

# **Preparatory lab ID1020 Algorithms and Data structures**

2019-05-02

# Table of Contents

1	Introduction.....	1
2	What you should know from previous courses .....	1
3	Requirements on your code.....	2
3.1	Comments – documentation .....	2
3.2	Requirement: Comments in your code .....	3
3.3	Why you should design tests for your code .....	3
3.4	Requirement: Tests.....	3
3.5	Requirement: Where your test code should be placed.....	5
4	On the assignments .....	5
5	Assignment 1: Set up a development environment .....	5
6	Assignment 2: A text changing filter in C.....	5
7	Assignment 3 a, b: Placing data in an array.....	6
8	Assignment 4 a, b: Linked lists in JAVA .....	6
8.1	Assignment 4a: Single linked lists.....	7
8.2	Assignment 4b: Double linked circular lists .....	8
9	Assignment 5: Binary trees.....	10
10	Background: Redirecting input/output .....	12
10.1	I/O to from the terminal .....	12
10.2	Redirecting I/O .....	13
10.3	Redirecting output.....	14
10.4	Redirecting input.....	15
10.5	Pipeing: Redirecting the output from one process to be the input of another process.....	16
10.6	An example how to pipe output/input between processes .....	16
11	Background: Some useful commands/things in UNIX/LINUX.....	17
11.1	man pages.....	17
11.2	Shells and shellscript .....	18
11.3	Accessing the LINUX servers at KTH Kista.....	18
12	Background: Basic functions for I/O in C.....	18
13	Background: References, pointers, addresses and memory management.....	19
13.1	Pointers and addresses in C.....	19
13.2	Memory management in C .....	21
13.3	Dynamic memory allocation in C .....	22
13.4	References in JAVA .....	23
14	Background: How to build linked data structures in C.....	24
15	Background: Common pitfalls when programming C.....	26
15.1	Test for equality in the condition of if, for or while statements.....	26
15.2	Confusing bit-wise and logical operators.....	27
15.3	Indexing out of bounds and addressing errors .....	27
16	Background: The best pieces of advice I can give! .....	28
17	Index.....	29

## 1 Introduction

The goal of the Algorithms and Data structures course is that you should learn about algorithms and data structures used to address common problems encountered when designing software. In many cases there are more than one algorithm/data structure that could be used to solve a specific problem. In such cases you should understand how to select an appropriate algorithm/data structure. The best choice of algorithm/data structure may depend on parameters such as the size and characteristics of the input, memory usage and execution time etc. You should also be able to tailor algorithms/data structures to efficiently solve specific problems

With this said it should be evident that this course is not a programming primer but a course which builds on that you know how to program. It is not programming language dependent in the sense that the algorithms and data structures can be implemented in different languages. However, as different programming languages have different pros and cons you should be able to select the appropriate language to solve a specific problem.

In this course we will use the JAVA language for most of the assignments as an example of an object oriented language. But we will also use the C programming language in cases where its lower overhead and more direct access to system calls<sup>1</sup> can result in more efficient solutions. You should know these languages from the ID1018 and IS1200 courses.

Experiences from previous course rounds show that in many cases the 7-8 months that have passed since students studied ID1018 and the 5 months since IS1200 means that much of the knowledge from these courses may have faded. Thus this preparatory lab recapitulates the knowledge you should have and also advances it a little bit in the assignments. Many of the problems you will solve in this lab are directly useful when you do the regular labs in ID1020.

## 2 What you should know from previous courses

When designing courses and chains of courses one base the course design on the assumption that students entering the course have met the learning outcomes and know the content of pre-requisite courses. However, time is a factor and knowledge may fade.

For ID1020 you should have working knowledge of the following:

Concept	JAVA	C	Example
Development environment	X	X	Eclipse, gcc
Simple debugging	X	X	Eg. what is built in to the IDE <sup>2</sup> you have used (for example Eclipse) and/or gdb
Basic datatypes and variables in C and Java	X	X	int, double, char, char *, int * string
Arrays	X	X	
Flow control	X	X	if, while, for

<sup>1</sup> System calls are the functions that implements the interface to the Operating system, i.e. the services a user program can get from the operating system. Examples of such services can be to read/write files or to allocate memory

<sup>2</sup> IDE – Integrated Development Environment

Methods, functions	X	X	
Parameter passing	X	X	value, reference, pointer
Classes	X		
Instantiating objects	X		
References, pointers, addresses	X		Build linked data structures by using objects, C: <code>struct</code> and pointers
Pointers, addresses, dereferencing		X	Parameter passing, calculating addresses, dereferencing. Build linked data structures by using <code>struct</code> and pointers
Simple I/O	X	X	Read and write numbers, characters and text to/from a terminal (window)
Memory management		X	<code>malloc()</code> , <code>free()</code>
Arguments to <code>main()</code>	X	X	

### 3 Requirements on your code

The requirements on your code in this course are basic professional requirements. The code that you will develop as an engineer will be used in applications that people depend on – hence you should make sure that it is correct, efficient and possible to maintain. This means that all code you write should be well documented and tested.

#### 3.1 Comments – documentation

Comments and documentation serve two purposes. The most intuitive is to serve the purpose of explaining to others what problem your code is supposed to solve and how it does it. Equally important is that by writing documentation you will understand your code better and possibly see where it can be improved or even where it might be incorrect.

Documentation may come as: i) documentation that is separate from the code; and ii) documentation integrated into the code in the form of comments. In real projects you will have both types of documentation. However, in this course you are only required to write well-commented code.

The advantage of having well-commented code is that by not separating the code from the documentation (comments) the documentation is more likely to reflect any changes to the code. Hence, it should not come as a surprise that there are systems available, such as **Javadoc**, which can collect specially formatted comments from (JAVA) code and generate separate documentation from the comments. Javadoc can parse JAVA code to generate HTML-documents with pre-formatted headings from comments in the code. Javadoc is integrated in many IDE's such as Eclipse. While it is not a requirement to use Javadoc in this course, it is recommended that you learn how to use it.

- How to write comments for Javadoc is explained in:  
<https://www.oracle.com/technetwork/java/javase/tech/index-137868.html>.

### 3.2 Requirement: Comments in your code

The minimum requirements on your code in this course is that you have inserted the following comments into it:

- Start each source code file with a comment stating:
  - Who is the author
  - When the code was generated, when it has been updated
  - What problem the code solves
  - How it is used (how it is executed, what input it takes, what it outputs)
  - What it has been based on (if you have “borrowed” code, then you must state what code and from what source)
- Each class and method should have a comment describing it and its purpose
- Each “non-intuitive” declaration and statement should have an explanatory comment

### 3.3 Why you should design tests for your code

One of the things that distinguishes an engineer from an amateur is that we would expect the engineer to create solutions that are both efficient and correct. Faulty or erroneous software can at best be annoying. But incorrect code could easily lead to more severe consequences and in the worst case threaten people’s lives. Lives may be jeopardized if the safety systems of an auto pilot systems in an aircraft does not work correctly or if a defibrillator fails to restart a heart. Correctness and efficiency are key elements of this course. So what can we do to build systems that work correctly?

Ideally we would like to guarantee the correctness of our systems by proving it. This is referred to as *verification* and is in most cases out of scope. We usually cannot verify the whole chain necessary to execute a program consisting not only of our code but also of computer hardware, operating system, compilers, run-time environments and networks.

In most cases we have to stick to the second best alternative which is to validate our systems. *Validation*<sup>3</sup> means that we try to ensure/make it plausible that a system works properly. In software development this is a process which starts already in the planning stages of a project where we should study the problem and try to understand it in detail before we start designing solutions for it. In the design process we should also design tests early on for how to test that the system we implement works as intended. Testing is one method for validating the implementation of a system. Testing and tests are central to most system development and project methods such as Agile project methods and Test Driven Development (TDD).

*What is important to realize as a student is that to be able to design proper tests you also have to gain a thorough understanding of how the systems you design and the algorithms you use work in detail. Thus testing the algorithms in the course and how you use them is a good way to learn and understand the internals of how the algorithms and data structures work.*

### 3.4 Requirement: Tests

In this course all code you write should have proper unit (module) tests. Unit testing refers to the testing of the individual units a system is built from. In particular the tests you design should perform/allow for glass/white box testing of your code. This means that the tests should allow for testing the code with all possible/plausible inputs and should test all parts of the code (i.e. all methods/branches). The input should not only be the expected or correct input but also non-expected

---

<sup>3</sup> It is quite common that the concepts of verification and validation are confused on the WWW

and incorrect input. Testing all parts of the code means that if you detect that parts of the code never can be reached, (dead code), then these parts of the code should be removed.

