HTTP-СЕРВЕР

HTTP-СЕРВЕР 1
Задание 2
Пояснения 3
Ход работы 4
Интерфейс 4
Вспомогательные функции 5
Сервер без поддержки соединения 6
Сервер с поддержкой соединения 14
Приложение Указатели (динамические переменные) 17
Объявление указателей 17
Выделение и освобождение памяти 18
Действия с данными 19
Действия с указателями 20

Задание

Реализовать HTTP-сервер, открывающий клиенту указанный в настройках каталог. Сервер должен:

- прослушивать указанный порт (по умолчанию 80);
- обрабатывать запросы по методам GET и HEAD и сообщать об этом (заголовок Allow);
- поддерживать keep-alive соединения (при условии запроса соединений такого рода пользователем);
- вести список подключенных клиентов и разрывать соединение после заданного тайм-аута;
- сообщать свое имя (заголовок Server);
- отправлять пользователю, помимо самих данных, тип передаваемых данных в формате MIME (заголовок Content-type);
- передавать длину файла в заголовке HTTP-ответа (заголовок Content-length);
- возвращать коды состояния 200, 400, 404, 405.

При сдаче работы необходимо продемонстрировать запросы к серверу из ранее разработанного TCP-клиента с получением всех возможных ответов и загрузку сайта в браузере. Для демонстрации подготовить сайт минимум из 2-х страниц и 1 картинки.

Показать исходный код программы.

Отчет оформлять необязательно.

Баллы:

10 баллов – рабочий сервер без поддержки соединения;

13 баллов – рабочий сервер с поддержкой соединения;

+ 1 балл за аккуратную разметку исходного кода;

+ 1 балл за комментарии к коду;

+ 1 балл за каждую функцию сервера, реализованную не как в методичке (по другому принципу)

+ 3 балла за каждую дополнительную функцию сервера, разработанную полностью самостоятельно.

Пояснения

HTTP-сервер – частный вид веб-сервера. Его задача – отправлять клиентам (браузерам) страницы сайта и другие файлы, которые находятся на компьютере сервера. Содержимое сайта обычно размещается в отдельной папке, откуда ее и берет сервер.

В процессе работы сервер «слушает» стандартные порты HTTP (80 или 8080).

Когда приходит сообщение, сервер должен извлечь из него необходимую информацию: что запрашивает клиент, поддерживать ли соединение. Т.е. сервер должен уметь получать и расшифровывать HTTP-сообщения.

В ответ сервер должен отправить не просто файл, а HTTP-ответ, т.е. он должен уметь составлять этот ответ и помещать в него файл. Также нужно оправлять сообщение об ошибке, если:

а) запрос неверный (код 400);

б) файл не найден (код 404);

в) неподдерживаемый метод HTTP (код 405).

В режиме *без поддержки соединения* сервер разрывает связь сразу после отправки сообщения.

Но большинство современных клиентов работают в режиме *с поддержкой соединения* и отправляют несколько запросов в течение одного подключения. Это позволяет экономить трафик и ускорять загрузку сайтов. В этом случае сервер не должен обрывать соединение, пока клиент сам не отключится. Выглядит просто, если клиент ведет себя «правильно». Но что, если клиент «завис», неправильно работает или возникли проблемы в сети? Соединение будет «висеть» бесконечно?

Если бы сервер просто оставлял все открытые соединения, то в реальном режиме работы его пришлось бы перезагружать чуть ли не каждый час, чтобы сбросить «мертвые» подключения.

Поэтому необходимо вести список всех подключений и автоматически сбрасывать их через определенное время (тайм-аут).

Таким образом, функционал сервера включает:

1. Выбор корневой директории, в которой размещается сайт, и прослушиваемый порт.
2. Кнопки запуска и остановки сервера.
3. Сокет для получения и отправки сообщений.
4. Считывание и разбор (на сленге – парсинг) входящих HTTP-запросов.
5. Проверка правильности запроса, наличия файла. Отправка сообщения об ошибке.
6. Чтение файла, определение его MIME-формата и размера. Отправка HTTP-ответа с файлом или без него.
7. Ведение списка подключения с указанием тайм-аута, отключение клиентов.
8. Ведение журнала (лога) подключения для контроля работы.

Разрабатываемый нами сервер имеет лишь минимальный функционал. Реальные серверы могут, например:

– поддерживать несколько сайтов одновременно;
– отправлять разные данные в зависимости от особенностей клиента (не отправлять неподдерживаемые типы файлов, разные файлы для разных браузеров и др.);
– проводить аутентификацию пользователей и назначать права доступа файлам для обеспечения безопасности;
– распределять нагрузку для улучшения быстродействия – на нашем сервере каждый пользователь вынужден ждать, пока мы отправим файл предыдущему пользователю;
– отправлять файлы по частям и поддерживать докачку файлов (для менеджеров загрузок);
– вместе с сообщениями об ошибках отправлять стандартные страницы с соответствующей информацией (например, “404.htm”);
– получать и использовать cookies.

