

**ETH**

Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# Informatik I (D-ITET)

**Group 7, 17:00-19:00, ETZ E9**

## Exercise 7

**(14/11/2011)**



# Series 6 – Exercise 1 – Stacks with Dynamic Arrays

```
#include <stdlib.h>
#include <iostream>
#include <string>
using namespace std;

int* data; //Pointer auf das Array
int numberOfElements; //Anzahl Elemente im Stack
int sizeOfArray; //Gegenwärtige Kapazität des Arrays

void init() {
    numberOfElements = 0;
    sizeOfArray = 2;
    data = new int[sizeOfArray];
}

void clear() {
    delete[] data;
    numberOfElements = 0;
}

int size() {
    return numberOfElements;
}
```

```
int pop(){
    if (numberOfElements < 1){
        cout << "Stack underrun: Stack is empty!" << endl;
        return -1;
    } else {
        numberOfElements--;
        return data[numberOfElements];
    }
    return 0;
}

void push(int element){
    // Falls Kapazität erschöpft: Neues Array erstellen
    if ( sizeOfArray <= numberOfElements){
        int* tempdata = new int[sizeOfArray];
        for (int i = 0; i < sizeOfArray; i++){
            tempdata[i] = data[i];
        }
        sizeOfArray = 2* sizeOfArray;
        delete[] data;
        data = tempdata;
        delete[] tempdata;
    }
    data[numberOfElements] = element;
    numberOfElements++;
}
```

# Series 6 – Exercise 1 – Stacks with Lists and Pointers

```
#include <stdlib.h>
#include <iostream>
#include <string>
using namespace std;

struct item{
    int key; //Value of the stack element
    item* next; //Pointer to the next item
};

item* first; //Pointer auf oberstes Stapелеlement
int numberOfElements;
```

```
void init() {
    first = NULL;
    numberOfElements = 0;
}
```

```
int size() {
    return numberOfElements;
}
```

```
void clear() {
    while (numberOfElements > 0)
        pop();
}
```

```
int pop(){
    int result;
    if (first != NULL){
        item* temp = first;
        result = first->key;
        first = first -> next;
        delete temp;
        numberOfElements--;
    } else{
        //Stack ist leer -> Fehler
        cout << "Stack underrun: Stack is empty!" << endl;
        result = -1;
    }
    return result;
}
```

```
void push(int element){
    item* ni = new item;
    ni->key = element;
    ni->next = first;
    first = ni;
    numberOfElements++;
}
```

# FILE INPUT/OUTPUT (I/O)

- A C++ program views input or output as a **stream** of bytes.
  - The bytes in the input can come from the keyboard, a file in the hard disk ...
  - The bytes in the output can flow to the display, a printer, a file ...

A **stream** acts as an intermediary between the program and the stream's source or destination.

- This enables a C++ program to treat e.g. input from the keyboard **in the same manner** it treats input from a file.
- You have already seen how the standard input **cin** and output **cout** objects (defined in the **iostream** class library) work.
- In the exercise you will learn a basic scheme to communicate with files either for input or for output.
- In the same way you had to define: `#include <iostream>` you now have now to add at the beginning of your code:

```
#include <fstream>
```