**Example:** Assume we would like to test the correctness of a method with two parameters of type *double* which divides the parameters *a* and *b* and returns the result ( $a/b$ ). For this simple example we could test with a number of combinations of *a* and *b* to cover the cases of using positive, negative and zero values. We could also test with different datatypes such integer or floating point values as parameters (i.e. we need to know if the method should work properly with both integer and floating point values as parameters?). The values used should generate results where we know what to expect as results. In this case we could make a list of the combinations of values for *a* and *b* and the expected results. Testing with  $b=0$  should probably result in an error. Moreover, one could design tests to test the accuracy of the division and the maximal/minimal values of the parameters that can be used while the output is still correct.

a	b	result
1	2	0.5 (assuming the method should work properly with integer input)
-1	2	-0.5
1	-2	-0.5
-1	-2	0.5
1.0	2	0.5
1	2.0	0.5
1.0	2.0	0.5
-1.0	2.0	-0.5
1.0	-2.0	-0.5
-1.0	-2.0	0.5
1	0	error
1	0.0	error
1.0	0	error
1.0	0.0	error

What one should learn from the above is that it is difficult and in many cases impossible to (manually) do exhaustive testing<sup>4</sup> of even simple code with few parameters. In fact there has been an example of where a large hardware manufacturer shipped many thousands (millions) of processors used in PC's that had an error in the floating point division.

***However, that testing is hard should not prevent us from designing good tests!***

---

<sup>4</sup> Exhaustive testing is hard/impossible to do even with automated tools

### 3.5 Requirement: Where your test code should be placed

So where should you place your test code? In JAVA each class can have a class-method `main()`. Thus you should place your test code for each class (unit) in the class main-method.

In C you cannot have more than one `main()` function in a program. In fact the names of globally visible functions and variables need to be unique in a program to avoid name collisions. So in this course you place your test code in a function declared as `static void test()`<sup>5</sup> for each assignment.

## 4 On the assignments

By solving the following assignments you will create solutions you can build on for many of the regular lab assignments. If you plan on taking the ADK course you may want to go the extra mile and try to solve all assignments in both JAVA and C.

## 5 Assignment 1: Set up a development environment

**Assignment 1:** Set up a development environment in which you can create JAVA and C programs (it may be two separate environments), execute the programs and debug them.

My preferred development environment for C and other languages is to use a LINUX/UNIX system<sup>6</sup> and tools such as EMACS for editing and GNU utilities like gcc (C/C++ compiler) and gdb (debugger). How to set up an environment to log on to the LINUX-servers in Kista is briefly described in Section 11.3.

## 6 Assignment 2: A text changing filter in C

**Assignment 2:** Create a filter written in C that reads characters from `stdin` until an end-of-file marker (EOF) is read. For every character read the filter should check if the character is the character 'a' in which case it should output an 'X' to `stdout` otherwise it should output the character read to `stdout`.

Tests:

- 1) test the filter by manually insert text from the keyboard. EOF is normally represented by <ctrl>-d
- 2) write a small text file with input. Feed the text file as input to the filter by redirecting the input (see Section 10) to be read from the text file

Hints: Use the functions `getchar()` and `putchar()` (see Section 12) , a `main()` function containing one `while`-loop and one `if-else` statement (it need not be more complex). The code will look the same regardless if you use Windows or UNIX/LINUX

Extra: Try to implement the same program in JAVA and test it in the same way as the C-program.

---

<sup>5</sup> The static declaration of the function `test()` will make it invisible outside the source code file (i.e. the unit of compilation) in which it is declared (i.e. the name/identifier will not appear in the symbol table)

<sup>6</sup> It is *not* a requirement that you develop your code in a LINUX/UNIX environment

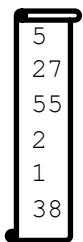
## 7 Assignment 3 a, b: Placing data in an array

**Assignment 3a:** Create a JAVA-program which:

- 1) Reads a positive number from `stdin` into an `int` variable named `nrElements`. The value is the number of elements in a dynamically allocated array of integers.
- 2) Creates an array of integers with `nrElements` elements.
- 3) Reads `nrElements` integers from `stdin` and stores them in the array.
- 4) Prints the elements of in the array to `stdout` in reverse order compared to how they were input.

Execute the program reading the input from the keyboard.

Create a text file containing the input (see the example below of a file where the first number is the number of elements followed by the five elements) and execute the program feeding the text file as input by redirecting the input to be read from the file instead of from the keyboard.



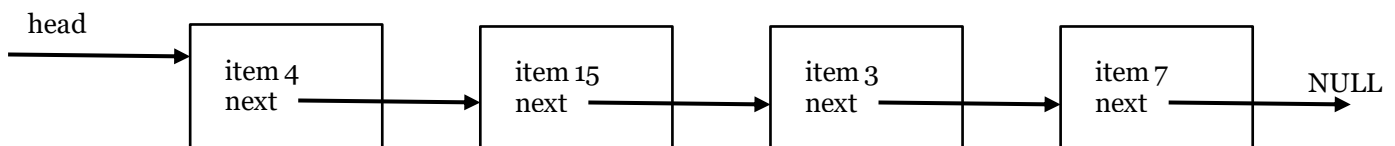
```
5
27
55
2
1
38
```

**Assignment 3b:** Redo the assignment by writing a C-program which does the same thing as the JAVA-program and execute it reading input from the keyboard and from the text file respectively. (See Section 13 for how to manage memory allocation in C)

## 8 Assignment 4 a, b: Linked lists in JAVA

In many applications there is a need to handle and store sets of data that may vary in size also during the execution. One commonly used technique to build data structures to hold such data sets is to use linked data structures.

In JAVA a linked data structure is built from objects which contain data and references to other objects in the data structure. The simplest example of such a data structure is a single linked linear list. The essential parts of code for a class that can be used to instantiate the elements (nodes) of such a data structure is found in Section 13.4. Below you find an illustration of a linked list built from such objects where `item` is used to store an integer value and `next` is a reference to the next element in the list (or `NULL` if the element is the last element in the list). `head` is a reference to the first element in the list. The list in the figure contains elements with the data 4, 15, 3, 7.



In the course book<sup>7</sup> there is a nice description of how to build single linked lists to implement stacks and FIFO-queues on pages 142-154 (a PDF of this chapter is found here <https://algs4.cs.princeton.edu/13stacks/>) The following assignment builds on this.

---

<sup>7</sup> The complete book is available online together with many other resources relevant for the course at <https://algs4.cs.princeton.edu/>

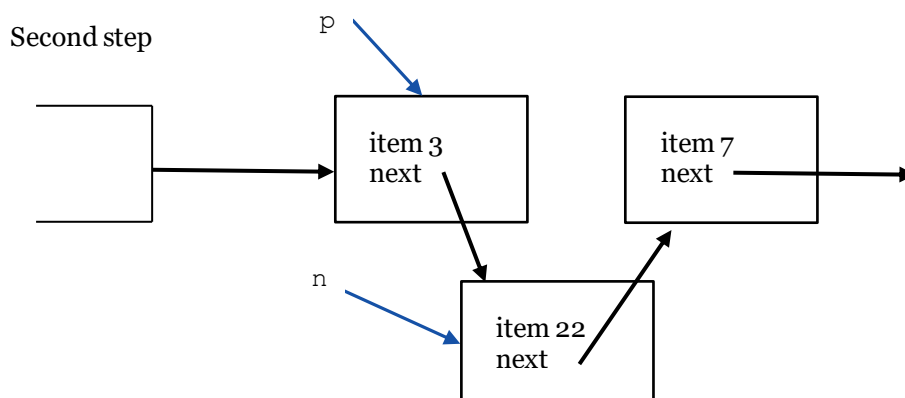
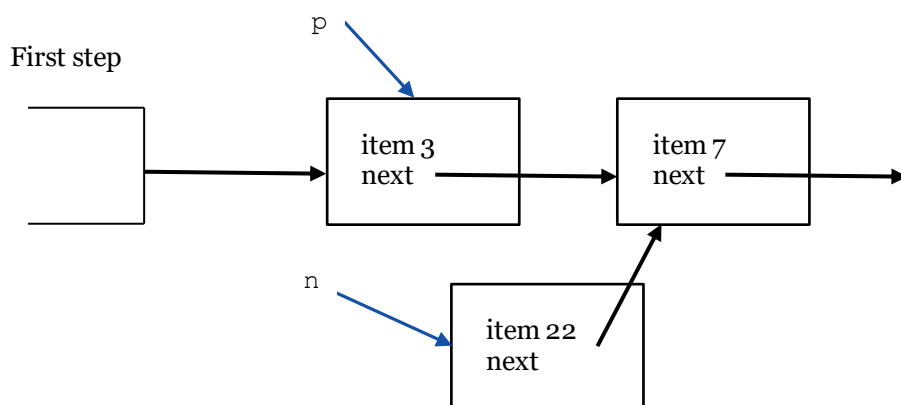
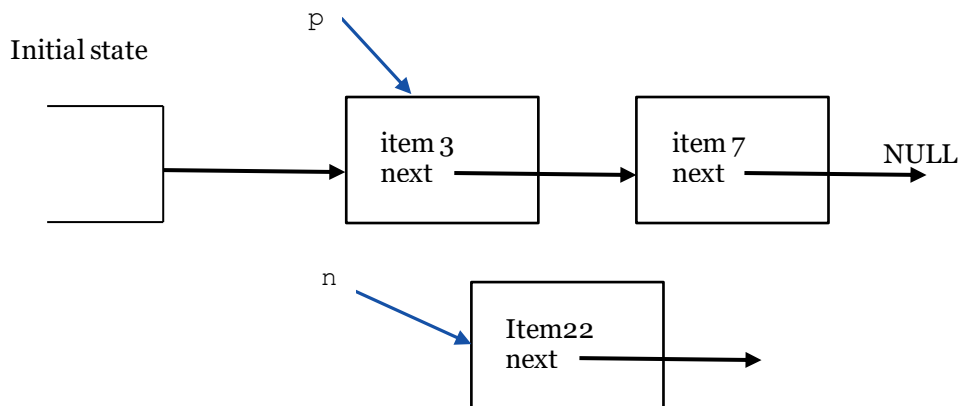


