Introduction to Numerical Methods
SS 23 Lecture Notes

Igor Dimitrov

May 03, 2023

CONTENTS

I Programming Tutorial 3

1 Introduction 5

2 Unix 7
- 2.1 Filesystem . . . 7
- 2.2 Shell . . . 8
- 2.3 I/O Streams . . . 11

3 Version Control with Git 15
- 3.1 Getting Started . . . 15
- 3.2 Fundamental Git Commands and Terms . . . 17

4 Hello World 21
- 4.1 Cing Config . . . 22
- 4.2 Name Collisions and Namespaces . . . 23

5 Variables 25
- 5.1 Basic Data Tytpes in C++ . . . 25
- 5.2 Variable Declarations . . . 26
- 5.3 Statements and Expressions . . . 26
- 5.4 Blocks and Scope . . . 29
- 5.5 Expressions and Operators . . . 30

6 Arrays 35

7 Input and Output 37

8 Flow of Control 39
- 8.1 If-Statement . . . 39
- 8.2 While Loops . . . 40
- 8.3 For Loops . . . 43
- 8.4 Numerical Solution of the Pendulum Problem . . . 44

9 Functions 47
- 9.1 Recursion . . . 48
- 9.2 Iteration . . . 49

10 Templates 51

11 Classes 53

II Numerical Methods 55

12 Introduction 57

Index 59

Introduction to Numerical Methods Uni Heidelberg summer semester 23 Lecture Notes.

Part I

Programming Tutorial

CHAPTER

ONE

INTRODUCTION

C++ programming tutorial notes

Some links:

- hedgedoc link
- git repo for hdNUM likn
- c++ reference cppreference.com

CHAPTER

TWO

UNIX

Unix was initially released in 1969. Many Unix-based operating systems emerged:

- **Free-**, **Net-**, and **OpenBSD** based on UC Berkeley's version of Unix.
- **macOS**, partially based on FreeBSD and NetBSD
- **iOS**, based on macOS
- **illumos/OpenIndiana**, based on Solaris and System V
- **Linux (1991)**, stricly speaking not a Unix, but compatible at most
- **Android**, a strongly modified Linux developed by Google.
- many various **embedded systems**

Some facts:

- 95% of 500 fastest computers run Unix
- Most web servers run Unix
- Most scientific communities use Unix/Linux.

Most software installed under Linux is open source, i.e. it's source code is free to insepct and learn from.

2.1 Filesystem

Each use has a **home directory**, usually under `/home/<username>`. Users don't have write priviliges outside of their home directory. The administrator is called **root** in unix and can write everywhere. Each program has a **working directory**. File access is always relative to this working directory. Working directory can be switched.

Linux provides usual GUI for interacting with the system but the **shell** still has advantages:

- automating repetitive tasks
- many taks can be done faster than with GUI
- many Unix programs don't have a GUI.

2.2 Shell

2.2.1 Commands

General command syntax:

```
$ cmd -sv --opt1 --opt2 arg1 arg2
```

The behaviour of a command can be changed by providing it **options**. Options are provided either beginning with a dash `-` or double dash `--`. Single-dash is followed by short options names consisting of a single letter. Short options can be combined by directly combining the letters: `-sv` above stands for `-s` and `-v`.

Some basic commands:

`pwd` outputs the full path of the current working directory:

```
pwd
```

```
/home/igor/Documents/uni/ss23/nummethSS23/docs/text/progtut
```

`exit` closes the current shell

`echo` prints out its argumetns to standard ouput:

```
echo hey there
```

```
hey there
```

If a command is used wrongly, it usually prints a short message help message to the standard output:

```
# rm
```

The option `--help` usually prints a succint help message to the standart output

```
rm --help
```

More comprehensive help can be optained with the `man` command, that opens the manual entry of a command within the **man pages**

```
man rm
```

`cd` to change directories:

```
cd ../../imgs
```

```
pwd
```

```
/home/igor/Documents/uni/ss23/nummethSS23/docs/imgs
```

- `..` stands for directory one level above
- `-` previous directory
- `.` this directory

```
pwd
cd ..
pwd
cd -
pwd
cd .
pwd
```

```
/home/igor/Documents/uni/ss23/nummethSS23/docs/imgs
/home/igor/Documents/uni/ss23/nummethSS23/docs
/home/igor/Documents/uni/ss23/nummethSS23/docs/imgs
/home/igor/Documents/uni/ss23/nummethSS23/docs/imgs
/home/igor/Documents/uni/ss23/nummethSS23/docs/imgs
```

- `ls` to list files in the directory.
- show files beginning with `.` with `ls -l`
- `ls` accepts a path as argument
- `ls -l` shows additional infomration. `-l` stands for **long**.

```
ls
```

```
deltas2.excalidraw.png          gitsnapshots.excalidraw.png
expression_cpp.excalidraw.png   pend2.png
expression.excalidraw.png       pend.data.png
funct.excalidraw.png            pendulum.excalidraw.png
```

```
ls -l
```

```
total 368
-rw-r--r-- 1 igor igor 90137  3. Mai 17:59 deltas2.excalidraw.png
-rw-r--r-- 1 igor igor 22263  3. Mai 17:47 expression_cpp.excalidraw.png
-rw-r--r-- 1 igor igor 15359  3. Mai 17:48 expression.excalidraw.png
-rw-r--r-- 1 igor igor 15990 28. Apr 12:51 funct.excalidraw.png
-rw-r--r-- 1 igor igor 88981  3. Mai 17:48 gitsnapshots.excalidraw.png
-rw-r--r-- 1 igor igor 44084 28. Apr 12:10 pend2.png
-rw-r--r-- 1 igor igor 50463 26. Apr 18:25 pend.data.png
-rw-r--r-- 1 igor igor 34042 26. Apr 16:54 pendulum.excalidraw.png
```

```
ls ~/Documents
```

```
 01.html
 2022-09-25.pdf
 2022-09-26.md
 2022-09-26.pdf
 220817_0001E7094BA6933C1EED87C1DC4BB8978778.pdf
 220902_ZTBM005056A262E41EDD8ADB804F91C5DD3E.pdf
 article-spotter-software-description.pdf
 bobo-yt
 bookmarks_6_18_22.html
 Books
 hamster-exports
 haskell-trial
 hello_files
 hello.html
 hello.pdf
```

(continues on next page)

(continued from previous page)

```
 hello.qmd
 IDB
 igor.diary
 igor.diary.~previousversion~
 IgorDimitrov_G429055818_Sepa_lastschriftMandat.pdf
 index.html
 jupbookexamples
 jupyter-notebooks
 learning
 learning_bash
 learning-haskell
 learning_prog
 learning_sage
 learn_prog_cpp
 MasonicInfluenceOnBrainTransformation.doc
 Mathematics.md
 MATLAB
 mindforger-repository
 ML,-AI---Data-Sci-Learning-Resources.html
'ML, AI & Data Sci Learning Resources.md'
 newdir
'New Document.rnote'
'Notable - Export (5950)'
'Notable - Export (DB2C)'
'Notable - Export (E58D)'
 notes
 Notes
 number-systems.pdf
 obsidian-home
 pascal-programs
 Personal_Home.pdf
 pockettube_subscriptions_list_2022-09-29-16_18.csv
 portfolio
'pyrex-auflaufform-2er-set-mit-deckel (1).jpg'
 quantecon-mini-example
 quarto-trial
 quick
 rnote
 Rplots.pdf
 sample.yaml
 SimpleDiary
 some_folder
 some_programs
 some.tex
 sqltrial02.pdf
 sqltrial.pdf
 starting-vscode
 Systematic_Programming_an_Introduction.bibtex
 TheNoteBook
 The_UNIX_System.bibtex
 TmForever
 trial.html
 trial.md
 uni
'Untitled Folder'
'Wolfram Mathematica'
'Zettlr Tutorial'
```

```
cd /home/igor/Documents/uni/ss23/nummethSS23/docs/text/progtut/testdir
pwd
```

```
/home/igor/Documents/uni/ss23/nummethSS23/docs/text/progtut/testdir
```