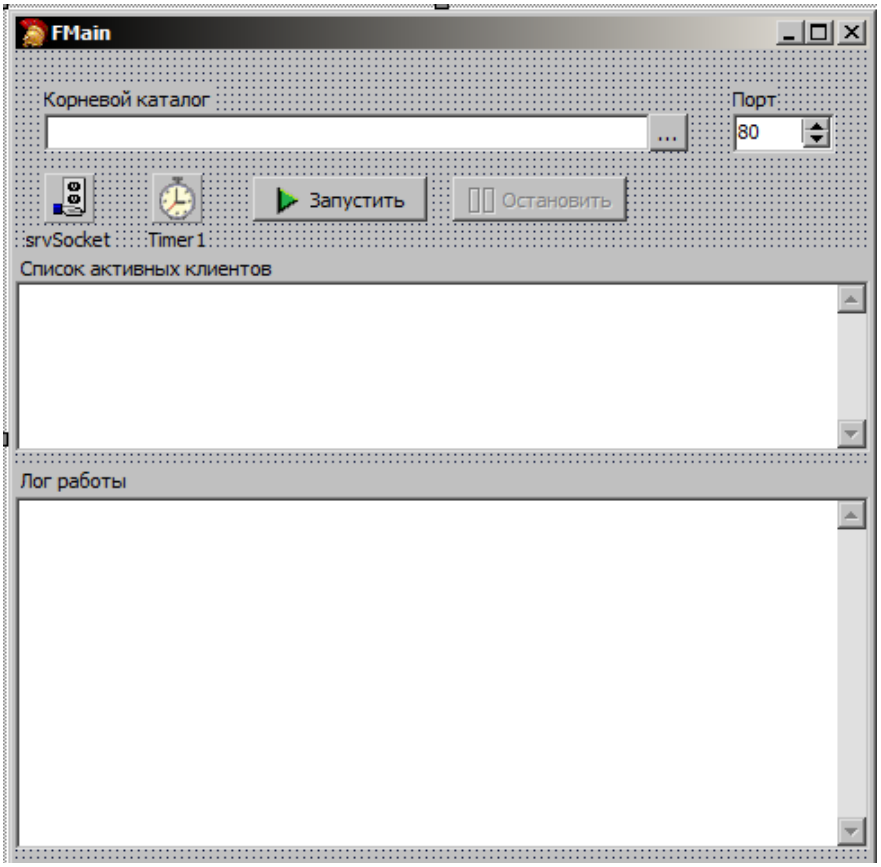
Ход работы

Интерфейс

Разместим на форме `FMain` элементы:

- сокет сервера `srvSocket: TServerSocket`;
- поле для ввода порта `edPort: TSpinEdit`, значение по умолчанию 80;
- поле и кнопка для ввода корневой директории, в которой размещен сайт `edRootDir: TEdit`; `btnRootDir: TButton`;
- кнопки запуска и остановки сервера `btnStart`, `btnStop: TBitBtn`; или `TButton`, кнопка остановки по умолчанию отключена;
- поле для вывода списка клиентов `mmClients: TMemo`, только для чтения;
- поле лога `mmLog: TMemo`, только для чтения;
- таймер `Timer1.TTimer`, период 1000мс;

Элементы желательно разместить на панелях `TPanel` с настройкой выравнивания `Align`. Либо запретите изменять размер формы – свойство `BorderStyle: bsSingle`.



Вспомогательные функции

При работе с текстом часто возникают проблемы с кодировкой, когда всё, вроде бы, работает, но вместо нормального текста получаются «зюки».

Тип `String` в Pascal и Delphi 7 подразумевает кодировку ASCII (ANSI), каждый символ – один байт. Однако в более новых версиях Delphi тип данных `String` стал использовать кодировку UTF (Unicode), каждый символ – 2 байта.

Все сообщения HTTP передаются в кодировке ASCII. Поэтому попытка записать их в переменную типа String приводит к появлению «кракозябр».

Чтобы избежать проблем с кодировкой, можно использовать строковые типы данных, для которых кодировка задана явно:

`AnsiString` – текст в ASCII (равнозначно String в Delphi 7 и ранее);

`WideString` – текст в Unicode (равнозначно String в новых версиях Delphi).

Но функция сокетов для получения текста `Socket.ReceiveText` возвращает тип `String`, мы не можем это поменять. Ее можно использовать в Delphi 7, а для более новых версий нужно вместо `Socket.ReceiveText` вписать в начало программы и использовать следующую функцию:

```
function ReceiveAnsiText(Socket: TCustomWinSocket): AnsiString;
// получение текстовых данных из сокета
begin
SetLength(Result, Socket.ReceiveBuf(Pointer(nil)^, -1));
SetLength(Result, Socket.ReceiveBuf(Pointer(Result)^, Length(Result)));
end;
```

Чтобы аккуратно оформить лог, каждую запись будем снабжать меткой времени и отделять горизонтальной линией. Чтобы сократить код, вынесем эти стандартные действия в отдельную процедуру:

```
procedure TFMain.AddToLog(Str: String);
//добавление сообщения Str в лог
begin
mmLog.Lines.Add('');    //пустая строка
mmLog.Lines.Add(TimeToStr(Now));    //время
mmLog.Lines.Add(Str);   //сообщение
mmLog.Lines.Add('------------------------');  //горизонтальная линия
end;
```

Чтобы получить доступ к `mmLog`, эту процедуру нужно объявить в разделе `public` главной формы `FMain`:

```
  public
    { Public declarations }
    procedure AddToLog(Str: String);
```

Все действия в процедуре выполняются с `mmLog.Lines`, поэтому можно записать короче, используя оператор `with`:

```
with mmLog.Lines do
  begin
  Add('');
  Add(TimeToStr(Now));
  Add(Str);
  Add('------------------------');
  end;
```

Сервер без поддержки соединения

Сначала реализуем сервер без поддержки соединения. При получении запроса он обрабатывает его, отправляет ответ и сразу завершает соединение.

Выбор корневой директории

Действие при нажатии на кнопку `btnRootDir`.

