university of groningen

faculty of mathematics and natural sciences

The Value of Literate Programming in Academic Courses

Bachelor thesis - Computing Science

Thursday 11th July, 2013

Student: Marc Holterman

Primary supervisor: Prof. Dr. G. R. Renardel de Lavalette

Secondary supervisor: Drs. P. Dykstra

Abstract

One of the most appealing and fundamental aspects of Literate Programming is the change of perspective towards the reader. Reading a literate program, like reading a book, should to be an enjoying experience. Where modern-day programming languages focus mainly on providing a computer with instructions, literate programming focuses on providing the reader with the best possible documentation about what the program is suppose to do. This thesis is a quest to determine the value of literate programming concerning academic courses given at the University of Groningen.

Contents

1 Introduction **5**

1.1 Thesis statements . . . 6

2 Literature Review **7**

2.1 Literate Programming . . . 7

2.2 The web of abstract things . . . 8

2.3 Structured Programming . . . 9

3 Research **10**

3.1 Advantages . . . 11

3.2 Disadvantages . . . 11

4 How to Program Literately **12**

4.1 Tools . . . 12

4.1.1 WEB . . . 13

4.1.2 CWEB . . . 13

4.1.3 NOWEB . . . 13

4.2 LaTeX . . . 15

4.3 Noweb in Courses . . . 15

5 Literate Programming using Noweb **16**

5.1 Pythagorean Triple . . . 16

5.1.1 Problem Statement . . . 16

5.1.2 The Root Chunk . . . 17

5.1.3 The triple . . . 18

5.1.4 The loop . . . 18

5.1.5 Main program . . . 19

6 Conclusion **20**

7 **Future Work** **21**

8 **Acknowledgement** **22**

9 **Bibliography** **23**

A **Quickstart** **24**

B **Noweb userguide** **26**

B.1 Literate Programming 26

B.1.1 Scope 26

B.2 Structure 27

B.2.1 Source 28

B.3 Noweb 29

B.3.1 Notangle 30

B.3.2 Noweave 31

B.4 Final words 40

C **How to install** **41**

C.1 Mac users 41

C.2 Unix users 43

D **Module** **44**

D.1 Noweb 47

D.1.1 Notangle 47

D.1.2 Noweave 48

Chapter 1

Introduction

This thesis aims to offer insight into the value of literate programming with regard to academic courses. It also explains the concepts of literate programming and how these aspects can be incorporated into courses given at the University of Groningen.

Programming is like writing a letter, everyone has his own style. New programmers who are yet unfamiliar with the art of programming will develop their own unique style just as they learned writing when they were younger. It is a unique style that develops along similar lines as someone his handwriting. Therefore, it can be quite a challenge to read through the lines of code and unravel the programmers intentions. In other words, programs just as handwriting, can be difficult to read. If you would ask a programmer to revise his work two weeks after his last edit, they might already trouble to figure out their own intentions [6]. Source code from which the reader cannot straight away determine the purpose requires supporting documentation. Documentation in natural language that clarifies the programmers intentions. However writing a good coherent sentence about your own code can be quite a challenge.

Literate programming offers an approach to programming whilst focusing on documentation. One of the most appealing aspects of literate programming is the focus on the reader. The reader occupies a central position to which the programmer has to justify his thoughts and intentions. This change of perspective opens up new ways to evaluate programs.

1.1 Thesis statements

Writing, designing and creating programs is not all about the source code alone. Very often different people work on the same programs and rewrite each others source code. This requires understanding of the source code, which sometimes, is hard to understand [1]. Therefore documentation or comments that clarify certain pieces of code is of utter importance.

Documentation is becoming more and more important. Writing good documentation should get equal if not more attention than the actual coding [2]. Documentation helps to readers to understand the source code and makes is easier for a programmer to explain his intention in natural language. Students who are starting to learn the concepts of programming will also be introduced to the writing of documentation. They should be stimulated to write solid coherent documentation not only for the reader but also for their own good. The concept of literate programming offers useful guidelines in this regard. It centralizes the reader, and changes the perspective of contemporary programming. I would like to investigate the aspects of literate programming and the possibilities is provides. Dig into the all the facets and the ideology behind it. Thereby examine the benefits of literate programming with regard to students and the possibility to integrate this concept into first year courses given at the University of Groningen.

Chapter 2

Literature Review

2.1 Literature Programming

Literate programming is a combination of two words namely literature and programming. Where literature means interest, knowledge and the ability to read and write coherently, programming stands for instructing a computer. The combination literate programming means something along the line of critical thinking and writing coherently with regard to the instruction of a computer.

Literate programming is a combination of two words namely literature and programming. Where literature means interest, knowledge and the ability to read and write coherently, programming stands for instructing a computer. The combination literate programming means something along the line of critical thinking and writing coherently with regard to the instruction of a computer.

Literate programming is invented and created by Donald E. Knuth in 1981. He introduced the term in an article in which he described the ideology of literate programming [2]. The rationale idea behind literate programming is the change of perspective towards the reader of your program. When literate programming one does always keep the reader in a central position the main focus of the programmer diverges from instructing and providing a computer with instructions towards explaining the reader what we want a computer to do [2]. The reader becomes the main point of interest for a literate programmer, everything he does has to be clarified in natural language in a coherent sentence that describes the intention of that piece of source code.

This shift of perspective is one of the fundamental ideas of literate programming. Programs are meant to be well-written and polished which makes reading an enjoyable experience. This is exactly the intention of the Donald E. Knuth when he founded the term literate programming. Programming is not just instruction a computer, programming should be an experience as well as for the programmer as for the reader. Writing a program is like writing a book, it should be a pleasure for the reader to read through the pages and be easy to understand the meaning of each part. This shift, the centralization of the reader is one of the most

appealing aspects of literate programming.

Programs are written in programming languages. These languages that are designed for the communication with machines, or in particular a computer, certainly not for the communication with the reader. We have designed our own communication tools for that purpose, namely the so called natural language or ordinary language which is far more readable than source code alone, some might disagree. Since we cannot instruct computers with ordinary language yet, we have to rely on programming languages and documentation. Documentation is a piece of ordinary language weaved through the source code in order to clarify the code it is not interpreted by the compiler and mainly used to elucidate the code. While you would normally write the documentation to support you code and thus writing it in sequence after the source code, literate programming forces you to write documentation and source code interleaved with each other.

2.2 The web of abstract things

Let us consider a program to be a web of things, things that represent the different parts of a program. These parts are all weaved and tangled together through connections and relations that are defined within the program [3]. Through all these connection it can be difficult to grasp and understand the flow of the program from first sight even for experienced programmers let alone a reader who might not have the technical background. Thus in order to explain software in this matter all the individual parts of the web have to be explained on their own. The individual parts of a software program are placed in a order required for the computer to interpret, an order which is most likely less understandable for the reader. Literate programs have the ability to present every piece of the web in an order which is best suited for humans to read. In fact the programmer can design and create his own web pieces and express them in any preferred order. This is because it is unlikely that the order of a normal program is also the best suited order for human to read. The ability to create own web pieces and represent them in any preferred order, all for the benefit of the reader is another great aspect of literate programming.

2.3 Structured Programming

In order to let your audience or reader understand a piece of software, the documentation representing that software should be clear and convenient to read. The documentation as well as the program should have a evident structure to enhance the readability of the document. Given the aspects of literate programming above, namely, the ability to change the order in which a programmer can represent their program parts and the main focus on the reader, there is a lot of freedom as to represent a document. The programmer can show all different pieces in any order and has to give a coherent explanation for each individual piece of the web. This will leave the programmer with a greater responsibility and will most likely make him think more carefully about the situation and how to split up the problem into smaller pieces. This responsibility will cause the programmer to think more carefully about the designing decisions which will eventually lead to a better understanding of his own code as well as the understanding of the audience to whom he shows his work.

Chapter 3

Research

The question is whether or not the aspects of literate programming are valuable enough to be used in academic courses given at the University of Groningen. Since literate programming offers opportunities to approach certain programming problems the main focus of this thesis lies on courses that have the art of programming as their main objective. Courses that learn students a programming language or introduce certain data structures. Furthermore, I am looking into first year courses for the incorporation of literate programming this narrows it down but there are still a few courses to consider. There are three courses that nominate to be my point of interest. There is Imperative Programming, a C course given in the first trimester. Furthermore there is Algorithms and Data Structures, an advanced course in C, taught in the third trimester and finally there is Object Oriented Programming first is a java course that introduces the aspects of object oriented programming.

First of all there is Imperative Programming. Imperative programming is given in the first trimester of the first year. Thus new arriving students will have there first contact with a programming course within the walls of the university from this moment. This is immediately the reason why I decided not to put my focus on this course. Since this is the very first course and it is about the basics of the programming language C, students who do not have any previous experience with programming will need all there afford to fully understand the concepts of programming as well as the ins and outs of the programming language C.

Object Oriented Programming is about the concept of programming in an object oriented way. Literate programming dates back from an era before the introduction of this programming style. With that the course is taught in java, which also is the first contact with that language.

Algorithms and Data Structures is an advanced course with regard to the programming language C. It gives an insight in fundamental data structures like stacks and queues and some algorithms that work with them. The programming problems that students have to solve are also getting beyond the level of Imperative programming since they will become bigger and harder to solve. I found this course best suited to incorporate the aspects of literate programming mainly because of the problem size and level. Students that will enter

this course will have there basic knowledge of C and will start to tackle more intriguing problems. Problems in which it is not immediately evident in which direction one has to think.

3.1 Advantages

The first and in my opinion most appealing aspects of literate programming is the position of the reader. The reader is the main point of interest, every decision has to be well considered because it has to be explained to the audience.