## 8.1 Assignment 4a: Single linked lists

### Assignment 4a:

- 1) Write code that implements a single linked list implementing a stack (i.e. a LIFO queue) based on the code from the book (see above). The code should have a method to insert a new element and a method to remove and return the first element in the list. For the sake of simplicity you can assume that the data stored in the elements are integer numbers.
- 2) Implement a method to print the data in the elements in the list (without removing elements). For instance if you were to print the list in the example above you should print: 4 15 3 7
- 3) Create a method to insert a new element (pointed to by the reference **n** in the example below) after an element in the list pointed to by a reference **p**. The first step is to set the **next** reference in the element pointed to by **n** to point to the same element as the next reference in the element pointed to by **p**. The second step is to make the **next** reference in the element pointed to by **p** set to point to the element pointed to by the reference **n**

Example:

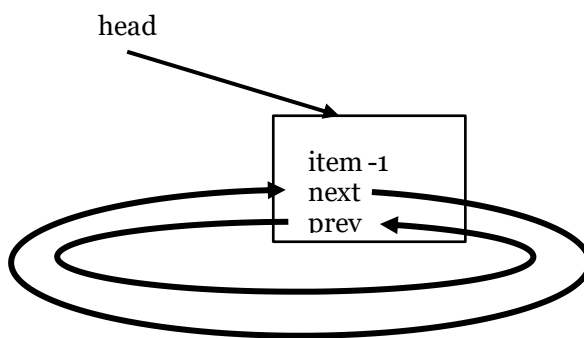


- 4) Write a method to insert new elements sorted in ascending order in the list according to the values of the data stored in the elements. Inserting elements in such an order implements an example of a simple priority queue<sup>8</sup>.
- 5) Implement tests in `main()` to test your implementation.

## 8.2 Assignment 4b: Double linked circular lists

Operations on single linked linear lists are a little bit tricky to implement requiring auxiliary pointers when inserting elements in an ordered fashion and they require that one handle the special cases for the first and last elements a little bit differently. By adding an extra reference (`prev`) in the elements that points to the previous element in the list and making the list circular one can make lists that trade some memory space (the extra reference) against being simpler to handle in many cases.

The smallest double linked circular list contains only one element where the `next` and `prev` references point to the element itself:



Whether or not one should use the element pointed to by `head` as only a marker of where the circular list ends/starts, or as a regular element of the list where we can store data, is an issue that there are different opinions on. I prefer to use this element only as a marker, as it otherwise would create unnecessary problems on how to update the `head` reference/pointer if we try to insert a new element at the beginning of the list (i.e. think of how you would handle the case if we tried to insert a new element which should be the new start/end of the list, i.e. replacing the element in the figure above. In which case the `head` reference/pointer should point to the new element instead of the element containing -1).

Inserting a new element in a list after another element involves resetting four references/pointers:

- 1) Set the `next` reference of the new element to point to the same element as the `next` reference in the element after which the new element is to be inserted
- 2) Set the `prev` reference of the new element to point to the element after which it is inserted
- 3) Set the `next` reference of the element after which the new element is to be inserted to point to the new element.
- 4) Set the `prev` reference in the element pointed to by the `next` reference in the new element to point to the new element.

---

<sup>8</sup> A priority queue is a queue for which the element with the highest priority in the queue is always returned when an element is removed (dequeued) from the queue.

Example: Assume we insert a new element after the element pointed to by `head`. Assume we have a reference `n` which points to the new element. In pseudo code the operations will look like this:

- 1) `n->next = head->next`
- 2) `n->prev = head`
- 3) `head->next = n`
- 4) `n->next->prev = n`

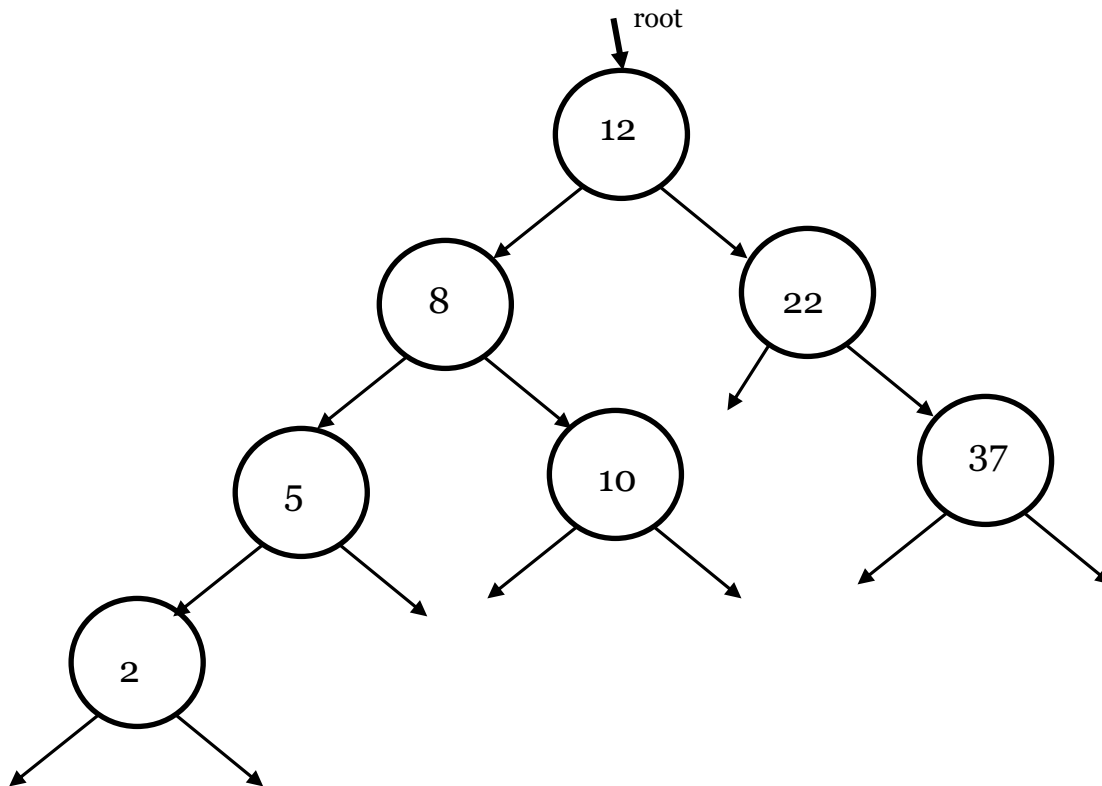
#### Assignment 4b:

- 1) Extend the class implementing elements (nodes) for the single linked list to implement a double linked list. That is add an extra reference `prev` and change the method to instantiate a new object from the class so that both the `next` and `prev` references point to the newly created object, i.e. a new object is in itself a minimal circular list.
- 2) Implement a method to insert a new element into the list after an element pointed to by a reference. This method should have two parameters, a reference to the element after which the new element is to be inserted and a reference to the new element.
- 3) Assume the data in an element is an integer value that can be used as a key. Implement a method which inserts new elements to the list sorted in ascending order by the value of the data/key.
- 4) Implement a method to print the content of the list. Start from the `head` element (the first element in the list) and traverse the list following the `next` references until you reach the element pointed to by `head` again.
- 5) Implement a method to remove an element from a list. This method need only one parameter, a reference to the element being removed. Return a reference to the element removed. The `prev` and `next` references in the removed element should be set to point to the removed element. You may need to think extra on how to handle the case if one tries to remove the `head` of the list while the list contains other elements also.
- 6) Implement tests in `main()` to test your implementation

Hint: When trying to understand how and in which order to set the references it is helpful to draw pictures explaining step by step how to do this.

## 9 Assignment 5: Binary trees

A binary tree<sup>9</sup> is a linked data structure where each node (element) can have up to two descendants normally named `left` and `right`. The first node in a tree is called the `root`. In a binary search tree all nodes also contains a `key` by which the nodes in the tree are ordered. The normal ordering of the nodes in a binary tree is that all nodes in the left sub-tree of the root has keys which are smaller (in some sense) than the key in the root, while all elements in the right sub-tree of the root has elements with keys larger or equal to the key in the root. This order should hold for all sub-trees in the tree. Below is an example of an ordered tree.



Trees are interesting by many standards. They can allow the user to efficiently look-up/search for elements in the data structure and have inspired some of the most efficient sorting algorithms known. Also the easiest way to implement operations on trees often is to implement recursive methods. A recursive method is a method which calls itself.