# Simple File Input

1. Create an **ifstream** object to manage the input stream.

e.g. `ifstream fin;`

2. Associate that object with a particular file.

e.g. `fin.open("input.dat");`

3. Use the object in the same way you would use **cin**.

e.g. `char ch; fin >> ch;`

# Simple File Output

1. Create an **ofstream** object to manage the output stream.

e.g. `ofstream fout;`

2. Associate that object with a particular file.

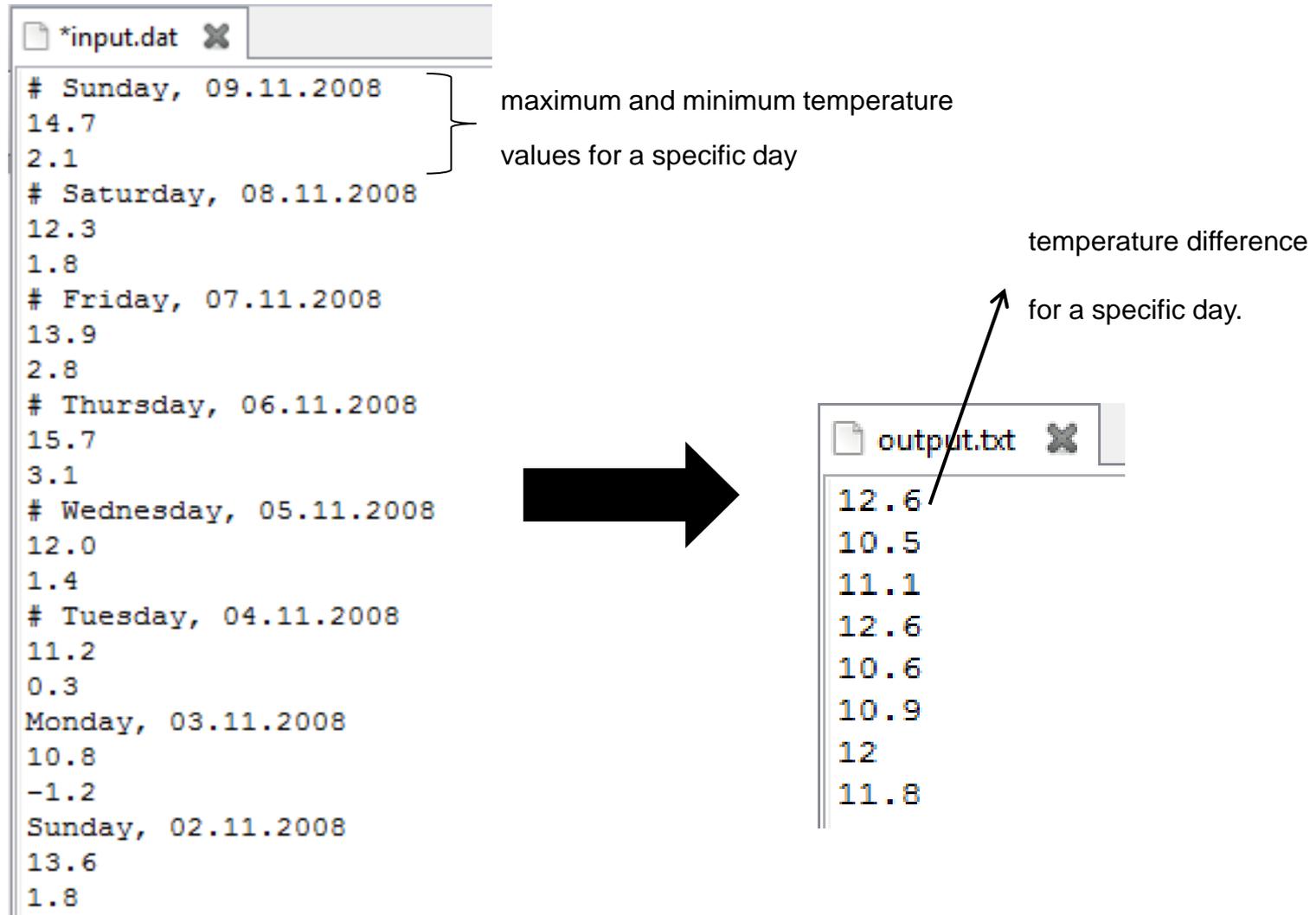
e.g. `fout.open("output.dat");`

3. Use the object in the same way you would use **cout**.

e.g. `fout << "This is a message";`

When you are done with the files, it's good practice to close them: `fout.close(); fin.close();`

# FILE I/O – An example



# FILE I/O – An example

```
#include <iostream>
```

```
#include <fstream>
```

← Header for file streams

```
using namespace std;
```

```
int main()
```

```
{
```

```
    ifstream instream("input.dat");
```

← Input file stream

```
    ofstream ostream("output.txt");
```

← Output file stream

```
    double t_max;
```

```
    double t_min;
```

```
    double diff;
```

```
    string s;
```

```
    ...
```

# FILE I/O – An example

```
while(!instream.eof()) {  
    while(instream.peek() == '#') {  
        getline(instream, s);  
    }  
    instream >> t_max;  
    instream >> t_min;  
  
    diff = t_max - t_min;  
    ostream << diff << endl;  
    getline(instream, s);  
}  
}
```

While you haven't reached the end of the file

peek() returns the next character in the input buffer, without extracting it.

Read the two doubles, representing the minimum and maximum values.

