Universität Zürich UZH

Physik-Institut

Programmieren in C++
FS 2025

Lecture 3: Data processing in C/C++

1 Introduction to data processing

Up to this point, we have been learning how to use some of the most basic tools for creating a program. Today, we will focus on building programs that process and analyze data. We will keep our examples simple, working with small datasets that fit in the *stack* and operate entirely in RAM.
Now that we understand how to use `if` statements and `for` loops, we will explore how to apply them to process data. In this session, we will cover *linear search*, the `switch–case` construct, and *escape conditions*.

2 Linear search

The term *linear search* refers to searching sequentially through a set of data. In a linear search, each element in the dataset is examined one by one.

2.1 Match

The simplest example is searching for a target value by checking each element in sequence until a match is found. The following code demonstrates one of the simplest implementations using a `for` loop:

```cpp
int some_integers[13] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
int target = 10;
int target_index;
for (int i = 0; i < 13; i++){
  if (some_integers[i] == target){
    cout << "The target is found and it has the index " << i << endl;
    target_index = i;
  }
}
```

This approach is not optimized and leaves plenty of room for improvement. In particular, it becomes ineffective as soon as more than one element in the `array` matches the target value.

```cpp
int some_integers[14] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 10};
int target = 10;
int target_index;
for (int i = 0; i < 14; i++){
  if (some_integers[i] == target){
    cout << "The target is found and it has the index " << i << endl;
    target_index = i;
  }
}
```

For this modified `array` of integers, the program will indicate where the target values are, but it will only store one index. We can easily fix this by storing the target indices in an `array` or, better yet, in a `vector`. A `vector` is preferable here since we do not know in advance how many values in the `array` will match the target.

```cpp
// linear.cxx
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char *argv[]){ // Main begins
  // Variable declaration
  int some_integers[14] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 10};
  int target = 10;
  vector<int> target_index;
  for (int i = 0; i < 14; i++){ // Linear search begins
    if (some_integers[i] == target){ // Target found if statement begins
      cout << "The target is found and it has the index " << i << endl;
      target_index.push_back(i);
    } // Target found if statement ends
  } // Linear search ends
  // Check to make sure that all of the indices have been saved properly.
  for (int i : target_index){ // Checking for loop starts
    cout << i << endl;
  } // Checking for loop ends
  return 0;
} // Main ends
```

This approach is not limited to integers. It can be used to search for matches in strings, characters, or any other property that can be programmed. For example, if you have an `array` of `string`s listing the types of pets people own, you can search to find how many people have a dog — and identify who they are.

```cpp
// linear2.cxx
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char *argv[]){ // Main begins
  // Variable declaration
  string owner[14] = {"Bob", "Donald", "Alice", "Elodie", "Chris", "Johannes", "Roland",
  ↪ "Micael", "Ben", "Ashley", "Paul", "Steven", "Peter", "Alvaro"};
  string pets[14] = {"dog", "cat", "lion", "dog", "bird", "turtle", "hamster", "dog",
  ↪ "guinea pig", "pig", "fish", "dog", "rat", "cat"};
  string target = "dog";
  vector<int> target_index;
  for (int i = 0; i < 14; i++){ // Linear search begins
    if (pets[i] == target){ // Target found if statement begins
      cout << "The " << target << " is found and it has the index " << i << endl;
      target_index.push_back(i);
    } // Target found if statement ends
  } // Linear search ends
  // See who owns a dog
  for (int i : target_index){ // Checking for loop starts
    cout << owner[i] << " has a " << target << endl;
  } // Checking for loop ends
  return 0;
} // Main Ends
```

2.2 Minimum and maximum

Linear search can be applied in different ways. For example, the following code performs a linear search to find both the maximum and minimum values in a given `array`:

```cpp
// minmax.cxx
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char *argv[]){ // Main begins
  // Variable declaration
  vector<int> numbers = {1, 2, 3, 54, 1, 532, 14, 3, 14, 31, 4, 321, 5, 35, 21, 5, 3215,
  ↪  324, 324, 321, 4, 3214, 321, 43, 14, 32, 14, 32, 14, 32, 41, 32, 432, 1, 432, 14, 321,
  ↪  43, 24, 321, 4, 3214, 32, 431, 35, 324321, 161, 6, 17, 34121, 7, 78, 53, 45, 24,
  ↪  -3143, 35, 432, 65, 437, 321};
  int max = -999999;
  int min = 999999;
  vector<int> target_index;
  for (int value : numbers){ // Linear search begins
    if (value > max){ // max if statement begins
      max = value;
    } // max found if statement ends
    if (value < min){ // min if statement begins
      min = value;
    } // min found if statement ends
  } // Linear search ends
  cout << "max is: " << max << endl;
  cout << "min is: " << min << endl;
  return 0;
} // Main ends
```