As for students, who are starting to familiarize them self with programming languages this concept does has a certain value. When first encountered with a problem students will start to generate a stream of thoughts, that may or may not be in the right direction. Literate programming forces students to subdivide their problems in to smaller pieces merely because they have to justify their decisions towards the reader, when the problem is too comprehensive it is difficult to give a coherent explanation in natural language. These subdivided problems can on their turn be divided down to where the problem is solvable with a few lines of code for which it is easier to verbalize their intentions. This whole subdividing and breaking down of a problem into bite-sized pieces will make them more aware of the decisions they make as well as a better understanding of the code.

When using the aspects of literate programming one will not only increase the readers understanding of the program but also increases their own knowledge. The possibility to abstract away from source code and use your own defined abbreviations will also contribute to this fact. Students can define their own pieces of the code and create and abstract web that represents their program.

3.2 Disadvantages

One of the only disadvantages that I could fine was and is about time. Literate programming takes time, not only does it take time to create and setup the literate programming environment, it also takes significantly more time to write a literate program. Especially when considering smaller sized problems, problems that are solvable within a few lines of code. It is just not worth it from a teaching perspective to let students create a literate program when the solution is right in front of them. The fact that it is sometimes a hassle and that it is time consuming is one of the only downsides I encountered.

Chapter 4

How to Program Literately

In a nutshell. When literate programming one is writing the source code and the program documentation in one single source file. One of the main differences in regard to the conventional programming approach is that literate programs offer the ability to change the order of the source code from a machine-imposed sequence to one convenient to the human mind, this is an important characteristic of literate programming according to Donald E. Knuth [2]. Thereby one needs to be able to explain the reader the meaning and the intentions with specific pieces of source code to increase the readability and making it an enjoying experience to review the code for an audience.

To achieve such an environment in which we can write source and documentation interleaved with one another in any order we prefer we need to look at some possible tools in this regard.

4.1 Tools

Tools in general are used to achieve certain goals. We want to create a literate program, thus we need a literate programming tool. When one wants to create a literate program you need to be able to use and interleave documentation and source in the same manner.

There are a dozen literate programming tools out there and the numbers are still increasing [4]. With the research question in mind, the investigation of the value of literate programming in courses given at the University of Groningen, I have compared a few literate programming tools and have chosen one which I think is the most appropriate for our case.

4.1.1 WEB

The philosophy behind WEB the combination of two worlds. On one hand a programmer who want to provide his readers with the best possible documentation needs to rely on a typesetting language. On the other hand he needs a programming language for the programming of their software. Neither type of programming language can provide proper documentation for itself that satisfies the aspects of literate programming. Thus there is a need to combine the strengths of typesetting languages and programming languages to obtain a system that is useful to fulfill this need, the need to literate program [2].

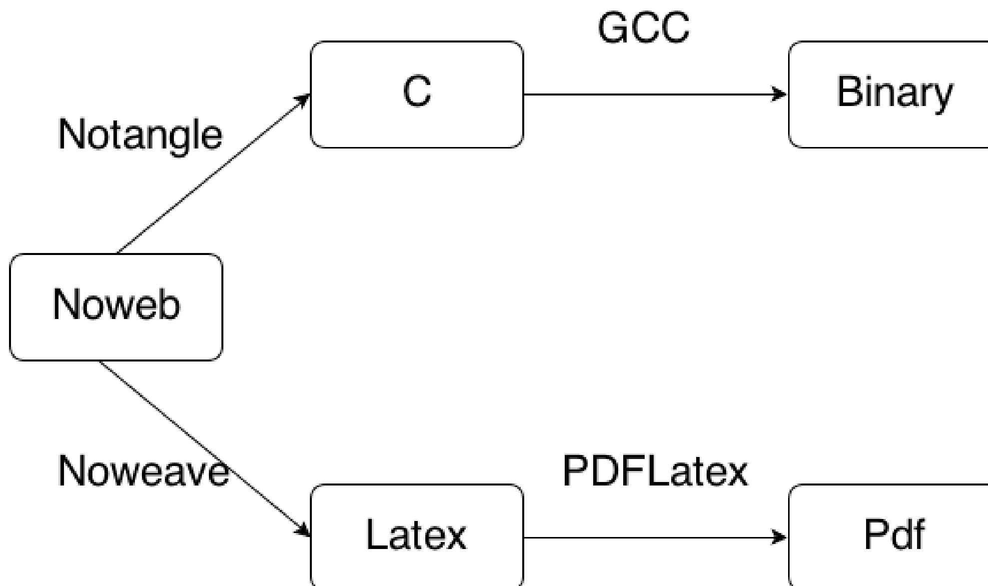
WEB is a literate programming tool developed by Donald E. Knuth in 1981 [3]. He developed WEB to support the programming language Pascal and the typesetting language TeX which he used for documentation purposes. We too want to use a variant of TeX namely, LaTeX however at the university of Groningen the first year courses are mainly focused on the C programming language thereby excluding WEB from the list of plausible tools.

4.1.2 CWEB

CWeb is a variant of Donald E. Knuth programming tool WEB. It is developed to support the programming language C [8] and a typesetting language like TeX. This literate programming would suit us better than WEB because it is designed for the programming language C, which is used in most first year courses at the University of Groningen. However it is still language dependent on TeX and it less extensible and simple than our last choose.

4.1.3 NOWEB

Noweb provides an environment for writing literate programs and is developed by Norman Ramsey in the late 90's. Noweb is mainly designed to be a simple, easily extensible and language independent tool to ease the practice of literate programming [1]. In literate programming one writes the program code and documentation in one single source file, by convention this file has the `.nw` extension. In order to process this source file Noweb provides two main subroutines to either weave the file into a well formatted document or tangle to file to a source file which can be interpreted by a compiler [7]. Both these routines can be called through the command line.



The figure above shows the structure of a literate program using noweb. The `.nw` can be processed by either of the two subroutines. The subroutines are listed below.

```
noweave program.nw > program.tex
notangle program.nw > program.c
```

A Noweb file is represented by a sequence of chunks which can be placed in any particular order. Chunks can either contain lines of code, or lines of documentation. The code chunks are named and can linked by referring to other code chunk, the documentation chunks are unnamed. Noweb interprets the source in a sequential order. Thus for each code chunk, the references are expanded recursively [6]. This gives the programmer the possibility the write the code in a sequence that is more convenient for the reader, the code chucks are eventually tangled together in the right order.

Each code chunk begins with «chunk name» in which the name corresponds with the reference of that chunk. Documentation chunks start with a @ followed by a white space or newline. Code chunks contain source code and references to other code chunks. They may have the same name in which case they will be concatenated upon tangling. Notangle extracts the program by expanding the chunks beginning by the defined root, which is by default «*». Chunks are terminated implicitly by the beginning of another chunk [6]. Documentation chunks contain information or text about the program and are ignored by notangle whilst building the source code. Code can be quoted using double square brackets [[quote]] . Noweave copies both the source code and documentation to the standard output thus generating the required report. Noweave can work with Tex or LaTeX or even HTML markup languages [6].

Tangling and weaving are not stand-alone programs but a set of filters through which the `.nw` file is piped. This pipelining system makes noweb both flexible and extendable hence they can be modified to the users needs which will change the behavior of noweb [1].

4.2 LaTeX

Tex, or Latex in this case is the underlying language of a noweb program. Programmers nowadays should familiarize themselves with documentation formatting language as well as the learning and understanding of program language. If you posses the knowledge for using Latex learning Noweb shouldn't require to much effort since the biggest part of Noweb is the understanding of LaTeX with a few extensions [2].

4.3 Noweb in Courses

Material written for computer science courses often contain pieces of source code and algorithms. These snippets are meant to give a better understanding of the subjects and thus require the be correct. Generally the snippets are supported by results indicating the code's intention. For students to fully understand the purpose of an algorithm it is important that they can reproduce the stated results. However, the algorithm is implemented, executed and tested in a different environment than the material and inserted by copy-paste techniques in a later stage making it very error prone [5].

Computer science courses often contain a practical component in which students implement the techniques stated in the material. Thus making it very important that the example code compiles and the results really correspond to the same algorithm as used in the material. Noweb provides an environment which omits the need to copy by embedding the source code in the material itself. Noweb files are documents with embedded source code rather than source code snippets with embedded comments [5].

Chapter 5

Literate Programming using Noweb

Let us consider a problem which we have to solve using the aspects of literate programming. Keep in mind that the reader is the main point of interest, we want to make sure that we explain everything in a convenient way for the reader. I have chosen to investigate a problem from the website `http://projecteuler.net`. They offer an wide amount of small programming problems and I have taken one of the smaller one to show how the aspects of a typical literate program.

5.1 Pythagorean Triple

A Pythagorean triplet is a set of three natural numbers, a b c, for which,

$$a^2 + b^2 = c^2$$

For example, 32 + 42 = 9 + 16 = 25 = 52.
There exists exactly one Pythagorean triplet for which $a + b + c = 1000$.

Find the product abc.

5.1.1 Problem Statement

We have to find and a b and c for which the product $abc = 1000$. We have to check the different values of the three variables so the native method would involve three loops. However, if we have two values we can reason about the third since the sum of all three values should be 1000.

We also need to check given three values whether it is an Pythagorean triple. A function doing this for us would be appreciated.

5.1.2 The Root Chunk

The root chunk is defined followed by the name of the program. We named our program Pythagorean triple merely because the this is the name of the problem. You are aloud to chose any name you prefer for any chunk, only the root chunk is mandatory.

17a ⟨* 17a⟩≡ 60c ▷

⟨*pythagorean triple* 17b⟩

Next we define the steps and subdivide our problem into smaller portions. We need to include some libraries in the preamble plus some other constants that we need in our program. For example the 1000 value would be a nice constant to refer to since that's the value of the triple we are looking for.

17b ⟨*pythagorean triple* 17b⟩≡ (17a)

⟨*include libraries* 17c⟩
⟨*global constants* 17d⟩
⟨*program* 18a⟩

We need a library to write something to the standard output, which is stdio in the case of c. If we need more libraries in the future we can simply append them. Thereby I have also declared the constant 1000, which is the value of the Pythagorean triple we are looking for.

