UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS132 - Imperative Programming

Practical 3 Specifications

Release Date: 03-03-2025 at 06:00

Due Date: 07-03-2025 at 23:59

Late Deadline: 09-03-2025 at 23:59

Total Marks: 60

Read the entire specification before starting with the practical.

Contents

1 General Instructions

- *Read the entire assignment thoroughly before you begin coding.*
- This assignment should be completed individually.
- **Every submission will be inspected with the help of dedicated plagiarism detection software.**
- Be ready to upload your assignment well before the deadline. There is a late deadline which is 48 hours after the initial deadline which has a penalty of 20% of your achieved mark. **No extensions will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or structure).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of C++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use C++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software to generate submitted code is strictly forbidden and will be considered as plagiarism.

2 Overview

In this practical, you will be exposed to the string data type, as well as inputting data into your program and outputting the data to the terminal. You will also be exposed to apply some mathematical operators and string manipulation.

3 Background

User Datagram Protocol (UDP) is a simple and fast way for computers to send data over a network without establishing a connection. UDP does not guarantee that messages will arrive in order or even at all, making it less reliable but much faster. It is lightweight, meaning it has minimal extra information in each message, which helps reduce delays. This makes UDP ideal for applications where speed is more important than accuracy, such as online gaming, video streaming, voice calls (VoIP), and DNS lookups. While some data may be lost, UDP ensures quick communication without delays in error checking and retransmission.

A **UDP packet** (or datagram) consists of a small header and the actual data being sent. The header is **8 bytes** long and contains four fields in this order:

- **Source Port**[1] **(2 bytes**[2]**)** – Identifies the sender's port number.
- **Destination Port (2 bytes)** – Identifies the receiver's port number.
- **Length (2 bytes)** – Specifies the total size of the packet (data)[3].
- **Checksum (2 bytes)** – Used for basic error detection, but errors are not corrected.

The **data section** contains the actual information being transmitted, and starts after the checksum portion of the packet. The data section then continues till the end of the packet. Since UDP does not guarantee delivery, there are no extra fields for tracking or resending lost packets. This makes UDP fast and efficient for real-time applications such as streaming, gaming, and VoIP.

Consider the following information:

- Source Port: 7822
- Destination Port: 6674
- Message: HelloWorld

Using the algorithms, you need to develop in this practical, we can express the UDP packet as follows:

```
0001111010001110 0001101000010010 0000000000110010 1111111100000000
0011100100010110 1011011101011001 1101000101011000 11
```

Which can then be extracted to:

- Source Port: 0001111010001110
- Destination Port: 0001101000010010
- Length: 0000000000110010
- Checksum: 1111111100000000
- Message: 00111001000101101011011101011001110100010101100011

The precise details of the encoding of the message and the creation of the checksum will be discussed later in this practical.

[1]Ports have a range of possible values ranging from 0 to 65535.

[2]A single byte is represented by 8 bits.

[3]Note: In the normal UDP packet, the length will be for the header and the data, but for this practical, we will only consider the data.

Task	Marks
1_InputOutput	8
2_UDP_Binary[4]	10
3_UDP_Other[5]	10
4_NetworkInterface	12
5_FormUDP[6]	4
6_Printing[7]	10
Testing	6

Table 1: Mark allocation

4 Your Task:

You are required to implement all of the functions laid out in this section. You have only been supplied with the `Header` files and will need to create your own `Implementation` files. See Section 7 for the names of the files you need to upload.

4.1 InputOutput

This file is responsible for the generic input and output pre-defined processes that is utilised in this practical.

4.1.1 Functions

- `std::string askForStringInput(std::string message)`
  - This function is responsible for displaying the `message` to a user and then retrieving input from the user.
  - The `message` printout should be followed with a newline.
  - The input that is received from the user should be stored in a string variable before returning the variable.
- `int askForIntInput(std::string message)`
  - This function is responsible for displaying the `message` to a user and then retrieving input from the user.
  - The `message` printout should be followed with a newline.
  - The input that is received from the user should be stored in an `int` variable before returning the variable.
- `void printOut(std::string message)`
  - This function should just display the `message` followed by a newline.

4.2 UDP

This file contains all the functionality required to create and extract data from UPD Packets.