You will encounter many problems in this course which are easiest, and in many cases most efficiently, solved by recursive methods. In real life we often use recursion to solve problems such as sorting hand of cards without ever spending a second thought on that we use recursion. However, when we learn how to program we often start by learning how to program sequential solutions without recursion. This often unconsciously prevents us from finding the most natural solutions to problems which may well be recursive solutions. So what properties does a description of an algorithm have which indicates that the algorithm (solution to the problem) could be recursive? If you discover that you can describe a solution to a problem in a way that you can handle part of the problem directly and that the remaining part of the problem, which now is smaller than the original problem, can be solved in the same way as the original problem – then you have described a recursive solution.

That each problem solved in each iteration (recursive call) should be smaller than the problem from where the recursive call is made is important. If this would not hold the recursion would never terminate.

---

<sup>9</sup> Binary trees have at most two branches/sub-trees connected to a node. In general trees may have any number of branches connected to a node.

A method to insert an element in an ordered tree can be described in pseudo code as:

```
insert (root, newNode)
  if newNode->key < root->key then
    if root->left == NULL then root->left = newNode
    else insert(root->left, newNode)
  else
    if root->right == NULL then root->right = newNode
    else insert(root->right, newNode)
```

So how can we show that the problem is smaller in each recursion-step in the above method/function? We know that we cannot have trees of infinite depth, nor should there be any circular structures in a tree. Thus when we do a recursive call we will traverse the tree along some branch and we know that the depth of the sub-tree (the parameter root in the method/function) will be less than the depth of the tree we came in to the function with. Thus we will eventually reach a node which has no sub-tree where we want to insert the new node.

When traversing a tree to process its content there are three distinct ways of doing it referred to as *prefix*, *infix* and *postfix*. These names refer to in which order we process the content of a node in comparison to when we process the left and right sub-trees of the node respectively.

Prefix traversal can be described in pseudo code as:

```
prefixTraversal(root)
  if root != NULL then
    process root->data
    prefixTraversal(root->left)
    prefixTraversal(root->right)
```

Infix traversal can be described in pseudo code as:

```
infixTraversal(root)
  if root != NULL then
    infixTraversal(root->left)
    process root->data
    infixTraversal(root->right)
```

Postfix traversal can be described in pseudo code as:

```
postfixTraversal(root)
  if root != NULL then
    postfixTraversal(root->left)
    postfixTraversal(root->right)
    process root->data
```

### Assignment 5:

- 1) Assume the processing of data is to print the key. Show what the output would be if the example tree above is traversed in prefix, infix and postfix order.
- 2) Show how a binary search tree would be built if you insert data that are: i) un-ordered; ii) ordered in ascending order; iii) ordered in descending order
- 3) Create a class that can be used to implement a binary search tree where the keys are integer values. When a new object is instantiated the `left` and `right` references should be set to `NULL`
- 4) Implement a method to insert elements in a binary search tree.
- 5) Implement methods to traverse and print the key values in the nodes to `stdout` which traverses the tree in prefix, infix and postfix order respectively.
- 6) Implement a method to search for a key in your binary tree. The method should take a reference to the root and an integer key value to search for as parameters. If the key is found in the tree then the method should return `TRUE` else it should return `FALSE`.
- 7) Implement tests in `main()`

## 10 Background: Redirecting input/output

In this course you will learn about and work with algorithms and data structures designed to be able to work also with large input data sets. The input data is normally stored in files as is often the output. Thus for a program to work on such a data set it needs to be able to read and possibly write files. However, you have not learned how to read and write files in any of the courses which are pre-requisites for this course. So do you need to learn how to access files from your JAVA and C-programs?

The answer to this is no – you need not to learn how to do this!<sup>10</sup> It is sufficient that you know how to read and write from/to the keyboard/terminal/window. In most operating systems you can redirect input/output to/from a process<sup>11</sup> to read/write from/to files or other processes. The following is a description of how I/O is done in UNIX/LINUX. However, most of these principles also apply to other operating systems such as the Microsoft operating systems.

### 10.1 I/O to from the terminal

All I/O goes through the operating system. In order to have one simple interface for a process to communicate with its environment (terminal, keyboard, files etc.) all I/O is performed as if the process was reading/writing from/to files. However, these need not be files on a disk and are referred to as streams. Streams may lead to/from all types of I/O devices such as disks, terminals, keyboards, mice etc.

When a process starts to execute it has three streams open, one for input and two for output. The input stream is referred to as standard input (`stdin`) and takes its input from the keyboard. The first output stream is referred to as standard output (`stdout`). `stdout` leads to the terminal/window in which the process was started. The standard output stream is buffered which means that when you output (write) something to this stream the data is actually written to a buffer in the operating system. This allows

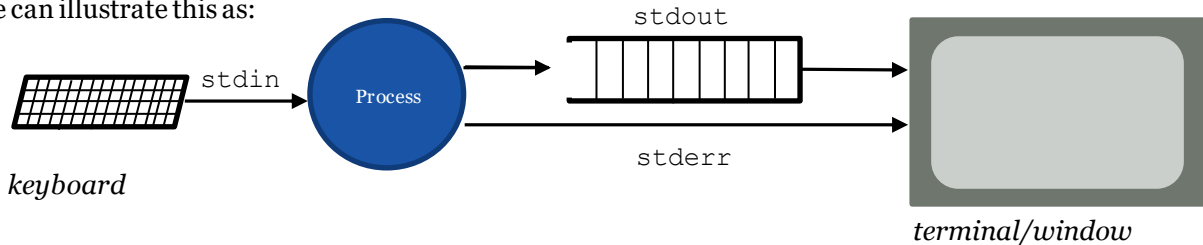
---

<sup>10</sup> Although learning how to read and write files is quite simple. The basic system calls you need to learn are: `open()`, `close()`, `read()`, `write()`, `lseek()`. And the related methods in languages like JAVA and the higher level functions in the C/C++ libraries are very similar to these system calls.

<sup>11</sup> A process is a program being executed

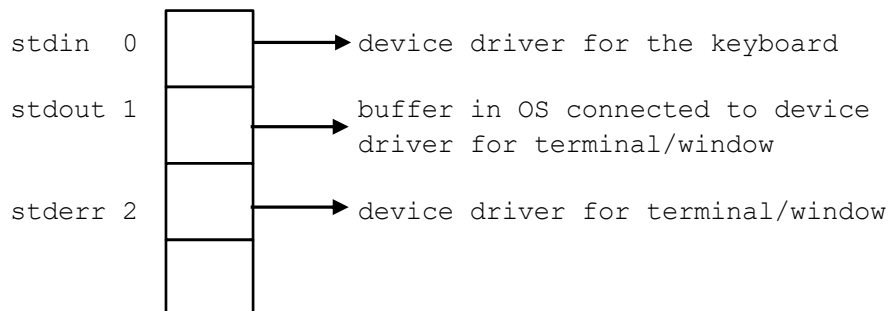
the operating system to output the content of the buffer when it is not occupied with other tasks. There is a second output stream open referred to as standard error (`stderr`). By default this also leads to the terminal/window in which the process was started. The difference between `stdout` and `stderr` is that `stderr` is not buffered which means that when something is output from the process on `stderr` the operating system has to output it to the terminal/window immediately without delay. Thus, if you have written an error message to `stdout` it may reside in the output buffer when a process crashes in which case the user never will see it. Had it instead been written to `stderr` before the process crashes, then it would also have been output to the terminal.

We can illustrate this as:



The open streams are implemented by that the operating system keeps a small table of what is called file descriptors for the open streams of the process. Each entry for an open stream identifies the device driver for the I/O-unit that the stream is attached to. The device driver implements functions corresponding to the system calls used to `read()`, `write()` data which the process calls when it wants to read/write data from/to the stream. For example, if a process calls `write()` to output something on a stream it results in that the operating system looks up the `write()` function in the device driver referred to by the entry in the file descriptor table corresponding to that stream and it will call that function with the parameters provided in the `write()` call made by the process. We can illustrate the file descriptor table for the example in the figure above as:

filedescriptor table for the process



So for example, what happens when we read something from the keyboard (`stdin`) is that we read from the device that is referred to from file descriptor 0 which normally would be the keyboard. And when we write to the terminal/window (`stdout`) we write to the device referred to from file descriptor 1.

## 10.2 Redirecting I/O

So what if we would like to have our program read from a file instead of the keyboard when it is executed, or write its output to another file rather than the terminal/window. Does this mean that we have to change the code for our program? The answer to this is no!

What we can do is to instruct the command interpreter<sup>12</sup> (normally referred to as a shell) to redirect the input/output to be read from/written to files. When redirecting input the command interpreter will set the `stdin` file descriptor to point to the device driver responsible for the file we want to read

<sup>12</sup> The command interpreter is the program running when you interact with the operating system using the command line. In other words, it is the program which outputs the prompt and then reads and interpret/performs the commands you write on the command line.