`cp` to copy files

```
echo hey >file.txt
ls
cp file.txt copy.txt
cat file.txt copy.txt
```

```
A  B  C  file.txt
hey
hey
```

`mkdir` creates directories

```
mkdir subdir
mv copy.txt subdir
ls subdir
```

```
copy.txt
```

`rm -r` remove directory and all of its contents with the **recursive** option `-r`:

```
rm -r subdir
```

```
ls
```

```
A  B  C  file.txt
```

search files for a pattern with `grep`

```
grep hey file.txt
```

```
hey
```

2.3 I/O Streams

Unix programs commpunicate with the system with **I/O Streams**. Streams are unidirectional, they can be either written to, or they can be read from.

Every program has three open streams when its starts executing:

- **stdin**: The standard input reads user input from the console and is connected to the **file descriptor 0**
- **stdout**: Normal results of a program are outputed to the standard output. Standard output is connected to the **file descriptor 1**
- **stderr**: Diagnostic messages and errors are output to the standard error, which is connected to the **file descriptor 2**.

2.3.1 Redirection of I/O Streams

Redireaction of I/O streams is one of the most powerful concepts in Unix. Be default the standard streams are connected to the terminal but sometimes it is sensible to redirect the stream to files:

Let's create a file "A" with the content "hey"

```
echo hey >A
cat A
```

```
hey
```

The direcctory now contains a single file "A"

```
ls
```

```
A
```

we can redirect the output of `ls` to another file called 'B'

```
ls >B
```

let's see the contents of the folder now;

```
ls
```

```
A  B
```

and the contents of the file B:

```
cat B
```

```
A
B
```

maybe somewhat unexpectedly for the novice, the file B contains its own name. The reason is that during the redirac-tion step we typed the command:

```
ls >B
```

Before `ls` is executed by the shell, the shell processes the whole line, and therefore creates the file B due to the `>B` part of the command. Thus before `ls` is executed B is already contained in the directory, therefore its own name along with A is written to it.

Redirecting **stdin** is achieved with <. Thus redirecting input to come from the file "B", effectively reading it is done the following way:

```
cat <B
```

```
A
B
```

stderr is redirected via `2>`. To output the standard error message to a file named "C" we do:

```
ls nonexistent 2> C
```

```
ls
```

```
A  B  C
```

```
cat C
```

```
ls: cannot access 'nonexistent': No such file or directory
```

CHAPTER

THREE

VERSION CONTROL WITH GIT

Instead of saving different versions of files under different names, version control can be elegantly achieved with dedicated version control systems like **git**.

Git is a **distributed version control** system, that allows to track changes of any set of files, compare different versions, create branches for different development versions, and to coordinate collaborative manipulation of the files, among other capabilities.

It was originally developed by **Linus Torvalds** in 2005, to be used for the development of the Linux kernel.

Since git is a **distributed version control system** as opposed to a **clint-server version control system**, every Git directory on every computer is a full-fledged **repository** with complete history and full version-tracking capabilities, independent of network access or a central server.

Repository Database with all the information and data that git uses to carry out the version control. Saved under `.git` folder at the root of the project folder

Commit A global snapshot of all tracked files, a long with a commit message that describes what has been changed. Commit is a fundamental versioning command; user executes it, when he wants to record the current version of the files.

Branch A self-contaied sequence of commits, accessible under a branch name, that are parallel to the main development/version branch. A branch represents a prallel development version

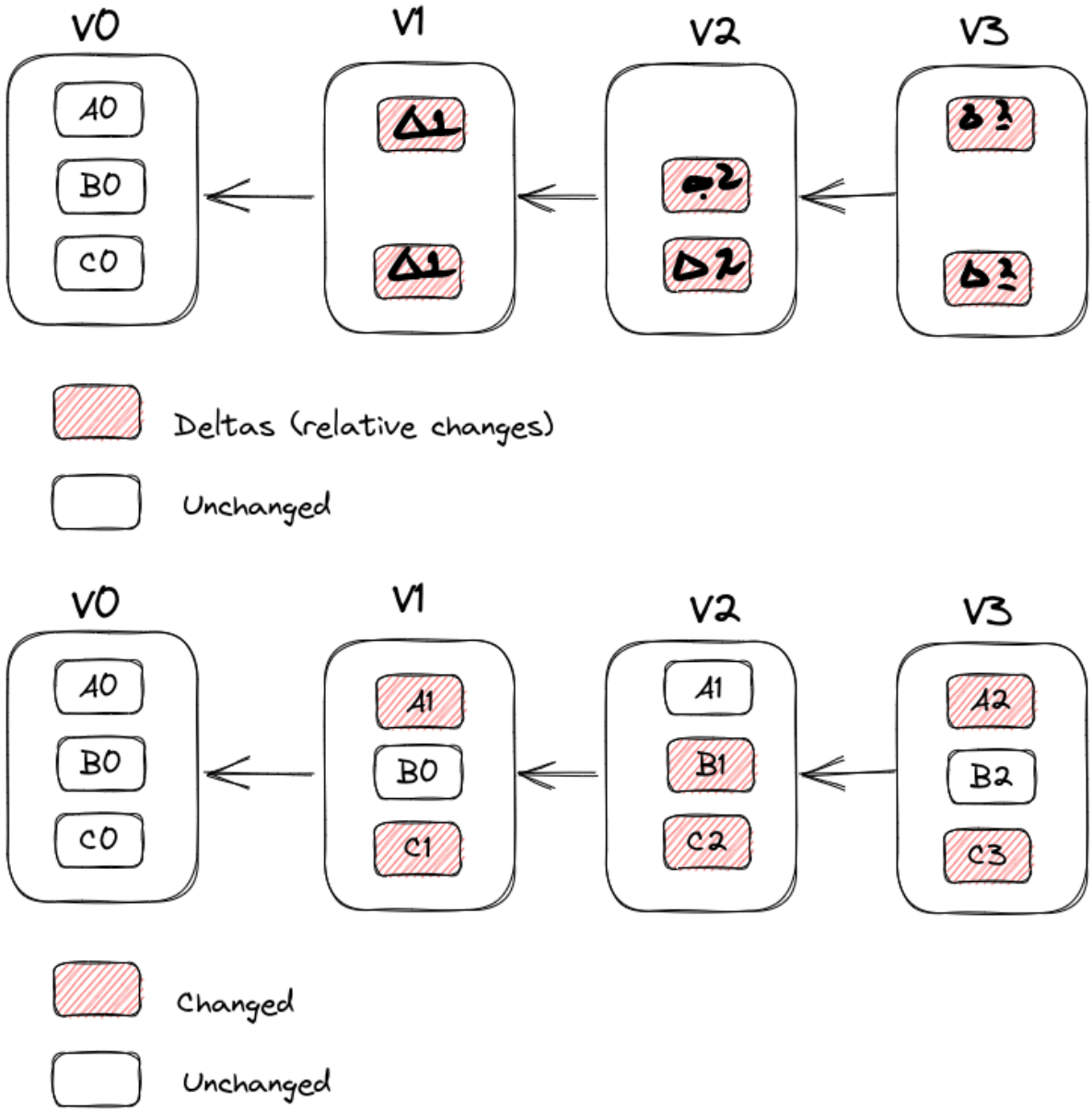
Tag A permanenet name for a commit - e.g. a release name.

