C++ Programmeermethoden Assignment4 Compensation

Bas Terwijn

June 4, 2024

1 Introduction

In this assignment we will create our own computer game and practice using Modern C++ concepts: dynamic polymorphism, RAII, STL Containers, and STL Algorithms. You will be graded on your software being "simple" (as in not overly complex). Software that is more simple is easier to understand and easier to change. These characteristics are important for software that is in use in the real world as often software needs to be changed regularly. We will use Kate Gregory's description of simple software from her ***"Simplicity: not just for beginners"*** talk, take some time to watch it first. It is also part of the exam material!

2 Tasks

Next, install and run ***VirusGame*** on your computer using the installation instructions provided, have a look at the documentation and code to get somewhat familiar with it. Then work on the following tasks.

2.1 task1: Dynamic Polymorphism

Currently in VirusGame.cpp all Virus units are stored in a static array:

```
Virus units[max_nr_units];
```

But we want to be able to add objects of other classes as units besides only the "Virus" objects. In addition we want to handle the "player" object as just another unit so the code gets simpler. Therefore change the "units" array so that units of different types can be added to it by using dynamic polymorphism.

2.2 task2: Avoid Duplicate Code

Avoid having duplicate code or expressions or said differently: don't repeat yourself (DRY). The Virus::step() function is currently already a duplicate of Player::step(). Find a good way to avoid that and any other duplication, maybe using 'inheritance'.

2.3 task3: RAII

When using dynamic polymorphism you will often have to dynamically allocate memory when you instantiate objects. This could be done using the "new" keyword. However, you will then also have to remember to deallocate the memory using the "delete" keyword to avoid memory leaks when it is no longer needed. Forgetting or double deallocation is a big security risks. A better alternative is to use RAII (Resource Acquisition Is Initialization). With RAII the deallocation of a resource is put in a destructor of an object so that is automatically done only once when an object goes out of scope and its destructor gets called. Use RAII in your code to make sure all resources are deallocated automatically. For more information see:

- ***RAII documentation***
- video ***Back to Basics: RAII in C++, Andre Kostur***

2.4 task4: STL Containers

The modern ***Standard Template Library containers*** are the preferred data structures in C++. Prefer std::vector over a static array as:

- it can grow to arbitrary size
- it knows its own size
- it doesn't decay to a pointer when passed to a function
- an assignment makes a full (deep) copy
- it has only little overhead compared to a static array

Therefore replace the static array mentioned in task1:

```
Virus units[max_nr_units];
```

and the "nr_units" counter with a std::vector. Also prefer ***STL containers*** if you decide to add other data structures.

2.5 task5: STL Algorithms

ES.1 of C++ Core Guidelines recommends using the standard library over "handcrafted code". Therefore use as much as possible the functions defined in the ***STL Algorithms Library*** instead of for example raw for-loops. For a gentle introduction to STL Algorithms see the ***"105 STL Algorithms in Less Than an Hour"*** talk by Jonathan Boccara.

2.6 task6: Game Extension

Change the game so that it is similar to the classic game Space Invaders in figure 1. See the ***"Space Invaders 1978 - Arcade Gameplay"*** youtube video for an example.

In this game different types of aliens move left and right and slowly come down to attack the player. The player can move left and right and can shoot the aliens. The aliens also sometimes shoot back. You don't necessarily have to make a exact copy of the game or use any graphics (colored circles or square will do fine) or write any text as long as you have a somewhat similar game play. Adding more game features will increase the points you earn.

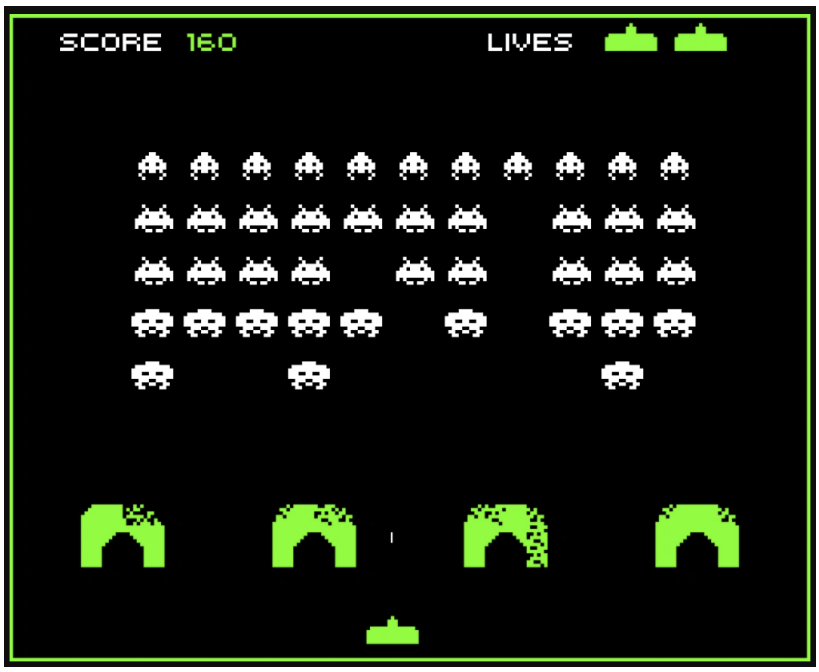

Figure 1: Space Invaders.

2.7 task7 (advanced, optional): SOLID principles

As source code grows from a few hundred lines to many thousands of lines it tends to get harder and harder to change. This is one of the most difficult problems in software engineering. There are different schools of thought on how to alleviate this problem. One of these is using the SOLID principles which focuses on reducing code dependencies. For the SOLID principles see the ***"Breaking Dependencies: The SOLID Principles"*** and ***"Free Your Functions!"*** talks by Klaus Iglberger. Try to use the SOLID principles to decide how to (re)structure your code. Describe the code structure decisions you made in relation to the SOLID principles in the README.md file with references to your source code.

3 Grading

Your grade will follow from which tasks you complete to a satisfactory level:

task	**points**
task1: Polymorphism	1.5
task2: Avoid Duplicate Code	1
task3: RAII	1.5
task4: STL Containers	1
task5: STL Algorithms	1
task6: Creative Extension	4
task7: SOLID principles (bonus)	+1

Points will be deducted for code that is not simple as described by the before mentioned ***"Simplicity: not just for beginners"*** talk by Kate Gregory.

4 Submission

Submit your code as a zip file of the whole VirusGame project before the deadline on Canvas. Remove the compiled executables (the "build" directory) and other derivatives that are not needed to compile your code in order to reduce the size. If you add other dependencies (additional libraries) describe them in the README.md so graders can install them. Double check that your submission contains all required files and compiles before you submit.