17c ⟨*include libraries* 17c⟩≡ (17b) 19b ▷

```
#include <stdio.h>
```

17d ⟨*global constants* 17d⟩≡ (17b)

```
#define MAX 1000
```

5.1.3 The triple

In order to determine whether three values are a Pythagorean triple, it would be useful to have a function or routine that checks this concept. We will need to test whether the three values are a Pythagorean triple. Let's introduce a function that does that and returns a boolean whether the tree values are indeed a Pythagorean triple.

18a ⟨*program* 18a⟩≡ (17b) 19a ▷

⟨*function triple test* 18b⟩

Given the three values a, b and c the function returns 1 if $a^2 + b^2 = c^2$. Furthermore it also checks this for the different positions of a, b, and c.

18b ⟨*function triple test* 18b⟩≡ (18a)

```
int isPythagoreanTriple(int a, int b, int c)
{
  if (a*a + b*b == c*c)
    return 1;

  if (a*a + c*c == b*b)
    return 1;

  if (b*b + c*c == a*a)
    return 1;

  return 0;
}
```

With this function we can check if the three given values are indeed a Pythagorean triple.

5.1.4 The loop

Now that we can check whether three values are a Pythagorean triple, we first need three values to test on. Simplest option is to simply loop over all the possible values and check whether they are a triple. Bare in mind that we only need two loops. Because if you have two values you can determine the third one since the addition of the three has to be equal to MAX. $a + b + c = 1000$.

To obtain the values a, b and c within the body of the double loop we can use the following statement. We can now use this abstraction within the source of our program to increase the readability.

18c ⟨*calculate the values for a b and c* 18c⟩≡ (19a)

```
c = MAX - a - b;
```

5.1.5 Main program

Thus let's fold all the information together and conclude our program to determine whether three values are a Pythagorean triple and there addition is equal to 1000.

19a ⟨*program* 18a⟩+≡ (17b) ◁18a

```
int main(int argc, char *argv[])
{
  int a, b, c;

  for (a = 1; a < MAX; ++a)
  {
    for (b = 1; b < MAX; ++b)
    {
      ⟨calculate the values for a b and c 18c⟩
      if (a + b + c == MAX)
      {
        if (isPythagoreanTriple(a, b, c))
        {
          ⟨print solution 19d⟩
          ⟨exit program 19c⟩
        }
      }
    }
  }
  ⟨exit program 19c⟩
}
```

To finalize this literate program we have to print the solution to the screen and exit the program. Exiting the program in the C programming language can be done by returning 0 or returning EXIT_SUCCESS, which is the same. I would like to use EXIT_SUCCESS, this is defined in the stdlib library this lets add this to our library declarations.

19b ⟨*include libraries* 17c⟩+≡ (17b) ◁17c

```
#include <stdlib.h>
```

19c ⟨*exit program* 19c⟩≡ (19a)

```
return EXIT_SUCCESS;
```

The problem description tells us that we have to print the product of the values a, b and c to the standard output. We can use the function printf which is defined in the standard c library for this purpose.

19d ⟨*print solution* 19d⟩≡ (19a)

```
printf("Triple: %d, %d, %d\n", a, b, c);
printf("Product: %d \n ", a*b*c);
```

Chapter 6

Conclusion

When literate programming one specifies the source code and documentation in one single document for which the programmer can specify any preferred order for the ease of human understanding. The program code can be extracted by tangling the document into a form which is understandable for an interpreter and thus can be compiled. The documentation can be obtained by weaving the document into a form ready to be interpreted from which a typesetter can create a nicely formatted readable document.

With regard to academic courses the aspects of literate programming add up providing a couple of valuable things. The change of perspective forces students to think more carefully about their design decisions merely because they have to justify them to the reader. Through this it becomes compulsory for students to subdivide their problems into bite-sized pieces which will not only increase the readers understanding but also their own. This problem reduction enhances the students in their programming capabilities as well as giving them a guideline to solve more complicated problems.

Literate programming comes to its best advantage when solving medium to large problems. Problems that are too small will not benefit from the concepts of literate programming simply because the problem is not really reducible any more. The problems that are written using literate programming will result in a well documented program that is an enjoyable experience for the audience to read. With that there is the ability to generate a working piece of software that matches the documentation. This property makes literate programming also a very good tool to write course material in since one will avoid the copy and pasting of pieces of source into their documentation. Noweb as a literate programming tool is thus well-suited for writing and maintaining course material for computer science courses.

I have used and worked with literate programming with great pleasure, I still use it because it a fun and educational to program and solve problems. I found the writing of literate programs a very addictive activity and will definitely keep on using it. This thesis is for a reason writing in noweb.

Chapter 7

Future Work

There are a few things that have to be done in order to incorporate literate programming successfully in to one of the courses given at the University of Groningen.

Students will want to create literate programs on the practical systems. Therefore noweb has to be installed and work properly on the systems. This however should not be too much of hassle because Noweb is written for Unix based systems and obtainable though the advanced packaging tool.

The judging system justitia should be adjusted to work with noweb files. The judges that judge source code should be able to handle a noweb file. Since the structure and extensibility of noweb allows the user to pipeline the outputs to another programs input it is possible to pipe notangles output towards the gcc compiler. This should make it possible for a judge to extract the C source code from a noweb file and judge this in a normal manner.

Chapter 8

Acknowledgement

I would like to thank Piter Dykstra for his help with the installation and startup of the literate programming environment on my computer and the enthusiastic conversation we had about literate programming, this made me very eager to embrace the aspects of literate programming and make them my own.

Furthermore I would like to thank Gerard Renardel for helping me with my thesis and the feedback he has given me throughout this last semester. The regular meetings helped me to keep on going with this project.

Chapter 9

Bibliography

[1] Andrew L. Johnson and Brad C. Johnson. “Literate Programming using noweb”. In: *Linux Journal, issue 42*, 1997.

[2] Donald E. Knuth. “Literate Programming”. In: *The Computer Journal, Vol.27, NO.2*, 1984, pp. 97–110.

[3] Donald E. Knuth. *The WEB System of Structured Documentation*. Tech. rep. Stanford University, 1983.

[4] Friedrich Leisch. “Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis”. In: *Proceedings of CompStat 2002*, 2002.

[5] Sebastien Li-Thiao-Te. “Literate program execution for teaching computational science”. In: *Procedia Computer Science 9*, 2012, pp. 1723 –1732.

[6] Norman Ramsey. “Literate Programming Simplified”. In: *IEEE Software.The Institute of Electrical and Electronics Engineers, Inc.* 1994, pp. 97–105.

[7] Wayne Sewell. *Weaving a program.* New York, NY: Van Nostrand Reinhold, 1989.

[8] H. Thimbleby. “Experiences of ’Literate Programming’ using cweb”. In: *The Computer Journal, Vol.29, NO.3*, 1986, pp. 201–111.

Appendix A

Quickstart

This is a quickstart guide that gives any user that want to learn noweb a rocket start of how to play with the literate programming concept. I will point out how to program hello world in a few steps.

1. Make sure noweb is installed correctly and noweave and notangle can be called on the commandline.
2. Create a noweb file. Open a terminal and type `touch hello.nw`
3. Open the file with a text editor. Type `open hello.nw`
4. Create a latex section and call it Hello World.
5. Create the code chunk that prints hello world.

   «print hello world»=

   printf("Hello World! \n");
6. Create a code chunk that includes your needed c library.

   «include library»=

   #include <stdio.h>
7. Create the code chunk that causes your program to exit.

   «exit program»=

   return 0;
8. On a seperate line type an @ to indicate that the next lines are documentation.
9. Write something very intriguing about your program.
10. Define the root chunk.

    «*»=

    «hello world»
11. Complete the program.

    «hello world»=

    «include library»

    int main()

{

«print hello world»

«exit program»

}

12. Open the terminal and extract the c code with the command `notangle hello.nw > hello.c`
13. Compile the code `gcc hello.c -o hello` and check if it works.
14. Extract the documentation using the command `noweave hello.nw > hello.tex`
15. Create a pdf with the command `pdflatex hello.tex` and look at your first literate program.

Appendix B

Noweb userguide

B.1 Literate Programming

Literate programming is a combination of source and documentation in such a way best suited for human beings to read. When writing literate programs the focus lies on explaining the reader what the code is suppose to do rather than instruction the computer how to run the program. The placement of the reader in a more central position, rather then the computer creates a new perspective on contemporary programming. The programmer has to justify every decision towards the reader and explain the reader of the intentions in a compact coherent sentence. This will most definitely force the programmer to think more critically and will result in a more structured way of programming. When problems are not solvable within a few lines of code, and thus not explainable in a few sentences, the programmer will have to subdivide the problem into smaller bite-sized pieces. The reduction of a problem will cause the programmer to think more carefully about certain choices and will result in a better understanding about the program for both the programmer and the reader. This is one of the most appealing aspects of literate programming.

B.1.1 Scope

This guide will cover the basics of literate programming and is meant to be a guideline to those who want to create literate programs. It will use examples written in the programming language C, with LaTeX as its typesetting language.

B.2 Structure

In general a literate program is written in one single file, in which both source and documentation are written. A literate program file has an extension which by convention we name *.nw*. This file is later processed by a literate programming tool, which is able to extract the source code and the documentation seperately. Were documentation would normally be of lesser importance it is now your and butter. Every decision, problem statement, problem indication or particular function details now requires thinking and clarification on their own.

First things first, since we are writing both documentation and source code in one file we need to be able to distinguish between them. This is done by dividing the file into blocks, so called chunks. We have documentation and code chunks. Each code chuck starts with «chunk_name» followed by a new line or whitespace. After the declaration of a code chunk the programmer can write source code, or divide the code chunk into smaller chunks. If you define a so called root chunk, «*» the literate programming tool extracting the source code will start from there and recursively call the other blocks linked to this one. The documentation chunk starts with a @ followed by a newline or whitespace. After which you can write documentation in natural language and use typesetting tools provided by for example LaTeX. These chunks can be placed in any order even the code chunks can be placed in such a way more convenient for the human reader.