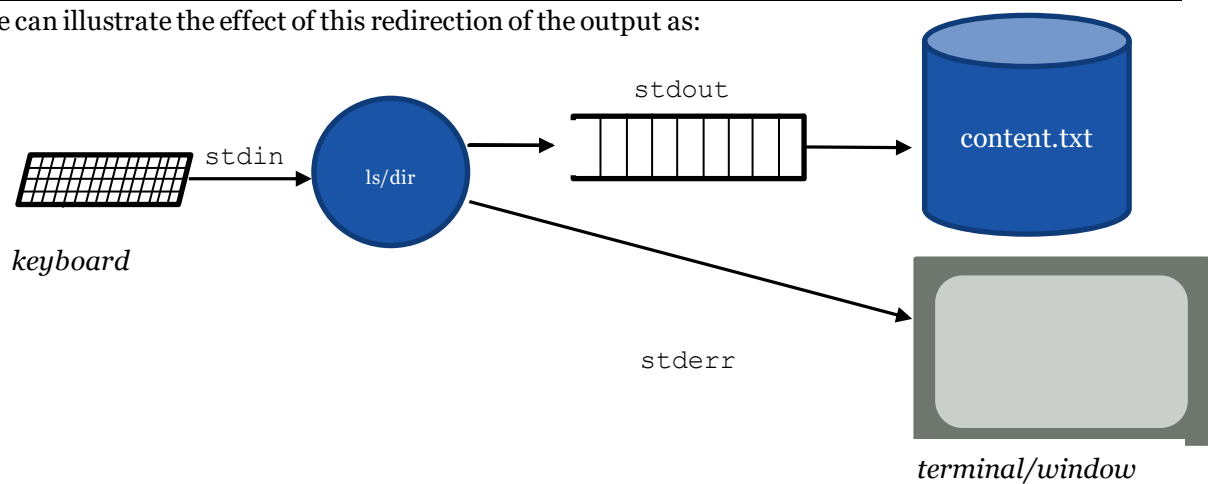
from and when redirecting output it will set the `stdout` file descriptor to point to the device driver responsible for the file we want to output to.

### 10.3 Redirecting output

To redirect the output from a process one uses `>`. The commands to redirect output from a process executing `program` to a `file` are:

UNIX/LINUX	Windows
<pre>program &gt; file</pre> <p><i>example: output the listing of the directory<sup>13</sup> to a file named <code>content.txt</code></i></p> <pre>ls &gt; content.txt</pre>	<pre>program &gt; file</pre> <p><i>example: output the listing of the directory<sup>14</sup> to a file named <code>content.txt</code></i></p> <pre>dir &gt; content.txt</pre>

We can illustrate the effect of this redirection of the output as:



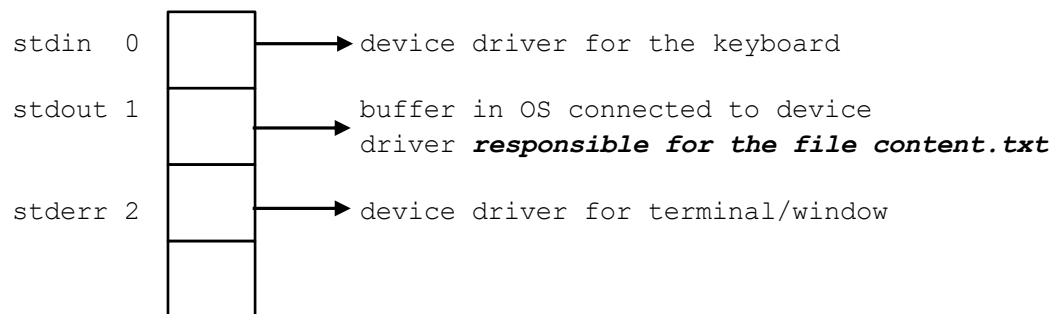
<sup>13</sup> `ls` is the program executed to list the contents of a directory in UNIX/LINUX

<sup>14</sup> `dir` is the command/program executed to list the contents of a directory in Windows



What actually happens in the above examples is that the device driver associated with `stdout` is reset in the process in which the program (`ls/dir`) is executed:

filedescriptor table for the process executing `ls/dir`

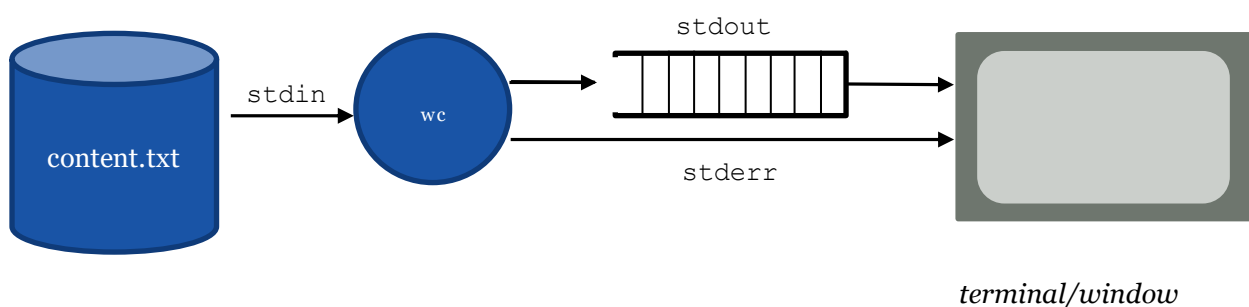


#### 10.4 Redirecting input

To redirect the input to a process one uses `<`. The commands to redirect input to the process executing program to be read from a file are:

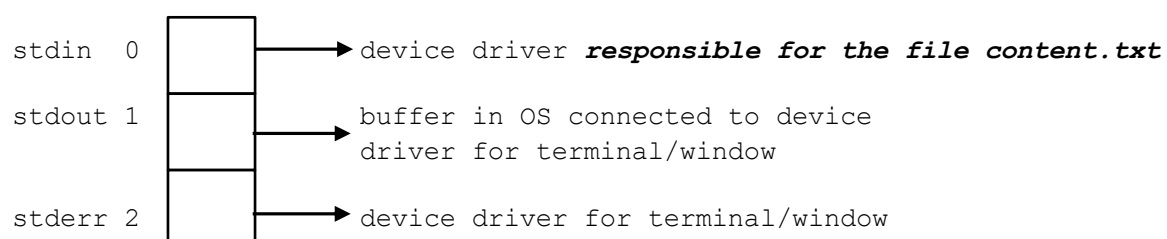
UNIX/LINUX	Windows
<code>program &lt; file</code>  <i>example: set the input for a process counting chars etc. to come from the file <code>content.txt</code></i>  <code>wc &lt; content.txt</code>	<code>program &lt; file</code>

We can illustrate this as:



What actually happens in the above example is that the device driver associated with `stdin` is reset in the process in which the program (`wc`) is executed:

filedescriptor table for the process executing `wc`



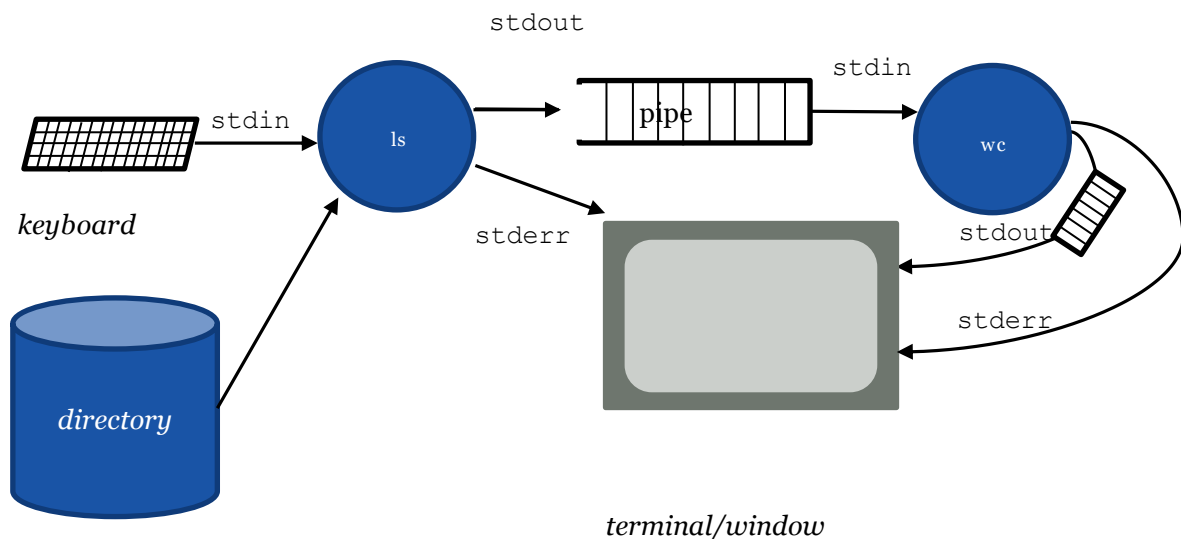
Of course it is possible to redirect both the input and output (`stdin` and `stdout`) for one process.

## 10.5 Piping: Redirecting the output from one process to be the input of another process

It is also possible to redirect the output from one process to be the input of another process. This is done by `|`. What actually happens in this case is that the output (`stdout`) from the first process is redirected to a small buffer in the operating system called a pipe, while the input (`stdin`) for the second process is redirected to read from the pipe

UNIX/LINUX
<pre>process1   process2</pre>
example: set the output from a process listing the content of a directory to be the input of another process counting characters, words and lines
<pre>ls   wc</pre>

We can illustrate the effect of this as (a *directory* is a special type of file found in the file system which *ls* reads and displays in a user friendly format):



## 10.6 An example how to pipe output/input between processes

When logged on to a UNIX/LINUX computer your environment is set up by a number of *environment variables*<sup>15</sup>. The environment variables determine things such as which directory is your home directory (`HOME`) which is the directory you will be attached to when logging on to the machine, which directories to search<sup>16</sup> when you issue a command (`PATH`) or what pager (`PAGER`) to display pages in.