# Binary file IO

- Files can be accessed in binary mode – in C++ this means the file is represented as a sequence of chars – the program is responsible for interpreting this sequence

- Files can be opened in binary mode:

```
ifstream fin(filename, ios::in | ios::binary);
```

- Reading/writing in binary mode:

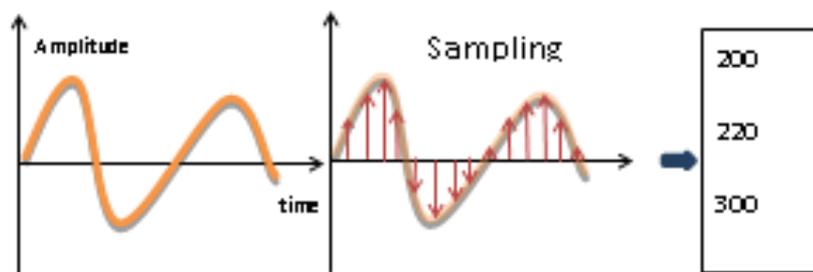
```
short s;
```

```
fin.read((char*)&s, sizeof(s));
```

```
fout.write((char*)&s, sizeof(s));
```

# Audio Data

- Audio data is usually represented on digital computers as a sequence of equally (time) spaced samples
- Each sample represents the amplitude (strength) of the audio at a given point in time
- Sometimes more than one stream is given – e.g. for stereo 2 channels are sampled



## Ex7 – Q2.2

- In this exercise you will be writing a small program to process .wav files
- You are given a template for the program with several unimplemented functions – your job is to implement them
- The program receives arguments from the command line, opens files as necessary and outputs to a file or to the screen

## Ex7 – Q2.2

The template defines a struct to hold a .WAV file header:

```
struct WAVEHEADER {  
    char ChunkID[4];//          Contains the letters "RIFF" in ASCII form  
    int ChunkSize;  
    int Format;  
    char Subchunk1ID[4];  
    int Subchunk1Size;  
    short AudioFormat;  
    short NumChannels;  
    int SampleRate;  
    int ByteRate;  
    Short BlockAlign;  
    short BitsPerSample;  
    char Subchunk2ID[4];  
    int Subchunk2Size;  
};
```

## Ex7 – Q2.2

The following function should print the .wav file information to the screen in the following format:

Codec: PCM, 44100 Hz, 16bit, 1 channel: Length 9.81 sec

```
void printWaveHeaderInfos( WAVEHEADER *hdr) {
    if(NULL == hdr) {
        cerr << "Keine Waveheaderinformationen gefunden" << endl;
    }

    // output
    // codec, sampling rate, sampling precision, channels, length

    /* TODO */
}
```

## Ex7 – Q2.2

The following function should read the header from a .wav file into a newly allocated struct, or return NULL on a read error:

```
WAVEHEADER *readWaveHeader(ifstream *fin) {  
    WAVEHEADER *hdr = NULL;  
  
    //benutze die read Funktion um binaere Daten zu lesen  
    hdr = new WAVEHEADER;  
  
    /* TODO */  
  
    printWaveHeaderInfos(hdr);  
    return hdr;  
}
```

## Ex7 – Q2.2

The following function should convert a .wav file into a text representation, and output this representation into a text file - It should write one number (short) on a separate line per sample:

```
void convertWaveFile( char infile[], char outfile[] )
{
    cout << "Trying to convert wavefile to text:" << endl;

    /* TODO */

    cout << "done" << endl;
}
```

## Ex7 – Q2.2

The following function should adjust the volume of sample and output the new data into a new .wav file.

The function adjusted sample is the original sample times the volume, capped by the maximum sample:

$$s' = \text{cap}(s \times \text{gain})$$

```
void volumeWaveFile( char infile[], char outfile[], double gain )
{
    cout << "Changing Volume:" << endl;

    /* TODO */

    cout << "done" << endl;
}
```

## Ex7 – Q2.2

The following function should mix 2 audio files and output the result into a third audio file.

If one sample is shorter the program should fill in 0s for the missing samples.

Mixing is addition:

$$s' = \text{cap}(s_1 + s_2)$$

```
void mixWaveFiles( char infile1[], char infile2[], char outfile[])
{
    cout << "Mixing 2 audio files:" << endl;

    /* TODO */

    cout << "done" << endl;
}
```

## Ex7 – Q2.2

The following function should add echo to a given audio stream, and output the result to another stream.

Echo is added by adding a scaled version of a sample a constant distance earlier:

$$s[i]' = \text{cap}(s[i] + s[i - \text{delay}] \times \text{echo\_gain})$$

```
void echoWaveFile(  
    char infile[], char outfile[],  
    double delay, double echo_gain )  
{  
    cout << "Adding echo:" << endl;  
    /* TODO */  
    cout << "done" << endl;  
}
```

## Ex7 – Q2.2

- Work on each function separately, make sure it works and only then move on to the next (Use your own main function for testing)
- Whenever you have shared functionality create a function with that functionality and use it – do not copy-paste code!
- Use only as much memory as necessary – don't store the whole audio file in memory – only for the echo function you have to keep a buffer but keep its size to a minimum
- Make sure that all memory allocated is later deallocated
- Make sure that samples do not over/underflow