Universität Zürich[UZH]
Physik-Institut

Lecture 6: Large data and templates in C/C++

1 Handling large data in C/C++

Now that we are familiar with handling and processing data on the *heap* and the *stack*, we will turn our attention to working with larger datasets. For the first part of today's lecture, we will focus on exercises and practical programming, applying the concepts we've learned so far. As a result, there will be very little new theory introduced.

1.1 Sampling

Before handling very large datasets, it's important to ensure that your code works correctly with smaller datasets. For instance, if you want to create and manipulate a large number of integers, consider the following example:

```cpp
// declare_large_heap.cxx
#include <iostream>
using namespace std;
void check(char c = 0){
  cout << "Crash point " << c << endl;
}
int main(){
  check();
  long int max = 1000;
  cout << "Expected size of allocated stack memory: \t" << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int a[max]; // allocation of a on the stack
  check('a');
  cout << "Value of pointer: a = \t" << a << endl;
  //  free(a); // gives an error - a resides on stack and is automatically deallocated
  max = 1000000000;
  cout << "Expected size of allocated heap memory: \t " << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int *e = (int*) malloc(max * sizeof(int)); // allocation of e on the heap
  check('e');
  cout << "Value of pointer: e = \t" << e << endl;
  free(e); // free the memory reserved for e on the heap
  max = 10000000000;
  cout << "Expected size of allocated heap memory: \t" << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int *f = (int*) malloc(max * sizeof(int)); // allocation of f on the heap
  check('f');
  cout << "Value of pointer: f = \t" << f << endl;
  free(f); // free the memory reserved for f on the heap
  return 0;
}
```

You will find that the program passes all checkpoints without issues, no matter how much memory you ask `malloc` to allocate. The reason is that `malloc` only reserves the memory, i.e. requests it from the system. If it can be filled at runtime, is not clear at this stage. If you allocate more memory than available in total on the system (basically RAM plus *swap*), `malloc` will return a nullptr, which tells you that the requested memory will not be available. On my virtual system with 4 GB of RAM and 2 GB of *swap*, the output of the above code is (it depends on the memory configuration of the system):

```
$ ./run_declare_large_heap.exe
Crash point
Expected size of allocated stack memory:         3.72529e-06 GB
Crash point a
Value of pointer: a =          0x7ffc1bbb7e30
Expected size of allocated heap memory:          3.72529 GB
Crash point e
Value of pointer: e =          0x774b4674d010
Expected size of allocated heap memory:         37.2529 GB
Crash point f
Value of pointer: f =          0
```

The corresponding version relying on the more modern `C++` *heap* memory allocation commands `new` and `delete` instead of `malloc` and `free` looks as follows:

```
// declare_large_newdelete.cxx
#include <iostream>
using namespace std;
void check(char c = 0){
  cout << "Crash point " << c << endl;
}
int main(){
  check();
  long int max = 1000;
  cout << "Expected size of allocated stack memory: \t" << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int a[max]; // allocation of a on the stack
  check('a');
  cout << "Value of pointer: a = \t" << a << endl;
  //  delete[] a; // gives an error - a resides on stack and is automatically deallocated
  max = 1000000000;
  cout << "Expected size of allocated heap memory: \t" << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int* e = new int[max]; // allocation of e on the heap
  check('e');
  cout << "Value of pointer: e = \t" << e << endl;
  delete[] e; // free the memory reserved for e on the heap
  max = 10000000000;
  cout << "Expected size of allocated heap memory: \t" << double(max) * sizeof(int) / (1024 * 1024 * 1024) << " GB" << endl;
  int* f = new int[max]; // allocation of f on the heap
  check('f');
  cout << "Value of pointer: f = \t" << f << endl;
  delete[] f; // free the memory reserved for f on the heap
  return 0;
}
```

Note that the resulting executables behave slighlt differently: Whereas `malloc` returns a nullptr if the memory requested to be allocated is too large, `new` throws a `std::bad_alloc` error to indicate that the allocation has failed and exits with a non-zero return value (unless this error is caught by exception handling):

```
$ ./run_declare_large_newdelete.exe
Crash point
Expected size of allocated stack memory:         3.72529e-06 GB
Crash point a
Value of pointer: a =          0x7ffec8c85a10
Expected size of allocated heap memory:          3.72529 GB
Crash point e
Value of pointer: e =          0x7aaccdf4d010
Expected size of allocated heap memory:          37.2529 GB
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

Aside from the different ways of signaling whether memory allocation was successful, the behaviour of `malloc/free` and `new/delete` is essentially the same in this context. Therefore, in what follows, we will adhere to the recommended C++ syntax.

Suppose you now want to work with `*f`, i.e., to actually fill it with data. It is only at this stage that the *heap* memory allocation becomes effective. Consider the following sample code:

```cpp
// declare_large_and_fill_newdelete.cxx
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  int max = 700000000;
  cout << "Expected size of allocated memory: " << double(max) * sizeof(int) / (1024 * 1024
  ↪  * 1024) << " GB" <<endl;
  int* f = new int[max]; // allocation of f on the heap
  cout << "Value of pointer: f = \t" << f << endl;
  // random number assignment
  for (int i = 0; i < max; i++){
    f[i] = rand();
  }
  // checking the assigned numbers (and their addresses)
  for (int i = 0; i < max; i++){
    if (&f[i] != nullptr){
      if (i < 10){ cout << i << ":\t" << f[i] << "\t" << &f[i] << endl; }
    }
  }
  delete[] f; // free the memory reserved for f on the heap
  f = nullptr;
  return 0;
}
```

The behaviour of the resulting executable depends on the chosen value of `max` relative to the total and available memory resources of the system. On my virtual machine with 4 GB of RAM and 2 GB of swap space, the chosen value works fine, and the executable produces the following output:

```
$ ./run_declare_large_and_fill_newdelete.exe
Expected size of allocated memory: 2.6077 GB
Value of pointer: f =         0x794d543b6010
0:        1804289383        0x794d543b6010
1:        846930886        0x794d543b6014
2:        1681692777        0x794d543b6018
3:        1714636915        0x794d543b601c
4:        1957747793        0x794d543b6020
5:        424238335        0x794d543b6024
6:        719885386        0x794d543b6028
7:        1649760492        0x794d543b602c
8:        596516649        0x794d543b6030
9:        1189641421        0x794d543b6034
```

It is very important to ensure that your code releases all *heap*-allocated memory once it has finished using it. While modern operating systems will typically reclaim all memory when a program terminates, this should not be relied upon. Failing to free memory inside the code is considered bad practice: the same program that seems harmless in one situation might exhaust system memory in another, for example if memory is allocated and never freed inside a function that is called repeatedly. For this reason, you should make it a habit to always free every piece of memory you allocate.

It is actually quite instructive to observe the memory usage during execution on the *system monitor* (similiar to Fig. 1).

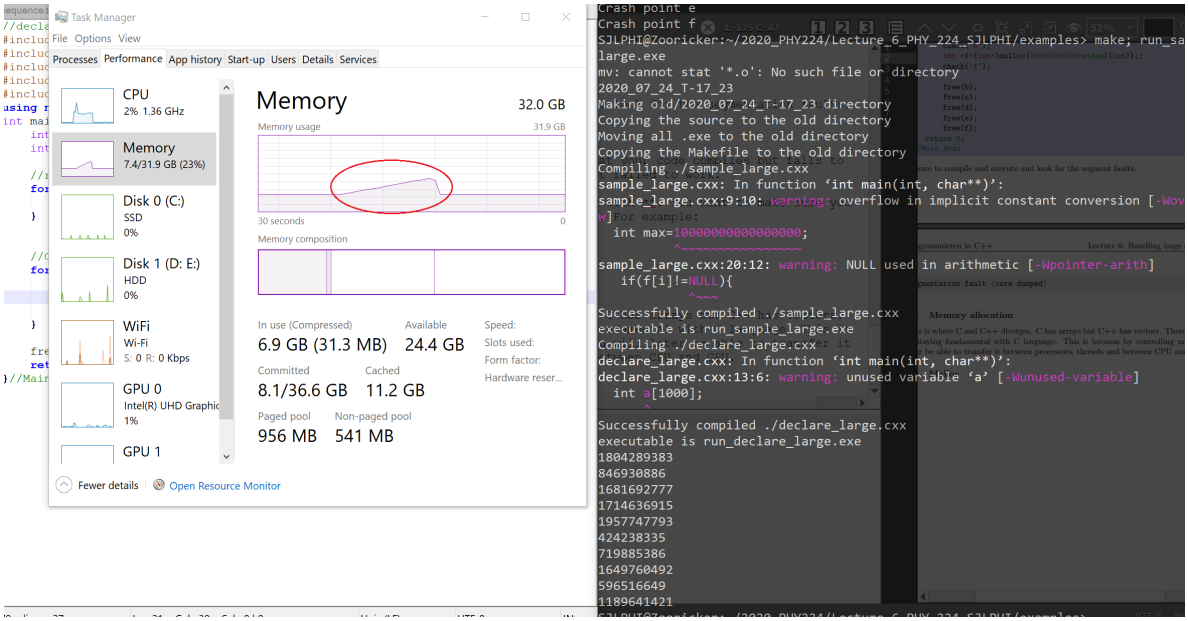

Figure 1: The memory usage is highlighted and peaks at the end of the execution of the above code.

Be aware, however, that if `max` in the above program exceeds the available memory (RAM + swap), the program may fail in two different ways. If the allocation request itself cannot be satisfied, a `std::bad_alloc` exception will be thrown immediately (as early as line 7). In contrast, if the allocation succeeds but the available memory is insufficient to actually use the reserved space — remembering that other programs also consume memory — the operating system may terminate the program during the `for` loop when the memory is first accessed (line 11). This illustrates why careful consideration of memory limits is crucial when working with large data allocations.

Warning: Testing memory limits can destabilize your system if you are not careful. Always be prepared to terminate the process manually. In practice, most operating systems are designed to intervene and kill the process before the entire system becomes unresponsive. If you run these tests inside a virtual machine, the worst-case scenario is usually a crash of the virtual machine itself rather than your primary operating system.

1.2 Freeing memory

The following example illustrates again how memory is allocated, used, and then freed. In this code, memory is allocated twice: once initially, and then again after the first block of memory has been released. Pay attention to how the memory is freed in between, ensuring that the program does not leak memory:

```
1  // declare_large_fill_and_destroy_newdelete.cxx
2  #include <iostream>
3  using namespace std;
4  int main(int argc, char *argv[]){
5    int max = 700000000;
6    cout << "Expected size of allocated memory: " << double(max) * sizeof(int) / (1024 * 1024
     ↪  * 1024) << " GB" <<endl;
7    int* f = new int[max]; // allocation of f on the heap
8    cout << "Value of pointer: f = \t" << f << endl;
9    // random number assignment.
10   for (int i = 0; i < max; i++){
11     f[i] = rand();
12   }
13   // checking the assigned numbers (and their addresses)
```

```
14    for (int i = 0; i < max; i++){
15      if (&f[i] != nullptr){
16        if (i < 10){ cout << i << ":\t" << f[i] << "\t" << &f[i] << endl; }
17      }
18    }
19    delete[] f; // free the memory reserved for f on the heap
20    cout << "Value of pointer: f = \t" << f << endl;
21    f = nullptr; // set f to nullptr
22    cout << "Value of pointer: f = \t" << f << endl;
23    int* g = new int[max]; // allocation of g on the heap
24    cout << "Value of pointer: g = \t" << g << endl;
25     // random number assignment
26    for (int i = 0; i < max; i++){
27      g[i] = rand();
28    }
29    // checking the assigned numbers (and their addresses)
30    for (int i = 0; i < max; i++){
31      if (&g[i] != nullptr){
32        if (i < 10){ cout << i << ":\t" << g[i] << "\t" << &g[i] << endl; }
33      }
34      else { int j = 0; }
35    }
36    delete[] g; // free the memory reserved for g on the heap
37    cout << "Value of pointer: g = \t" << g << endl;
38    g = nullptr; // set g to nullptr
39    cout << "Value of pointer: g = \t" << g << endl;
40    return 0;
41  }
```

Note that it is good practice not only to release *heap* memory once it is no longer needed, but also to reset the corresponding pointer. This does not prevent memory leaks, but setting the pointer to `nullptr` ensures it remains in a well-defined state, rather than pointing to some random memory address that might be used for other purposes in the meantime.

The output of this sample code is:

```
Expected size of allocated memory: 2.6077 GB
Value of pointer: f =         0x7e1eccbb6010
0:        1804289383        0x7e1eccbb6010
1:        846930886        0x7e1eccbb6014
2:        1681692777        0x7e1eccbb6018
3:        1714636915        0x7e1eccbb601c
4:        1957747793        0x7e1eccbb6020
5:        424238335        0x7e1eccbb6024
6:        719885386        0x7e1eccbb6028
7:        1649760492        0x7e1eccbb602c
8:        596516649        0x7e1eccbb6030
9:        1189641421        0x7e1eccbb6034
Value of pointer: f =         0x7e1eccbb6010
Value of pointer: f =         0
Value of pointer: g =         0x7e1eccbb6010
0:        912961091        0x7e1eccbb6010
1:        2056082205        0x7e1eccbb6014
2:        2139653295        0x7e1eccbb6018
3:        846344994        0x7e1eccbb601c
4:        183377509        0x7e1eccbb6020
5:        590176192        0x7e1eccbb6024
6:        148315237        0x7e1eccbb6028
7:        1601732327        0x7e1eccbb602c
8:        1757586571        0x7e1eccbb6030
9:        1150070753        0x7e1eccbb6034
Value of pointer: g =         0x7e1eccbb6010
Value of pointer: g =         0
```

Note that it is normal and expected to get the same address when you allocate, free, and allocate again an object with exactly the same size. However, you must not rely on this behaviour — it is an implementation detail, not part of the `C++` standard. The reason is that the *memory allocator* keeps track of freed blocks — and if you request a new block with the same size, the previous is just a perfect fit.

As in the previous example, it is instructive to observe the memory usage in the system monitor (similar to Figure Fig. 2). Note that memory freed by a program is not always returned to the operating system immediately; instead, it may remain reserved in the process's *heap* for future allocations. For `array`s, you can use the `malloc.h` header to monitor the *heap* memory currently held by the program.

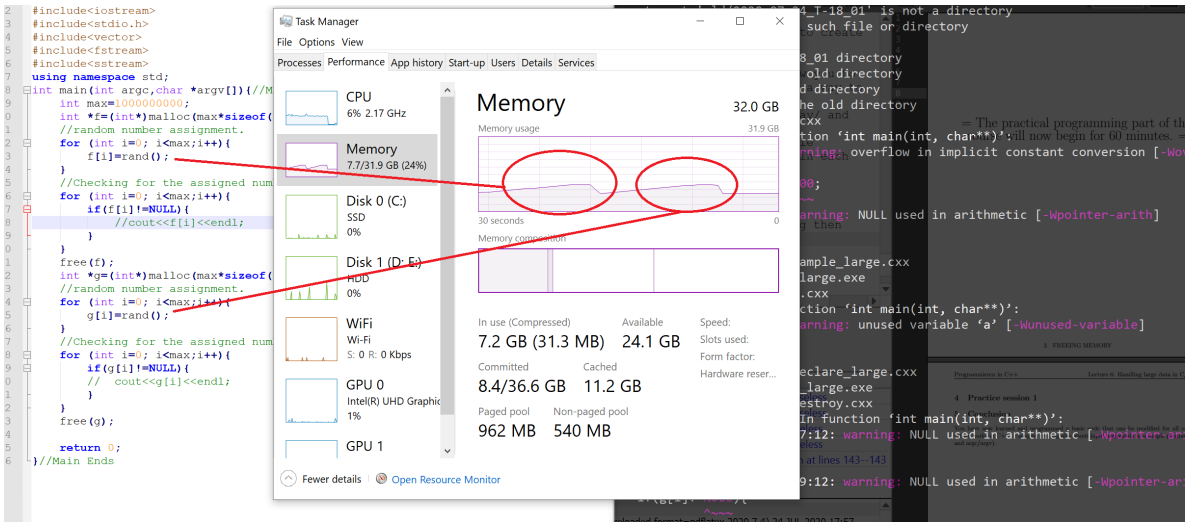

Figure 2: You can see when the memory is being used and freed up within the program.

In the following practical session, we will focus on declaring and then destroying large variables.

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

2 Practical Session – Part I

Handling large data in C/C++ — Extend Your Functions Package

For the following exercises, it is instructive to open the system monitor to observe how memory usage changes during different memory-related operations — and potentially to detect memory leaks. To have full control over the program's execution, it can be useful to implement an optional feature that pauses the program and waits for a key press before continuing (for example, controlled via a command-line argument to `main`).

1. Use the `sizeof` function to calculate the maximum number of the following in 1.5 GB:
   - `int`
   - `short int`
   - `long int`
   - `long long int`
   - `bool`
   - `string`
   - `char`
   - `float`
   - `double`
   - `long double`
2. Create a program that declares, assigns, frees, and deletes the maximum number of each type above, making sure the total memory usage stays below 1.5 GB).
3. Create a random boolean generator that produces an outcome of `true` or `false`.
4. Create a random generator that covers the full range of `unsigned int`.
5. Using the results from exercises 4 and 5, create a function that lets you use a `bool` as the sign of an `unsigned int`, i.e. to effectively store a 5-byte signed `unsigned int`.
6. Create a 2-dim. `array` $(2 \times n)$ of 1 GB and fill it with random unsigned integers.
7. Create a function to calculate the mean, median, sample standard deviation, sample skewness, maximum, and minimum of that `array` for each dimension.

Freeing memory

1. Create two quadratic 2-dim. `array`s ($n \times n$) of 0.5 GB each, and fill them with random integers between `-9999` and `9999`.
2. Create a general function to calculate the mean, median, sample standard deviation, sample skewness, maximum, and minimum of such an `array` along each direction (i.e. for rows and columns).
3. Create a function to perform element-wise addition, subtraction, and multiplication of two 2D `array`s.
4. Modify the above function to store the results and free the memory of the two `array`s used in the calculation.
5. Modify the above function to store the results and free the memory of the two `array`s used in the calculation on-the-fly, such that never much more memory than for the two basic `array`s is required.
6. Create a `vector` of 1 GB, and fill it with random integers between `-9999` and `9999`.
7. Create a function that takes the above `vector`, multiplies each element by 3, outputs a new `vector`, and destroys/frees the old one.

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

3 Function Templates

So far, we have written functions for specific data types. However, many operations follow exactly the same algorithm regardless of the type of data being processed. In such cases, writing a separate function for every possible type would quickly become repetitive. Function templates allow us to write a general function once and let the compiler generate the appropriate version when it is needed.

As an example, consider a function that prints a two-dimensional `vector` in a specific format. We would like the function to work with different element types, such as `int`, `double`, or `string`. Furthermore, we would like to be able to send the output to different kinds of streams, such as the terminal, a file, or a `stringstream`. A first, fully general version could therefore use two template parameters:

```
template <typename S, typename T>
void print_matrix_template(S & output, vector<vector<T> > & data){
  for (vector<T> & row : data){
    output << "( ";
    for (T & value : row){
      output << right << setw(8) << value << " ";
    }
    output << ")" << endl;
  }
}
```

Here, `S` represents the type of the output stream, while `T` represents the type of the elements stored in the two-dimensional `vector`. The following example demonstrates that the same function can now be called with different combinations of streams and element types:

```
// template_function.cxx
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;
template <typename S, typename T>
void print_matrix_template(S & output, vector<vector<T> > & data){
  for (vector<T> & row : data){
    output << "( ";
    for (T & value : row){
      output << right << setw(8) << value << " ";
    }
    output << ")" << endl;
  }
}
int main(){
  vector<vector<int> > integers = {{1, 2, 3}, {4, 5, 6}};
  vector<vector<double> > doubles = {{1.2, 3.4}, {5.6, 7.8}};
  vector<vector<string> > strings = {{"one", "two"}, {"three", "four"}, {"five", "six"}};
  cout << "integers through cout:" << endl;
  print_matrix_template(cout, integers);
  cout << "doubles through ofstream to file output_template.txt" << endl;
  ofstream file("output_template.txt");
  print_matrix_template(file, doubles);
  cout << "strings through stringstream:" << endl;
  stringstream stream;
  print_matrix_template(stream, strings);
  cout << stream.str();
  return 0;
}
```

In each case, the compiler automatically deduces the concrete stream type and the element type from the respective function arguments. This allows us to use the same function implementation without explicitly writing separate versions for `int`, `double`, and `string`, or for each individual stream type.

3.1 Reducing the Number of Template Parameters

Although the first version is very general, we do not actually need to make the output stream type a template parameter in this case. All the streams used above provide a common output interface. We can therefore specify this interface directly by using an `ostream` reference. The function now contains only one template parameter, `T`, because only the type of the data elements needs to vary. Nevertheless, we can still use the function with the same three output destinations:

```
1  // template_function_ostream.cxx
2  #include <iostream>
3  #include <iomanip>
4  #include <fstream>
5  #include <sstream>
6  #include <vector>
7  using namespace std;
8  template <typename T>
9  void print_matrix_ostream(ostream & output, vector<vector<T> > & data){
10   for (vector<T> & row : data){
11     output << "( ";
12     for (T & value : row){
13       output << right << setw(8) << value << " ";
14     }
15     output << ")" << endl;
16   }
17 }
18 int main(){
19   vector<vector<int> > integers = {{1, 2, 3}, {4, 5, 6}};
20   vector<vector<double> > doubles = {{1.2, 3.4}, {5.6, 7.8}};
21   vector<vector<string> > strings = {{"one", "two"}, {"three", "four"}, {"five", "six"}};
22   cout << "integers through cout:" << endl;
23   print_matrix_ostream(cout, integers);
24   cout << "doubles through ofstream to file output_ostream.txt" << endl;
25   ofstream file("output_ostream.txt");
26   print_matrix_ostream(file, doubles);
27   cout << "strings through stringstream:" << endl;
28   stringstream stream;
29   print_matrix_ostream(stream, strings);
30   cout << stream.str();
31   return 0;
32 }
```

The important point is that the function only requires an output stream that provides the operations defined by `ostream`. Therefore, the concrete type of the stream does not need to be known by the function.
For the moment, we will simply use this common stream interface as provided by the standard library. The underlying mechanism that allows different stream types to be used through the same interface is based on *class inheritance* and *polymorphism*, which we will study in the next lecture.
This example illustrates an important principle when working with templates: a template parameter should only be introduced when the function genuinely needs to support different types that cannot already be handled through a common interface.

3.2 Automated Type Deduction

Throughout this section, we have seen several examples in which the compiler automatically deduces types from the surrounding context. When calling a function template, the compiler deduces the template arguments from the function arguments. C++ provides a similar mechanism for variables through the `auto` keyword.

```
int x = 42;
auto y = 42;
```

In this simple example, `y` is automatically deduced to have type `int`, just like `x`.

In the example of the previous section, this approach is particularly convenient for the range-based loops over the template variable `vector<vector<T> > data`:

```
  for (auto & row : data){
    for (auto & value : row){
      // ...
    }
  }
```

The exact type of `row` depends on the type of the elements stored in `data`, and therefore on the template argument `T`. Instead of explicitly writing `vector<T>` and `T`, `auto` allows the compiler to deduce the correct type automatically. The complete example then reads, using the `auto` keyword:

```
// template_function_auto.cxx
#include <iostream>
#include <iomanip>
#include <fstream>
#include <sstream>
#include <vector>
using namespace std;
template <typename T>
void print_matrix_auto(ostream & output, vector<vector<T> > & data){
  for (auto & row : data){
    output << "( ";
    for (auto & value : row){
      output << right << setw(8) << value << " ";
    }
    output << ")" << endl;
  }
}
int main(){
  vector<vector<int> > integers = {{1, 2, 3}, {4, 5, 6}};
  vector<vector<double> > doubles = {{1.2, 3.4}, {5.6, 7.8}};
  vector<vector<string> > strings = {{"one", "two"}, {"three", "four"}, {"five", "six"}};
  cout << "integers through cout:" << endl;
  print_matrix_auto(cout, integers);
  cout << "doubles through ofstream to file output_auto.txt" << endl;
  ofstream file("output_auto.txt");
  print_matrix_auto(file, doubles);
  cout << "strings through stringstream:" << endl;
  stringstream stream;
  print_matrix_auto(stream, strings);
  cout << stream.str();
  return 0;
}
```

Although `auto` is not itself a template mechanism, both `auto` and function templates rely on the compiler's ability to deduce types automatically.

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

4 Practical Session – Part II

Function templates

1. In a previous exercise, you have created functions to calculate the mean, median, sample standard deviation, sample skewness, maximum, and minimum of an `array`/`vector` of `double`s. Make use of the template feature to generalize these functions to deal with different numeric types, such as `int`, `double`, `float`, while choosing an appropriate return type for each statistical measure.

   Note that template types may not only be used in the arguments of a function, but also as its return type:

```
template <typename T>
T calculate_minimum(vector<T> & data){
  T minimum = data[0];
  for (T value : data){
    if (value < minimum){ minimum = value; }
  }
  return minimum;
}
```

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

5 Conclusion

You now have all the tools needed to perform data analysis in a general sense. In the future, however, you will rarely be writing short, self-contained programs. Instead, you will often work in groups. In collaborative settings, it is usually best to divide the project into smaller tasks rather than having everyone work on a single, large piece of code.
This is one of the main advantages of object-oriented programming: it allows a project to be broken into independent, manageable components, enabling multiple people to work on different parts simultaneously.
In the next class, we will start exploring objects and classes.