<sup>15</sup> On a Windows machine the corresponding information is found in the “registry”

<sup>16</sup> Many (most) commands that you issue such as `ls`, `cd` etc. are programs that will be executed. `PATH` contains a list of directories that are searched to find the executable (such as the executables named `ls`, `cd`) when you issue the command.

If you want to inspect your environment variables you can execute:

```
printenv
```

This will result in a long listing of your environment variables. The listing is not ordered which makes it less useful. To order the listing in alphabetic order we can send the output from `printenv` to `sort`.

```
printenv | sort
```

The listing will now be sorted but may still be too long to fit in one page. To have one page at a time displayed we can send the output from `sort` to a pager such as `less` (to exit `less` press the 'q'-key).

```
printenv | sort | less
```

`printenv`, `sort` and `less` are examples of programs called *filters*. A filter reads input from `stdin` until an end-of-file marker (EOF) is found, manipulates what it has read and outputs the result to `stdout`. Well-designed filters which solve a well-defined problem (i.e. a single thing) efficiently, are powerful and can be combined to solve more complex problems by piping.

## 11 Background: Some useful commands/things in UNIX/LINUX

Working in a UNIX/LINUX environment has some advantages. Such as UNIX/LINUX systems having command interpreters capable of interpreting a programming language which makes it possible to issue complex commands and they have an excellent built-in manual.

### 11.1 man pages

The built-in manual is often referred to as “*man pages*”. In the man pages you can find detailed information on user commands, system calls and library functions for different languages. For a library function you will get information on things such as: parameters and return values, which header files to include, possible problems associated with the function, related functions etc.

The man pages are divided into several sections. Section 1 contains built-in and user commands implemented by the shell, Section 2 contains the system calls and Section 3 contains library functions for different programming languages.

To search for information in the man pages you give the command:

```
man -k item
```

Where *item* is the keyword you are searching for. There is a limitation to the `man` command in that you can only search for one keyword in a search.

To display the information on a specific command/function you simply give the command `man keyword` to the command interpreter, where *keyword* is the name of the command/function. `man` will display information on what it finds from the first section where it finds a match for the *keyword*. This means that if you are searching for something that exists under the same name as both a user command in Section 1 and a system call in Section 2 and possibly as a library function in Section 3, then `man` will only display the man page from Section 1.

If we for example search for the system call to send signals to a process which is named `kill` with the command `man kill`, then `man` will display information on the user command `kill` from Section 1 where it finds the first match. The Section is indicated by the section number in parenthesis after the name in the page header of the man page, i.e. `kill (1)` refers to Section 1 while `kill (2)` refers to Section 2.

To instruct `man` to display<sup>17</sup> information from a specific Section of the man pages you can specify the Section you want `man` to search in, in the command by adding the section number. The following command will search for `kill` in Section 2:

```
man 2 kill
```

Note: in some shells you would give the section number with a slightly different syntax such as `-s2` instead of only 2. To find out the exact syntax you should check the man pages for `man` with the command `man man`. While you can find man pages on-line on the web, the built-in man pages have the distinct advantage of being up-to-date and relevant for the version of the operating system and programming languages libraries installed (provided the installation has been done correctly).

## 11.2 Shells and shellscript

Other advantages of using UNIX/LINUX and its command interpreters, the shell(s), is that the shells normally can interpret a small programming language called shellscript. By using shellscript you can program your own more advanced commands, series of commands such as detailed searches, or have the shell run several instances of programs/sets of executions in experiments etc.

## 11.3 Accessing the LINUX servers at KTH Kista

The EECS school has several LINUX-servers available to students. The names of the LINUX-servers are listed here:

```
atlantis.sys.ict.kth.se
avril.sys.ict.kth.se
malavita.sys.ict.kth.se
subway.sys.ict.kth.se
```

You can access them by several tools such as `putty` or `X-Win32`. `X-Win32` is a program which allows the servers to open windows on your computer. The application is free to download and install for KTH-students from: <https://www.kth.se/student/kth-it-support/software/download>

You will need a tool to get tickets in the Kerberos authentication system to be able to access the servers. On Windows you would use a tool called Network Identity manager found in the “Kerberos for Windows” package. To get new tickets in the tool you select `Credential`, to obtain new credentials (tickets), and then give your KTH user name, `KTH.SE` as realm and then your KTH-password.

In `X-Win32` you set up a connection by setting the host (värd) to one of the names in the listing above, your identity (inloggningsnamn) as your KTH-identity, command should be: `xterm -ls` and password is your KTH-password.<sup>18</sup>

How to log on to the servers using different tools are described here: <https://intra.kth.se/it/arbetspa-distans/unix/how-to-connect-1.82881>

## 12 Background: Basic functions for I/O in C

To use the appropriate (correct) tools is important for all engineers. For a programmer it is important to be able to use an appropriate programming language to solve a specific problem. In some cases where execution speed and/or resource usage (such as memory) is important it may be an advantage to use C instead of JAVA. C has less overhead than JAVA due to a less complex run-time system and it has efficient implementations of library functions to access the services of the operating system by

---

<sup>17</sup> `man` will display the man page in an application called a `pager`. Typically it will be either `more` or `less` that is used for the `pager`, where `less` is more powerful than `more`. Which `pager` to use is determined by the environment variable `PAGER`. To exit the `pager` hit the ‘q’-key.

<sup>18</sup> The Linux servers use a distributed file system which you can access from a Windows machine also. That requires that you download and use OpenAFS.

system calls. However, the drawback of using C is that it may be harder to develop your code and you need to thoroughly understand what you are doing to avoid errors.

In this course you will find that particularly for I/O heavy applications C will have an advantage over JAVA. In many cases it is easier to program I/O in C compared to doing it in JAVA and the execution time can often be heavily reduced. Reduction of the execution time by up to a factor 10 is not unrealistic<sup>19</sup> for specific problems.

In this course you will find it useful to be able to use the following functions for I/O in C:

Function	Application
<code>getchar()</code> , <code>getc()</code>	Read and return a character from <code>stdin</code>
<code>putchar()</code> , <code>putc()</code>	Write a character to <code>stdout</code>
<code>scanf()</code>	Read formatted input from <code>stdin</code> such as numbers, strings etc.
<code>printf()</code>	Write formatted output to <code>stdout</code> such as numbers, strings etc.
<code>isalpha()</code> , ...	<code>isalpha</code> is one of a family of functions to determine the type of a character such as if the character is alphanumeric, low-case, upper-case, white space etc.

To find out the details of the above functions we recommend that you use the `man` facility on a UNIX/LINUX machine.

## 13 Background: References, pointers, addresses and memory management

References, pointers and addresses are related concepts which are important when working with data structures. References and pointers are needed if we want a method/function to be able to change the value of a variable in C or a member of an object in JAVA. It also enables us not to copy large amounts of data between methods/functions when we use arrays or objects.

### 13.1 Pointers and addresses in C

In C you may calculate the addresses of variables and functions, i.e. where in memory a variable or a function is stored. This is done by the `&` operator. By placing an `&` in front of the name of a variable or function we will calculate its address.

```
int a = 47;           // declare a variable a and initialize it to 47

&a                   // the address of the variable a
```

A variable containing the address of something is called a *pointer* in C. A pointer should always have a well specified type defining how to interpret what is found at the address the pointer holds. A pointer is declared by placing an asterisk in front of the name of a variable in the declaration.

---

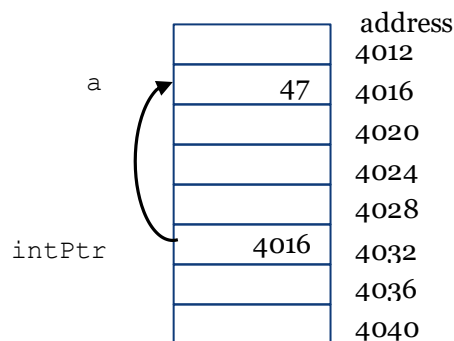
<sup>19</sup> The possibility to drastically reduce the execution time by using C is highly relevant for or anyone planning to take the "ADK"-course in the Computer Science Master

```
int a, *intPtr; // intPtr has the type "pointer to int"

intPtr = &a;    // intPtr is set to point to a,
                // i.e. intPtr will contain the address of a
```

When reading a declaration in C to determine the type of a variable it is easiest to read the declaration right-to-left (provided there are no parentheses changing the order in which to assess the type). In the above example we start by reading: *intPtr is a*, then we find an asterisk meaning, *pointer to*, and finally we see *int*. Thus we can conclude that the type is: *intPtr is a pointer to int*.