Для выбора корневой директории сайта воспользуемся функцией, которая находится в файле `FileCtrl` (его необходимо прописать в `uses`).

```
SelectDirectory(const Caption: string; const StartDir: WideString;
 out ChosenDir: string): Boolean;
```

Она вызывает стандартное системное окно выбора папки и возвращает `True`, если пользователь подтвердил выбор, `False` – если нажал «Отмена». Сам путь к выбранной папке записывается в переменную `ChosenDir`. `Caption` – текст в заголовке окна, `StartDir` – каталог по умолчанию.

```
procedure TFMain.btnRootDirClick(Sender: TObject);
// выбор корневой директории
var RootDir: string;
begin
RootDir := edRootDir.Text;    //что было выбрано ранее
if SelectDirectory('Укажите корневой каталог', RootDir, RootDir) then  //выбор
  edRootDir.Text := RootDir;  //сохранение
end;
```

Запуск и остановка сервера

Действия по нажатию кнопок `btnStart`, `btnStop`.

При запуске сервера необходимо проверить доступность корневой директории, отключить изменение настроек сервера (кнопки и поля ввода) и открыть сокет:

```
procedure TFMain.btnStartClick(Sender: TObject);
begin
if DirectoryExists(edRootDir.Text) then    //если указан корневой каталог
  begin
  // считываем параметры
  srvSocket.Port := StrToInt(edPort.Text);
  // откл. кнопки
  btnRootDir.Enabled := False;
  edRootDir.Enabled := False;
  edPort.Enabled := False;
  btnStart.Enabled := False;
  btnStop.Enabled := True;
  // открываем сокет
  AddToLog('Запуск сервера');
  srvSocket.Open;
  end
else
  MessageBox(0, 'Указанная корневая директория не существует', 'Ошибка', 0);
end;
```

Остановка сервера – выключение сокета и включение кнопок.

```
procedure TFMain.btnStopClick(Sender: TObject);
// остановка сервера
begin
// закрываем сокет
srvSocket.Close;
AddToLog('Остановка сервера');
// вкл. кнопки
btnRootDir.Enabled := True;
edRootDir.Enabled := True;
edPort.Enabled := True;
btnStart.Enabled := True;
btnStop.Enabled := False;
end;
```

Обработка входящих запросов

Реализовывать всю обработку входящих запросов в одной процедуре неудобно. Выделим отдельные процедуры и функции.

Первое действие – получение полного пути к файлу, исходя из корневой директории и указанного в запросе URL. Эти данные будем передавать в функцию в качестве параметров, а в результате получать полный путь к файлу:

```
function FileFromURL(URL: AnsiString; Root: AnsiString = ''): String;
```

Примеры значений Root:

```
C:\
C:\Sites\MySite
D:\Server\www\
```

Примеры значений URL:

```
/
/index.html
```

```
/src/image003.png
/ru/
```

Кратко, говоря, путь к файлу = `Root + URL`.

```
Result := Root + URL;
```

Но сначала нужно проверить ряд моментов.

В URL папки отделяются /, а в Windows \. Нужно организовать замену символа / на \. Не будем писать цикл, а воспользуемся стандартной функцией:

```
URL := StringReplace(URL, '/', '\', [rfReplaceAll]);
```

Если в конце `Root` стоит косая черта, ее нужно удалить, т.к. она уже присутствует в начале `URL`.

```
if copy(Root, Length(Root), 1) = '\' then
  Delete(Root, Length(Root), 1);
```

Еще необходимо учитывать, что не-ASCII символы в URL записываются в виде escape-последовательностей. Для декодирования escape-последовательностей в URL можно использовать готовую функцию `HTTPDecode` из файла `HTTPApp` (прописать в `uses`). Результат `HTTPDecode` нужно еще перекодировать из UTF-8 с помощью стандартной функции `UTF8Decode`.

```
URL := UTF8Decode(HTTPDecode(URL));
```

Наконец, если клиент запрашивает директорию без указания конкретного файла, то следует поискать в ней файл с именем “index”, тип – веб-страница (html, htm, phtml, shtml – какой первым найдем).

Примечание Некоторые серверы в этом случае показывают содержимое папки (если она существует), но это небезопасно.

Для проверки существования файла используется стандартная функция `FileExists`, для извлечения имени и расширения файла из полного пути – `ExtractFileName` и `ExtractFileExt`.

```
if (copy(Result,Length(Result),1) <> '\')
    and (ExtractFileExt(Result) = '') then
  Result:= Result + '\';
if (ExtractFileName(Result) = '') then
  begin
  if FileExists(Result + 'index.html') then
    Result:= Result + 'index.html' else
  if FileExists(Result + 'index.htm') then
    Result:= Result + 'index.htm' else
  if FileExists(Result + 'index.phtml') then
    Result:= Result + 'index.phtml' else
  if FileExists(Result + 'index.shtml') then
    Result:= Result + 'index.shtml';
  end;
```

В ответе необходимо указать MIME-формат файла. Определять MIME-формат будем по расширению файла. Поскольку расширений файлов существует огромного количество, проверка вручную будет слишком громоздкой.

Мы извлечем необходимые сведения из реестра Windows.