The output should be as follows:

```
max is: 324321
min is: -3143
```

Note that the variables for `min` and `max` are initialized to 999999 and -999999, respectively, before starting the linear search. This is because, when performing a linear search for the minimum and maximum values, the computer can only determine them by comparing each current value to a reference value. Since you do not know beforehand how large the maximum will be, it is best to give it a very small reference value (e.g., -999999). Likewise, since you do not know how small the minimum will be, it is best to give it a very large reference value (e.g., 999999).

Let's consider another example with a `vector` of random integers:

```
// rand_minmax.cxx
#include <iostream>
#include <vector>
using namespace std;
int main(int argc, char *argv[]){ // Main begins
  // Define the threshold for the random number generation.
  const int rand_threshold = 999999; // constant integers cannot be changed once declared.
  vector<int> numbers; // empty numbers vector
  int sign_generator = 1; // a sign generator to assign + or -
  cout << "RAND_MAX = " << RAND_MAX << endl;
  for (int i = 0; i < 1000000; i++){ // random vector generator begins
    sign_generator = -1; // By default the sign is negative
    if(rand() % 2 == 0){ // roll the dice, if even = positive number. Statistically this is
    ↪  50% of the numbers generated
      sign_generator = 1; // 50% of the numbers will be positive
    } // sign generator ends
    numbers.push_back(sign_generator * (rand() % rand_threshold)); // new random number
    ↪  generated and signed
  } // random numbers created
  // variables for linear search
  int max = -999999;
  int min = 999999;
  vector<int> target_index;
  for (int value : numbers){ // Linear search begins
    if (value > max){ // max if statement begins
      max = value;
    } // max found if statement ends
    if (value < min){ // max if statement begins
```

```
      min = value;
    } // max found if statement ends
  } // Linear search ends
  cout << "max is: " << max << endl;
  cout << "min is: " << min << endl;
  return 0;
} // Main ends
```

Here we used the built-in `C` pseudo-random number generator `rand()`, which returns integer pseudo-random numbers between `0` and `RAND_MAX`. Note that `rand()` is *deterministic*, i.e. it always produces the same sequence when you run the program. You can change the *random seed* though. See https://www.cplusplus.com/reference/cstdlib/rand/ for reference. Note that the qualitiy of these pseudo-random numbers is quite poor in several respects, but it is sufficient for our purposes.

In `C++` you don't need to manually guess the starting values of `min` and `max`. You can use the standard library header `limits` to query the smallest and largest values any data type can hold instead, see https://cplusplus.com/reference/limits/numeric_limits/ for reference. For `int`, e.g. you can do this:

```
// numeric_limits_int.cxx
#include <iostream>
#include <limits>   // for std::numeric_limits
using namespace std;
int main() {
  cout << "Minimum value for int: " << numeric_limits<int>::min() << endl;
  cout << "Maximum value for int: " << numeric_limits<int>::max() << endl;
}
```

Now, let's process some data using linear search.

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

---

3 Practical Session – Part I

Extending your functions package from the previous session

1. Write a random-number generator function (one for `int`, one for `double`) that takes upper and lower limits (possibly signed) and returns a random number in that range.
2. Writa linear-search functions (one for each combination) that return all exact matches of an `int` in an `array`/`vector` of `int`s, and of a `char` in a `string`.
3. Write linear-search functions (one for each combination) that can return the minimum /maximum value in an `array`/`vector` of `double`s.

So far, we have only passed function arguments *by value*. This means that the original variable is copied into a new variable that exists only inside the function. Any changes made to this copy do not affect the original variable outside the function. To allow a function to modify the original variable, you can pass the argument *by reference:*

```
void function_by_reference(double & d); // by reference: original value can be changed
void function_by_value(double d); // by value: a copy is taken; original value unchanged
```

In this case, the original variable is accessed directly. Note that `array`s behave differently: they are treated as pointers, so when passed to a function, the pointer (i.e., the address of the first element) is copied. This means that the function accesses the same underlying data as the caller, and any modifications to the `array` elements inside the function are visible outside as well.