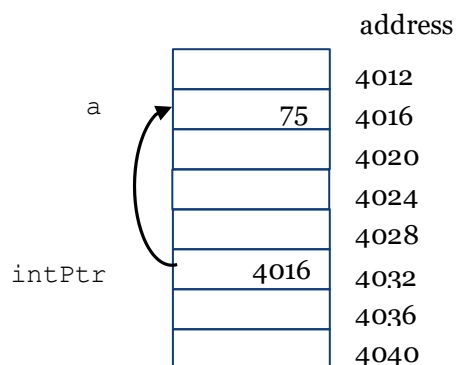
To understand how pointers work it is a good idea to try to draw a picture of what is happening in the memory. For the above example there are memory allocated for two variables: *a* which most likely is a 32 or 64<sup>20</sup> bit integer and *intPtr* which is a 32 or 64<sup>21</sup> bit pointer to *int*. In the figure below we only display addresses which are a multiple of four bytes assuming a 32 bit system (though normally we would expect to be able to address each byte individually).



Dereferencing a pointer means that we follow the pointer to go to the address it contains/points to. This is done by using the asterisk in front of a pointer variable in an executable statement (i.e. not in a declaration of a variable). The following example shows how we can use our *intPtr* to change the value stored in the variable *a*.

```
*intPtr = 75;
```

This means that we follow the pointer to the address 4016 where *a* is stored and interprets what starts at the address 4016 as an *int* and changes (i.e. writes) the value 75 to this address. Resulting in the following situation.



<sup>20</sup> This is normally determined by if we have a 32bit or 64bit processor

<sup>21</sup> This is determined by if we have a 32 bit or 64 bit operating system

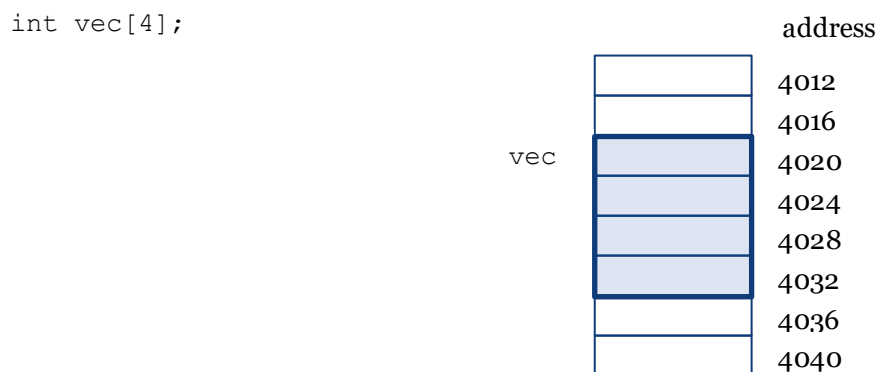
### 13.2 Memory management in C

In JAVA the run-time system will handle memory allocation/deallocation. When you instantiate an object in JAVA the underlying system will reserve (allocate) memory for the object. When the object no longer can be reached through any direct or indirect reference, the system will automatically free (deallocate) the memory by a mechanism called garbage collection<sup>22</sup>.

In C on the other hand there are no built in mechanisms that automatically allocates or deallocates (garbage collects) memory for dynamically generated objects (in fact there is no such thing as objects as defined in JAVA in C either). This means that the programmer explicitly has to manage memory allocation in many situations.

Assume we want to create an array with a number of elements which is not known a priori (before execution starts). How can we do that? Before we consider that problem we need to understand how arrays with a size known at compile-time works.

In C, memory is automatically allocated for an array which has a size which is known at compile time. The name of the array will be a constant pointer to the first element in the array. Array indices always start at zero (0) to facilitate addressing in the array. Consider the following example:



In the above example we have reserved the addresses from 4020 through 4035 to store the four elements of the array `vec`. `vec` is a static pointer (constant which cannot be changed) with the value 4020 which is the starting address of the array.

When referring to an element of the array such as `vec[2]` the C language will find this element by calculating its starting address and dereferencing that address in the following way:

`*(vec + 2 * sizeof23 (int)) -> *(4020 + 2*4) -> *(4028)`

`vec[X]` means that we should access the  $X$ th element of the array. In our case  $X$  is two, which is the third element which is located at address 4028. This address is calculated by taking the starting address of the array, 4020, and adding  $X$  times the size of the element in bytes. Note that to be able to calculate addresses in this way we have to know the type of the pointer, i.e. what type of data it points

---

<sup>22</sup> Note that when a process terminates the operating system will automatically reclaim all memory it has allowed the process to use. That is, even if you forget to deallocate memory in a process when it is executing the memory will be reclaimed when the process terminates.

<sup>23</sup> `sizeof()` is an operator in C which calculates the size of an element/type in bytes

to. JAVA and virtually any other programming language implementing arrays use address calculations as the above when indexing into arrays.

Warning: You should also note that it is possible to index out of bounds in C without any run-time system detecting this. Thus in the above example we could for example do operations such as `vec[20] = 0;` or `vec[-2] = 14;` which would overwrite memory spaces possibly used for other purposes without any run-time errors. This could lead to erroneous behavior of your program (if for instance the value of a variable has been changed un-intentionally) or that the program eventually (at a later time) crashes. Thus these kind of errors can be very hard to detect. And it is virtually impossible to detect them by printing trace messages or using debuggers. The best way to avoid such problems is to be careful and fully understand what one is doing.

### 13.3 Dynamic memory allocation in C

So what if we want to create an array where we do not know the size of the array until we execute the program? In this case we would have to allocate memory dynamically (on the heap<sup>24</sup>). Dynamic memory allocation is done by calls to the library function `malloc()`.

```
void * malloc(size_t size)
```

`malloc` will allocate a contiguous memory space of `size` bytes and return the address to where the space starts. If `malloc` could not find a free memory space of the desired `size`, i.e. there is no available free memory, it will return a `NULL` pointer. `void *` is a generic pointer type which can be used to address (point to) any valid address. In an assignment you should cast a pointer of type `void *` to the type of the pointer you assign it to.

When a memory space allocated by `malloc()` no longer is used it should be returned to the run-time system<sup>25</sup> by a call to `free()`. The `ptr` parameter to `free()` should be a pointer to an area allocated by `malloc()`. Note that `free()` does not check that this condition holds and you can call `free()` with any valid address which very likely could result in that your program no longer would function correctly.

```
void free(void * ptr)
```

---

<sup>24</sup> If you do not know what the heap is you need not worry – you will learn about that in the Operating systems course. However, a process normally has four memory areas: a **stack** which is used for activation records created when functions/methods are called, **BSS** which is used for static/global data, the **heap** used for dynamically (during run-time) allocated data and the **text** area where the instructions for the methods/functions of the program is stored.

<sup>25</sup> Think of the run-time system as a set of functions/methods used to implement different services a process needs. In this case it is the library functions to manage dynamically allocated memory, `malloc()`, `free()`, `realloc()`, which keeps track of and manages memory obtained from the operating system through system calls such as `mmap()` or `brk()` (`brk()` is old and no longer used).



Now we have all the necessary elements to be able to dynamically allocate memory for an array in C which has a size that is not known until execution time. The following code snippet shows how this can be done (note that we do not show how to free the memory when we no longer have use for the array):

```
int size, //number of elements in the array
    *dynVec; // a pointer to the allocated memory

printf("give the number of elements: ");

scanf("%d", &size); // read the number of elements and store
                    // at the address where size has been
                    // allocated

assert(size > 0); // make sure size is > 0, else throw error

//allocate memory, cast the pointer returned to type int *

dynVec = (int *)malloc(size*sizeof(int));

dynVec[0] = 14; //use dynVec as a normal array
```

### 13.4 References in JAVA

JAVA does not give the programmer the same freedom of calculating addresses and accessing any address for the reason of avoiding what could be the potential cause of errors and possible security risks.

In fully object oriented languages<sup>26</sup> there is no need to have references other than for being able to point to objects. Thus we will think of a reference in JAVA is a variable that contains the address of where an object is stored in memory. Consider the following example:

```
private class Node
{
    Item item;
    Node next; // reference variable that contains a
} //reference to a Node object

//instantiate an object from the class node
//and set first to be a reference to the new object

private Node first = new Node();
```

In the case above where we instantiate a `Node` object by `new`, the JAVA run-time system (virtual machine) will allocate memory for the new object and return a reference to the new object and store the reference (~address) to it in `first`. What goes on behind the scenes is similar to allocating memory by `malloc()` in C and then initiating the fields in the new object (`item`, `next`) to zero/`NULL` (in this case) and then returning the address to the memory allocated for the object.

---

<sup>26</sup> JAVA is not a fully object oriented language as it allows for basic data types such as `int`, `double` etc. for performance reasons where the variables of such types are not objects in an object oriented meaning.

## 14 Background: How to build linked data structures in C

In C there are no concepts of objects or classes. Instead you have to rely on something called `struct` to implement elements/nodes. A `struct` is a way of managing several fields of data as one item. The syntax is as follows:

```
struct name {  
    field variable declarations  
} declaration of structs ;
```

Items set in boldface are mandatory, items in italics are optional.

