



Lecture 4: Data handling in C/C++

1 Introduction to data handling

In the previous class, we discussed how to process and analyze data within our code. In practice, however, the data we need to work with is rarely hard-coded into a program. More often, it is provided in a specific format that we must first import into variables before we can use it. You will not receive or send an e-mail containing a neatly prepared **array** or **vector** of integers. Instead, data is far more likely to arrive in the form of a text file containing numbers, characters, strings, and multiple lines.

In today's class, we will focus on techniques for importing and exporting data in C/C++. More specifically, we will concentrate on how to read data into our programs and prepare it for processing, as well as how to write it back out in specific output formats.

1.1 ifstream

In C/C++, there is a built-in file-streaming library called **fstream**. To use it, you must include it at the beginning of your program:

```
1 #include <fstream>
```

Suppose you are given a `.txt` data file with the following content:

```
1 some_numbers.txt
2 1
3 2
4 3
5 4
6 5
7 6
8 7
9 8
10 9
11 10
12 11
13 12
14 13
15 10
```

The following code demonstrates how to import the dataset shown above. Take a careful look at it and analyze what each part of the code does:

```
1 // data_import.cpp
2 #include <iostream>
3 #include <vector>
4 #include <fstream>
5 using namespace std;
6 int main(int argc, char *argv[]){ // Main begins
7     // Variable declaration
8     ifstream input_data; // This is the variable that will be used to import data.
9     cout << "input through for loop" << endl;
10    input_data.open("some_numbers.txt"); // open the input file
```

```

11  string line_holder; // This string will hold one input line at a time.
12  for (int i = 0; i < 15; i++){ // getline for loop
13      getline(input_data, line_holder); // You can read one line at a time using getline.
14      cout << line_holder << endl;
15  } // getline for loop ends
16  input_data.close(); // Always close your file as soon as you're done working with it.
17  // Each call to getline advances to the next line in the file.
18  // If you need to read the file again from the beginning, close and then reopen it.
19  cout << "input through while loop" << endl;
20  input_data.open("some_numbers.txt"); // (re-)open the input file
21  while (getline(input_data, line_holder)){ // while input begins
22      cout << line_holder << endl;
23      // Here you are directly pushing each line from input_data into line_holder.
24      // While loop ends when the file ends as there is no more line to push into line_holder.
25  } // while input ends.
26  input_data.close();
27  return 0;
28 } // Main ends
    
```

And the output is:

```

for method
some_numbers.txt
1
2
3
4
5
6
7
8
9
10
11
12
13
10
while method
some_numbers.txt
1
2
3
4
5
6
7
8
9
10
11
12
13
10
    
```

A **while** loop is the natural choice for importing data, especially when the size of the dataset is unknown. By contrast, a **for** loop requires the number of input lines to be specified manually. In both approaches we use the **getline** function, which reads an **fstream** line by line. This means it continues reading until the next occurrence of the *delimiter* `'\n'`, which is discarded. Note that a custom delimiter can be provided as an optional third argument instead of the default delimiter `'\n'`.

In this example, we also know that all data following the first line consists of integers. Since we actually do not want to import these values as **strings**, it is better to read them directly into an **array** or a **vector** of integers:

```

1  // data_import_int.cpp
2  #include <iostream>
3  #include <vector>
4  #include <fstream>
5  using namespace std;
6  int main(int argc, char *argv[]){ // Main begins
7      // Variable declaration
8      ifstream input_data; // This is the variable that will be used to import data.
9      input_data.open("some_numbers.txt");
10     string line_holder; // This string will hold one input line at a time.
11     // Removing the "some_numbers.txt" in the header by getting the first line
12     // (typically headers are longer than 1 line).
13     for (int i = 0; i < 1; i++){ // getline for loop
14         getline(input_data, line_holder);
15     } // getline for loop ends
16     // At this point, we know that the remainder of the file contains only integers,
17     // but we don't know how many. Let's create a vector<int> and load them in.
18     int dummy_integer; // This integer will hold one integer at a time.
19     vector<int> input_integers;
20     while (input_data >> dummy_integer){ // while input begins
21         // keep extracting integers until input fails, into dummy_integer
22         // Any content that is not of integer type will make the while condition false
23         // and thus the loop stop.
24         input_integers.push_back(dummy_integer);
25     } // while input ends.
26     input_data.close();
27     // Now let's print and make sure we get what we imported:
28     for (int i = 0; i < input_integers.size(); i++){
29         cout << "input_integers[" << i << "] = " << input_integers[i] << endl;
30     }
31     return 0;
32 } // Main ends
    
```

Running the above code produces the following output:

```

input_integers[0] = 1
input_integers[1] = 2
input_integers[2] = 3
input_integers[3] = 4
input_integers[4] = 5
input_integers[5] = 6
input_integers[6] = 7
input_integers[7] = 8
input_integers[8] = 9
input_integers[9] = 10
input_integers[10] = 11
input_integers[11] = 12
input_integers[12] = 13
input_integers[13] = 10
    
```

As mentioned earlier, a typical file header often spans multiple lines. It is important to inspect the file carefully and determine where the relevant data begins.