3.1 Getting Started

3.1.1 Some Core Asepcts of Git

Snapshots vs Deltas

Most other version control systems save differences with delats - file based relative changes. Git on the other hand saves complete snaphot of the whole filesystem, but to save space unchnaged files aren't saved again, but rather links to the previous unchanged version.



Most Operations are Local

most operations like

- browsing history
- viewing changes between versions of files (diffs)
- commiting
- checking out a branch
- …

are all local and therefore very fast.

Checksumming

All files stored by git are checksummed. Since this functionality is built-in on lowest level of git, it is usually impossible to lose information. The checksumming mechanism is **SHA-1 hash** - a 40 character string calculated based on files or a directories contents. Entities are stored as checksum strings in git database, not by their file names. Git also generally only adds data, therefore it is usually impossible to lose information, enables free experimenation with the source code.

To querry the **state** of the different files in your repository use `git status`

3.2 Fundamental Git Commands and Terms

One of the most fundamental notions to any VCS, and not just git, is the **working directory/tree**. It is the collection of concrete version of folders and files that are currently checked out and are being edited.

Working Directory The collection of files and folders, that are at a specific version, that are currently checked-out and being editted/processed by the user.

Another component of the VCS's is the **database**. After files and folders have been eddited and modified, and the programmer wants to save that as a new **version**, the programmer **commits** this current version to the VCS database. Different VCS have different approaches in storing these changes, e.g. deltas, snaphots.

The VCS uses the databases to create versions of the folders and files that were commited by the programmer. Populating the working directory with a certain version is called **checking out** that version.

Thus basically in usual VSC there are two "areas", the working directory and the database.

But git adds a **third area** called the **staging area** or **index**. After programmer edits and modifies files, the programmer must add files explicitely to the index, in order that these files get commited in the next commit. The **unstaged** files, even if modified will **not** be commited.

Index / Staging Area The collection of files that are marked to be **committed** in the next commit

Unstaged / Modified files Files that are modified but are not explicitely added to the index.

The ability to explicitly decide what will be committed in the next commit gives the programmer more flexibility, e.g. some modified files might be ommitted from the current commit, and be commited on some other commit.

Let's demonstrate these concepts, by first initialize a repository with `git init`.

```
git init
```

This creates a subdirecctory named `.git`, which is essentially a database containing all files necessary for git to carry out version control.

Currently the directory is empty. Let's add some files to the directory:

```
echo "I am A" >A
echo "I am B" >B
ls
```

```
A  B
```

Although these files are in the directory, git doesn't track them, i.e. they haven't been added to the .git database yet. Such files are called **untracked**

Untracked Files that are not in the `.git` database, i.e. files that git doesn't track.

Let's verify that these files are indeed untracked with the `git status` command:

```
git status
```

As suggested by the help message, we can add these files to the database with `git add <filename>`:

```
git add A B
git status
```

Now these files are tracked, but more specifically they are **staged**. When untracked files are first added to the DB they are also staged.

Staged staged files are files that will be commited on the next commit

Let's execute our first commit with the `git commit -m <message>` command

```
git commit -m 'first commit'
```

and check the status

```
git status
```

Note the **working tree clean** message output. The working tree is clean when there are no unstaged modified files.

Let's modify files and not yet stage them:

```
echo "adding new lines to A" >>A
```

Check the status:

```
git status
```

Let's modify B and add it to the staging area. (But not add A yet):

```
echo "adding a new line to B" >>B
git add B
git status
```

We have added the modified B file to the staging area with the `git add <filename>` command. A is modified but still not have been added to the staging area/index and will not go in the next commit.

Now let's perform this commit, that will record the new version of B:

```
git commit -m "changing B"
```

Now the status command shows:

```
git status
```

We can use the short version of the command with `git status -s`

```
git status -s
```

```
 M A
```

3.2.1 Tracke

3.2.2 The Three main States

One of the most fundamental concepts to understand git is the three main states that a tracked file can be in:

1. **Mo

CHAPTER

FOUR

HELLO WORLD

C++ is suitable both as a high-level application and software developed language, as well as as low-level systems programming language.

Various programming paradigms are supported, including:

- imperative,
- object oriented
- generic

Goals of the language are efficiency, performance and fexibility. Applied in various fields ranging from embedded controllers to super computers. Allows direct access to hardware resources.

some additional mottos of the language:

- zero-cost abstractions
- pay only for what you see

Libraries are included with the `#include` directive.

- `<iostream>`: standard library for standard input/output (keyboard and screen)
- `<vector>`: container library for dynamic arrays, i.e. arrays whose size changes during run-time. Implemented as amortized arrays (rather than linked lists, contrary to what one would naively assume).
- `"hdnum.hh"`: header file to include the custom `hdnum` library.

Note: standard libraries are enclosed with angle brackets, progammer-defined header files are enclosed with double quotes

---

```
#include <iostream>
#include <vector>
#include "../../../hdnum/hdnum.hh"
```

and print out some messages to standard outpout:

```
std::cout << "hello" << std::endl;
std::cout << "1 + 1 = " << 1 +  1 << std::endl;
```

```
hello
1 + 1 = 2
```

```
#include <string>

void print(std::string msg)
{
```

(continues on next page)

(continued from previous page)

```
    std::cout << msg << std::endl;
}

std::string greeting = "hello, world";
print(greeting);
```

```
hello, world
```

Full program text:

```
#include <iostream>
#include <vector>
#include "../../../hdnum/hdnum.hh"

int main()
{
    std::cout << "hello" << std::endl;
    std::cout << "1 + 1" << 1 + 1 << std;endl;

    return 0;
}
```

To compile a single program called `hello.cpp` enter the command:

```
$ g++ -o hello -I../../../ hello.cc
```

where `-I../../` is the include option indicating where the header files are located. In this case the headers are located in a folder **three levels above** the folder where `hello.cc` is located and is being compiled.

To run the compiled program enter `./hello` at the CLI

4.1 Cing Config

To include the `hdnum` library in the jupyter notebook cling environment, following must be executed at the beginning of a notebook:

```
#pragma cling add_include_path("/home/igor/Documents/uni/ss23/nummethSS23/hdnum/")
#include <iostream>
#include <cmath>
#include <complex>


#include "src/densematrix.hh"
#include "src/exceptions.hh"
#include "src/lr.hh"
#include "src/newton.hh"
#include "src/ode.hh"
#include "src/opcounter.hh"
#include "src/pde.hh"
#include "src/precision.hh"
#include "src/qr.hh"
#include "src/rungekutta.hh"
#include "src/sgrid.hh"
#include "src/timer.hh"
#include "src/vector.hh"
```

4.2 Name Collisions and Namespaces

Assume Alice and Bob have written libraries, `alice.hh` and `bob.hh` respectively, that both contain the function `greeting()`. Following will generete a compiler error, due to compiler not knowing which `greeting()` function to accept

```
#include <alice.hh>
#include <bob.hh>

int main(int argc, char** argv)
{
    greeting(); //compiler error!
}
```

This problem is reffered to as **name collision** and is resolved by using **namespaces** the following way:

```
#include <alice >
#include <bob>

int main(int argc, char** argv)
{
    alice::greeting();
    bob::greeting();
}
```

i.e. with the syntax `libraryname::libraryentity`.

