Technisch Wetenschappelijke Software
Scientific Software
Session 7: C++ - introduction and performance

Pieter Appeltans, Wouter Baert & Emil Loevbak Leuven, November 19, 2020

Introduction

This session aims for you to develop an understanding of the following concepts which will come back in the homework assignments:

- header and source files;
- compiling programs in C++;
- the `std::vector`-class;
- the C++ standard library;
- timings in C++;
- code optimalization and compile options.

As a minimum, you should be able to answer the questions in **boldface**. The syntax you need for the session can be found in the demo, but you can ask further questions to the assistants in the session. If you have questions later, please post them on the Toledo forum. That way other students can see the questions that have already been asked, as well as their answer.

Demo

Compiling your first C++ program

Header versus source files There are two kinds of files in C++, namely source files and header files. The former have the extension `.cpp` (but `.cc` and `.cxx` are also sometimes used). These source files contain the implementation of the provided functionality and need to be compiled. Header files, on the other hand, are included by other files and can not be compiled on their own. They commonly have the extension `.hpp`. Conceptually you can think of a header file as describing the interface of a function or class to the source file by which it is included. We want to emphasize the word *conceptually* because header files can be much more than just an interface. For example, the use of templates, which we will cover next week, requires that the functionality is implemented in the header file.

Including header files You can include header files by using either angle brackets `#include <library_header.hpp>` or quotation marks `#include "own_header.hpp"`. Using the former restricts the search for the header file to the system and library paths. When using quotation marks, the compiler will first search for the header file in the current directory. If the compiler does not find the header file in the current directory, it continues its search for the header file as if it was included with angle brackets.

Compiling your code There are several revisions of the C++ standard: C++11, C++14 and C++17 to name some of the recent ones. The latest revision, C++20, was approved on September 4th 2020. We will be using some features introduced by the C++11, C++14 and C++17 standards in the exercise sessions as they simplify many things. For those that are interested, we will also provide some material that discusses novel functionality of the C++20 standard, but this material

is optional and out of scope for the assignments and the exam. To compile our C++-code we will use the GNU C++ compiler `g++`. Compilation of C++-code is similar to compiling Fortran-code. For example, if we want to compile the file `main.cpp` which uses functionality from `source.cpp` with `g++`, we have to use the following commands:

```
g++ -Wall -std=c++17 -O3 -c source.cpp          # -> source.o
g++ -Wall -std=c++17 -O3 -c main.cpp            # -> main.o
g++ -std=c++17 -O3 -o main source.o main.o      # -> main
```

in which the flag `-std=c++17` indicates that we want to use the C++17 standard, the flag `-Wall` enables all the warnings that `g++` can give and the flag `-O3` determines the optimization level. These compile statements can be replaced by the following single line command:

```
g++ -Wall -std=c++17 -O3 -o main source.cpp main.cpp
```

This command creates an executable `main` which we can execute from the command line by

```
./main
```

You may also use the Clang compiler.

```
clang++ -Wall -std=c++17 -O3 -o main source.cpp main.cpp
```

Avoid multiple inclusions using preprocessor macros To avoid multiple inclusions of the same header file, the contents of a header file must be protected by an include guard.

```
#ifndef PACKAGE_FILENAME_HPP      //ifndef -> if not defined
#define PACKAGE_FILENAME_HPP 0
// Content of header file
#endif
```

If the header file is included for the first time, the macro `PACKAGE_FILENAME_HPP` is set and the content of the header file is added to the compilation unit. When the header file is included a second time, the macro `PACKAGE_FILENAME_HPP` already exists and the content of the header file is skipped. The name of the macro is free to choose but it is recommended to use the format `PACKAGE_FILENAME_HPP`. For an example of what can go wrong when a header file is included multiple times, see `https://en.wikipedia.org/wiki/Include_guard`.

C++ Standard Library

The entire C++ Standard Library is defined in the namespace `std`. In the following we will discuss some important functionalities of the C++ Standard Library.

The `std::vector`-class In the lectures, you encountered several container classes, which can store multiple elements of the same type, and discussed their (dis)advantages. Here we will limit ourself to the `std::vector`-class, which is defined in the `<vector>`-header file (`https://en.cppreference.com/w/cpp/container/vector`).