To read in the actual data, we use the extraction operator (`>>`) here, which reads from the `fstream` and splits the input into tokens separated by *whitespace characters*, such as spaces, tabs, or newlines. If a token cannot be interpreted as an integer, the condition of the `while` loop evaluates to `false`, and the `ifstream` stops reading further input.

Real-world datasets are usually more complex and often multi-dimensional. Nevertheless, the method for importing the data remains basically the same. Suppose you are given a slightly more complex file, as shown below:

```

1  some_data.txt
2  This is the header and it will start in the next couple of lines.
    
```

```

3
4
5 Or not.
6 x      y      z
7 1      2      87
8 2      4      99
9 3      7      33
10 4      3      11
11 5      2      634
12 6      9      213
13 7      11     41
14 8      31     532
15 9      312    12
16 10     56     76
17 11     13     31
18 12     89     321
19 13     137    12
20 10     1478   94
    
```

There are several ways to handle such a header. The simplest approach is to check the line numbers and skip all lines until line 7. For a multi-dimensional dataset, you can modify the while loop to read and store more than one element from each line:

```

1 // data_import_int_3D.cxx
2 #include <iostream>
3 #include <iomanip>
4 #include <vector>
5 #include <fstream>
6 using namespace std;
7 int main(int argc, char *argv[]){ // Main begins
8     // Variable declaration
9     ifstream input_data; // This is the variable that will be used to import data.
10    input_data.open("some_data.txt");
11    string line_holder; // This string will hold one input line at a time.
12    // Removing the six header lines by getting but abandoning them
13    for (int i = 0; i < 6; i++){ // getline for loop
14        getline(input_data, line_holder);
15    } // getline for loop ends
16    // At this point we know that everything else in the file are integers, explicitly
17    // three per lines, but we don't know how many lines with three integers there are.
18    // Let's create three dummy integers and three vectors of integers to load them.
19    int dummy_x, dummy_y, dummy_z;
20    vector<int> x, y, z;
21    while (input_data >> dummy_x >> dummy_y >> dummy_z){ // while input begins
22        // keep extracting integers until input fails, directly into the three
23        // temporary variables dummy_x, dummy_y, dummy_z, i.e. one for each dimension
24        x.push_back(dummy_x);
25        y.push_back(dummy_y);
26        z.push_back(dummy_z);
27    } // while input ends.
28    input_data.close();
29    // Now let's print and make sure we get what we imported:
30    // By construction, x, y, and z must have the same size.
31    for (int i = 0; i < x.size(); i++){
32        cout << right << "x[" << setw(2) << i << "] = " << setw(8) << x[i] << "   y[" << setw(2)
33        << i << "] = " << setw(8) << y[i] << "   z[" << setw(2) << i << "] = " << setw(8) <<
34        z[i] << endl;
35    }
36    return 0;
37 } // Main ends
    
```

As you can see, data separated by tabs or spaces is very easy to handle. Note that we have used the `iomanip` package to format the `cout` output, here by fixing the width of certain output to `n` through the command `setw(n)`. Running the above code produces the following output:

x[0] =	1	y[0] =	2	z[0] =	87
x[1] =	2	y[1] =	4	z[1] =	99
x[2] =	3	y[2] =	7	z[2] =	33
x[3] =	4	y[3] =	3	z[3] =	11
x[4] =	5	y[4] =	2	z[4] =	634
x[5] =	6	y[5] =	9	z[5] =	213
x[6] =	7	y[6] =	11	z[6] =	41
x[7] =	8	y[7] =	31	z[7] =	532
x[8] =	9	y[8] =	312	z[8] =	12
x[9] =	10	y[9] =	56	z[9] =	76
x[10] =	11	y[10] =	13	z[10] =	31
x[11] =	12	y[11] =	89	z[11] =	321
x[12] =	13	y[12] =	137	z[12] =	12
x[13] =	10	y[13] =	1478	z[13] =	94

Now, let's move on to writing data.