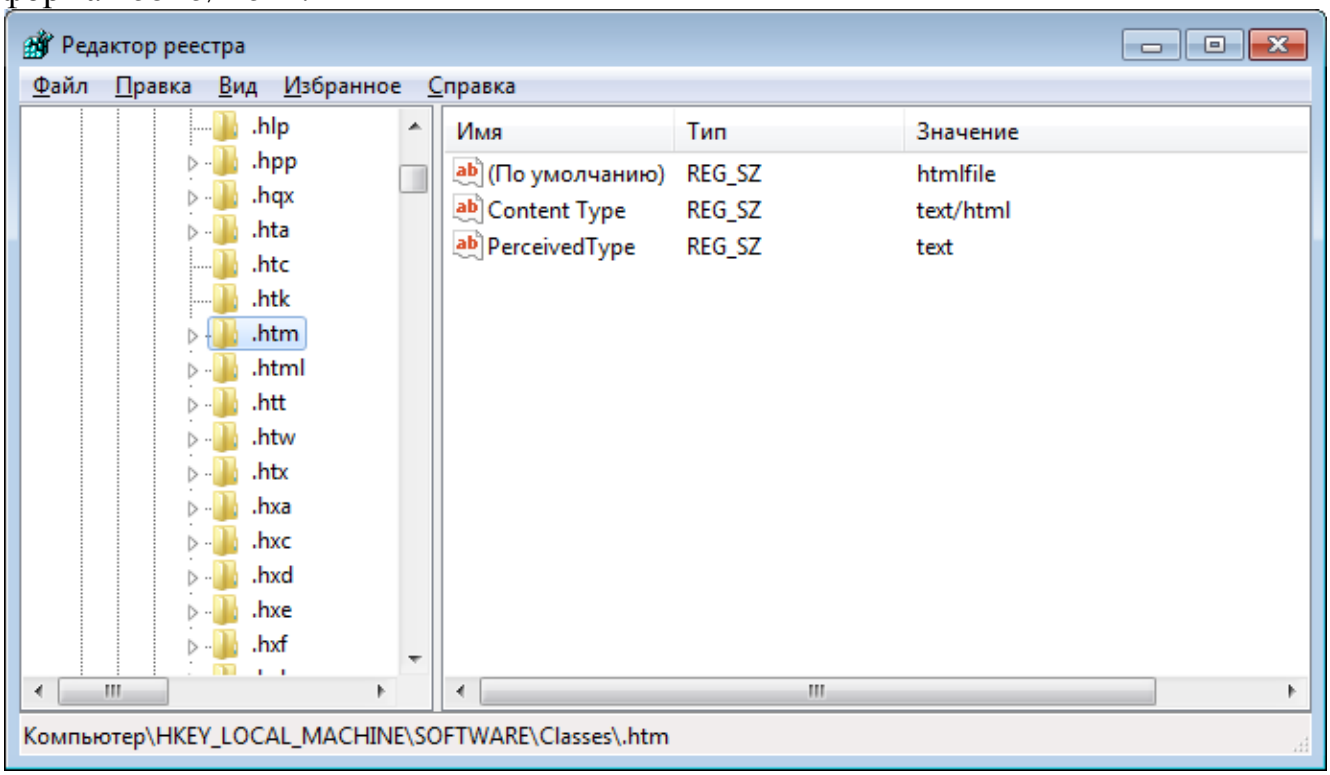
Объявите функцию, в которую мы передаем расширение файла, а получаем название MIME-формата:

```
function MIMEFromExt(Ext: AnsiString): AnsiString;
```

Сразу впишите значение по умолчанию – пустая строка:

```
Result := '';
```

Все известные системе расширения файлов и соответствующие им MIME-форматы хранятся в ветке реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Classes\`. Загляните в редактор реестра, чтобы посмотреть, как это организовано (могут потребоваться права администратора). В меню Пуск выберите пункт «Выполнить...» (если его там нет, посмотрите через поиск или в Программы – Стандартные), введите `regedit`. Найдите указанную ветку, а в ней, например, расширение `.html`. Видим, что в параметре `Content Type` указан его MIME-формат `text/html`.



Это не очень надежный способ, т.к. содержание этой ветки зависит от того, какие программы установлены на компьютере. Кроме того, небезопасно давать доступ к системному реестру для веб-сервера. В идеале, перечень известных MIME-форматов должен храниться в самом сервере.

Для работы с реестром в Delphi используется специальный класс `TRegistry` (необходимо прописать в `uses` файл `Registry`).

Объявите в функции переменную

```
var reg: TRegistry;
```

Доступ к реестру похож на работу с файлом. Необходимо:

1) создать в переменной `reg` объект класса `TRegistry`, указав метод доступа – только чтение `KEY_READ` (либо только запись, либо и запись, и чтение);

```
reg := TRegistry.Create(KEY_READ);
```

2) указать, что мы работаем в ветке `HKEY_LOCAL_MACHINE` в свойстве `RootKey`;

```
reg.RootKey := HKEY_LOCAL_MACHINE;
```

3) попытаться считать значение из ветки `\SOFTWARE\Classes\` + расширение файла, параметр `Content Type`;

```
if reg.OpenKey('\SOFTWARE\Classes\' + Ext, False)
  then Result := reg.ReadString('Content Type');
```

4) закрыть реестр и освободить память методом Free;

```
reg.Free;
```

5) если не удалось найти тип файла в реестре, то считаем, что это просто двоичные данные `'application/octet-stream'`

```
if Result = '' then
  Result := 'application/octet-stream';
```

Однако, шаги 2 и 3 – небезопасные. Если по каким-то причинам не удалось считать данные из реестра (нет прав доступа, нет нужной ветки), то сервер выдаст сообщение об ошибке, а шаги 4 и 5 не выполнятся, т.е. сервер ничего не ответ клиенту, да еще и реестр оставит открытым.

Чтобы избежать такой ситуацией, воспользуйтесь «оберткой» `try finally end`.

Небезопасные операции (1-3) необходимо поместить в секцию `try`, а обязательные (4-5) – в секцию `finally`.