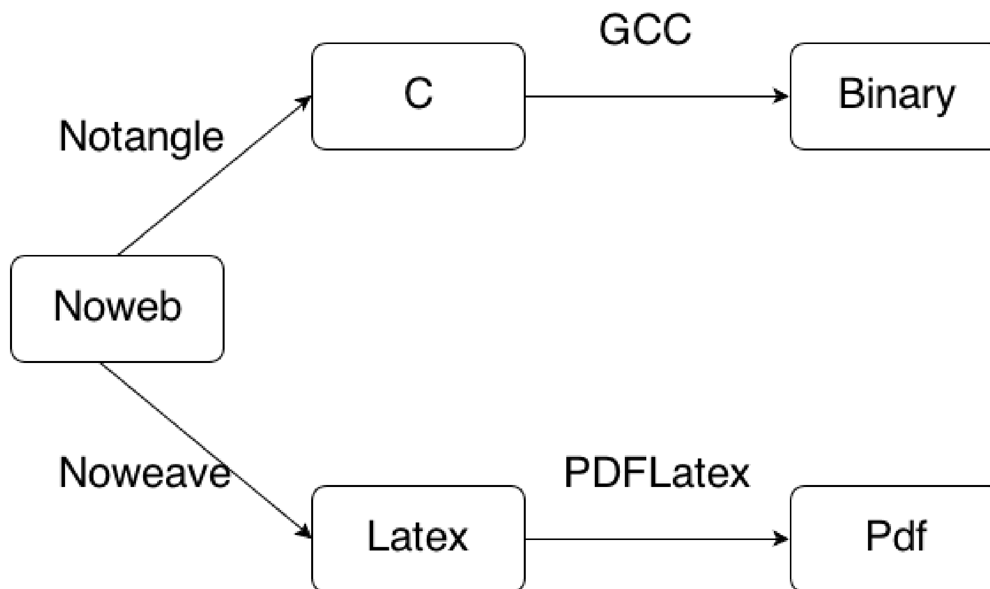
B.2.1 Source

This is an minor example of the infamous hello world example. The noweb file below shows a very minor example of a literate program using noweb. The documentation and code chunks in the program below are distinguished by the @ and «name» keywords.

```
1  \documentclass[a4paper, 10pt, twoside]{article}
2  \usepackage{noweb}
3
4  \begin{document}
5  \pagestyle{noweb}
6
7  @
8  \section*{Hello world}
9  This is an example of how to solve a problem using literate programming. The
       problem or goal here is to print the sentence hello world. \\
10
11 The actual printing of Hello World can be abstracted by introducing a code chunk
       that does that for us.
12
13 <<print hello world>>=
14 printf("Hello World!\n");
15
16 @
17 We cannot simply call the printf function, this is in the standard C library. Thus
        we create another code chunk to do this for us. Doing this has two advantages.
        One we can simply use the @<<libraries>> chunk in our coding and we can call
       it again to add libraries to the preamble on the fly when we need them.
18
19 <<libraries>>=
20 #include <stdio.h>
21
22 @
23 This is pretty much it, the only thing left is the definition of the program.
24
25 <<*>>=
26 <<libraries>>
27
28 int main(int argc, char *argv[])
29 {
30   <<print hello world>>
31   <<exit program>>
32 }
33 @
34
35 Exiting the program can be done through returning 0.
36 <<exit program>>=
37 return 0;
38 @
39 \end{document}
```

B.3 Noweb

Noweb is a literate programming tool used to process literate programs. Norman Ramsey created Noweb based on Donald Knuth implementation of Web, a tool for literate Pascal programs. Normans Web, Noweb is easier to use and language independent. Noweb has a number of interesting features from which two are of utter importance. These two routines can be called from the command line and generate their output to the standard output. They respectively weave and tangle the code providing both the documentation file and the ordinary source code files. These can then be processed by your own processors, for example and compiler and typesetting command. The routines are called `notangle` and `noweave`.



The figure above shows the structure of a literate program using noweb. The `.nw` can be processed by either of the two subroutines. The subroutines are listed below.

```
noweave program.nw > program.tex
notangle program.nw > program.c
```

Tangling and weaving are not stand-alone programs but a set of filters through which the `.nw` file is piped. This pipelining system makes noweb both flexible and extendable hence they can be modified to the users needs which will change the behaviour of noweb.

B.3.1 Notangle

The routine notangle is embedded in noweb and extracts the source code into a form suitable for further processing by an interpreted or compiler. Notangle will look for the specified root chunk and output this chuck line by line until another chunk is encountered. It will recursively add all the source code linked to this root chunk. If there is no root chunk specified, notangle will search for the default root chunk «*» and process from there. When there are two chunks with the same name, they are concatenated together in order of appearance.

Notangle flags:

-Rname
Expand the «name» code chunk. The -R option can be repeated, in which case each chunk is written to the output. If no -R option is given, expand the chunk named «*»

Now consider the source file stated above. We can process this source file with `notangle` which produces standard output. By simply running typing the following into the command line.

```
notangle hello.nw > hello.c
```

This generates a C source file which can be further interpreted by an C compiler.

```
1 #include <stdio.h>
2
3
4 int main(int argc, char *argv[])
5 {
6   printf("Hello World!\n");
7
8   return 0;
9 }
```

B.3.2 Noweave

The Noweb file can be processed by noweave to extract documentation which can then be further processed by a typesetting interpreter. Noweave extracts the contents of the documentation chunks as well as formatting the code chunks with cross references to one another. The output is written to the standard output and can thus be redirected to another file. Noweave can work with LaTeX, TeX and even HTML thought we only use LaTeX here.

Noweave flags:

-latex
Emit LaTeX, including wrapper in article style with the noweb package and page style. (Default)

-tex
Emit TeX, including wrapper in article style with the noweb package and page style. see **-latex**

-n
Don't use any wrapper (header or trailer). This option is useful when noweave's output will be a part of a larger document. See also **-delay**.

-delay
By default noweb write file information and preambles to the standard output before the first chunk. In general you don't want that if you wish to include your own LaTeX headers and packages for example. This delays the information, however with this option you do have to specify your begin and end document scopes. **-n** is included.

-x
For LaTeX, add a page number to each chunk name identifying the location of that chunk's definition, and emit cross-reference information relating definitions and uses.

-index
Buid cross-reference information (or hypertext links) for defined identifiers. When noweave **-index** is used with LaTeX, the control sequence \nowebindex expands to an index of identifiers.

Let's consider our Hello World example again. We can process this source file with `noweave` which produces standard output. Since the **-latex** flag is set by default we only need to redirect the output of noweave as well as the flags for indexing and delaying. By using **-delay** and **-x** we accomplish these needs. The delay flag is set because most of the time someone wants to include their own packages and definitions in the preamble of the latex file. Remember by doing that you do need to specify where in the noweb file your document starts and ends, like you would do in a latex file.

```
noweave -latex -delay -x hello.nw > hello.tex
```

Hello world

This is an example of how to solve a problem using literate programming. The problem or goal here is to print the sentence hello world.

The actual printing of Hello World can be abstracted by introducing a code chunk that does that for us.

32a ⟨*print hello world* 32a⟩≡ (32c)

```
printf("Hello World!\n");
```

We cannot simply call the printf function, this is in the standard C library. Thus we create another code chunk to do this for us. Doing this has two advantages. One we can simply use the «libraries» chunk in our coding and we can call it again to add libraries to the preamble on the fly when we need them.

32b ⟨*libraries* 32b⟩≡ (32c)

```
#include <stdio.h>
```

This is pretty much it, the only thing left is the definition of the program.

32c ⟨* 32c⟩≡

```
⟨libraries 32b⟩

int main(int argc, char *argv[])
{
  ⟨print hello world 32a⟩
  ⟨exit program 32d⟩
}
```

Exiting the program can be done through returning 0.

32d ⟨*exit program* 32d⟩≡ (32c)

```
return 0;
```

Primes

Now that we have seen how to create a literate program that prints hello world, which is not to exciting, I want to look at something more sophisticated. Let have a look at a problem which is called Primes. This is still relatively small thought gives a better indication of the usage of noweb.

Problem Description

Given the input N, write a program that print the first N prime numbers to the standard output.

Problem Indication

First thing that comes to mind is the ability to distinct prime numbers from normal numbers. Since I am not directl sure of how to solve this problem lets create a routine that might help us on this regard. We are writing C code thus we have to create a function, If choosen to call the function isPrime because we are programming imperatively.

isPrime

How to determine if an integer x is a prime number. There are some naive approaches, but this is what I found the cleanest method. Prime numbers are always odd expect for the number 2. Thus we have to check if a number is odd or even. This can be done using the modulo operator.

⟨*check if a number is even*⟩≡
```
if (isEven(number))
    return 0;
```

The isEven function can be defined as follows.

⟨*isEven function*⟩≡
```
int isEven(number)
{
  return (number % 2 == 0);
}
```

Furthermore a prime number has to be bigger then 0 because we are talking about natural numbers. Thereby the number 2 is the only even prime thus we have to filter this out before checking the parity.

⟨*check if a number is smaller than 3*⟩≡
```
if (number < 3)
    return (number == 2);
```

Last but not least we have to check all other cases for numbers larger than 3. However we do not have to check all the possibilities, only up to the root of the number x. This is because if x was not a prime, there would be two factors in the number x that would both be bigger than the square root of x.

⟨*check the rest bigger than 3*⟩≡
```
for (idx = 3; idx < sqrt(number); idx += 2)
{
  if (number % idx == 0)
  {
    return 0;
  }
}
```

We can now conclude the function using the previous declared chunks.