1.2 ofstream

Similar to `ifstream`, you must import `fstream` libraries in order to use `ofstream`. `ofstream` is extremely similar to using a `cout` command. The following is an easy example. Let's re-use the random-number generator from a previous example to output some data.

```

1  // data_export.cpp
2  #include <iostream>
3  #include <cstdlib>
4  #include <vector>
5  #include <fstream>
6  using namespace std;
7  int main(int argc, char *argv[]){ // Main begins
8      // Define the threshold for the random number generation.
9      const int rand_threshold = 999999; // constant integers cannot be changed once declared.
10     vector<int> numbers; // empty numbers vector
11     int sign_generator = 1; // a sign generator to assign + or -
12     for (int i = 0; i < 1000000; i++){ // random vector generator begins
13         sign_generator = -1; // by default the sign is negative
14         if (rand() % 2 == 0){ // roll the dice, if even = positive number.
15             // Statistically, this should hold for 50% of the numbers generated
16             sign_generator = 1; // 50% of the numbers will be positive
17         } // sign generator ends
18         numbers.push_back(sign_generator * (rand() % rand_threshold));
19         // new random number generated and signed, and added to numbers
20     } // random numbers created
21     // variables for linear search
22     int max = -999999;
23     int min = 999999;
24     vector<int> target_index;
25     for (int value : numbers){ // Linear search begins
26         if (value > max){ // max if statement begins
27             max = value;
28         } // max found, if statement ends
29         if (value < min){ // min if statement begins
30             min = value;
31         } // min found, if statement ends
32     } // Linear search ends
33     cout << "max is: " << max << endl;
34     cout << "min is: " << min << endl;
35     // Now, let's create a header and write all of the random numbers into a text file.
36     ofstream output;
37     output.open("random_numbers.txt");
38     output << "random_numbers.txt" << endl;
39     output << "max is: " << max << endl;
40     output << "min is: " << min << endl;
41     for (int value : numbers){ // Linear search begins
42         output << value << endl;

```

```

43 } // Linear search ends
44 output.close();
45 return 0;
46 } // Main ends
    
```

The resulting output file `random_numbers.txt` should appear as follows:

```

1 random_numbers.txt
2 max is: 999998
3 min is: -999995
4 -931732
5 -638629
6 -238759
7 ...
    
```

And at the same time, the shell output is:

```

max is: 999998
min is: -999995
    
```

Note that we could use the `iomanip` package to format the `ofstream` output just in the same way as we did for the `ostream` output with `cout` previously.

Now, let's move on to exporting some 3D data with a minimal header.

```

1 // data_export_3D.cpp
2 #include <iostream>
3 #include <cstdlib>
4 #include <vector>
5 #include <fstream>
6 using namespace std;
7 int get_random_sign(){
8     int sign_generator = -1; // a sign generator to assign + or -
9     if (rand() % 2 == 0){ // roll the dice, if even = positive number.
10         // Statistically, this should hold for 50% of the numbers generated
11         sign_generator = 1; // 50% of the numbers will be positive
12     } // sign generator ends
13     return sign_generator;
14 }
15 int main(int argc, char *argv[]){ // Main begins
16     // Define the threshold for the random number generation.
17     const int rand_threshold = 999999; // constant integers cannot be changed once declared.
18     ofstream output;
19     output.open("random_numbers_3D.txt");
20     output << "random_numbers_3D.txt" << endl;
21     output << "x \t y \t z" << endl;
22     for (int i = 0; i < 100000; i++){ // random vector generator begins
23         output << get_random_sign() * (rand() % rand_threshold) << "\t" << get_random_sign() *
24             << (rand() % rand_threshold) << "\t" << get_random_sign() * (rand() % rand_threshold)
25             << endl;
26     } // random numbers created
27     output.close();
28     return 0;
29 } // Main ends
    
```

The resulting output file `random_numbers_3D.txt` should appear as follows:

```

1 random_numbers_3D.txt
2 x         y         z
3 -931732 -638629 -238759
4 762141  -642610 491377
5 521161  -515893 384966
6 ...
    
```