Теперь у нас есть все средства, чтобы сформировать ответ на запрос и отправить его. Вынесем это действие в приватную процедуру формы:

```
procedure TFMain.Response(Request: AnsiString; Socket: TCustomWinSocket);
```

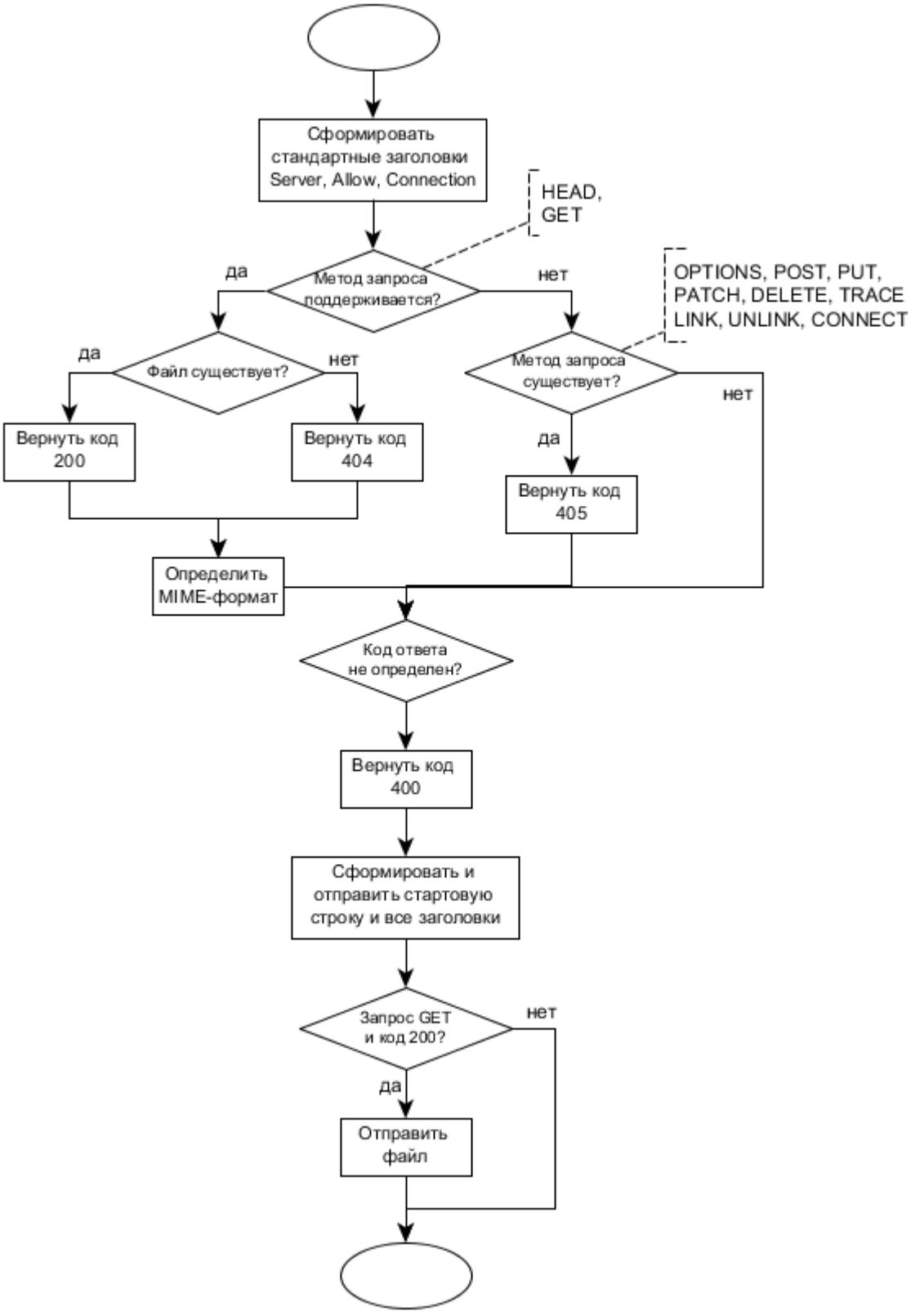
`Request` – полный текст входящего запроса, `Socket` – через какой сокет мы отправляем ответ. С точки зрения «красивого» программирования правильнее было бы не использовать сокет в этой функции, а вернуть код и текст ответа, но, в данном случае, это сделает код более громоздким и сложным для понимания.

Не забудьте объявить ее в описании формы.

Логика работы функции показана на блок схеме ниже.

Функция должна:

1. Определить код ответа (зависит от метода запроса, существует ли файл). Поддерживаемые коды: 200 (OK), 400 (Неверный запрос), 404 (Файл не найден), 405 (Неподдерживаемый метод).
2. Сформировать заголовки (Server, Allow, Connection – всегда одинаковые, Content-Type, Content-Length зависят от файла).
3. Отправить стартовую строку и заголовки.
4. Если нужно (метод GET, код 200), отправить файл.
5. Занести отправленный ответ в лог. Вместо данных из файла указать строку `<Содержание файла *имя файла*>`.



Нам потребуются строковые и числовые переменные

```
Method: AnsiString; // метод запроса
FileName: AnsiString; //имя запрашиваемого файла
ResponseCode: Word; // код ответа (беззнаковое целое)
StartLine, Headers: AnsiString; // части ответа
LogMessage: AnsiString;     //запись в лог
```

Для отправки файла воспользуемся типом `TFileStream` – двоичный поток данных:

```
  RequestedFile: TFileStream; //поток для чтения файла
```

Кроме файловых, существуют и другие типы потоков (stream) – данные в оперативной памяти, данные из сети, с какого-то устройства и др. Очень легко передавать данные из одного потока в другой. В том числе, у сокетов есть метод `SendStream` – отправка данных из указанного потока.

