) одинаковая десятичная разрядность, крайняя справа цифра у третьего числа (**c**, которое поначалу должно быть равно 0) всегда будет на четном месте, так как при его формировании мы работаем с длинами **a** и **b** как с числами одной четности, сумма которых всегда четна, и длина **c** как раз и есть позиция крайней справа цифры. Это значит, что формирование **c** нужно в любом случае начинать с последнего разряда **b**.

При этом каждый взятый из **a** или **b** разряд мы должны сместить на необходимую позицию влево, чтобы добавлять разряды **c**, используя операцию сложения.

Мы сделаем это с помощью вспомогательной переменной **z**, которая перед входом в цикл будет равна 1. В цикле же она будет умножаться на последний добытый разряд **b** (при этом выражение $z * b \% 10$ нужно прибавить к **c**), затем умножить **z** на 10 и проделать то же самое с последним разрядом **a** и снова умножить **z** на 10.

Кстати, при этом нужно не забыть своевременно отбросить уже рассмотренные разряды чисел.

Так как разрядность чисел неизвестна, нам нужен цикл с предусловием. В силу одинаковой десятичной разрядности **a** и **b** мы можем сделать условие по обнулению любого из них, так как второе при этом также обнулится. Возьмем условие $a <> 0$.

Таким будет основной цикл:

```
while (a != 0)
{
    c = c + z * (b % 10);
    z = z * 10;
    b = b / 10;
    c = c + z * (a % 10);
    z = z * 10;
    a = a / 10;
}
```

В итоге конечное число **c** будет сформировано в таком виде (все направления справа налево): первая цифра **b**, первая цифра **a**, вторая цифра **b**, вторая цифра **a** и так далее до самых последних разрядов слева.

Кстати, скобки в двух операторах нужны для правильного понимания компилятором приоритета выполняемых арифметических операций.

Без них **z** умножится на соответствующее число, и остаток от деления именно этого числа прибавится к **c**, что неправильно.

Алгоритм на естественном языке:

- 1) Ввод **a** и **b**.
- 2) Обнуление переменной **c**.
- 3) Присвоение переменной **z** числа 1.
- 4) Запуск цикла с предусловием $a <> 0$. В цикле:
 1. Прибавляем последний разряд **b** в текущий разряд **c**, определяемый с помощью множителя **z**.
 2. Умножаем **z** на 10.
 3. Избавляемся от последнего разряда в **b**.

4. Прибавляем последний разряд **a** в текущий разряд **c** с помощью множителя **z**.
 5. Умножаем **z** на 10.
 6. Избавляемся от последнего разряда в **a**.
- 5) Вывод **c**.

Код:

```
Программа
{
int c, z, a, b;
print "Введите числа a и b:";
input a, b;
c = 0;
z = 1;
while (a != 0)
{
c = c + z * (b % 10);
z = z * 10;
b = b / 10;
c = c + z * (a % 10);
z = z * 10;
a = a / 10;
}
print "Сформированное число с чередованием разрядов:";
print c;
}
```

Задача № 10. Вычислить экспоненту с заданной точностью

Формулировка.

Дано действительное число **x**. Вычислить значение экспоненциальной функции (то есть показательной функции e^x , где e – математическая константа, $e \approx 2,718281828459045$) в точке **x** с заданной точностью **eps** с помощью ряда Тейлора:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Примечание 1: показательными называются функции вида a^x , где a – некоторое действительное число, x – независимая переменная, являющаяся показателем степени.

Примечание 2: ряд Тейлора – это представление функции в виде суммы (возможно, бесконечной) некоторых других функций по особым правилам (требующим детального математического обоснования, что в данном случае нам не нужно).

Решение.

Не вникая в теоретическую часть и считая представленную формулу корректной, попробуем разобраться в том, что же нам необходимо сделать для того, чтобы решить эту задачу:

1) Нам дана некоторая точка на оси Ox , и мы должны вычислить значение функции e^x в этой точке. Допустим, если **x = 4**, то значение функции в этой точке будет равно $e^4 \approx 2,71828^4 \approx 54,598$.

2) При этом вычисление необходимо реализовать с помощью заданной бесконечной формулы, в которой прибавление каждого очередного слагаемого увеличивает точность результата.

3) Точность должна составить вещественное число **eps**, меньшее 1. Это означает, что когда очередное прибавляемое к сумме слагаемое будет меньше **eps**, необходимо завершить вычисление и выдать результат на экран.

Это условие обязательно выполнится, так как математически доказано, что каждое следующее слагаемое в ряде Тейлора меньше предыдущего, следовательно, бесконечная последовательность слагаемых – это бесконечно убывающая последовательность.

Теперь разберемся с вычислением самого ряда. Очевидно, что любое его слагаемое, начиная со второго, можно получить из предыдущего, умножив его на **x** и разделив на натуральное число, являющееся номером текущего шага при последовательном вычислении (примем во внимание то, что тогда шаги нужно нумеровать с нуля).

Значение **x** нам известно на любом шаге, а вот номер текущего шага (будем хранить его в переменной **n**) придется фиксировать.

Создадим вещественную переменную **expf** (от англ. *exponentialfunction* – экспоненциальная функция) для накопления суммы слагаемых.

Будем считать нулевой шаг уже выполненным, так как первое слагаемое в ряду – константа 1, и в связи с этим **expf** можно заранее проинициализировать числом 1:

$$expf = 1;$$

Так как мы начинаем вычисления не с нулевого, а с первого шага, то также нужно инициализировать значения **n** (числом 1, так как следующий шаг будет первым) и **p** (в ней будет храниться значение последнего вычисленного слагаемого):

$$n = 1;$$

$$p = 1;$$

Теперь можно приступить к разработке цикла.

С учетом заданной точности **eps** условием его продолжения будет **abs(p) >= eps**, где **abs(p)** – модуль числа **p** (модуль нужен для того, чтобы не возникло ошибки, если введено отрицательное **x**).

В цикле необходимо домножить **p** на **x** и поделить его на текущий номер шага **n**, чтобы обеспечить реализацию факториала в знаменателе, после чего прибавить новое слагаемое **p** к результату **expf** и увеличить **n** для следующего шага:

```
while (abs(p) >= eps)
{
    p = p * x / n;
    expf = expf + p;
    n = n + 1;
}
```

После выхода из цикла нужно осуществить форматированный вывод результата **expf** на экран с некоторым количеством цифр после точки, например, пятью.

Отметим, что если при этом введенное **eps** содержало меньше 5 цифр после точки, то сформированное значение **expf** будет, соответственно, неточным.

Код:

```
Программа
{
    float x, eps, expf, p;
    int n;
    print "Введите число x:";
    input x;
    print "Введите точность eps:";
    input eps;
    expf = 1;
    n = 1;
    p = 1;
    while (abs(p) >= eps)
```

```
{  
  p = p * x / n;  
  expf = expf + p;  
  n = n + 1;  
}  
print "Экспонента числа равна ";  
printexpf;  
}
```


Е.А. Редько, С.В. Летучий, А.В. Крупский

**Информатика: методическое пособие для участников
летней (очной) сессии**

ВЫПУСК 10

Методическое пособие по математике для учащихся и преподавателей

Подписано к печати 28.07.2016 г. Формат 60x84 1/16.
Тираж 65 экз.

КГБОУ ДО ХКЦРТДиЮ
680000, г. Хабаровск, ул. Комсомольская, 87

© КГБОУ ДО ХКЦРТДиЮ, 2016
© Е.А. Редько, С.В. Летучий, А.В. Крупский, 2016