4. Write linear-search functions (one for each case) that can return the following quantities from an `array`/`vector` of `double`s:
   (a) minimum/maximum and how many times they appear.
   (b) median $\hat{\mu}$ and how many times it appears.
   (c) sum of all numbers.
   (d) average $\bar{\mu}$; same function also holds for sample mean $\bar{x}_n = \sum_i^n x_i$.
   (e) sum of the squares of all numbers.
   (f) population variance $\sigma^2 = \frac{1}{N}\sum_i^N x_i^2 - \left(\frac{1}{N}\sum_i^N x\right)^2$ and standard deviation σ.
   (g) sample variance $s^2 = \frac{1}{N-1}\sum_i^N (x_i - \bar{x}_N)^2$ and standard deviation s.
   (h) population skewness $\frac{1}{N}\sum_i^N \left(\frac{x_i-\bar{\mu}}{\sigma}\right)^3$.
   (i) sample skewness $\frac{1}{(N-1)(N-2)}\sum_i^N \left(\frac{x_i-\bar{x}_N}{s}\right)^3$.
   (j) how many numbers are contained in $\bar{\mu} \pm \sigma$, $\bar{\mu} \pm 2\sigma$ and $\bar{\mu} \pm 3\sigma$.
5. Test all of the above using an `array`/`vector` with 10000 random `double`s in the range (-999999., 999999.).

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

---

4 Data sorting

As you may have noticed in the last exercise, many operations can be performed using a single `for` loop. Some libraries also provide built-in functions to compute properties such as the average, median, maximum/minimum, and others. Keep in mind that each time you call such a function, it typically loops over the entire `array` or `vector`.
For example, with 1000000 integers, modern computers can perform these operations very quickly. You can even measure the execution time using the built-in `bash time` function as follows:

```
$ time ./run_rand_minmax.exe
max is: 999998
min is: -999995

real    0m0.144s
user    0m0.031s
sys     0m0.047s
```

Increasing from 1000000 to 100000000 numbers results in the following `time` chart:

```
$ time ./run_rand_minmax.exe
max is: 999998
min is: -999998

real    0m5.183s
user    0m4.125s
sys     0m0.984s
```

There is a way to calculate the approximate computational cost of each operation, but we will not cover that in this course. What you should keep in mind, however, is that each time you call a built-in function that requires looping over all elements in an `array` or `vector`, it consumes time by accessing each element one by one. If you instead program your code so that multiple operations are performed within a single loop, it will be much more efficient.
For example, you should try to write your own function that computes maximum, minimum, average, standard deviation, median, etc. all in one pass through the data, rather than running separate loops for each calculation.
For the duration of this course, we will not be dealing with data sets large enough for this to become critical. Still, depending on your field and application, you may one day need to handle data sets well beyond 100000000 integers.
Another important aspect to pay attention to is that in most cases we also need to categorize and sort our data. This process is sometimes called binning, categorization, or even clustering. Let's look at the following example:

```cpp
// switch_case0.cxx
#include <iostream>
using namespace std;
void checker(int i = 0){
  cout << "I got " << i << endl;
}
int main(int argc, char *argv[]){ // Main begins
  int input = 2;
  if (input == 1){ checker(1); }
  if (input == 2){ checker(2); }
  if (input == 3){ checker(3); }
  if (input == 4){ checker(4); }
  return 0;
} // Main ends
```

The output is the following:

```
I got 2
```

The above code checks the condition separately for inputs `1`, `2`, `3`, or `4`. As you can see, this requires writing a new `if` statement for every case, which quickly becomes tedious and repetitive. Moreover, what if we want to perform a sequence of hardware operations in an organized way? This is where `switch−case` construct becomes useful.