Note: namespaces can be nested:

```
library::sublibrary::function()
```

Note: C++ delivers a **standard library** as a part of the compiler. Standard library is accessed with the namespace `std`:

```
std::cout <<"hey" <<std::endl;
```

Namespaces can be declared manually in source code as well:

```
namespace bob {
    void greeting()
    {
        std::cout << "hey\n";
    }
} // end namespace bob
```

```
bob::greeting();
```

```
hey
```

CHAPTER

FIVE

VARIABLES

A **domain** is a range of possible values exhibiting some uniform pattern. Objects belonging to the same domain have the same **type**. An object x belonging to a domain D this is written mathematically as:

$$x \in D \quad \text{(Set theoretical notation)}$$

or equivalently as

$$x : D \quad \text{(Type theoretical notation)}$$

In C++ we have:

```
D x;
```

This is called the **declaration** of **variable** x of the **data type** D.

A (mathematical) domain can be finite, countably infinite, or uncountanbly infinite.

Note: A **mathematical structure** is a domain equipped with distinguished **special elements**, **relations** and **operations**. Relations and operations are usually binary.

Example: Domain of integeres $\mathbb{Z}$ along with the special elements 0 and 1, order relation $<$, arithmetic operations $+, \times$.

$\mathbb{Z}$ **countably infinite**.

---

computers are finite, and physical components (the arithmetic and logic unit) performing the operations are capable of holding finitely many distinct representations of objects.

Therefore the mathematical structures realized by computer hardware are not $\langle \mathbb{Z}, 0, 1, +, \times$ and $\langle \mathbb{R}, 0, 1, +, \times \rangle$ but their finite approximations $\langle \tilde{\mathbb{Z}}, 0, 1, \oplus, \otimes \rangle$ and $\langle \mathbb{F}, 0, 1, \oplus, \otimes \rangle$, where the basic axioms are not always satisfied.

5.1 Basic Data Tytpes in C++

basic(atomic) data types of a programming langauge are the types directly provided by the language, as opposed to the data types defined by the programmer using the mechanisms of the langauge.

Some high-level programming langauges provide basic numerical data types that correspond to the ideal mathematical types $\mathbb{R}$ and $\mathbb{Z}$. But C++ provides only low level basic data types that are directly represented by the computer and directly operated on by the ALU. This data types are `int`, `float` and `double` correspong to $\tilde{\mathbb{Z}}$ and $\mathbb{F}$, and operated on by the floating point unit and integer unit, respectively.

Type	Range	Implements	Represents
`int`	[-2^31^, 2^31^-1]	IEEE int	$\mathbb{Z}$
`unsigned int`	[0, 2^32^ - 1]	-	$\mathbb{N}$
`float`	[-3.4e38, 3.4e38]	IEEE float	$\mathbb{R}$
`double`	[-1.80e+308, 1.80e+308]	IEEE double	$\mathbb{R}$
`char`	ASCII characters	ASCII characters	letters and others
`string`	strings of ASCII	-	-

Table: List of Basic Data Types in C++

5.2 Variable Declarations

variables can be declared unitialized. Usually some default value like 0 is assigned to them.

```
unsigned int i;
std::cout << i << std::endl;
```

```
0
```

or initialized

```
double x(3.14);
float y(1.0);
short j(3);
std::cout << x << " " << y << " " << j << std::endl;
```

```
3.14 1 3
```

full program text:

```
#include <iostream>

int main()
{
    unsigned int i;
    double x(3.14);
    float y(1.0);
    short j(3);
    std::cout << x << " " << y << " " << j << std::endl;

}
```

5.3 Statements and Expressions

5.3.1 Statements

An object[1] of a certain type can take different values during its existense. This transformation of values is called **change of state** of the object.

Computers can change the sate of an object residing in memory. This is called an **action**. The **instructions** to peform actions in a given programming language are called **statements**.

[1] Here we use the word "object" not in the object-oriented sense, but in it most general sense; simply standing for any concrete sequence of bits (corresponding to some variable of a certain type) residing in memory.

The change of a variable's state as specified by a statement is said to be the **effect** of the statement.

5.3.2 Expressions

Expressions on the other hand, do **not** transform the state of the variables, but merely denote **values**.

Expressions are formed according to the rules of some formal notation (like the language of arithemtic)

Some expressions: `1 - (3 / 6)`, `1 + x * (y % 10)`

Thus:

Attention:

- expressions **denote values**,
- statements **have effects**

Note: Understanding this dichotomy between statements and expessions is fundamental.

5.3.3 Assignment

The most basic statement is the **assignment**. Its has the basic form:

```
x = E;
```

Its **effect** is to update the value of the variable `x` on the left hand side of = to the **value** denoted by the **expression** `E` on the right-hand side of the assignment operator. In C++:

```
#include <iostream>

int x(1);  //declare & initialize x
int y = x; //declare y and assign the value denoted by the expression "x" to y.

std::cout << x << " " << y << " " << std::endl;
```

```
1 1
```

The expression on the right hand side of the assignment can contain the variable being updated. Expression is evaluated before assignment:

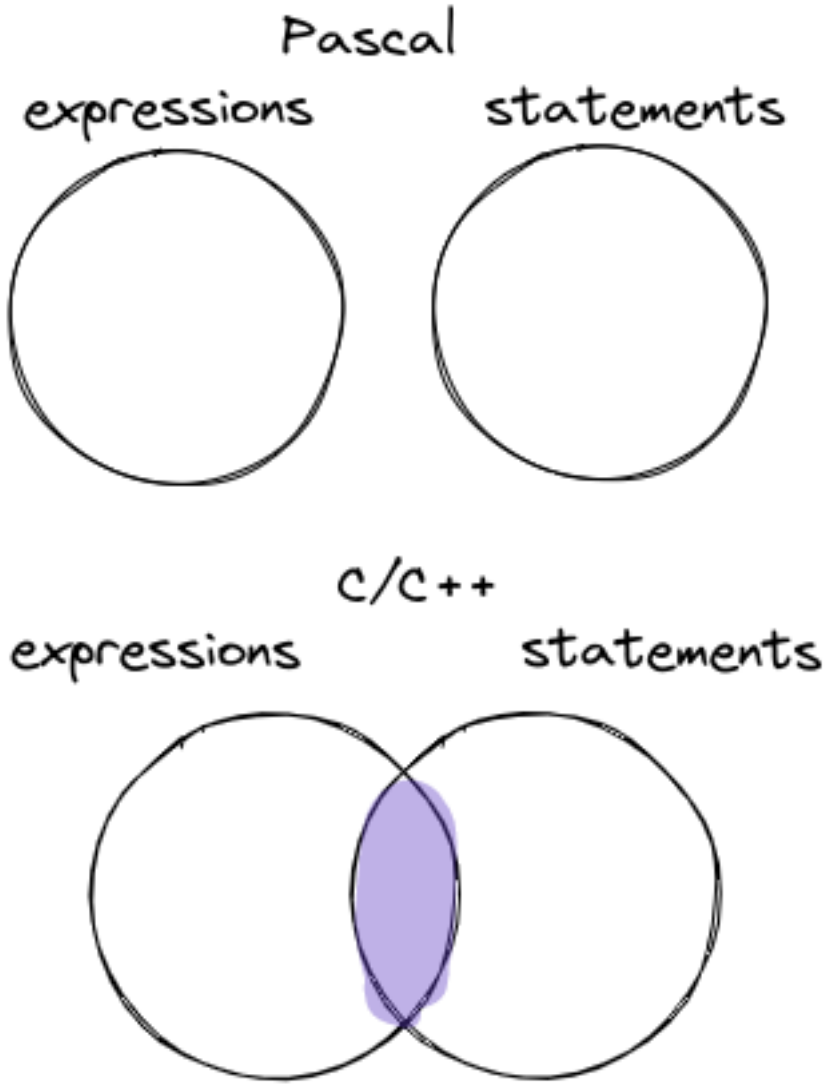
```
y = (y * 3) + x; //assign the value denoted by (y * 3) + x to y.
std::cout << y << std::endl;
```

```
4
```

5.3.4 Side Effects

In some programming languages like Pascal the world of statements and the world of expressions are completely distinct. A given syntactical entity is either a statement or an expression.