Сформируем стандартные заголовки, начало стартовой строки, код ответа пока неизвестен:

```
Headers := 'Server: SimpleServer v1.1'#13#10;
Headers := Headers + 'Allow: HEAD, GET'#13#10;
Headers := Headers + 'Connection: Close'#13#10;
StartLine := 'HTTP/1.1 ';
ResponseCode := 0;
```

Сочетание #13#10 ознчает переход на новую строку (для Windows). # означает, что дальше идет ASCII-код символа в десятичном виде. #13 – CR (клавиша Enter), #10 – LF переход на новую строку. Подробнее см. в Википедии, статья «Перевод строки».

Таким образом, в «однострочную» переменную мы помещаем несколько строк заголовков.

Считываем метод запроса. В `Request` он идет сразу же в начале запроса и до первого пробела. Собственно, из запроса мы пока берем всего лишь 2 вещи – метод запроса и URL. Все остальное игнорируется.

```
Method := copy(Request, 1, pos(' ', Request) - 1);
```

URL находится между методом и `' HTTP'` (с пробелом):

```
FileName := copy(Request, 1, pos(' HTTP', Request) - 1);
Delete(FileName, 1, Length(Method) + 1);
```

Сразу можно преобразовать URL к пути к файлу через ранее написанную функцию:

```
FileName := FileFromURL(FileName, RootDir);
```

Теперь, если метод `HEAD/GET`, если файл существует – код ответа 200.

```
if (Method = 'HEAD') or (Method = 'GET') then
  begin
  if FileExists(FileName) then
    begin
    ResponseCode := 200;
    StartLine := StartLine + '200 OK';
```

Файл загружаем в переменную `RequestedFile`. Для этого создаем объект `TFileStream`, передав путь к файлу, метод доступа – только чтение.

```
    RequestedFile := TFileStream.Create(FileName, fmOpenRead);
```

Нужно определить MIME-формат и размер файла. Для определения размера файла в байтах, у `TFileStream` есть свойство `Size`.

```
    Headers := Headers + 'Content-Type: '
                + MIMEFromExt(ExtractFileExt(FileName)) + #13#10;
    Headers := Headers + 'Content-Length: '
```

```
                + IntToStr(RequestedFile.Size) + #13#10;
    end
```

Если файл не найден, ответ 404, длина данных – 0 байт, в логе отдельным сообщением указываем, что файл не найден:

```
  else
    begin
    ResponseCode := 404;
    StartLine := StartLine + '404 Not Found';
    Headers := Headers + 'Content-Length: 0'#13#10;
    AddToLog('файл ' + FileName + ' не найден');
    end;
  end
```

Если метод не GET и не HEAD, проверяем, а существует ли такой метод вообще. Если он есть в списке методов HTTP, то код – 405.

```
else if (Method = 'OPTIONS') or (Method = 'POST') or (Method = 'PUT')
   or (Method = 'PATCH') or (Method = 'DELETE') or (Method = 'TRACE')
   or (Method = 'LINK') or (Method = 'UNLINK') or (Method = 'CONNECT') then
  begin
  ResponseCode := 405;
  StartLine := StartLine + '405 Method Not Allowed';
  end;
```

Наконец, если после всех этих проверок, мы так и не определили код ответа (равен 0), значит наш сервер не смог распознать запрос, код – 400:

```
if ResponseCode = 0 then
  StartLine := StartLine + '400 Bad Request';
```

Теперь мы сформировали `StartLine` и `Headers`, можно отправить их и вписать в сообщение лога с указанием, кому именно отвечаем:

```
Socket.SendText(StartLine + #13#10 + Headers + #13#10);
LogMessage := 'ответ клиенту ' + Socket.RemoteAddress + #13#10
  + StartLine + #13#10 + Headers + #13#10;
```

В конце не забываем добавить пустую строку.

Если нужно, отправляем файл, иначе – освобождаем память. Особенностью метода `SendStream` является то, что после него память освобождать не нужно. Не забываем вписать в LogMessage, отправил ли сервер файл.

```
if ResponseCode = 200 then
  if Method = 'GET'
    begin
    Socket.SendStream(RequestedFile);
    LogMessage := LogMessage + '<Содержимое файла ' + FileName + '>';
    end
  else
    RequestedFile.Free;
```

Добавляем в лог полученное сообщение:

```
AddToLog(LogMessage);
```

И, наконец, запишем обработчик запроса от клиента. Напомню, что за получение запросов отвечает событе `onClientRead` сокета-сервера.

```
procedure TFMain.srvSocketClientRead(Sender: TObject; Socket: TCustomWinSocket);
```

Необходимо считать в текстовую переменную полученный запрос (функцию мы писали в самом начале).

```
Request := ReceiveAnsiText(Socket);
```

Затем внести в лог информацию о пришедшем запросе

```
AddToLog('получен HTTP-запрос от '
        + Socket.RemoteHost + ' (' + Socket.RemoteAddress
        + ':' + IntToStr(Socket.RemotePort) + ')');
AddToLog('запрос от клиента ' + Socket.RemoteAddress + #13#10
         + Request);
```

Формируем отклик через нашу процедуру `Response`:

```
Response(Request, Socket);
```

Поскольку наш сервер пока не поддерживает соединение, сокет сразу закрываем:

```
Socket.Close;
```

Для избежания возможных ошибок, в начало функции следует добавить проверку доступности сокета:

```
if Socket.Connected then
```

В большинстве случаев она не нужна, т.к. если с сокета пришло сообщение, он, видимо открыт. Но, если сервер сильно загружен, между моментом прихода сообщения и моментом, когда сервер его обработает, может пройти достаточно много времени, и клиент уже отключится. Без такой проверки это вызовет ошибку.