4.1 Switch case

The following example achieves the same result as the previous code, but this time using the `C++ switch−case` construct:

```
// switch_case1.cxx
#include <iostream>
using namespace std;
void checker(int i = 0){
  cout << "I got " << i << endl;
}
int main(int argc, char *argv[]){ // Main begins
  cout << "(int)(NULL) = " << (int)(NULL) << endl;
  int input = 2;
  switch (input){
  case 1: checker(1);
    break;
  case 2: checker(2);
    break;
  case 3: checker(3);
    break;
  case 4: checker(4);
    break;
  }
  return 0;
} // Main ends
```

The output is:

```
I got 2
```

Notice that the `break` statements have to be placed after each `case` block. Also, the individual `case` labels do not need to appear in numerical order. Let's look at a small variation:

```
// switch_case2.cxx
#include <iostream>
using namespace std;
void checker(int i = 0){
  cout << "I got " << i << endl;
}
int main(int argc, char *argv[]){ // Main begins
  int input = 2;
  switch (input){
  case 2: checker(2);
  case 1: checker(1);
  case 4: checker(4);
    break;
  case 3: checker(3);
    break;
  }
  return 0;
} // Main ends
```

The output is:

```
I got 2
I got 1
I got 4
```

You may wonder why this is important. This feature is particularly useful when you are sorting or categorizing items, since you can group several possible values together without writing separate `if` statements. For example:

```cpp
// switch_case3.cxx
#include <iostream>
#include <vector>
using namespace std;
constexpr unsigned int sitranslate(const char* input_string, int char_index = 0){
  return !input_string[char_index] ? 5381 : (sitranslate(input_string, char_index + 1) * 33) ^ input_string[char_index];
}
int main(int argc, char *argv[]){ // Main begins
  vector<string> animals = {"dog", "mouse", "rodent", "pet", "domesticated_animal", "dog"};
  for (string animal : animals){
    int input = sitranslate(animal.c_str());
    switch (input){
    case sitranslate("dog"): cout << "This is a dog, ";
    case sitranslate("pet"): cout << "This is a pet, ";
    case sitranslate("domesticated_animal"): cout << "This is a domesticated animal.";
      cout << endl;
      break;
    case sitranslate("mouse"): cout << "This is a mouse, ";
    case sitranslate("rodent"): cout << "This is a rodent.";
      cout << endl;
      break;
    }
  }
  return 0;
} // Main ends
```

The output is:

```
This is a dog, This is a pet, This is a domesticated animal.
This is a mouse, This is a rodent.
This is a rodent.
This is a pet, This is a domesticated animal.
This is a domesticated animal.
This is a dog, This is a pet, This is a domesticated animal.
```

One limitation is that `switch` in `C++` does not work directly with `string`s. Sometimes people use helper functions that translate strings into integers so that they can be handled in a `switch` statement. (If you are curious, there is a detailed discussion on this topic here: https://stackoverflow.com/questions/2111667/compile-time-string-hashing, but this goes beyond the scope of our course and you do not need to worry about it.)

Similarly, `switch`–`case` statements can be applied to computer input and output. This is particularly useful in programs or hardware that require human interaction, because it allows us to clearly organize and control how the computer should respond to different inputs.

As an example, the following code shows how to use `switch`–`case` to map key presses for a simple computer game. Here the keys `W`, `A`, `S`, `D` act like arrow keys, and the keys `Q` or `X` allow the user to exit the program.

```cpp
#include <iostream>
using namespace std;
#include <curses.h> // needs to be compiled with the flag -lncurses
```

```
// Defining the key presses to integers so that it can be called from case
#define KEY_UP 65
#define KEY_DOWN 66
#define KEY_LEFT 68
#define KEY_RIGHT 67
int main(int argc, char *argv[]){ // Main begins
  initscr(); // initiating screen for ncurses library
  noecho(); // eliminating echoing of the key presses on the screen
  // coordinates
  int x = 2, y = 2;
  move(x, y);
  int hit_key = 0; // initializing key press
  bool loop = true; // escape condition for the while loop
  while (loop){
    switch (hit_key = getch()){
    case 'x':
    case 'q':
      cout << "Pressed e(x)it or (q)uit key";
      cout << "aborting" << endl;
      loop = false;
      break;
    case KEY_UP:
    case 'w':
      move(1, 1);
      addstr("pressing up   ");
      x--;
      move(x, y);
      break;
    case KEY_DOWN:
    case 's':
      move(1, 1);
      addstr("pressing down ");
      x++;
      move(x, y);
      break;
    case KEY_LEFT:
    case 'a':
      move(1, 1);
      addstr("pressing left ");
      y--;
      move(x, y);
      break;
    case KEY_RIGHT:
    case 'd':
      move(1, 1);
      addstr("pressing right");
      y++;
      move(x, y);
      break;
    }
  }
  endwin(); // returning the screen to normal before exiting.
  return 0;
} // Main ends
```