⟨*determine if a number x is a prime*⟩≡
```
int isPrime(int number)
{
  int idx;

  ⟨check if a number is smaller than 3⟩
  ⟨check if a number is even⟩
  ⟨check the rest bigger than 3⟩

  return 1;
}
```

If none of the checks return false, then the number is a prime number.

I also want to put this function declaration in a declaration chunk because If we want to declare more functions we can simply add them there.

⟨*functions*⟩≡
```
⟨isEven function⟩
⟨determine if a number x is a prime⟩
```

Libraries

The necessary libraries needed for the declaration of isPrime are a few standards plus the mathematics library. We need the standard library and input output library plus possible the math library to calculate the square root of and integer to speed up the actuall prime calculation.

```
⟨include libraries⟩≡
  #include <stdio.h>
  #include <stdlib.h>
  #include <math.h>
```

Global variables

Done, next on the list are the global variables. Since I do not want any input interaction yet, for the sake of simplicity I want to define the number of primes I would like to gerenate. This is to be N. Furthermore we need something to save our calculated prime number in, an array for that sake.

```
⟨global variables⟩≡
  #define N 10
```

Main program

Now that we have defined the libraries the function to test if a number is indeed a prime number we can start working on the main program. First define the root chunk from which everything is going to expand.

```
⟨*⟩≡
  ⟨include libraries⟩
  ⟨global variables⟩
  ⟨functions⟩

  int main()
  {
    ⟨print the first N primes⟩

    return 0;
  }
```

Printing the primes

Now that we can check whether a number is a prime we have to find them. The easiest way to to simply start of at 0 and start looping over all natural numbers untill we found N primes. We have to create a counter which reminds us when to stop searching for primes. Thereby we need a running number that increases every iteration of the loop.

⟨*print the first N primes*⟩≡

```
int primes_so_far = 0;
int running_number = 0;

while (1)
{
   ⟨print if running number is prime⟩
   ⟨increase running number⟩
   ⟨stop the loop when N equals running number⟩
}
```

This is basically the heart of the program. When a number is prime print it. With that we have to increase the primes found variable.

⟨*print if running number is prime*⟩≡

```
if (isPrime(running_number))
{
    primes_so_far++;
    printf("%d\n", running_number);
}
```

We have to increase the running number each iteration.

⟨*increase running number*⟩≡

```
running_number++;
```

Finnaly we have to stop looping over all natural numbers as soon as we found N primes.

⟨*stop the loop when N equals running number*⟩≡

```
if (N == primes_so_far)
    break;
```

The noweb document

```
1  \section*{Primes}
2
3  Now that we have seen how to create a literate program that prints hello world,
       which is not to exciting, I want to look at something more sophisticated. Let
       have a look at a problem which is called Primes. This is still relatively small
        thought gives a better indication of the usage of noweb. \\
4
5  \subsection*{Problem Description}
6  Given the input N, write a program that print the first N prime numbers to the
       standard output. \\
7
8  \subsection*{Problem Indication}
9
10 First thing that comes to mind is the ability to distinct prime numbers from
       normal numbers. Since I am not directl sure of how to solve this problem lets
       create a routine that might help us on this regard. We are writing C code thus
       we have to create a function, If choosen to call the function isPrime because
       we are programming imperatively. \\
11
12 \subsubsection*{isPrime}
13
14 How to determine if an integer $ x $ is a prime number. There are some naive
       approaches, but this is what I found the cleanest method. Prime numbers are
       always odd expect for the number 2. Thus we have to check if a number is odd or
        even. This can be done using the modulo operator. \\
15
16 <<check if a number is even>>=
17 if (isEven(number))
18     return 0;
19 @
20
21 The isEven function can be defined as follows. \\
22
23 <<isEven function>>=
24 int isEven(number)
25 {
26   return (number % 2 == 0);
27 }
28 @
29
30 Furthermore a prime number has to be bigger then 0 because we are talking about
       natural numbers. Thereby the number 2 is the only even prime thus we have to
       filter this out before checking the parity. \\
31
32 <<check if a number is smaller than 3>>=
33 if (number < 3)
34     return (number == 2);
35
36 @
37 Last but not least we have to check all other cases for numbers larger than 3.
       However we do not have to check all the possibilities, only up to the root of
       the number x. This is because if x was not a prime, there would be two factors
       in the number x that would both be bigger than the square root of x. \\
38
39 <<check the rest bigger than 3>>=
40 for (idx = 3; idx < sqrt(number); idx += 2)
41 {
```

```
42    if (number % idx == 0)
43    {
44      return 0;
45    }
46  }
47  @
48
49  We can now conclude the function using the previous declared chunks. \\
50
51  <<determine if a number x is a prime>>=
52  int isPrime(int number)
53  {
54    int idx;
55
56    <<check if a number is smaller than 3>>
57    <<check if a number is even>>
58    <<check the rest bigger than 3>>
59
60    return 1;
61  }
62  @
63
64  If none of the checks return false, then the number is a prime number. \\
65
66  I also want to put this function declaration in a declaration chunk because If we
        want to declare more functions we can simply add them there. \\
67
68  <<functions>>=
69  <<isEven function>>
70  <<determine if a number x is a prime>>
71  @
72
73  \subsubsection*{Libraries}
74
75  The necessary libraries needed for the declaration of isPrime are a few standards
        plus the mathematics library. We need the standard library and input output
        library plus possible the math library to calculate the square root of and
        integer to speed up the actuall prime calculation. \\
76
77  <<include libraries>>=
78  #include <stdio.h>
79  #include <stdlib.h>
80  #include <math.h>
81  @
82
83  \subsection*{Global variables}
84
85  Done, next on the list are the global variables. Since I do not want any input
        interaction yet, for the sake of simplicity I want to define the number of
        primes I would like to gerenate. This is to be N. Furthermore we need something
         to save our calculated prime number in, an array for that sake. \\
86
87  <<global variables>>=
88  #define N 10
89  @
90
91  \subsection*{Main program}
92  Now that we have defined the libraries the function to test if a number is indeed
        a prime number we can start working on the main program. First define the root
```

```
    chunk from which everything is going to expand. \\
93
94  <<*>>=
95  <<include libraries>>
96  <<global variables>>
97  <<functions>>
98
99  int main()
100 {
101   <<print the first N primes>>
102
103   return 0;
104 }
105 @
106
107 \subsection*{Printing the primes}
108 Now that we can check whether a number is a prime we have to find them. The
        easiest way to to simply start of at 0 and start looping over all natural
        numbers untill we found N primes. We have to create a counter which reminds us
        when to stop searching for primes. Thereby we need a running number that
        increases every iteration of the loop. \\
109
110 <<print the first N primes>>=
111 int primes_so_far = 0;
112 int running_number = 0;
113
114 while (1)
115 {
116   <<print if running number is prime>>
117   <<increase running number>>
118   <<stop the loop when N equals running number>>
119 }
120 @
121
122 This is basically the heart of the program. When a number is prime print it. With
        that we have to increase the primes found variable. \\
123
124 <<print if running number is prime>>=
125 if (isPrime(running_number))
126 {
127     primes_so_far++;
128     printf("%d\n", running_number);
129 }
130 @
131
132 We have to increase the running number each iteration. \\
133
134 <<increase running number>>=
135 running_number++;
136 @
137
138 Finnaly we have to stop looping over all natural numbers as soon as we found N
        primes. \\
139
140 <<stop the loop when N equals running number>>=
141 if (N == primes_so_far)
142     break;
143 @
```

B.4 Final words

These examples are meant as guidelines to hold on to whilst literate programming using noweb. By looking at these examples you are meant to generate an understanding of the concepts of literate programming and the usages of noweb. It includes a few words about the basics of literate programming and the usage of noweb.

Appendix C

How to install

C.1 Mac users

Noweb requires the ICON binaries to install properly.

1. Download ICON **here** *http://www.cs.arizona.edu/icon/*
2. Unzip and put the folder into your */usr/local/bin*
3. Download Noweb **here** *http://www.cs.tufts.edu/ nr/noweb/*

   or from Noweb's master distribution ftp **here** *ftp://www.eecs.harvard.edu/pub/nr*
4. Unzip the downloaded Noweb version
5. Open the folder go to *src/*
6. Open the Makefile and change the following:

   LIBSRC=**icon**

   ICONC=**icont**

   TEXINPUTS=**$(shell kpsewhich -expand-var='$$TEXMFLOCAL')/tex/latex/noweb**
7. Open a terminal
8. Navigate to *noweb/src*
9. Type to following command: `export PATH=$PATH:/usr/local/bin/icon-v950/bin`
10. Now install Noweb: `sudo make all install`
11. Open your **.bash_profile** and add the following line `export PATH=$PATH:/usr/local/noweb`
12. SET and ready to use Noweb

Done!

Common errors

Error step 10

The terminal might throw you a `getline error` during the installation. This is very unfortunate but is a known bug with versions < noweb-2.11b.

In order to complete the installation you will have to go deep into your binaries.

10.1. Open Finder and press: **(cmd + shift + g)** or naviage to *Go* and click *Go to Folder*

10.2. In the popup dialog type: **/usr/include**

10.3. Find and open the file `stdio.h`

10.4. Comment out line `.449` with the prototype of the `getline()` function

10.5. Now try step 10 again: `sudo make all install`

10.6. Go back to the stdio.h file and uncomment line `.449 getline()`

10.7. Continue at step 11.

C.2 Unix users

If your Unix distribution provides you with the Advanced Packaging Tool, installing noweb is as easy as installing any other piece of software. Simply type the following in the command line.

```
sudo apt-get install noweb
```

Done!

If your system does not provide the packaging tool I highly recommend getting it since it makes installing software on Unix distributions extremely easy. Although you can also install noweb using the following steps.