To create a `std::vector` of size 10 containing integers we can use the following command:

```
std::vector<int> v(10);
```

This creates an integer vector, the elements of which are initialized with the default value for `int`, namely 0. We can also initialize the elements of the vector with one using:

```
std::vector<int> v(n,1);
```

The type between the angle brackets (in this case `int`) defines a Template argument. Templates will be explained in more detail in the next exercise session. All you need to know for now is that we can make a vector with elements of another type by replacing `int`. For example, `std::vector<float>` creates a vector of single precision floating point numbers and `std::vector<double>` creates one with double precision floating point numbers. It is even possible to create a vector of vectors `std::vector< std::vector<double> >`.

To access an element in a `std::vector`, you can use the `[]`-operator. For example, to access the fifth element of `v`, you have to use

```
v[4];
```

Important: Notice that, in contrast to Fortran, indexing in C++ starts from 0.

Printing to the standard output stream To gain access to IO-functionality of C++ (`http://en.cppreference.com/w/cpp/io/basic_iostream`) you have to include the `<iostream>`-header file.

The following program reads an integer from the standard input stream and prints this integer to both the standard output stream and the standard error stream.

```
int main(){
    int n;
    std::cout<<"Enter a positive number"<<std::endl;
    std::cin>>n;
    std::cout<<"Output n= "<<n<<std::endl;
    std::cerr<<"Error n= "<<n<<std::endl;
    return 0;
}
```

In the exercise sessions and the homework assignments, we will often redirect the error output to a separate file. This can be done by executing `./exec 2>err.log`, with `exec` the name of your executable and `err.log` the name of the file.

Algorithms on containers The Standard Template Library (STL) provides several generic routines for containers based on iterators. Some of these algorithms that are useful for this course, are `std::copy`, `std::sort`, `std::fill`, `std::accumulate`, `std::shuffle`, `std::iota`, `std::for_each` and `std::transform`.

The following example illustrates how the `std::copy`-algorithm can be used. First, it creates two vectors `v1` and `v2`. It subsequently copies the content of `v1` to `v2`.

```
std::vector<int> v1(10),v2(10);
std::copy(v1.begin(),v1.end(),v2.begin());
```

This `std::copy`-function works as follows. In the first step, the first element of `v1` (indicated by `v1.begin()`) is copied to the first element of `v2` (`v2.begin()`). Next, both iterators (`v1.begin()` and `v2.begin()`) are increased by one position. Subsequently, the second element of `v1` is copied to `v2`. This is repeated until the end of `v1` (`v1.end()`) is reached.

Debugging using assert statements To debug your code, you can use assert statements, defined in the `<cassert>`-header file (`https://en.cppreference.com/w/cpp/error/assert`). Assert statements can be used to verify certain conditions at run-time. For example, to verify whether a variable `n` is strictly positive, we can use the following syntax:

```
assert(n>0)
```

If the assert statement fails, the execution of the program is stopped and an error message is displayed. It is important to know that assert statements are executed unless disabled. These assert statement thus cause an overhead when you are no longer debugging. Luckily, you can disable all assert statements by adding the `-DNDEBUG`-flag during compilation:

```
g++ -std=c++17 -DNDEBUG -o main source.cpp main.cpp
```

Finally, to run certain lines of code only during debugging (eg. some additional print statements), you can use the following syntax

```
#ifndef NDEBUG
// Put here your code that will only be executed during debuging
#endif
```

Command line arguments To access the command line arguments passed to your program, you have to add two input argument to the main function:

```
int main(int argc, char* argv[]){
...
}
```

in which `argc` will be equal to the number of command line arguments plus one and `argv` will be an array containing the command line arguments. The first element of `argv` is, however, the name of the executable. The actual command line arguments are stored starting from index 1. These command line arguments are passed as character strings and need therefore to be converted to the correct type. This is where the `<cstdlib>`-library (`https://www.cplusplus.com/reference/cstdlib/`) comes in handy. This library defines, among others, functions to convert character strings to integers (`atoi`), floats (`strof`) and doubles (`atof`).

Exercises

Question 1: Sum of a vector

Create a file named `question1.cpp` and include the following header files at the beginning of your code: `algorithm`, `iostream`, `iomanip`, `numeric`, `random`, `vector`, `cassert` and `chrono`. Next, define a main function, that has one command line argument (see example above). Now, complete the following steps.