Be mindful that the above example uses a library called `libncurses`. Note that cursor movements with `libncurses` are not compatible with standard `iostream` functions, so you should use `ncurses` functions like `addstr()` instead. Also, your program will not compile unless you link the `ncurses` library by adding the `-lncurses` flag during compilation, for example:

```
g++ -o run_switch_case4.exe switch_case4.cxx -Wall -lncurses
```

or something along these lines in your `Makefile`:

```
inputcode=switch_case4.cxx
CPP=g++
SFLAG=-Wall
PFLAG=-lncurses
@$(CPP) -o $(addprefix run_,$@.exe) $(SFLAG) $(intputcode) $(PFLAG)
```

5 Escape conditions

As you have seen, `switch`–`case` constructions and `while` loops work together to create interactive programs. In fact, nearly all modern devices — computers, phones, remote controls, mice, and even cars — operate within an infinite loop, constantly listening for `case` switches. However, there are times when you want to turn these devices off. In the previous example, recall lines 15 to 24

```
  int hit_key = 0; // initializing key press
  bool loop = true; // escape condition for the while loop
  while (loop){
    switch (hit_key = getch()){
    case 'x':
    case 'q':
      cout << "Pressed e(x)it or (q)uit key";
      cout << "aborting" << endl;
      loop = false;
      break;
```

and lines 55 to 56

```
  endwin(); // returning the screen to normal before exiting.
  return 0;
```

You may notice that this `case` sets the boolean controlling the loop to `false`, causing the `while` loop to terminate and then to `break` out of the current iteration. The following is a summary of escape conditions:

```
break;   // will take you out of the current iteration.
return;  // will take you out of the current function, which is \main in the above case.
exit(0); // will end the program completely.
```

Note that in `run_switch_case4.exe` we always call `endwin();` before exiting `main`. Without this call, your terminal will be broken. In case you have tried, you can usually fix it with the following command (although you will not be able to see what you type):

```
reset
```

If you are not careful with escape conditions, it is very easy to break your terminal session, and in some cases even destabilize the operating system, potentially causing freezes or crashes. Therefore, you must ensure that all of your commands include an escape condition (otherwise the program will be stuck in the infinite loop) and/or a break.

While I have not observed actual hardware damage from a misbehaving program, poorly written code can cause malfunctions in cars, phones, and other devices, potentially leading to serious consequences. Let's now practice implementing proper escape conditions.

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

---

6 Practical Session – Part II

Sorting data – Modify your functions package from our previous session

1. Create a function that determines all *prime numbers* less than or equal to a given input number and stores them in a `vector`.
2. Improve the above function so that previously determined prime numbers are reused when the function is called with a larger input value.
3. Write a `string`-valued function that returns a formatted output of all determined prime numbers.
4. Create a function that calculates the prime factorisation of an arbitrary input number. Store the results in a separate `vector` that contains only the exponents of the corresponding prime numbers appearing in the factorization.
5. Write a `string`-valued function that outputs the prime factorisation of a number in the format $N = p_1^{a_1} \times p_2^{a_2} \times \ldots \times p_n^{a_n}$, including only terms where $a_i \neq 0$.
6. Create two separate functions to calculate the *Greatest Common Divisor* (GCD) and the *Least Common Multiple* (LCM) of two arbitrary input numbers.
7. Generate a `vector` of 100 random numbers in the range $[2; 10000000]$. Sort them by increasing value of $\sum_{i=1}^{n} a_i$; if equal, put that with largest p_i first, and so on.
8. Calculate the pair-wise GCDs and LCMs of all numbers in the random `vector`.

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

---

7 Conclusion

You now have all the tools necessary to process data and analyze a given set of information. Everything we have covered so far is very important for this course and for any type of data analysis. What we are still missing is data handling. We have not yet discussed how to read data from or write data to files.

In the next class, we will cover data handling, where we will import and export data from the computer's storage and use a smaller portion of memory called the *stack* to perform analyses. Improper memory management can lead to issues such as *memory leaks*. The advantage of using the *stack* is that when the program terminates, the memory allocated on the *stack* is automatically returned to the operating system. However, the *stack* also has several limitations. For more information on memory allocation, you can visit: Stack vs Heap Memory Allocation The course on memory management will follow after the data handling session.

Please continue practicing at home and complete any exercises you could not finish today in preparation for tomorrow.