1. Download Noweb **here** *http://www.cs.tufts.edu/ nr/noweb/*
   or from Noweb's master distribution ftp **here** *ftp://www.eecs.harvard.edu/pub/nr*
2. Unzip the downloaded Noweb version
3. Open the folder go to *src/*
4. Open the Makefile and change the following:
   LIBSRC=**icon**
   ICONC=**icont**
5. Open a terminal
6. Navigate to *noweb/src*
7. Now install Noweb: `sudo make all install`
8. SET and ready to use Noweb

Done!

Appendix D

Module

Literate Programming: Poker

Literate programming is a combination of source and documentation in such a way best suited for human beings to read. When writing literate programs the focus lies on explaining the reader what the code is suppose to do rather than instruction the computer how to run the program. Basically the reader is placed in a central position, rather than the computer. Since the main focus lies on explaining the reader the intentions of the code the programmer is forced to verbalize his/her thoughts. Every decision has to be justified since we have to explain the reader of the our intentions with that particular piece of code.

In the next practical session you will get introduced with the basic concepts of literate programming.

File Structure

In general a literate program is written in one single file, in which both source and documentation are written. A literate program file has an extension which by convention we name *.nw*. This file is later processed by a literate programming tool, which is able to extract the source code and the documentation separately. Were documentation would normally be of lesser importance it is now your and butter. Every decision, problem statement, problem indication or particular function details now requires thinking and clarification on their own.

First things first, since we are writing both documentation and source code in one file we need to be able to distinguish between them. This is done by dividing the file into blocks, so called chunks. We have documentation and code chunks. Each code chuck starts with «chunk name»= followed by a new line or white space. After the declaration of a code chunk the programmer can write source code, or divide the code chunk into smaller chunks. If you define a so called root chunk, `<<*>>=` the literate programming tool

extracting the source code will start from there and recursively call the other blocks linked to this one. The documentation chunk starts with a followed by a newline or white space. After which you can write documentation in natural language and use typesetting tools provided by for example LaTeX. These chunks can be placed in any order even the code chunks can be placed in such a way more convenient for the human reader.

Source

This is an minor example of the infamous hello world example. It is a literate program which uses C code and LaTeX typesetting.

```
\documentclass[a4paper, 10pt, twoside]{article}
\usepackage{noweb}

\begin{document}
\pagestyle{noweb}

@
\section*{Hello world}
This is an example of how to solve a problem using literate programming. The
    problem or goal here is to print the sentence hello world. \\

The actual printing of Hello World can be abstracted by introducing a code chunk
    that does that for us.

<<print hello world>>=
printf("Hello World!\n");

@
We cannot simply call the printf function, this is in the standard C library. Thus
     we create another code chunk to do this for us. Doing this has two advantages.
     One we can simply use the @<<libraries>> chunk in our coding and we can call
    it again to add libraries to the preamble on the fly when we need them.

<<libraries>>=
#include <stdio.h>

@
This is pretty much it, the only thing left is the definition of the program.

<<*>>=
<<libraries>>

int main(int argc, char *argv[])
{
  <<print hello world>>
  <<exit program>>
}
@

Exiting the program can be done through returning 0.
<<exit program>>=
return 0;
@
\end{document}
```

D.1 Noweb

Noweb is a literate programming tool used to process literate programs. Norman Ramsey created Noweb based on Donald Knuth implementation of Web, a tool for literate Pascal programs. Normans Web, Noweb is easier to use and language independent. Noweb has a number of interesting features from which two are of utter importance. These two routines can be called from the command line and generate their output to the standard output. They respectively weave and tangle the code providing both the documentation file and the ordinary source code files. These can then be processed by your own processors, for example and compiler and typesetting command. The routines are called `notangle` and `noweave`.

D.1.1 Notangle

The routine notangle is embedded in noweb and extracts the source code into a form suitable for further processing by an interpreted or compiler. Notangle will look for the specified root chunk and output this chuck line by line until another chunk is encountered. It will recursively add all the source code linked to this root chunk. If there is no root chunk specified, notangle will search for the default root chunk `«*»` and process from there. When there are two chunks with the same name, they are concatenated together in order of appearance.

Notangle flags:

-Rname

Expand the «name» code chunk. The -R option can be repeated, in which case each chunk is written to the output. If no -R option is given, expand the chunk «*».

Now consider the source file stated above. We can process this source file with `notangle` which produces standard output. By simply running typing the following into the command line.

```
notangle hello.nw > hello.c
```

This generates a C source file which can be further interpreted by an C compiler.

```
1 #include <stdio.h>
2
3
4 int main(int argc, char *argv[])
5 {
6   printf("Hello World!\n");
7
8   return 0;
9 }
```

D.1.2 Noweave

The Noweb file can be processed by noweave to extract documentation which can then be further processed by a typesetting interpreter. Noweave extracts the contents of the documentation chunks as well as formatting the code chunks with cross references to one another. The output is written to the standard output and can thus be redirected to another file. Noweave can work with LaTeX, TeX and even HTML thought we only use LaTeX here.

Noweave flags:

-latex

Emit LaTeX, including wrapper in article style with the noweb package and page style. (Default)

-tex

Emit TeX, including wrapper in article style with the noweb package and page style. see **-latex**

-n

Don't use any wrapper (header or trailer). This option is useful when noweave's output will be a part of a larger document. See also **-delay**.

-delay

By default noweb write file information and preambles to the standard output before the first chunk. In general you don't want that if you wish to include your own LaTeX headers and packages for example. This delays the information, however with this option you do have to specify your begin and end document scopes. **-n** is included.

-x

For LaTeX, add a page number to each chunk name identifying the location of that chunk's definition, and emit cross-reference information relating definitions and uses.

-index

Buid cross-reference information (or hypertext links) for defined identifiers. When noweave **-index** is used with LaTeX, the control sequence \nowebindex expands to an index of identifiers.

Let's consider our Hello World example again. We can process this source file with `noweave` which produces standard output. Since the **-latex** flag is set by default we only need to redirect the output of noweave as well as the flags for indexing and delaying. By using **-delay** and **-x** we accomplish these needs. The delay flag is set because most of the time someone wants to include their own packages and definitions in the preamble of the latex file. Remember by doing that you do need to specify where in the noweb file your document starts and ends, like you would do in a latex file.

```
noweave -latex -delay -x hello.nw > hello.tex
```

Practical Assignment

In the card game poker, a hand consists of five cards and are ranked, from lowest to highest, in the following way:

- **High Card:** Highest value card.
- **One Pair:** Two cards of the same value.
- **Two Pairs:** Two different pairs.
- **Three of a Kind:** Three cards of the same value.
- **Straight:** All cards are consecutive values.
- **Flush:** All cards of the same suit.
- **Full House:** Three of a kind and a pair.
- **Four of a Kind:** Four cards of the same value.
- **Straight Flush:** All cards are consecutive values of same suit.
- **Royal Flush:** Ten, Jack, Queen, King, Ace, in same suit.

The cards are valued in the order:
2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace.

Note that the value 10 is coded as T.

If two players have the same ranked hands then the rank made up of the highest value wins; for example, a pair of eights beats a pair of fives (see example 1 below). But if two ranks tie, for example, both players have a pair of queens, then highest cards in each hand are compared (see example 4 below); if the highest cards tie then the next highest cards are compared, and so on. `http://projecteuler.net`

Example

Consider the following five hands dealt to two players:

Hand	Player 1	Player 2	Winner
1	5H 5C 6S 7S KD Pair of Fives	2C 3S 8S 8D TD Pair of Eights	Player 2
2	5D 8C 9S JS AC Highest card Ace	2C 5C 7D 8S QH Highest card Queen	Player 1
3	2D 9C AS AH AC Three Aces	3D 6D 7D TD QD Flush with Diamonds	Player 2
4	4D 6S 9H QH QC Pair of Queens Highest card Nine	3D 6D 7H QD QS Pair of Queens Highest card Seven	Player 1
5	2H 2D 4C 4D 4S Full House With Three Fours	3C 3D 3S 9S 9D Full House with Three Threes	Player 1

The file, `poker_test1.txt` and `poker_test2.txt` each contain a number **n** that represents the amount of games to be played, followed by **n** random hands dealt to two players. Each line of the file contains ten cards (separated by a single space): the first five are Player 1's cards and the last five are Player 2's cards. You can assume that all hands are valid (no invalid characters or repeated cards), each player's hand is in no specific order, and in each hand there is a clear winner.

Print for each hand the winner of that game!

Thus for example.

Input:

```
2
5H 5C 6S 7S KD 2C 3S 8S 8D TD
5D 8C 9S JS AC 2C 5C 7D 8S QH
```

Output:

```
Player 2
Player 1
```

Step1

Before you are going to compare the hands think of how you would represent a hand in your code. How would you represent a card? And how would you encode a player?

*(*hint*)A player has a hand, which consists of 5 cards, a card has a value and a suit.*

Step2

Getting the data into your program. The input file, or test file how to read from a file.

*(*hint*) Have a look at fscanf() or fgetc() and getchar()*

Step3

Comparison algorithm, given two players with their respective cards, how would you decide which hand is the strongest.

*(*hint*) Give each hand a rank that represents the strength, enum*

*(*hint*) When both hands are equals decide on border cases who is the winner*

Practical Assignment: Elaboration

Problem Statement

In the card game poker, a hand consists of five cards and are ranked, from lowest to highest, in the following way:

- **High Card:** Highest value card.
- **One Pair:** Two cards of the same value.
- **Two Pairs:** Two different pairs.
- **Three of a Kind:** Three cards of the same value.
- **Straight:** All cards are consecutive values.
- **Flush:** All cards of the same suit.
- **Full House:** Three of a kind and a pair.
- **Four of a Kind:** Four cards of the same value.
- **Straight Flush:** All cards are consecutive values of same suit.
- **Royal Flush:** Ten, Jack, Queen, King, Ace, in same suit.

The cards are valued in the order:
2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace.