1. Use an assert statement to verify that the number of provided input arguments is equal to one.
2. Convert this input argument to an integer and store the result in a variable named `n`:

   ```
   int n = atoi(argv[1]);
   ```
3. Assert that `n` is strictly positive.
4. Create a vector `v` of length `n`. Fill this vector with $1, 2, \ldots, n$ and shuffle its elements. Use the `std::iota`-function to fill the vector. You can use the following code to shuffle the vector:

   ```
   //Shuffle the elements of v using the given random number generator
   std::shuffle(v.begin(), v.end(), std::mt19937{std::random_device{}()});
   ```
5. Download `library.hpp` and `library.cpp` from Toledo. Add an appropriate `include`-statement at the beginning of `question1.cpp`.
6. In your main function, use the `print_vector`-function, as defined in `library.hpp` and implemented in `library.cpp`, to verify that order of the elements in the vector is random. Compile and run your code for $n = 10$.
   Hint: `./exec 10`
7. Debug the following code for printing the elements of a `std::vector<int>` in reverse order (found in `library.cpp`):

   ```
   void reverse_print_vector(std::vector<double> const & v){
       assert(v.size()>0);
       std::cout<<"("<<v.size()<<")[";
       for(decltype(v.size()) i=v.size()-1;i>=0;i--){
           std::cout<<v[i]<<" ";
       }
       std::cout<<"]"<<std::endl;
   }
   ```

Hint: Print `i` inside the for-loop.
In your main function, now also print the elements of `v` in reverse order. Compile and run your code again.

8. Write a function `sum` that uses a for-loop to calculate the sum of the elements of `v`. Make sure that you pass `v` *by reference*. Compile and run your code. Make sure that the calculated sum is correct for different values of n (**Hint:** $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$).

9. Calculate the same sum using the `std::accumulate`-function (`https://en.cppreference.com/w/cpp/algorithm/accumulate`) from the STL library. As the `std::copy`-function encountered in the demo, this functions uses iterators to loop over the elements of `v`. More specifically, the first argument of `std::accumulate` should be a pointer to the first element of `v`; the second argument should be a pointer to the last element of `v`; finally, the third and last argument should be an integer that indicates the initial value of the sum (in our case this is thus equal to 0).

Next, we will examine the performance of our sum function. In order to make an informative decision, we should at least measure the execution time of our function. We will use the high precision wall-clock timer, available in the `<chrono>`-header (`https://en.cppreference.com/w/cpp/chrono`), to perform timings. More precisely, the following code measures the elapsed wall-clock time in seconds.

```
auto t_start = std::chrono::high_resolution_clock::now();
... //Let the machine do hard, tough work
auto t_end = std::chrono::high_resolution_clock::now();
std::cout<<"elapsed time: "<<\
std::chrono::duration<double>(t_end-t_start).count()<< std::endl;
```

Now complete the following steps.

10. Modify your code such that your main function accepts two command line arguments. Convert this second command line argument to an integer and store the result in the variable `nb_runs`. Add an assert statement to make sure that `nb_runs` is strictly positive.

11. In your main function, create a for loop that calls your `sum`-function `nb_runs` times. Store the execution time of each run in a `std::vector` named `timings` (**What template argument should you use?**). Make sure that you print the resulting sum during each loop (to the standard error stream) else the compiler might perform a code optimization that elimates all but one run of your function.

12. Print these timings. Compile your code with optimization level `-O3`. Run the resulting executable with `n` equal to 5000 and `nb_runs` equal to 10. **What do you notice when examining the execution time of the first run(s)?**

13. Modify your code such that your main function accepts three command line arguments. Convert the third command line argument to integer and store the result in the variable `nb_warm_up`. This argument will be used in the next step to discard the first `nb_warm_up` timings. Assert that `nb_warm_up` is strictly positive and smaller than `nb_runs`.

