Institut für Informatik — TUM
Scientific Computing in Computer Science
R. Wittmann

WT 2015

Tutorial (Advanced Programming) Worksheet 6:

Assignment 1: Evaluation order

As your projects evolve over time they also get bigger and require the use of multiple source and header files for the sake of maintainability and easier collaboration. However, this separation of source and header files offers some pitfalls which we want to cover in this exercise. You can find a zip archive with the necessary stub implementations on moodle.

1. Read through the different source and header files and explain how the compiler should progress through the files.
2. When compiling the package you will notice that certain classes are not known, though they were defined in one of the header files. Explain this issue. If you run into messages like `undefined references`, you will have to add the `eo_include_1.cpp` file to your build process.
3. If you run the program you will notice that the expected value does not match the actual value in `Depends depends`. Explain this behaviour and repair the code.

Assignment 2: Unique Pointers

With the new C++11 standard two kinds of smart pointers have made it into the C++ language. `unique_ptr` replaces the now deprecated `auto_ptr`, while `shared_ptr` is completely new to the standard.

In this exercise, we focus on some aspects of how we can benefit from a *unique pointer* and how we can implement one. We therefore come up with a class of our own called `UniquePointer`. You can find the corresponding stub implementation on moodle. Be sure to compile this program with C++0x or C++11 support.

1. Discuss the use of smart pointers. Find scenarios where they may ease the life of programmers.
2. Implement the missing parts of class `UniquePointer` and test your program by uncommenting lines 79 to 104 in main. If your program works, continue by uncommenting lines 90 and 91 and test again.
3. Think about the TODO in line 100 and adjust the class in such a way that the operation in line 101 will be forbidden.

Class Assignment 3: Memory allocation schemes

There are many different ways to store arrays in C/C++. We will now write a small program which performs some read and write operations on differently stored arrays. We are especially interested in out-of-bounds effects for fixed size arrays where the size is controlled by a global constant N (e.g. 100) Consider the following program sketch:

1. Create an `int` array of size N for each of the following storage types and print the respective addresses of the arrays.
   (a) global
   (b) stack
   (c) static, using the singleton pattern.
   (d) heap, remember to cleanup the memory.
2. Define a function `memory_accesses` which takes an `int` pointer, a `start` and a `stop` index as arguments.
3. Then, the function should replace the existing values in this memory range with new but distinguishable entries.
4. Finally, the function should read and compare the values with the just written ones.

Now, we want to look how far we can go across the array boundaries without causing the program to crash. Alter the indices in such a way that the `start` index becomes smaller and the `stop` index becomes larger. In case of the heap storage, also rerun these procedures after the memory was freed using the old pointer. If your program crashes, explain this behaviour.