4.2.1 Functions

- `int hexBinToInt(std::string bits)`
  - This function is responsible for converting a string that consists of '0's and '1's to an integer value.
  - Your code should use the 16 left most characters in the `bits` array.
  - In order to do the conversion the following equation can be used:

$$\text{hexBinToInt(bits)} = \sum_{i=0}^{15} \text{bits}_i \times 2^{15-i} \tag{1}$$

  - In order to get the numerical value of the characters, '0' and '1', a function has been provided for you in the `ProvidedFiles`, which is discussed in Section 4.4.
  - Example: Take the following bits string: `0001111010001110`
    Using Equation (1), this can be calculated as:

$$\text{hexBinToInt}(0001111010001110) = (0\times2^{15})+(0\times2^{14})+(0\times2^{13})+...+(1\times2^{2})+(1\times2^{1})+(0\times2^{0})$$

  - Note: Due to looping structures not being covered yet, and the fact that the number of bits being used is known beforehand, you should hardcode this calculation.
  - *Hint: the pow C++ function can be used.*
  - This function will not be explicitly tested but will rather form part of the testing of the relevant functions that use it.
- `std::string extractSourcePortBin(std::string UDPPacket)`
  - This function needs to extract the **source port** from the passed in `UDPPacket`, and return it[8].
- `std::string extractDestinationPortBin(std::string UDPPacket)`
  - This function needs to extract the **destination port** from the passed in `UDPPacket`, and return it.
- `std::string extractLengthBin(std::string UDPPacket)`
  - This function needs to extract the **length** from the passed in `UDPPacket`, and return it.

[8]Reminder UDPPackets are highly structured, and follow a fixed format

- `std::string extractCheckSumBin(std::string UDPPacket)`
  - This function needs to extract the **checksum** from the passed in `UDPPacket`, and return it.
- `std::string extractDataBin(std::string UDPPacket)`
  - This function needs to extract the **data** from the passed in `UDPPacket`, and return it[9].
- `int extractSourcePortInt(std::string UDPPacket)`
  - This function needs to extract the **source port** from the passed in `UDPPacket`, convert it to `int` and then return it.
  - *Hint: a function already discussed can help with this conversion.*
- `int extractDestinationPortInt(std::string UDPPacket)`
  - This function needs to extract the **destination port** from the passed in `UDPPacket`, convert it to `int` and then return it.
- `int extractLengthInt(std::string UDPPacket)`
  - This function needs to extract the **length** from the passed in `UDPPacket`, convert it to `int` and then return it.
- `int extractCheckSumInt(std::string UDPPacket)`
  - This function needs to extract the **checksum** from the passed in `UDPPacket`, convert it to `int` and then return it.
- `std::string extractDataStr(std::string UDPPacket)`
  - This function needs to extract the **data** from the passed in `UDPPacket`, convert it to `letters` and then return it.
  - Each letter is represented by 5 bits. For example: A = 00000, B = 00001, H = 00111.
  - *Hint: there is a function in the `ProvidedFiles` file that can assist with this.*

[9]Remember the data part of the packet starts at a certain point and run till the end of the packet

4.3 NetworkInterface

This file acts as the interface that can communicate with the user to create or extract the UDP packet.

4.3.1 Members

- `std::string dataSourceRequest`
  - This member stores the request for the **source port** of the UDP packet.
  - Initialise the request as follows in the `NetworkInterface.cpp`:

    ```
    "Please enter the source port: "
    ```

- `std::string dataDestRequest`
  - This member stores the request for the **destination port** of the UDP packet.
  - Initialise the request as follows in the `NetworkInterface.cpp`:

    ```
    "Please enter the destination port: "
    ```

- `std::string dataDataRequest`
  - This member stores the request for the **data** of the UDP packet.
  - Initialise the request as follows in the `NetworkInterface.cpp`:

    ```
    "Please enter the message: "
    ```

4.3.2 Functions

- `std::string askUserForSourcePort()`
  - Using the function(s) already discussed, this function should prompt the user for a **source port** and store it as an `int`.
  - The inputted port should then be converted to a 16-bit length binary string and return it.
  - *Hint: There is a function in the `ProvidedFiles` which can be used for this purpose.*
  - You can assume only valid input will be entered[10].

[10]This includes the boundaries of the port number

- `std::string askUserForDestinationPort()`
  - Using the function(s) already discussed, this function should prompt the user for a **destination port** and store it as an `int`.
  - The inputted port should then be converted to a 16-bit length binary string and return it.
  - *Hint: There is a function in the* ***`ProvidedFiles`*** *which can be used for this purpose.*
  - You can assume only valid input will be entered[11].
- `std::string askUserForData()`
  - Using the function(s) already discussed, this function should prompt the user for **data** and store it as a `string`.
  - The inputted data should then be converted to a binary string, where each character is represented by 5 bits.
  - The function should then return this bit string.
  - *Hint: There is a function in the* ***`ProvidedFiles`*** *which can be used for this purpose.*
  - You can assume that only alphabetical characters are present in the input and that the data contains no spaces.
- `std::string formUDPFromUserInput()`
  - This function should prompt the user for all the information required to create a UDP packet.
  - You should first prompt the user for a **source** port, then a **destination** port, and lastly for the **data**, using the functions discussed above.
  - Remember that the functions will return these values as binary strings.
  - The **length** is calculated by obtaining the length of the binary string representation of the data, then converting this length to a 16-bit binary string.
  - The **checksum** should be hardcoded to the value of:

    ```
    "1111111100000000"
    ```

  - The final result is the contamination of the binary strings of the **source** port, **destination** port, **length**, **checkSum**, and **data**.
  - Using the input for the example in Section 3 the returned result should be:

    ```
    000111101000111000011010000100100000000000011001011111111100000000000111001000101101011011101011001110100010101100011
    ```

  - *Hint: It is recommended that you draw a flowchart or use pseudocode to plan out this function before actually implementing it. Tutors will also be instructed to only assist with this function if you have drawn up a flowchart or alternative representation for the algorithm.*

[11]This includes the boundaries of the port number

- `void prettyPrint(std::string UDPPacket)`
  - This function is used to printout out the `UDPPacket` in a more easily readable way.
  - The structure of the output is as follows:

    ```
    Source Port:_<binary representation of the source port>$
    Destination Port:_<binary representation of the destination port>$
    Length:_<binary representation of the length>$
    Checksum:_<binary representation of the checksum>$
    Message:_<binary representation of the message>$
    ```

    where:
    * $ indicates a newline
    * _ indicates a space
    * The < and > indicates what should be printed out.
  - An example output is shown below using the following input for the function:

    000111101000111000011010000100100000000000110010111111110000000000111001000101101011011101011001110100010101100011

    ```
    Source Port: 0001111010001110
    Destination Port: 0001101000010010
    Length: 0000000000110010
    Checksum: 1111111100000000
    Message: 00111001000101101011011101011001110100010101100011
    ```

- `void packetPrint(std::string UDPPacket)`
  - This function is used to print out the `UDPPacket` in a more readable way using an easier representation.
  - The structure of the output is as follows:

    ```
    Source Port:_<int representation of the source port>$
    Destination Port:_<int representation of the destination port>$
    Length:_<int representation of the length>$
    Checksum:_<int representation of the checksum>$
    Message:_<string representation of the message>$
    ```

    where:
    * $ indicates a newline
    * _ indicates a space
    * The < and > indicates what should be printed out.
  - An example output is shown below using the following input for the function:

    000111101000111000011010000100100000000000110010111111110000000000111001000101101011011101011001110100010101100011

    ```
    Source Port: 7822
    Destination Port: 6674
    Length: 50
    Checksum: 65280
    Message: HELLOWORLD
    ```

– Note the functions in the `ProvidedFiles` automatically capitalise the letters for you.

4.4 ProvidedFiles

This file contains functions that are provided. Do not try to understand what the implementation of these functions do. Just use them as per the descriptions below.

4.4.1 Functions

- `int bitToInt(char bit)`
  - This function converts a `bit` '0' and '1' to the `int` value of 0 and 1.
- `std::string intToBin(int number, int length)`
  - This function converts the `number` to a binary string of the specified `length`.
- `std::string stringToBin(std::string str, int length)`
  - This function converts a string consisting of alphabetical symbols only to a binary string.
  - Each letter's binary representation is padded such that the binary string has a length of `length`.
  - For example, if we call the function with "ABC" and 6, we will get the following result:

    ```
    000000000001000010
    ```

    where:
    * 000000 is A
    * 000001 is B
    * 000010 is C
- `std::string binToString(std::string str, int length)`
  - This function converts the passed in binary string, `str`, to its `string` representation, assuming each letter is represented by `length` bits.
  - For example, if the function is called with "001101000101" and 4, we get the result of: DEF

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [12] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 2:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%
40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 2: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **C++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using `namespace std` in any of the files.

[12] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

- You may only use the following libraries:
  - string
  - iostream
  - cmath
  - cstring
- You are supplied with a **trivial** main demonstrating the basic functionality of this assessment.

7 Upload Checklist

The following C++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- `InputOutput.cpp`
- `NetworkInterface.cpp`
- `UDP.cpp`
- `main.cpp`
- Any textfiles used by your `main.cpp`

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder. If you are unsure how to create an archive please review Tutorial 3's slides.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ -std=C++98 -g *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 20 submissions and your final submission's mark will be your final mark. Upload your archive to the Practical 3 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**