14. Use the provided `calculate_mean_stdev`-function to calculate the mean and standard deviation for the relevant timings. Since C++17, you can use the following syntax to store the elements of the resulting tuple in the variables `mean` and `st_dev`

    ```
    auto [mean,st_dev]=calculate_mean_stdev(timings,nb_warm_up)
    ```

15. At the end of your main file print the vector size `n` and the mean and standard deviation of the execution time on one row. For example for `n` equal to 2 the output of your program should look as follows.

    ```
    2 1.87726e-07 2.16605e-08
    ```

16. Next, we will examine the execution time of `sum` in function of the vector size. Compile your code with optimization level `-O0`. The following command will run your code for `n` equal to $2, 4, 8, \ldots, 1048576$:

    ```
    for i in `seq 1 20`; do ./sum $((2**$i)) 100 5 ; done | tee sum.out
    ```

    with `sum` the name of your executable. The resulting file `sum.out` should look as follows:

    ```
    2 1.87726e-07 2.16605e-08
    4 8.53158e-08 1.17301e-08
    8 8.28526e-08 1.62972e-08
    16 1.86326e-07 1.8688e-08
    ...
    131072 0.00039831 6.65425e-05
    262144 0.000770872 2.34155e-05
    524288 0.00155205 5.75422e-05
    1048576 0.00316305 0.000210601
    ```

17. Now recompile the code with option `-O3` and run

    ```
    for i in `seq 1 20`; do ./sum $((2**$i)) 100 5; done | tee sum_O3.out
    ```

18. To compare the results for optimization options `-O0` and `-O3` we will use the provide `plot.tex`-file from Toledo. To generate the figure run the following command in the terminal.

    ```
    pdflatex -shell-escape -interaction=nonstopmode plot.tex
    ```

19. (At home) Repeat your experiments where you pass `v` by value instead of by reference to `sum`.

20. (At home - optional) Repeat your experiments using the accumulate function.

Question 2: Sorting a vector

1. Implement the bucket sort algorithm (`https://en.wikipedia.org/wiki/Bucket_sort`) in C++ to sort the elements of a vector of doubles with values that lie in the range $[0, r)$. In a first step, this algorithm places the elements in m buckets. For example, suppose that the range r is equal to 1000 and that 10 buckets are used, then the range for each bucket is 100. This means that bucket 0 contains elements in the range $[0, 100)$, bucket 1 contains elements in the range $[100, 200)$, ... and bucket 9 contains elements in the range $[900, 1000)$. Next each bucket is sorted separately. Finally the sorted bucket elements are placed back in the original vector. **Some hints:**

    - For determining in which bucket an element belongs, take a look at the functions `std::ceil` and `std::floor` defined in the `<cmath>`-header file.
    - Type conversion or type casting can be done by `(type)`. This can be useful since dividing two integers results in an integer value.
    - To sort the buckets you can use the `std::sort`-function, defined in the `<algorithm>`-header file.
    - To add an element to the end of a `std::vector`, you can use the `push_back`-function (`https://www.cplusplus.com/reference/vector/vector/push_back/`).
    - To request that the vector capacity is at least enough to contain n elements, you can use the `reserve`-function (`https://www.cplusplus.com/reference/vector/vector/reserve/`).

2. Test your function by writing a `main`-function that call this function for `m` buckets with a vector of length `n` whose elements are uniformly distributed between 0 and `r`, where you pass `m`, `n` and `r` as command line arguments. To set the elements of the vector, you can use the following commands assuming `v` is the vector of doubles (requires the inclusion of the `<random>`-header).

```
std::random_device rd;
std::mt19937 generator(rd());
std::uniform_real_distribution<> distribution(0, r);
for(auto& vi:v){
vi=distribution(generator);
}
```

3. (At home) Shortly after the session, a reference implementation of the bucket sort algorithm will be available on Toledo. Compare this implementation with your code. How can you improve your code? What is the advantage of using a class and storing the buckets over multiple runs?

4. (At home - optional) Investigate the execution time of the `std::sort`-function for different optimization options (try at least `-O0` and `-O3`). Use command line arguments for the size of the vector, the range, the number of experiments and the number of discarded timings. Shuffle the vector before each experiment and print the first element at the end of each experiment.

5. (At home - optional) Compare your implementation of bucket sort with the standard sorting algorithm. What is the function of the number of buckets m? What are the (dis)advantages of both algorithms? Illustrate your explanation with (approximate) algorithmic complexities.