But in C/C++ a syntactical entity can be both a statement (have an effect) and an expression (denote a value). The action that such an expression performs is called the **side effect** of the expression.



Example: `j--` is both a statement and an expression. Its effect is to assign to the variable `j` the value of the expression `j - 1`. I.e. it is equivalent to `j = j - 1`. As an expression it denotes the value `j` before assignment.

```
int j = 10;

std::cout << j-- << std::endl; //side effect: decrement j.
std:: cout << j << std::endl;
```

```
10
9
```

Having expressions with side effects is academically less clean, but has practical benefits as it allows shorter programming constructs and succinct **idioms** by using statements as expressions, like:

```
int i = 10;
while (i--){
    std::cout << i << " ";
}
```

```
9 8 7 6 5 4 3 2 1 0
```

5.4 Blocks and Scope

5.4.1 Block

Blocks are compound sequences of statements treated as a unit. Blocks can be nested. Blocks are used to **structure** a program.

A block

```
{
    double x(3.14);
    double y;
    y = x;
}
```

A nested block:

```
{
    double x(3.14);
    {
        double y;
        y = x;
    }
    // y is invisible here
}
```

5.4.2 Scope

Scope is the portion of the code where a variable name is visible.

Potential Scope and Actual Scope

Potential scope of a name declared in a block begins at the point of declaration and ends at the end of the block. Thus scopes are delimited by blocks.

Actual scope is the same as the potential scope, but if an identical name is declared in a **nested block** than the potential scope of the name in hte nested block is exluded from the actual scope of the name in the enclosing block.

Note: There is a recursive aspect to the definition, but stated simply declarations in the nested block overwrite declarations in enclosing blocks

---

```
int i = 0; //scope of outer it begins
++i;
std::cout << i << std::endl;
{
    int i = 1;
    i = 42;
    std::cout << i << std::endl;
    int j = i + 1;
    std::cout << j << std::endl;
}
//j  std::cout << j << std::endl; //this would be an error, because is invisible␣
 ↪in this scope
```

```
1
42
43
```

The scope of a variable defines its **lifetime**. The variable ceases to exist after the program exist the scope where the variable is visible.

5.5 Expressions and Operators

5.5.1 Arithmetic

```
#include <iostream>

int a = 6;
int b = 2;
int c = a + b;

std::cout << a << " " << b << " " << c << "\n";
c = a * b;
std::cout << c << "\n";
c = a - b;
std::cout << c << "\n";
c = a / b;
std::cout << c << "\n";


c = c + a;
std::cout << c << "\n";
c = c * a;
std::cout << c << "\n";
c = c / a;
std::cout << c << "\n";
```

```
6 2 8
12
4
3
9
54
9
```

or equivalently and more succintly:

```
std::cout << c << " " << a << "\n";
c += a;
std::cout << c << "\n";
c *= a;
std::cout << c << "\n";
c /= a;
std::cout << c << "\n";
```

```
9 6
15
90
15
```

more complex and arbitrarily long expresson can be formed, whre precedence of operations follow the usual mathematical rules:

```
std::cout << c << " " << a << " " << " " << b << "\n";
c = b + a/b - a;
std::cout << c << "\n";
c = (a + b)/(a - b);
std::cout << c << "\n";
```

```
15 6  2
-1
2
```

5.5.2 Boolean

```
4 > 3
```

```
true
```

```
4 >= 3
```

```
true
```

```
4 <= 3
```

```
false
```

```
3 <= 4
```

```
true
```

```
4 == 4
```

```
true
```

```
3 == 4
```

```
false
```

```
3 != 4
```

```
true
```

```
not false
```

```
true
```

```
not true
```

```
false
```

As with number airthmetic arbitarily complex boolean expressions can be formed:

```
bool a = false;
bool b = true;
(a || b) && (not true || (b and (a or (!b && a))))
```

```
false
```

Note that for

- **conjuction** both `and` and `&&`
- **disjunction** both `or` and `||`
- **negation** both `!` or `not`

can be used, and the usual precedence rules of negation over conjuction over disjunction hold.

5.5.3 String

The library `<string>` provides the basic data type `string`, to represent sequences of characters and operations thereon.

```
#include <string>
std::string msg1 = "Hey";
std::string msg2 = "there";

std::cout << msg1 << "\n";
std::cout << msg2 << "\n";
```

```
Hey
there
```

Strings can be combined with +

```
std::string msg = msg1 + " " + msg2;
std::cout << msg;
```

```
Hey there
```

and compared with == or != for equality:

```
std::string a = "a";
bool check1 = (a == "b");
bool check2 = (a != "b");
```

```
std::cout << check1 << "\n";
std::cout << check2;
```

```
0
1
```

Warning: When combining or comparing strings there should **always** be variable on the left hand side

CHAPTER

SIX

ARRAYS

Array is a structured data type which is a collection of component variables of the **same type**. For a domain D, D^N is called data type of "D arrays of length N". If $x \in D^N$, then x is called a D array of length N.

To declare a D array x of length N in C++ we type:

```
D x[N];
```

Note: N must be set explicitely in compile time, that is, the length of the array must be known at compile time.

To declare a 10-dimensional `int` array in C++

```
int a[10];
```

All elements are initialized to 0:

```
for (int i = 0; i < 10; i++) std::cout << a[i] << " ";
```

```
0 0 0 0 0 0 0 0 0 0
```

array components are accessed with the syntax `a[index]`, where

$$0 \leq index < N$$

Thus

```
std::cout << a[9];
```

```
0
```

But a[10] is access past the array bounds and

```
std::cout << a[10]
```

outputs garbage.

We can update the components of the array:

```
for (int i = 0; i < 10; i++) a[i] = i;
for (int i = 0; i < 10; i++) std::cout << a[i] << " ";
```

```
0 1 2 3 4 5 6 7 8 9
```

CHAPTER

SEVEN

INPUT AND OUTPUT

C++ uses streams for I/O, found in `<iostream>`.

```
#include <iostream>
```

`std::cout << variable;` is used for output:

```
int var = 10;
std::cout << var;
```

```
10
```

`std::cin >> variable` for input:

```
std::cin >> var;
```

CHAPTER

EIGHT

FLOW OF CONTROL

Flow of control determines the order in which individual statetements, instructions or function calls of the program are executed.

The execution flow is controlled by **control flow statements** like

- if statements
- for loops
- while loops
- break statements

and so on.

- *If-Statement*
- *While Loops*
- *For Loops*
- *Numerical Solution of the Pendulum Problem*

8.1 If-Statement

execute a block of statements **if** a **condition** is satisfied. Syntax:

```
if (condition){
    statement1;
    statement2;
    statement3;
} else {
    statement4;
}
```

```
#include <iostream>

double x(3.14), y;
if (x >= 0) y = x;
else y = -x;
std::cout << x << " " << y;
```

```
3.14 3.14
```

```
if ( x == 3.14) std::cout << "yes";
```

```
yes
```

8.2 While Loops

Block of statements in the loop body are executed repetitevly as long as a certain condition on the state of the progarm is satisfied.