To make it easy to access the data from LibreOffice Calc (or Microsoft Excel), a commonly used file format is *Comma-Separated Values* (.csv). You can create such a file by modifying the code above as follows:

```

1  // data_export_3D_csv.cpp
2  #include <iostream>
3  #include <cstdlib>
4  #include <vector>
5  #include <fstream>
6  using namespace std;
7  int get_random_sign(){
8      int sign_generator = -1; // a sign generator to assign + or -
9      if (rand() % 2 == 0){ // roll the dice, if even = positive number.
10         // Statistically, this should hold for 50% of the numbers generated
11         sign_generator = 1; // 50% of the numbers will be positive
12     } // sign generator ends
13     return sign_generator;
14 }
15 int main(int argc, char *argv[]){ // Main begins
16     // Define the threshold for the random number generation.
17     const int rand_threshold = 999999; // constant integers cannot be changed once declared.
18     ofstream output;
19     output.open("random_numbers_3D.csv");
20     output << "random_numbers_3D.csv" << endl;
21     output << "x,y,z" << endl;
22     for (int i = 0; i < 100000; i++){ // random vector generator begins
23         output << get_random_sign() * (rand() % rand_threshold) << "," << get_random_sign() *
24             ↪ (rand() % rand_threshold) << "," << get_random_sign() * (rand() % rand_threshold) <<
25             ↪ endl;
26     } // random numbers created
27     output.close();
28     return 0;
29 } // Main ends
    
```

The output should look similar to the following:

```

1  random_numbers_3D.csv
2  x,y,z
3  -931732,-638629,-238759
4  762141,-642610,491377
5  521161,-515893,384966
6  ...
    
```

Note that reading .csv files in C++ is not entirely straightforward. It typically involves reading the data line by line, splitting each line by the comma separators (called *delimiters*) into **strings**, and then converting those **strings** into integers.

We will now start a 60-minute practical programming session.

2 Practical Session – Part I

Creating and reading files – Extend your functions package

1. Write a C++ program that generates and saves datasets. Each output file must include at least 10 header lines containing the following information:
 - title of the data,
 - file name and type,
 - dimension of the dataset,
 - total number of elements,
 - maximum of the elements,
 - minimum of the elements,
 - average (mean) of the elements,
 - median of the elements,
 - sample standard deviation of the elements,
 - sample skewness of the elements.

Create the following datasets (with double-valued data):

- (a) a one-dimensional dataset with 1000 random numbers.
 - (b) a two-dimensional dataset (table format, separated by tabs or spaces) with 10000 random numbers.
 - (c) a three-dimensional dataset (table format, separated by tabs or spaces) with 10000 random numbers.
 - (d) Same as (a)–(c), but saved as comma-separated files (`.csv`).
2. In your functions package, write three functions to read the datasets created in tasks 1 (a)–(c).
 3. Write functions that save (multi-dimensional) datasets stored in an `array` into a file.
 4. Write functions that save (multi-dimensional) datasets stored in a `vector` into a file.

We will now continue with a 15-minute theory session.

3 Parsing Structured Input and Type Conversion

As mentioned earlier, working with a `.csv` file as input is not entirely straightforward in C++. Suppose you are trying to read the file `random_numbers_3D.csv`, which looks as in Fig. 1 if imported to LibreOffice Calc (or Microsoft Excel).

1	A	B	C	D	E	F	G	H
2	x y z							
3		-931732	-638629	-238759				
4		762141	-642610	491377				
5		521161	-515893	384966				
6		457039	-595889	-958155				
7		22391	665356	-703603				
8		-981603	723693	-899292				
9		-20545	-479698	-379026				
10		-546833	899293	596297				
11		-666408	-803710	-270404				
12		242622	999212	497416				

Figure 1: Output `.csv` file from the above code.

Since each line must first be read as a `string` and then split into tokens using delimiters, the `sstream` library is required. To use it, simply add the following include directive:

```
1 #include <sstream>
```

Below is a good example of using `stringstream` to read data from a `.csv` file.

```
1 // read_csv.cpp
2 #include <iostream>
3 #include <vector>
4 #include <fstream>
5 #include <sstream>
6 using namespace std;
7 int main(int argc, char *argv[]){ // Main begins
8     ifstream input;
9     input.open("random_numbers_3D.csv");
10    string temp_string;
11    for (int i = 0 ; i < 2; i++){
12        getline(input,temp_string);
13    } // Headers have been read and skipped
14    vector<int> x, y, z;
15    int counter = 0;
16    char delimiter = ','; // comma as separator
17    while (getline(input, temp_string)){ // input while loop starts
18        stringstream temp_sstream(temp_string); // stringstream created in stack and
19        // then destroyed at the end of each iteration.
20        string temp2_string;
21        while (getline(temp_sstream, temp2_string, delimiter)){ // Separating each line by the
22            ↪ delimiter.
23            counter = counter % 3;
24            switch (counter){ // Switch case is used to determine x, y, or z
25            case 0: // This is x
26                x.push_back(stoi(temp2_string));
27                break;
28            case 1: // This is y
29                y.push_back(stoi(temp2_string));
30                break;
31            case 2: // This is z
32                z.push_back(stoi(temp2_string));
33                break;
34            } // switch case ends
35        }
36    }
37 }
```

```

34     counter++;
35     } // stringstream while loop ends
36 } // input while loop ends
37 input.close();
38 for (int i = 0; i < x.size(); i++){
39     if (i == 10){ break; }
40     cout << " " << x[i] << ", " << y[i] << ", " << z[i] << endl;
41 }
42 return 0;
43 } // Main ends
    
```