Сервер с поддержкой соединения

Добавим поддержку соединения и ведение списка клиентов. Список клиентов можно организовать в виде массива, списка или класса, но можно обойтись и списком подключений, который уже есть в `TServerSocket`.

Также, для удобства администрирования, мы будем отображать список подключенных клиентов в `TMemo`, который мы назвали `mmClients`. Более наглядной была бы таблица `TStringGrid`, но зато из `TMemo` список проще скопировать.

При подключении каждому клиенту будем задавать таймаут, по истечении которого соединение завершается сервером. Таймаут уменьшается по таймеру. Таймаут обычно назначают от нескольких секунд до нескольких минут. Для ускорения тестирования будем использовать таймаут, равный 10сек.

Таким образом, нам необходимо хранить для каждого клиента всего лишь одно число – таймаут. У каждого сокета есть поле с указателем на пользовательские данные `Data: Pointer`. Его можно использовать для любых своих целей.

Подробнее о работе с указателями см. в Приложении.

При подключении клиента (сразу, а не когда придет сообщение, т.е. в событии `onCLientConnect`) необходимо выделить память и вписать туда максимальное значение таймаута. `Pointer` – указатель на данные неопределенного типа, и сколько выделять памяти, непонятно. Поэтому мы объявим локальную переменную, которая явным образом ссылается на целое число:

```
var p: ^Integer;
```

Выделение памяти:

```
New(p);
```

Запись данных:

```
p^ := 10;
```

Если приходится часто менять таймаут, то лучше записать его в глобальную константу в начале программы, чтобы не искать по всему тексту, или хотя бы пометить узнаваемым комментарием.

Нового клиента дописываем в конец списка на форме вместе с таймаутом:

```
lstClients.Lines.Add(Socket.RemoteAddress
      + ' (' + Socket.RemoteHost + ':' + IntToStr(Socket.RemotePort) + ') '
      + IntToStr(p^) + 'c.');
```

По срабатыванию таймера `Timer1`, событие `onTimer`, необходимо для всех клиентов уменьшить таймут на 1. У кого он истек – отсоединить сокет и удалить из списка клиентов.

Но именно удалять из списка не очень удобно – нужно каждый раз искать нужную строчку. Проще сначала очистить список, а потом добавить только оставшихся клиентов.

```
lstClients.Clear;
```

Перебирать соединения будем в цикле `for`, а для краткости кода воспользуемся конструкцией `with`:

```
for i := 0 to srvSocket.Socket.ActiveConnections - 1 do
 with srvSocket.Socket.Connections[i] do
  begin
```

Для доступа к данным нам опять потребуется указатель на тип Integer.

```
  p := Data;
  p^ := p^ - 1;
```

Напрямую вычитать 1 из `Data` нельзя, т.к. неизвестно, что там за данные (это может быть и текст, и дробное число, и что-то другое), т.е. следующая запись неверна:

```
  Data^ := Data^ - 1;
```

Также сравнивать с числом можно только типизированный указатель. Если таймаут истек, закрываем сокет, иначе – заносим в список клиентов:

```
  if p^ <= 0 then
    Close
  else
    lstClients.Lines.Add(RemoteAddress
        + ' (' + RemoteHost + ':' + IntToStr(RemotePort) + ') '
        + IntToStr(p^) + 'c.');
  end;
```

Но клиент мог и не запрашивать поддержку соединения. Если в тексте запроса нет заголовка `'Connection: Keep-alive'`, то соединение обрываем сразу, в `srvSocketClientRead` (замените соответствующую строку):

```
  if (pos('connection: keep-alive', LowerCase(Request)) <= 0) then
```

```
    Socket.Close
  else;
  end;
```

Если клиент так и не пришлет сообщение, то он будет отключен по таймауту.

Также, в процедуре Response необходимо добавить соответствующий заголовок ответа:

```
if (pos('connection: keep-alive', LowerCase(Request)) <= 0) then
  Headers := Headers + 'Connection: Close'#13#10
else
  Headers := Headers + 'Connection: Keep-Alive'#13#10;
```

Еще одно действие, которое мы должны сделать – удалить из памяти таймер, привязанный к сокету. Сделать это можно в `onSocketDisconnect`:

```
Dispose(Socket.Data);
```

Пока клиентов мало, важность этой строчки невелика, но чем чаще клиенты подключаются и отключаются, тем больше будут утечки памяти.

Приложение

Указатели (динамические переменные)

Важным типом данных в программировании является **указатель** (**pointer**). Указатель, в отличие от обычной статической переменной, не содержит конкретных данных, а только хранит адрес ячейки в памяти, начиная с которой эти данные находятся.

Важное отличие указателей от обычных переменных – память под них выделяется динамически, т.е. во время работы программы, а не при ее запуске.

Т.е., например, когда вы объявляете переменную типа `Integer`, при запуске программы под нее сразу выделится 4 байта, для `Byte` – 1 байт, для `Word` – 2 байта, для `Double` – 8 байт и т.д. Эти байты будут зарезервированы, пока программа работает, даже если фактически использоваться не будут.

Когда объявляется переменная-указатель, в программе под нее резервируется всегда 4 байта (длина адреса памяти), а место под данные нужно выделить в тексте программы, там, где переменная используется и столько, сколько нужно. А когда данные больше не нужны, мы должны сами удалить их из памяти.

Объявление указателей

Переменную-указатель можно объявить как тип `Pointer` – *нетипизированный указатель* на произвольные данные.

```
var
  p: Pointer;
```

В этом случае размер хранимых данных неограничен, программист сам должен следить за этим.