The number of loops can in general depend on the program state and not be predictable in advance.

Syntax:

```
while (condition){
    statement1;
    statement2;
    statement3;
}
```

Upon exiting the loop the nagation of the while condition holds.

Simple example:

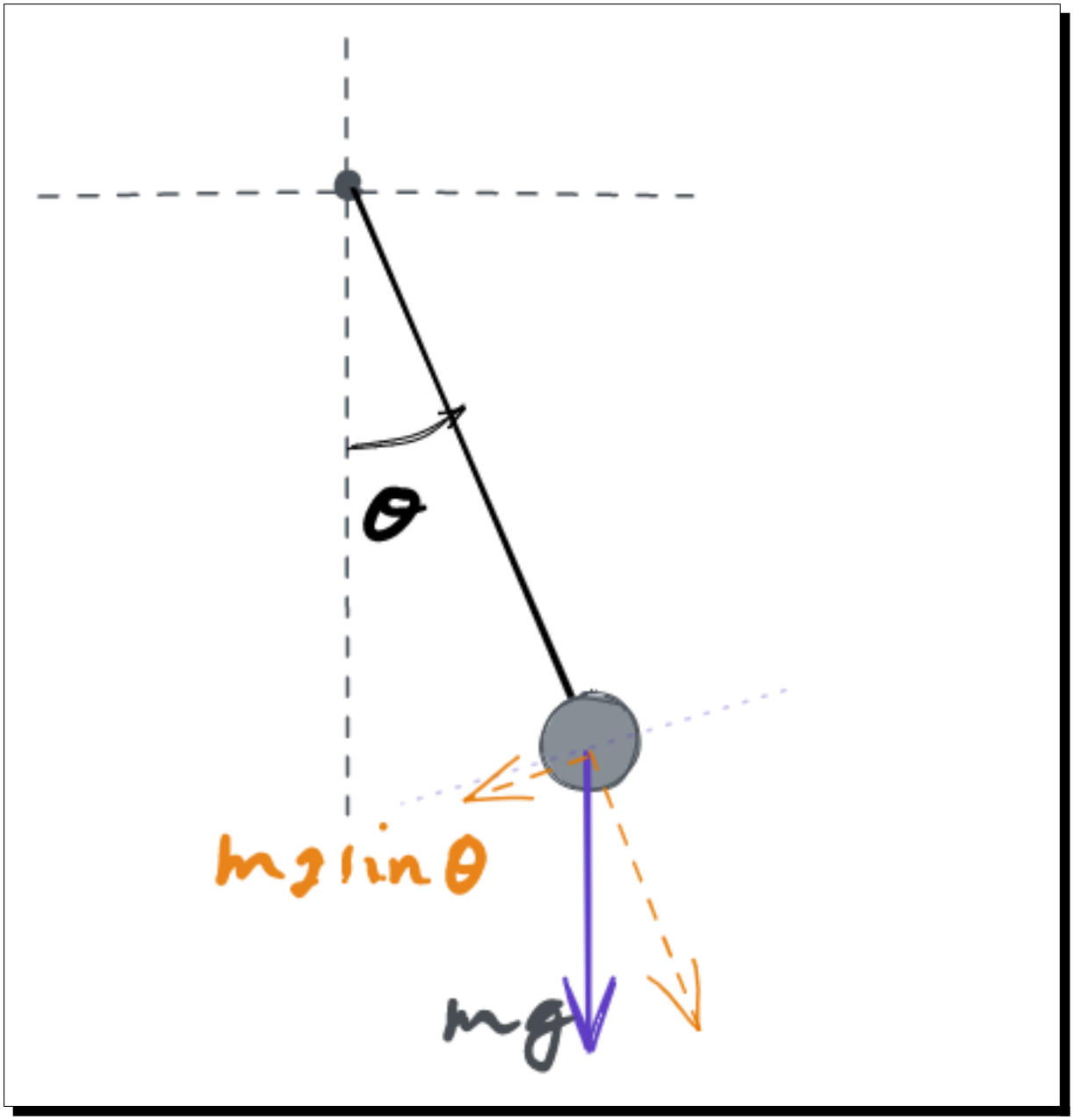
```
#include <iostream>

int i = 0;
while (i < 10){
    i++;
}
// condition i >= 10 holds
std::cout << i;
```

```
10
```

8.2.1 A More Comprehensive Example: Pendulum

Let's consider a simple 2D pendulum:

Simple Pendulum



The equations of motions for this ideal model can be easily shown to be:

$$\begin{aligned} & \ddot{\theta} = -\frac{g}{l}\sin(\theta) && \text{(Mechanics of motion)} \\ \Leftrightarrow \quad & \ddot{\theta} + \frac{g}{l}\sin(\theta) = 0 \\ \Leftrightarrow \quad & \ddot{\theta} + \frac{g}{l}\theta = 0 && (\sin\theta \approx \theta \text{ for small } \theta) \end{aligned}$$

Solving this differential equation with initial conditions $\theta(0) = \theta_0$, and $\dot{\theta}(0) = 0$ we arrive at the solution:

$$\theta(t) = \theta_0 \cos(\sqrt{\frac{g}{l}}t)$$

Following program computes this formula for each time value

$$t_i = i\Delta t, \quad 0 \leq t_i \leq T, \quad i \in \mathbb{N}_0$$

and outputs t and corresponding $\theta(t)$ values seperated by blankspace in a new line to the standard output.

```
#include <iostream>
#include <cmath>

double l(1.34); //Length of the pendulum chord in meters
double phi0(0.2); //Amplitude i.e. the initial angle in radians
double dt(0.05); //Time-step in seconds
double T(1.0); //End-time in seconds
double t(0.0); //Initial time value

while (t <= T){
    std::cout << t << " "
              << phi0 * cos(sqrt(9.81/l) * t)
              << std::endl;
    t += dt;
}
```

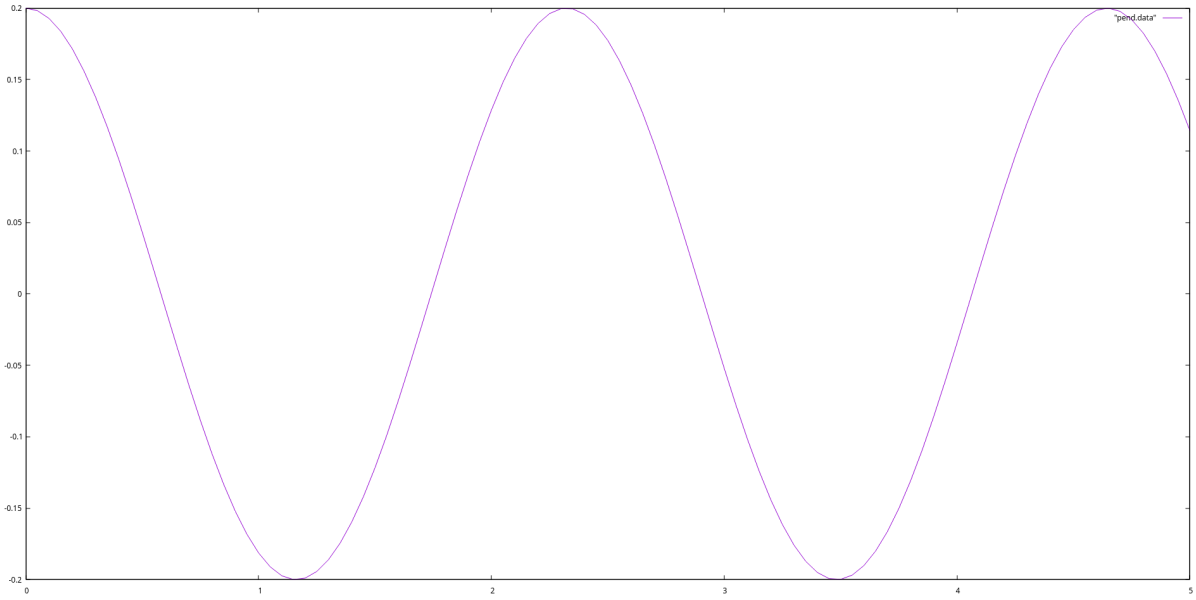
Full program text

```
#include <iostream>
#include <cmath>

double l(1.34); //Length of the pendulum chord in meters
double phi0(0.2); //Amplitude i.e. the initial angle in radians
double dt(0.05); //Time-step in seconds
double T(1.0); //End-time in seconds
double t(0.0); //Initial time value
int main(){
    while (t <= T){
        std::cout << t << " "
                  << phi0 * cos(sqrt(9.81/l) * t)
                  << std::endl;
        t += dt;
    }
}
```