A `struct` representing a person with the name and date of birth could be declared as:

```
struct person {  
    char * name;  
    int year;  
    int month;  
    int day;  
};
```

The following would create a `struct` variable from the data type we defined above:

```
struct person kalle;
```

The fields of `kalle` can be accessed by the dot (.) operator:

```
kalle.year = 97;
```

However, in most cases structs are handled by pointers. The following declares a pointer to a person `struct` and initiates it to point to `kalle`.

```
struct person *ptr = &kalle;
```

To access the fields of the `struct kalle` that `ptr` points to, we can use two methods:

```
(*ptr).month = 12;    // dereference ptr and then select a field  
  
ptr->month = 12;       // same as the above but uses the "arrow" operator
```

When working with structs it is often convenient and also helps the readability of the code to use the possibility to create your own names for data types by using `typedef`. The following creates new names for the data types `struct person` (`aPerson`) and a pointer to a `struct person` (`personPtr`).

```
typedef struct person {
    char * name;
    int year;
    int month;
    int day;
} aPerson, *personPtr;
```

That is, if we had had the above type definitions we could have used them to declare `kalle` and `ptr` as:

```
aPerson kalle;           // same as struct person kalle;
personPtr ptr = &kalle;  // same as struct person *ptr = &kalle;
```

If we want to dynamically create structures, for instance when creating a linked list, we have to dynamically allocate memory for the `structs`. The following code snippet defines a `struct` and associated names for `struct` elements and pointers suitable for implementing a single linked list and allocates memory for a new element, updates the fields in the element and then deallocates the element:

```
typedef struct x {
    int data;
    struct x *next;
} elem, *elemPtr;

elemPtr ny = (elemPtr) malloc(sizeof(elem)); //note: the memory space
                                              //allocated by malloc may
                                              //contain old data, i.e. you
                                              //cannot assume all
                                              //fields will be 0 (NULL)
                                              //(unless you initiate them)

ny->data = 4;

ny->next = NULL;

free((void *) ny);
```

## 15 Background: Common pitfalls when programming C

There are some common errors that inexperienced (and sometimes also experienced) C programmers may encounter. Below you find some of these.

### 15.1 Test for equality in the condition of `if`, `for` or `while` statements

In C all statements have a value which can be used for assignments but also for tests on whether or not it evaluates to TRUE/FALSE. In C a value of zero (0) evaluates to FALSE.

An error which may occur is that one confuses test for equality which is written by two equality signs `==`, with assignment which is a single equality sign. Assume we want to execute a `while`-loop as long as an integer variable `x` has the value 4962, *i.e. the condition has the form of a comparison of a variable and a constant*. We naïvely write the `while`-statement as:

```
while (x = 4962) ... //no error from the compiler as this is allowed in C
```

What we have created in the above example is an infinite loop (i.e. the loop will never terminate) as we assign the value 4962 to the variable `x` in the condition part of the `while` statement before we check the condition in each iteration of the loop. We then test to see if `x` is zero, in which case we break the loop, or in case it is non-zero we will execute the statement in the loop. As the variable `x` is always assigned the value 4962 before we check the condition, the value of `x` that we check in the condition, will always be non-zero and the loop will never terminate.

A simple way to avoid this is to always place the variable to the right of the equality signs and the constant to the left. Had we made the same mistake as above the compiler would have recognized it and issued an error telling us that we cannot assign a new value to a constant. This is often somewhat cryptically expressed as that we have: “an illegal lvalue”. In C-parlance an `lvalue` is the value to the left in an assignment and an `rvalue` is the value to the right.

```
while (4962 = x) ... //compilation error as we cannot assign to a constant
```

This error message would alert us so that we could correct the code to:

```
while (4962 == x) ...
```

## 15.2 Confusing bit-wise and logical operators

As C was designed also to cover low-level programming controlling hardware, it has not only logical but also bitwise operators. This may lead to hard to understand results if one confuses the bitwise operators for *or* which is written as a single vertical bar (|) and the bitwise *and* operator which is written as a single ampersand (&) with the logical *or* written as double vertical bars (||) and logical *and* written as double ampersands (&&). Consider the following example:

```
int x = 1, y = 2;

if(x && y) statement // statement will be executed as both x and y are
                      // true (non-zero)

if(x & y) statement  // statement will not be executed as bitwise AND of
                      // 1 & 2 is zero

x    0 1
y    & 1 0 bitwise and
-----
      0 0
```

## 15.3 Indexing out of bounds and addressing errors

C allows you to access basically any valid memory location in the (virtual) address space of the process in which your program executes. This is to allow C-code to control hardware and enable it to be used for implementing operating systems etc. Thus most C-compilers assume that you know what you are doing and allows for code to pass the compilation without error or warning messages that would occur in many other languages. There are also few run-time controls checking for instance if you index out-of-bounds of an array and overwrite things. That is if you are not careful you may overwrite data in your program, which in general only will result in faulty behavior. But you may also overwrite things vital for the execution of the program, such as the execution stack, which in most cases would cause your program to crash. In both these circumstances the effects of the overwriting, in the form of execution errors or a crash, may occur much stage in the execution which often means that it may be hard to find the cause of the problem. So how do you avoid such problems?

There are tools which can check your code before you compile it and issue warnings when it finds parts of the code which it suspects could be erroneous and there are other tools which may allow for more extensive run-time tests including searching for memory leaks. It may also be useful to know how to use a debugger.

***However, the best way to avoid such problems (and most problems encountered when programming in any language) is to truly understand what you are doing.***

## 16 Background: The best pieces of advice I can give!

When writing programs solving more complex problems or designing more elaborate data structures, such as linked data structures, it is a bad idea to immediately start coding and experimenting with the code to see what happens. In any other area of engineering it would be totally unconceivable not to start by creating a design or a blueprint for how to solve the problem. We should also adhere to this best-practice and use a more structured approach for our problem solving:

1. Make sure you understand the problem properly.
2. Try to break-down the problem into manageable pieces, i.e. pieces which you can solve and use to build the complete solution from
3. Describe your design in some kind of pseudo-code, draw pictures of your data structures and what the methods/functions operating on them should result in.
4. Design tests to test your solution/design<sup>27</sup>.
5. When you understand the problem and how you could solve it – only then should you start to finalize your break-down of the problem into well-defined classes and methods in JAVA and/or functions and datatypes (such as `structs`) in C.

Each method/function should preferably solve a single, well-defined problem efficiently. This will make it easier to check that the methods/functions are correct. Once you know that the building-blocks are correct, you can use them to build more complex functionality from.

6. Once you have a solution on paper which is simple and clean – only then and not earlier should you try to implement it in code.
7. A good advice, both when designing and coding, is to always strive for simplicity. If your design/code becomes too complex you should redesign your solution to simplify it. While most of us are hesitant to discard the work we have put into something, the fact is that trying to add even more complexity (code) to solve the problem in most cases simply adds more errors and makes the solution even harder to understand<sup>28</sup>.

Spending a lot of time on understanding the problem and designing solutions on paper may seem like a waste of time! And why delay the fun of coding and executing the code to see how (if) it works? The simple answer to this is:

***You will save lots of time and create better, more professional solutions by spending more (often much more!) time on understanding the problems and properly designing your solutions as compared to simply spending time mainly on coding.***

***And creating good and correct solutions is what being an engineer is about!***

---

<sup>27</sup> If you cannot design tests for your solution to the problem you probably do not understand your solution well enough. I.e. then you need to work more with your solution.

<sup>28</sup> To simply keep adding code to something that does not work or which one does not understand is a common mistake made by many novices to programming. Such an approach actually only makes the task of learning how to program unnecessarily difficult.

## 17 Index

&, 20  
\*, 20  
|, 28  
||, 28  
addresses, 19  
Agile, 3  
assert, 24  
asterisk, 20  
binary tree, 10  
bitwise operators, 28  
buffer, 13  
command interpreter, 14  
Comments, 2  
declaration, 20  
dereference, 25  
Dereferencing, 20  
device driver, 15  
double linked circular list, 8  
Dynamic memory allocation, 22  
EMACS, 5  
environment variables, 16  
EOF, 5  
file descriptors, 13  
filter, 17  
free(), 22  
garbage collection, 21  
getc(), 19  
getchar(), 5, 19  
GNU, 5  
head, 8  
HOME, 16  
home directory, 16  
Indexing out of bounds, 28  
infinite loop, 27  
*infix*, 11  
input, 14, 15  
instantiate, 24  
isalpha(), 19  
**Javadoc**, 2  
Kerberos, 18  
less, 17  
linked data structures, 6  
LINUX-servers, 18  
lvalue, 27  
malloc(), 22  
man pages, 17  
memory allocation, 21  
Network Identity manager, 18  
ordered tree, 10  
output, 14  
PAGER, 16  
PATH, 16  
pipe, 16  
*pointer*, 20  
pointers, 19  
*postfix*, 11  
*prefix*, 11  
printenv, 17  
*printf()*, 19  
priority queue, 8  
putc(), 19  
putchar(), 5, 19  
recursive methods, 10  
redirect, 12  
redirecting input, 14  
references, 24  
References, 19  
rvalue, 27  
scanf(), 19  
sections, 17  
shell, 14, 18  
shellscrip, 18  
single linked linear list, 6  
sizeof(), 22  
sort, 17  
stderr, 13  
stdin, 5, 12, 19  
stdout, 5, 12  
streams, 12  
struct, 25  
system calls, 13  
Test Driven Development, 3  
Test for equality, 27  
tests, 3  
typedef, 26  
*validation*, 3  
*verification*, 3  
void \*, 22  
X-Win32, 18