Зачастую на этапе написания программы уже известно, какие именно данные будут использоваться. Тогда используют *типизированный указатель* вида:

```
var
  p: ^Intger;
  q: ^Byte;
  s: ^String;
  a: ^TMyArray; //TMyArray - пользовательский тип, объявленный в type
```

Символ ^ означает, что переменная является указателем.

Для стандартных типов существуют встроенные типы-указатели:

```
var
  p: PIntger;
  q: PByte;
  s: PString;
```

Для типизированных указателей программа автоматически выделяет ровно столько памяти, сколько нужно.

Выделение и освобождение памяти

Для выделения памяти типизированных указателей используется процедура `New`, например:

```
var
  p: ^Intger;
begin
New(p);
//...
```

В результате в памяти будет выделено 4 байта под данные `Integer`, а в `p` будет записан адрес первого из этих 4-х байтов. Но никакие данные по этому адресу записаны не будут, там находится «мусор» – абсолютно что угодно, что было в памяти раньше.

Для освобождения памяти типизированного указателя используется процедура Dispose

```
var
  p: ^Intger;
begin
New(p);
//...
Dispose(p);
ens;
```

После этого 4 байта, выделенные под `Integer`, будут освобождены и могут использоваться другими программами. Но в самой переменной `p` по-прежнему хранится старый адрес данных! И если попытаться обратиться к этому адресу, в лучшем случае получим ерунду, в худшем – программа вылетит с ошибкой AccessViolation, т.е. ошибка доступа.

Поэтому после освобождения памяти рекомендуется в переменную вписать значение `nil`, т.е. отсутствие указателя.

```
var
  p: ^Intger;
begin
New(p);
//...
Dispose(p);
p := nil;
ens;
```

Для нетипизированного указателя программисту нужно явно вписать, сколько байт выделить, используется процедура `GetMem`:

```
var
  p: Pointer;
begin
GetMem(p, 100);
//...
```

Мы выделили 100 байт памяти.

Если мы хотим поместить в Pointer какой-то конкретный тип, например, тот же Integer, или собственный `TMyData`, длину которого мы не помним или не знаем, можно использовать функцию `SizeOf`, которая возвращает длину данных в байтах

```
var
  p, q: Pointer;
begin
GetMem(p, SizeOf(Integer));
//...
GetMem(q, SizeOf(TMyData));
//...
```

При освобождении памяти также нужно явно указать ее размер. Процедура называется `FreeMem`:

```
var
  p: Pointer;
begin
GetMem(p, 100);
//...
FreeMem(p, 100);
p := nil;
end;
```

Сколько памяти выделяли, столько и освобождаем. Замечание про `nil` верно и здесь.

Кстати, аналогично работают переменные-экземпляры классов. У каждого класса есть метод `Create` для выделения памяти и метод `Free` для освобождения памяти. В переменной также хранится адрес, который не определен, пока ее не создадут и соответствующий участок памяти не будет выделен. Условный пример:

```
var obj: TObject;
begin
obj := TObject.Create;  //память выделена
//...
obj.Free;   //память осовобождена
obj := nil; //подтираем адрес
end;
```

Действия с данными

Указатели нужны для того, чтобы работать с данными, на которые они указывают. Доступ к данным можно получить с помощью того же символа `^`, поставив его в конец переменной. В таком виде ими можно пользоваться как обычными переменными:

```
var
  p: ^Intger;
  a: Integer;
begin
a := 100;
New(p);
Readln(p^);
p^ := a + ^p;
a := p^ - a;
if a > p^ then
  Writeln(a)
else
  Writeln(p);
Dispose(p);
p := nil;
ens;
```

В Delphi допускается пропускать ^ для доступа к данным, но «красивый» код предполагает явное использование ^.

Обратная операция – получение адреса обычной переменной или другого объекта в памяти. Записывается оператором @ или функцией Addr

```
p := @a;
p := Addr(a);
```

В этом случае p становится полным синонимом a, т.к. их данные хранятся по одному и тому же адресу. Если теперь записать:

```
p := 50;
```

то изменятся оба значения: и p, и a. a^ и p^ всегда будут равны. Освобождать память в этом случае не нужно, т.к. там находится статическая переменная a.

Действия с указателями

С самими указателями тоже можно выполнять некоторые действия, т.к. адрес в памяти – это тоже целое число.

Можно присваивать указатели друг другу, если у них совпадают типы, или нетипизированному указателю присвоить любой другой:

```
var
  p,q: ^Double;
  u: Pointer;
begin
New(p);
p^ := 89;
q := p;
u := p;
```

В этом случае все три указателя ссылаются на *одни и те же* данные в памяти. Если записать:

```
p^ := ^p - 10;
```

то значение изменится для всех трех указателей.

Освобождать память нужно только один раз (через любую переменную).

```
Dispose(p);
p := nil;
q := nil;
u := nil;
```

Но если записать:

```
New(q);
q^ := p^;
```

то создастся копия значения из p в q. И если после этого записать в p другое число, на q это никак не повлияет. Освобождать память нужно от каждой переменной отдельно.

Значения указателей можно сравнивать между собой, и с nil.

```
p = nil //пустой, ни на что не указывает
p = q // указывают на одни и те же данные
p <> q //указывают на разные данные, или один из них, или оба = nil
p > q //адрес p больше адреса q (находится в памяти дальше)
```

Также можно прибавлять/вычитать к указателям целые числа или вычитать одно из другого:

```
p := q + 10; //через 10 байт после q
q - p = 10; //рассотяние между p и q равно 10 байт
p := @(a[10]) - 4; //адрес 9-го элемента массива a, если в нем сожержится Integer
```

Другие арифметические действия над указателями бессмысленны.