We can put this program in a file `pendulum.cc` and compile it the usual way. The program pendulum can be used to generate the plot of θ vs t with `gnuplot` using linux i/o redirection:

```
$ ./pendulum >pend.dat
$ gnuplot
gnuplot> plot "pend.dat"with lines
```

We obtain the following plot for `T = 5.0`:



8.3 For Loops

Another way to achieve repetition. Syntax:

```
for(start; condition; increment){
    command;
}
```

in C `while` and `for` are equivalent.

Example;

```
#include <iostream>

for (int i = 0; i < 5; i++) std::cout << i << " ";
```

```
0 1 2 3 4
```

The *pendulum example* from the previous section can be rewritten with a **for** loop:

```
#include <cmath>

double l(1.34);
double phi0(0.2);
double dt(0.05);
double T(1.0);

for (double t = 0.0; t <= T; t+=dt)
    std::cout << t << " " << phi0 * cos(sqrt(9.81/l) * t)
                          << std::endl;
```

8.3.1 For and While Equivalence

In C++ all for and while loops are equivalent and can be rewritten to one another.

for-loop

```
for (int i = 0; i < n; i++){
    //S1;
    //S1;
    //...
    //Sn;
}
```

while-loop

```
{
    int i = 0;
    while (i < n){
        //S1;
        //S2;
        //...
        //Sn;
        i++;
    }
}
```

8.4 Numerical Solution of the Pendulum Problem

We can solve the differential equation numerically, without resorting to the $\sin\theta \approx \theta$ approximation using the Euler method to rewrite the sytem as a first-order differential equation:

$$\dot{\phi} = u \qquad u(0) = u_0$$
$$\dot{u} = -\frac{g}{f}\sin\phi \qquad \phi(0) = \phi_0$$

With the time step Δt we can express this differential equation as:

$$\phi_{n+1} = \phi_n + \Delta t u_n$$
$$u_{n+1} = u_n - \Delta t(\frac{g}{l}\sin\phi_n)$$

Coding this difference scheme as a C++ we obtain:

```
#include <iostream>
#include <cmath>

double l(1.34);
double phi(3.0); //note that the initial phi can be large
double u(0.0);
double dt(1E-4);
double T(0.003);
double t(0.0);

while (t < T){
    std::cout << t << " " << phi << std::endl;
    double phiprev(phi);
    phi += dt * u;
```

(continues on next page)

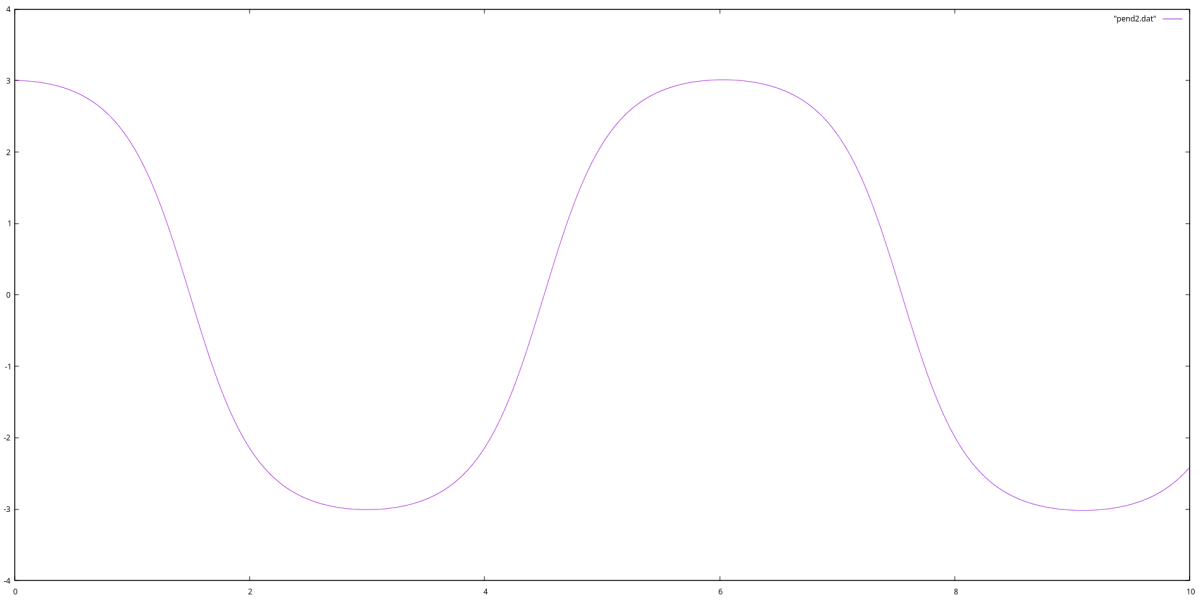
(continued from previous page)

```
    u -= dt * (9.81 / l) * sin(phiprev);
    t += dt;
}
```

Full program text with `T` initialized with `10` instead of small value as above:

```
#include <iostream>
#include <cmath>

double l(1.34);
double phi(3.0); //note that the initial phi can be large
double u(0.0);
double dt(1E-4);
double T(10);
double t(0.0);

int main()
{

    while (t < T){
        std::cout << t << " " << phi << std::endl;
        double phiprev(phi);
        phi += dt * u;
        u -= dt * (9.81 / l) * sin(phiprev);
        t += dt;
    }
}
```

we obtain following plot by plotting the data produced by this program:



CHAPTER

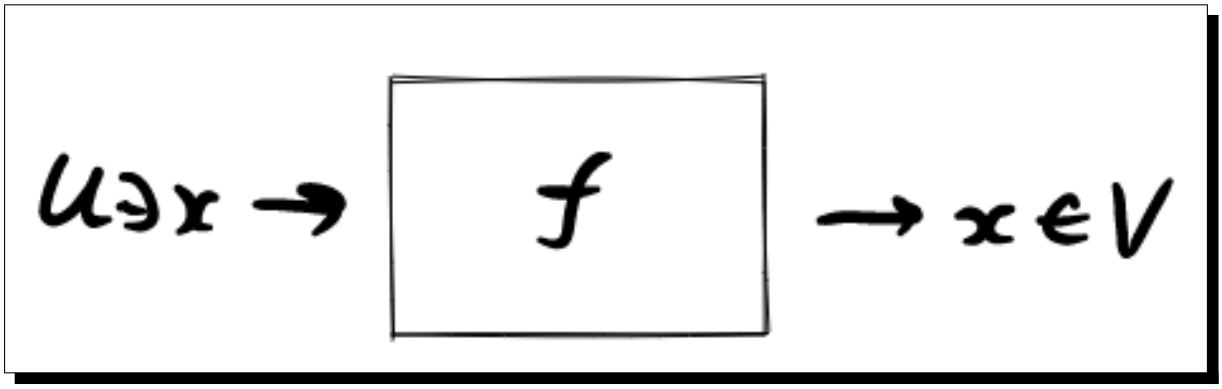
NINE

FUNCTIONS

In math we have the concept of a function; a mapping from one domain to another:

$$f : U \to V$$

Which defines an analogy between an object $x \in U$, that can thought as an "input", and an object $f(x) \in V$, that can be ragerded as the output.