The output of the code above is as follows:

```

-931732 , -694458, -638629
-238759 , -886105, -762141
-642610 , -203387, -491377
521161 , 899807, 515893
384966 , 89476, 457039
-595889 , -702861, -958155
22391 , 723140, 665356
-703603 , -515030, -981603
723693 , 134438, 899292
-20545 , -175639, -479698
    
```

In the example above, a new `stringstream` is created at the beginning of each `while` loop iteration. This is perfectly safe: the local `temp_sstream` object is created at the start of each iteration and automatically destroyed at the end. Since it only exists for the duration of one iteration, this approach is generally safe and avoids unnecessary manual memory management. A common mistake is to allocate new *heap* objects inside loops unless dynamic allocation is actually necessary. Repeated allocation and deallocation can introduce unnecessary overhead. In addition, dynamically allocated objects managed through *raw pointers* must be explicitly deallocated; otherwise, memory leaks can occur. Modern C++ typically avoids this problem by using automatic storage or RAII types such as `vector` and *smart pointers*. (The concepts of *stack* and *heap* memory management will be discussed in detail in the next class.)

In the example above, we also use the function `stoi` that converts a `string` containing digits into an integer. For details, see the C++ reference documentation: https://en.cppreference.com/w/cpp/string/basic_string/stol.

This function supersedes an older approach that you might encounter occasionally, where a string is first converted into a C string using the `c_str()` method, and subsequently converted to an integer using `atoi`, like instead of line 25:

```

1 x.push_back(atoi(temp2_string.c_str()));
    
```

The reason `c_str()` exists is largely historical: the C programming language has no built-in string type. Instead, C strings are represented as arrays of characters. C++ later introduced the `string` class, but many existing library functions (such as `atoi`) were originally designed for C-style strings. The `c_str()` method therefore provides a bridge between C++ `string` objects and traditional C strings.

As you may have noticed, `.csv` files are not the only file type that require non-trivial methods for data import. In practice, you may encounter a variety of formats, as well as issues such as corrupted data or even encrypted files, all of which demand careful line-by-line processing and manipulation of subsets of data.

In the next practical session, we will focus on handling different types of data and explore strategies for working with them effectively.

We will now start a 60-minute practical programming session.

4 Practical Session – Part II

Data handling practice – Extend Your Functions Package

1. Recall task 1 (d) of today's first Practical Session. Write functions that read the `.csv` files you created and verify their output by comparing it with the import result in `LibreOffice Calc` (or `Microsoft Excel`).
2. Check explicitly if the datasets read in from the `.txt` and `.csv` files produced in task 1 of today's first Practical Session agree with each other.
3. Create a general function that takes a delimiter character and a `stringstream` as input, and prints each separated element.
4. Create a general function that takes a delimiter character and a `string` as input, and prints each separated element.
5. Write a program that performs *Caesar cipher encryption and decryption* for all 26 possible shift keys. For more information on the *Caesar cipher*, please visit <http://practicalcryptography.com/ciphers/caesar-cipher/#:~:text=The%20Caesar%20cipher%20is%20one,become%20C%2C%20and%20so%20on..>

We will now conclude with a 15-minute question session.

5 Conclusion

You have now learned the basics of data handling. Not only can you analyze data, but you also have the ability to import and export different types of data, regardless of how it is organized. However, it is important to understand that, up to this point, we are still unable to handle very large data sets efficiently.

In many inefficient programs, data is repeatedly written to and read from the storage device. This process is typically very slow, especially on systems that still rely on tape or platter-based hard drives.

Efficient programs, on the other hand, load the data once from the hard drive and keep it in RAM until the program terminates. As long as your machine has enough RAM, you should avoid using the hard drive except for saving the final results of your analysis.

So far, we have only been working with a small portion of RAM known as the *stack*. In the next course, we will begin working with a larger section of RAM, called the *heap*.