Note that the value 10 is coded as T.

If two players have the same ranked hands then the rank made up of the highest value wins; for example, a pair of eights beats a pair of fives (see example 1 below). But if two ranks tie, for example, both players have a pair of queens, then highest cards in each hand are compared (see example 4 below); if the highest cards tie then the next highest cards are compared, and so on. `http://projecteuler.net`

Example

Consider the following five hands dealt to two players:

Hand	Player 1	Player 2	Winner
1	5H 5C 6S 7S KD Pair of Fives	2C 3S 8S 8D TD Pair of Eights	Player 2
2	5D 8C 9S JS AC Highest card Ace	2C 5C 7D 8S QH Highest card Queen	Player 1
3	2D 9C AS AH AC Three Aces	3D 6D 7D TD QD Flush with Diamonds	Player 2
4	4D 6S 9H QH QC Pair of Queens Highest card Nine	3D 6D 7H QD QS Pair of Queens Highest card Seven	Player 1
5	2H 2D 4C 4D 4S Full House With Three Fours	3C 3D 3S 9S 9D Full House with Three Threes	Player 1

The file, `poker_test1.txt` and `poker_test2.txt` each contain a number **n** that represents the amount of games to be played, followed by **n** random hands dealt to two players. Each line of the file contains ten cards (separated by a single space): the first five are Player 1's cards and the last five are Player 2's cards. You can assume that all hands are valid (no invalid characters or repeated cards), each player's hand is in no specific order, and in each hand there is a clear winner.

Print for each hand the winner of that game!

Problem Indication

The exercise states that each player has 5 cards and each card contains two pieces of information. The first indicates the value of the card and the second the color or type. The value has a range from 2 up to A which can be represented by a number. Namely count up from 10, to J = 11, Q = 12, K = 13, A = 14. The reason as to why we define this is because I want to start off with defining some global variables. The face cards.

53a ⟨*face cards* 53a⟩≡ (59b)

```
#define JACK 11
#define QUEEN 12
#define KING 13
#define ACE 14
```

Furthermore it would be handy to define the four different suits.

53b ⟨*suits* 53b⟩≡ (59b)

```
#define CLUBS 1
#define DIAMONDS 2
#define HEARTS 3
#define SPADES 4
```

These defines will be handy to refer to in the program.

Card

Each player has 5 cards, and like stated before a card is made up out of two pieces of information. Let's define a struct card which we can use for that purpose. The struct contains two integer fields for the suit and the value.

54a ⟨*card* 54a⟩≡ (59b)

```
typedef struct
{
  int value;
  int suit;
} card;
```

We can also define a player struct that consists of a hand, or array of 5 cards. Thereby it will come clever if we define a rank on which we check which of the hand is the strongest and a number points. I am doing this because I already thought of the problem what will happen if both players have the same Rank, for example two pair and we have to determine which of the two is the winner. With this we can also introduce our definition of two players as a global variable.

54b ⟨*player* 54b⟩≡ (59b)

```
typedef struct
{
  card cards[5];
  int rank;
  int points;
} player;
```

54c ⟨*global variables* 54c⟩≡ (60c) 60b ▷

```
player player1, player2;
```

We now need to invent the rank value. This represent the value of the hand which can be listed from high to low.

55 ⟨*card rank* 55⟩≡ (59b)

```
enum
{
   ROYAL_FLUSH     = 10,
   STRAIGHT_FLUSH  =  9,
   FOUR_OF_A_KIND  =  8,
   FULL_HOUSE      =  7,
   FLUSH           =  6,
   STRAIGHT        =  5,
   THREE_OF_A_KIND =  4,
   TWO_PAIR        =  3,
   ONE_PAIR        =  2,
   HIGH_CARD       =  1
} RANK;
```

Input

Since we have an input file which we have to read there has to be some input redirection to fill up our players cards. We have to open the input file and read it line by line. The first 5 cards on the input line are for player1 and the second 5 for player two. After this we have to compare the hands and declare one of them the winner.

We need to open the file and read the characters one by one untill we filled up all ten cards. Thus effectively reading 20 character ommiting white space. We can use to loops one for each player, in both loops we can read two characters at a time to fill the cards. We can use the C function fscanf to automatically ignore whitespace, we also need a token to put the newly read character in. The functions to set the players card value and suit are still to be

56 ⟨*read cards from input line* 56⟩≡ (59c)

```
void readCardFromLine(FILE *fp)
{
  int idx;
  char value, suit;

  for (idx = 0; idx < 10; ++idx)
  {
    if (idx < 5)
    {
      fscanf(fp, " %c %c", &value, &suit);
      setValue(&player1.cards[idx], value);
      setSuit(&player1.cards[idx], suit);
    }
    else
    {
      fscanf(fp, " %c %c", &value, &suit);
      setValue(&player2.cards[idx - 5], value);
      setSuit(&player2.cards[idx - 5], suit);
    }
  }
}
```

The first character on a card is the value, which can be a number between 2 and 10 or a character representing the face of the card. Thus we have to check whether the character is a digit smaller then 10, otherwise it is a face card and we have to determine which face card it is.

57 ⟨*set value function* 57⟩≡ (59c)

```
void setValue(card *card, char token)
{
  int value = token - '0';
  if (value < 10)
    card->value = value;
  else
  {
    switch(token)
    {
      case 'T':
      {
        card->value = 10;
        break;
      }
      case 'J':
      {
        card->value = JACK;
        break;
      }
      case 'Q':
      {
        card->value = QUEEN;
        break;
      }
      case 'K':
      {
        card->value = KING;
        break;
      }
      case 'A':
      {
        card->value = ACE;
        break;
      }
    }
  }
}
```

The second input symbol will be parsed on to the function below that determines what suit the current card is.

58a ⟨*set suit function* 58a⟩≡ (59c)

```
void setSuit(card *card, char token)
{
  switch(token)
  {
    case 'C':
    {
      card->suit = CLUBS;
      break;
    }
    case 'D':
    {
      card->suit = DIAMONDS;
      break;
    }
    case 'H':
    {
      card->suit = HEARTS;
      break;
    }
    case 'S':
    {
      card->suit = SPADES;
      break;
    }
  }
}
```

We can now read an input line and set the appropriate cards for the respective players. Thought we have to keep on reading from the file untill their is no more input. Thus we need some kind of while loop that does that forever. We also need to check who is the winner after each iteration in this main while loop. Let us introduce a loop that opens the file and checks if there is another line to read. This loop also has to invoke some checking function that declares a winner.

58b ⟨*main program* 58b⟩≡ (60c) 61a ▷

```
FILE *fp;
int idx;
fp = fopen(argv[1], "r");

if (fp == NULL)
{
  perror("Error while opening the file.\n");
  exit(EXIT_FAILURE);
}
```

Print cards

This function was only used for testing purposes, to check if the cards are read in properly.

59a ⟨*print cards* 59a⟩≡ (59c)

```
void printCards()
{
  int idx;
  for (idx = 0; idx < 10; ++idx)
  {
    if (idx == 0) printf("Player1: ");
    if (idx < 5)
    {
      printf("card[%d]= [%d][%d] ", idx,
                           player1.cards[idx].value,
                           player1.cards[idx].suit);
    }
    else
    {
      if (idx == 5) printf("\nPlayer2: ");
      printf("card[%d]= [%d][%d] ", idx - 5,
                           player2.cards[idx - 5].value,
                           player2.cards[idx - 5].suit);
    }
  }
  printf("\n");
}
```

Program

All the defines can be summarized within one chunk to make it easier to refer to if we want to add anything.

59b ⟨*defines* 59b⟩≡ (60c)

⟨*face cards* 53a⟩
⟨*suits* 53b⟩
⟨*card* 54a⟩
⟨*card rank* 55⟩
⟨*player* 54b⟩

All the functions defined so far can be summarized under another banner for the same purpose as the defines.

59c ⟨*function declarations* 59c⟩≡ (60c) 72a ▷

⟨*print cards* 59a⟩
⟨*set suit function* 58a⟩
⟨*set value function* 57⟩
⟨*read cards from input line* 56⟩

Main Program

We can now start defining our main program, before we continue to introduce more constants. The preamble of our program looks like this.

60a ⟨*libraries* 60a⟩≡ (60c)

```
#include <stdio.h>
#include <stdlib.h>
```

60b ⟨*global variables* 54c⟩+≡ (60c) ◁54c

```
int number_of_hands = 0;
```

We can also create the main program but omit the body just yet because we don't know how to implement that one yet.

60c ⟨* 17a⟩+≡ ◁17a

```
⟨libraries 60a⟩
⟨defines 59b⟩
⟨global variables 54c⟩
⟨function declarations 59c⟩

int main(int argc, char *argv[])
{
  ⟨main program 58b⟩
  return 0;
}
```

We have defined a few functions that was a result from the previous chapters, aswell as the main skeleton of our main function. The only thing missing now is the implementation of the actual algorithm that compares two hands and the processing of the input files. This is the main setup of the program, from here one we can start adding functions to the predefined blocks aswell as starting to build the main program.

The first number on the input line gives the amount of hands that is going to be played. This number can be used as a reference to count how many more games we have to decide.

61a ⟨*main program* 58b⟩+≡ (60c) ◁58b 72b▷

```
fscanf(fp, "%d", &number_of_hands);
```

Comparison Algorithm

Now the hardest part, how do we decide the winner based on the hands. Everything is set now, the only thing is deciding whom ever is the winner.

One way of tackling this problem is checking each hand and define the rank, from top to bottum. First check royal flush, then straight flush and so on. All the function are returning a boolean which tells us whether the player has that rank or not. The tricky part is the decision of the points. We will need this in case both players have the same hand rank.

Royal flush, this is the case if the player has a hand which has the same suit and they are listed from ace to 10. Thus because we are lazy we first check it the suit is the same. Followed by if the he has a street.