In programming this can be understood in terms of pre- and post-conditions. For the above function f the statement `x := f(x)` has the **effect**:

$$[\![U(x)]\!]\, x := f(x)\, [\![V(x)]\!]$$

Where $U(x)$ is the pre-condition of x belonging to the domain $U(x)$ and $V(x)$ is the post-condition of x belonging to the domain $V(x)$

For example let $f : \mathbb{R} \to \mathbb{R}$, $x :\mapsto x^2$; we define this in C++ as:

```
double f(double x) //funtction head/signature
{
    return x*x; //function body
}
```

Then

```
#include <iostream>
double x(3.0);
std::cout << x << "\n";
x = f(x);
std::cout << x << "\n";
```

```
3
9
```

In generel following holds for the function f :

```
// {x = X}
x = f(x);
// {x = X*X}
```

given there are no overflows or other similar phenomena related to computer representation of real numbers that violate the usual axioms.

Function calls are implemented by the function call stack mechanism. New frame is allocated at the top of the stack when a function call is made, where parameters passed to the function are freshly allocated in the stack frame to be used within the body of the function.

Variables used within the function frame go away after the execution of the function ends and control returns to the calling function. Therefore they are inaccessible to the calling function.

Communication between the caller and the calee is realized by `return` statements, and the values passed as parameters.

Following attempt to write a function that swaps variables doesn't work:

```
void swap(int a, int b){
    int temp;
    temp = a;
    a = b;
    b = temp;
};

int a(3), b(5);
swap(a, b);
std::cout << a << " " << b << "\n";
```

```
3 5
```

because of the "call by value" mechanism explained above. The values of `a` and `b` are determined and passed to the function, which in turns uses those values to initialize new variables `a` and `b` local to its frame. Within this frame the values of `a` and `b` are swapped, but after function returns, and control is back at the caller they go away.

9.1 Recursion

The stack based function call mechanism enables to realization of recursively defined functions, almost verbatim. Consider for example the followig function:

$$\begin{aligned} f &: \mathbb{N} \to \mathbb{N} \\ &: 0 \mapsto 1 \\ &: n \mapsto n \cdot f(n-1) \end{aligned}$$

This is the well-known factorial function. In C++ this is implemenetd directly as:

```
int f1(int n)
{
    if (n == 0) return 1;
    return n * f(n - 1);
}
```

```
std::cout << f1(10) << std::endl;
```

```
810
```

9.2 Iteration

Same function can be realized iteratively with **while**

```
int f2(int n)
{
    int g = 1, i = 0; //g == i! && i <= n;
    while (i < n){
        g *= i + 1; //g == (i + 1)!
        i++; //g == i!
    } //i == n
    return g;
}
```

```
std::cout << f2(10) <<std::endl;
```

```
3628800
```

Recursion vs Iteration

recursion

```
int f(int n)
{
    if (n == 0) return 1;
    return n * f(n - 1);
}
```

while-iteration

```
int f(int n)
{
    int f = 0, i = 0; //f == i!
    while (i < n){
        f *= i + 1; //f == (i + 1)!
        i++; //f == i!
    }
}
```

Note the pre- and post-conditions, and loop invariants, denoted as comments that guarantee the correctness of this program, as long as there is no overflow. The advandate of recursive implementation is that they follow the mathematical specification almost verbatim, and therefore there is no need to specify pre- and post-conditions or to prove the program correctness.

Equivalently **for** loops can be used:

```
int f3(int n)
{
    int res = 1;
    for (int i = 0; i < n; i++){
        res *= (i + 1);
    }
    return res;

}

std::cout << f3(10) << std::endl;
```

```
3628800
```

CHAPTER

TEN

TEMPLATES

Consider following function defintions in C++:

```
double f (double x)
{
    return x * x;
}
```

```
float f (float x)
{
    return x * x;
}
```

These two definitions compute the same function, with the only difference being the signature type defintions. It is possible to avoid this redundancy by using the **templates** abstraction mechanism, that allows specifying generic types:

```
template<typename T>
T f (T y)
{
    return y * y;
}

int x(3);
float y(5.0);
double z(7.0);
```

```
std::cout << f(x) << " " << f(y/x) << " " << f(z / y) ;
```

```
9 2.77778 1.96
```

CHAPTER

ELEVEN

CLASSES

Roughly speaking a **class** is an **extensible** program-code-template for creating **objects**, providing initial values for for state **member variables** and implementations of behavior **member functions** or **methods**.

Class may refer to following distinct but closerly interrelated contructs that are usually conflated in practice:

- **template**: the code text that defines the class
- **type**: the type of the object generated by instantiating an **instance** of the class
- **constructor**: the subroutine that creates objects belonging to the class

consider a definition of a class in some pseudo-language:

```
Class A{
    B x
    C y

    D f();
    E g();
}
```

The above code is the template.

`A a = A(c, e)`. The first `A` is the **type** declaration that the variable `a` is of type `A`. Second `A` is the **constructor** that initializes `a` with the values `c` and `e`.

As an example for a class definition in C++ let's define a simple vector class:

```
#include <iostream>
#include <cmath>

class Vector3d
{
    private:
        float x;
        float y;
        float z;

    public:
        float getX()
        {
            return x;
        }
        float getY()
        {
            return y;
        }
        float getZ()
        {
            return z;
```

(continues on next page)

(continued from previous page)

```
        }
        float getNorm()
        {
            return sqrt(x * x + y * y + z * z);
        }

        float dot(Vector3d u)
        {
            return x * u.getX() + y * u.getX() + z * u.getZ();
        }
}
```

Let's initialize the class:

```
#include <array>

auto v = std::array<int, 4>{1, 2, 3, 4};
for (auto& x : v) x *= x;
std::cout << std::accumulate(v.begin(), v.end(), 0)
          << std::endl;
```

```
30
```

```
int sum(int n)
{
    if (n == 0) return 0;
    return n + sum(n - 1);
}
```

```
sum(10)
```

```
55
```

```
int sum_it(int n)
{
    int i = 0, sum = 0; //sum == s(i)
    while (i < n){
        sum += i + 1; //sum == s(i + 1)
        i++ ;// sum == s(i)
    }
    // i == n, sum == s(n)
    return sum;
}

bool check = (sum(10) == sum_it(10));
check;
sum(10) == sum_it(10)
```

```
true
```

Part II

Numerical Methods

CHAPTER

TWELVE

INTRODUCTION

Introduction to Numerical Methods Notes.

INDEX

B

basic data type, 26
Branch, **15**

C

Commit, **15**

D

domain
    variable, 25

I

index include, 22

R

Repository, **15**

T

Tag, **15**

V

variable
    declaration, 26
    domain, 25