The hasFlush function. Because all the cards have to have the same value, we can simply look at one and check if they all match this one.

61b ⟨*function normal flush* 61b⟩≡ (72a)

```
int hasFlush(player *player)
{
  int idx;
  int suit = player->cards[0].suit;

  for (idx = 0; idx < 5; ++idx)
  {
    if (suit != player->cards[idx].suit)
      return 0;
  }
  return 1;
}
```

The function hasFlush(player player) returns 1 if this player has a flush and 0 is he hasn't. Since a royal flush is a combination of a flush and a street we can use a function to decide if the player has a street. I am designing this function to return the highest card of the street or 0 if there is no street. Thereby we need some function to determine the highest card of a hand.

62a ⟨*function street* 62a⟩≡ (72a)

```
int hasStraight(player *player)
{
  int value = getHighestCard(player);
  int idx;
  for (idx = 0; idx < 5; ++idx)
  {
    if (!handContains(player, value))
      return 0;

    value-;
  }
  return 1;
}
```

I have used two functions here which we have not build yet. A function to return the highest card of that hand, and a function that return 0 if the player does not have that card and return the amount if the player does have that card.

62b ⟨*function get highest card* 62b⟩≡ (72a)

```
int getHighestCard(player *player)
{
  int idx, value = 0;
  for (idx = 0; idx < 5; ++idx)
  {
    if (player->cards[idx].value > value)
      value = player->cards[idx].value;
  }
  return value;
}
```

62c ⟨*function contains* 62c⟩≡ (72a)

```
int handContains(player *player, int target)
{
  int idx;
  int counter = 0;
  for (idx = 0; idx < 5; ++idx)
  {
    if (player->cards[idx].value == target)
      counter++;
  }
  return (counter > 0) ? counter : 0;
}
```

The hand ranks

The royal flush decision is based on whether someone has a flush and a street with the ACE as the highest card.

63a ⟨*function royal flush* 63a⟩≡ (72a)

```
int hasRoyalFlush(player *player)
{
  if (!hasFlush(player))
    return 0;

  if (hasStraight(player) != ACE)
    return 0;

  return 1;
}
```

The street flush is a combination of having a street and a flush, being nearly the same as royal flush only it does not matter what the height of street doesn't matter.

63b ⟨*function street flush* 63b⟩≡ (72a)

```
int hasStraightFlush(player *player)
{
  if (!hasFlush(player))
    return 0;

  if (!hasStraight(player))
    return 0;

  return 1;
}
```

The four of a kind can be checked by walking over all the cards and check if he has four of these. We are using the ⟨*function contains* 62c⟩ function here.

64 ⟨*function four of a kind* 64⟩≡ (72a)

```
int hasFourOfAKind(player *player)
{
  int idx, value;
  int quartet = 0;
  int highcard = 0;
  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;

    if (handContains(player, value) == 4)
      quartet = value;
  }
  if (quartet != 0)
  {
    player->points = highcard;
    return 1;
  }
  return 0;
}
```

The determination of a full house is similar to the four of a kind approach, only now we have to find a pair and a three of a kind.

65 ⟨*function full house* 65⟩≡ (72a)

```
int hasFullHouse(player *player)
{
  int idx, value;
  int triple_value = 0, pair_value = 0;

  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;

    if (handContains(player, value) == 3)
      triple_value = value;

    if (handContains(player, value) == 2)
      pair_value = value;
  }

  if (triple_value != 0 && pair_value != 0)
  {
    player->points = triple_value * 100 + pair_value;
    return 1;
  }

  return 0;
}
```

The three of a kind is similar to the four of a kind just now we check if the hand contains three of the same values.

66 ⟨*function three of a kind* 66⟩≡ (72a)

```
int hasThreeOfAKind(player *player)
{
  int idx, value;
  int triple = 0;

  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;
    if (handContains(player, value) == 3)
      triple = value;
  }

  if (triple != 0)
  {
    for (idx = 0; idx < 5; ++idx)
    {
      if (player->cards[idx].value != triple)
        player->points += player->cards[idx].value;
    }
    player->points += triple * 100;
    return 1;
  }

  return 0;
}
```

The two pair value we have to check on two different pair, thus two different card that are presented twice. So we are going to look for a pair twice, only the second time we also check if the founded pair is not equal to the previously found pair. If one the the pairs are 0, thus we found less then 2 pairs return 0.

67 ⟨*function two pair* 67⟩≡ (72a)

```
int hasTwoPair(player *player)
{
  int idx, value;
  int pair_one = 0, pair_two = 0, highcard = 0;

  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;
    if (handContains(player, value) == 2)
    {
      pair_one = value;
    }
  }

  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;
    if (value != pair_one)
    {
      if (handContains(player, value) == 2)
        pair_two = value;
      else
        highcard = value;
    }
  }

  if (pair_one == 0 || pair_two == 0)
    return 0;

  if (pair_one < pair_two)
  {
    int temp = pair_one;
    pair_one = pair_two;
    pair_two = pair_one;
  }

  player->points = pair_one * 1000 + pair_two * 10 + highcard;
  return 1;
}
```

The pair function is simply checking if one card value contains twice. However the decision of the points is more tricky, since the pair itself has to yield in more points than the other cards.

68 ⟨*function pair* 68⟩≡ (72a)

```
int hasPair(player *player)
{
  int idx, value;
  int pair_value;
  for (idx = 0; idx < 5; ++idx)
  {
    value = player->cards[idx].value;
    if (handContains(player, value) == 2)
    {
      pair_value = value;
    }
  }
  if (pair_value != 0)
  {
    for (idx = 0; idx < 5; ++idx)
    {
      if (player->cards[idx].value != pair_value)
      {
        player->points += player->cards[idx].value;
      }
    }
    player->points += pair_value * 100;
    return 1;
  }

  return 0;
}
```

The cases

So now to continue, we can make a big check function to check all the cases for each hand and decide which player has which rank. While doing this the functions will also fill in the appropriate points.

69 ⟨*decide players hand* 69⟩≡ (72a)

```
void decideRank(player *player)
{
  if (hasRoyalFlush(player))
  {
    player->rank = ROYAL_FLUSH;
  }
  else if (hasStraightFlush(player))
  {
    player->rank = STRAIGHT_FLUSH;
  }
  else if (hasFourOfAKind(player))
  {
    player->rank = FOUR_OF_A_KIND;
  }
  else if (hasFullHouse(player))
  {
    player->rank = FULL_HOUSE;
  }
  else if (hasFlush(player))
  {
    player->rank = FLUSH;
  }
  else if (hasStraight(player))
  {
    player->rank = STRAIGHT;
  }
  else if (hasThreeOfAKind(player))
  {
    player->rank = THREE_OF_A_KIND;
  }
  else if (hasTwoPair(player))
  {
    player->rank = TWO_PAIR;
  }
  else if (hasPair(player))
  {
    player->rank = ONE_PAIR;
  }
  else
  {
    player->rank = HIGH_CARD;
    player->points += player->cards[0].value * 100000000
                    + player->cards[1].value * 1000000
                    + player->cards[2].value * 10000
                    + player->cards[3].value * 100
                    + player->cards[4].value * 1;
  }
}
```

There is one thing that needs some explanation and that is the else case. If the hand does not comply to any of the ranks the player only has a high card. Deciding the points has to be done by giving credits from the highest card down. I am doing an assumption here, the assumption that we sort the hands of each player. Something we will have to arrange in the next section.

Sorting

For sorting I will simply use bubble sort, bubbling the high values to the from and the lowest to the end. This way we will obtain a sorted array from high to low.

70 ⟨*sort players hands* 70⟩≡ (72a)

```
void sortHand(player *player)
{
  int i, j;
  for(i = 0; i < 5; ++i)
  {
    for (j = i + 1; j < 5; ++j)
    {
      if (player->cards[i].value < player->cards[j].value)
      {
        card temp = player->cards[i];
        player->cards[i] = player->cards[j];
        player->cards[j] = temp;
      }
    }
  }
}
```

The only thing that lasts up is one function that calls all these previously defined function and prints the winner of that hand. This function is called every iteration until there are no hands left to play. The first thing it should do is call the sorting algorithm on both hands to sort the hands. After which if has to decide that ranks and declare a winner.

71 ⟨*decide winner* 71⟩≡ (72a)

```
void decideWinner()
{
  sortHand(&player1);
  sortHand(&player2);

  decideRank(&player1);
  decideRank(&player2);

  if (player1.rank == player2.rank)
  {
    if (player1.points > player2.points)
    {
      printf("Player1\n");
    }
    else
    {
      printf("Player2\n");
    }
  }
  else
  {
    if (player1.rank > player2.rank)
    {
      printf("Player1\n");
    }
    else
    {
      printf("Player2\n");
    }
  }
}
```

These are all the functions to decide whether what kind of hand the player has. They need to be in the function declaration section so we define them here.

72a ⟨*function declarations* 59c⟩+≡ (60c) ◁59c
⟨*function get highest card* 62b⟩
⟨*function contains* 62c⟩
⟨*function pair* 68⟩
⟨*function two pair* 67⟩
⟨*function three of a kind* 66⟩
⟨*function street* 62a⟩
⟨*function normal flush* 61b⟩
⟨*function full house* 65⟩
⟨*function four of a kind* 64⟩
⟨*function street flush* 63b⟩
⟨*function royal flush* 63a⟩
⟨*sort players hands* 70⟩
⟨*decide players hand* 69⟩
⟨*decide winner* 71⟩

So finally we can fill the main body of our program. We have to read one line from the input which will be handled by our `readCardFromLine` function. Furthermore we have to decide a winner based on those cards with `decideWinner` and keep on doing this `number_of_hands` times.

72b ⟨*main program* 58b⟩+≡ (60c) ◁61a

```
for (idx = 0; idx < number_of_hands; idx++)
{
  readCardFromLine(fp);
  decideWinner();
}
```