Programming in C(++) and Mathematica

Lecture in SS 2018 at the KFU Graz

Axel Maas

Contents

Chapter 1

Introduction

1.1 The lecture

Programming is essentially the task of making a computer do something. I. e. its purpose is to transform some input to some output, using a computer to do the mapping. In physics, the usual reason to do so is that too many mathematical operations are required as that a human could perform them in any reasonable amount of time. Outside physics, many other topics arise, but in the end the main motivation is always that some task is more efficiently done by a computer than a human being and/or some task is so irrelevant that a computer can do it as well as a human, such that the human has time for other things, which are less trivial. Still, there is the same basic tenant: Transform an input to and output.

It is this task, which lies at the heart of writing a program. Therefore, before any program is created, the first step should always be to really consider the following questions:

- What is the input?
- In which form is the input available?
- What is the desired output?
- In which form should the output become available?
- What is the map?
- Are there any conditions which may alter the map?
- How can it be ensured that the output is correct?

All of these question are far less trivial than they may appear. The aim of this lecture is not to provide answers to these questions. The aim is, given the answers to these questions, how can the computer be told what it has to do.

Thus, before writing any program, one should sit down and answer these questions. The result of this are flow diagrams, as well as so-called use cases. By now, structured tools are available to do this, and partially these tools come with automatized code generators, which create at least part of the required code. However, this is usually not the case for physics. Even if it is, it requires first to understand how these generators work, i. e. how they program, to understand their output. The necessary understanding is only obtained by experience, and thus programming oneself.

Programming is not a skill which can be learned by reading or calculating. It is a skill which, like any craftsmanship, requires first of all experience. Thus programming oneself is the only way to really acquire the necessary abilities. Therefore, doing (exercises in) programming is the only way to really become a programmer.

1.2 Building an example

To develop the skills, it is necessary to do something. So, let's start.

It is not yet possible to really program, but it is possible to do so-called pseudo-code. Pseudo-code is a, more or less formalized, way to describe a program in natural language or with approximative programming statements. This pseudo-code will be turned into real code in due course.

The prototypical first program is the so-called 'Hello world' program. It does nothing else than putting the statement 'Hello world' on the screen. In terms of the previous section, this means there is no input, just output. A pseudo-code version of this program looks like

Listing 1.1: "Pseudo-code hello world"

```
1 Start
2 Print 'Hello world'
3 End
```

It shows that the code starts somewhere, then does the printing, and finally ends. That is all. But it already lets the computer do something. However, in practice more formalized languages will be used, of which there are many. But they belong to certain types, and these will be introduced now.

1.3 C, C++, and Mathematica

Just like ordinary languages, programming - speaking to the computer - can be done in many ways. Every way corresponds to a given programming language. These differ widely in details, but belong to a few basic classes. These classes differ in the level of abstraction from the actual computer, the hardware.

There are those which are very close to the hardware, the so-called low-level languages. They are often specific to a given type or group of hardware. It is rare that one is encountering this type of language nowadays in physics, and therefore little will be said on them in this lecture. These languages are very often hard to read for humans, as they are much closer to the machine.

The next ones are procedural languages. These are essentially languages based on single, human-readable, statements. A corresponding example in mathematics is $x = 1+2$. Such statements can be combined, like in mathematics more complicated functions can be composed from simple functions, but this is essentially the highest level of organization. These languages are widely used in physics, as they form a compromise between abstraction and efficiency. They will therefore be discussed at length in this lecture, using the example of C. Other important examples are Pascal, Fortran, or Cobol.

One level higher are the so-called object-oriented languages. They are no longer based on statements, but rather on entities, which are collections of data and operations on the data. They are the cornerstone of modern programming, especially when it comes to problems which are not essentially doing mathematical exercises. They will be treated in this lecture by the example of C++, a superset of C. Other examples are Java or (to some extent) Python.

Functional languages are even higher abstractions. Rather than using statements and entities, they are working with implicit definitions. In a sense, they are again more mathematical than object-oriented languages, as they are based on maps from input to output. But this moves them again more closer to the original idea of programming. The are also providing more formal and abstract manipulations. Mathematica, a language particularly suited for problems in mathematics, will be the example of choice in this lecture, even though it is a hybrid of a functional and a procedural language. Another example is Prolog.

It should be noted that programming languages, just as natural languages, evolve. And just like there are times at which a natural language becomes (re)codified, providing rules of how to write and spell it, there are times where the same happens for a programming language. These are so-called standards, often defined by international organizations. In professional programming most often the current and next-to-current standard are

prevalent. In physics, where in many areas one encounters projects running over decades, this is not necessarily so, and often decade-old versions need to be used. Therefore, in the context of this lecture, it will be attempted to only use elements which are available for some time, to ensure availability in practice. Therefore, in some case more recent standards may also provide more elegant or compact ways of dealing with the same problems.

Still, no matter the language, all programs have a similar basic structure.

1.4 Basic structure

The simplest possible concept for a program is that every program consists out of two different parts. One is the code, i. e. a set of instructions. The other is the data, i. e. information. This data separates into two different subsets. One is external data, e. g. input from a user. The other is the internal data, i. e. things the program 'knows', usually in the form of memory of the computer which processes the code. While a strict separation in both concepts, code and data, will be maintained throughout this lecture, the line between both can sometimes be blurry or even non-existent when starting advanced programming. Still, strict separation is pervasive in physics applications.

Every such program has a beginning and an end in its written form, the so-called code, even if the actual program may run for all eternity. Thus, there is always a single first statement in every program. However, there may be many more than one last statement, or even none. At the beginning, data is only available in the form of predefined constants and free memory, which can be used to store data during the runtime of the program.

One important concept is the recognition that data needs to be stored somewhere. It is therefore necessary that the program has access to this storage. For this purpose, the program needs an address for a chunk of data, which it can then retrieve from the storage. This address, often called a pointer, is itself data. However, addresses are stored at fixed places, on which the program is informed by the underlying system software, the operation system, when started.

The example in listing 1.1 has already the same structure, but does not use data. Therefore it is useful to have also an example in pseudo-code which is a bit more involved. Following the recipe in section 1.1, define first what it should do. Here, lets assume it takes a number from the user, which it multiples by a fixed factor and then prints the result. Not a big deal, but it involves now all the basic features: Internal and external data, operations on data, and user interaction. A pseudo-code version could look like

Listing 1.2: "Pseudo-code for user interaction"

```
1 Start
```

```
2 Put factor in memory
3 Ask user for number and store number in memory
4 Calculate the result and store it in the memory
5 Print the result
6 End
```

One important thing is that all the data the program uses needs to be somewhere for the computer. The action of 'remembering' is nothing happening automatically, it has to be done explicitly by the programmer in the code, and for each thing separately. In 1.1 this was not yet mentioned explicitly, but in principle line 2 of 1.2 would also be needed there. But in this case modern languages are quite helpful, as they help to avoid many of the details in this case, but not in the present one.

This shows that there are a lot of things to be kept in mind when writing a program. Therefore programming has a lot to do with how to manage things, and especially every program is a project.

Chapter 2

Computer structure

To really understand how to program requires at least basic knowledge of how a computer works. To write efficient code, which is often, but not always, necessary in physics, requires a rather deep understanding of computer architecture. The latter is not a goal of this lecture, given the limited amount of time, but is rather a lecture on its own.

2.1 CPUs and assembler

The basic principle of a computer today is still the concept of a von Neumann-machine. It is essentially the idea of sequentially performing operations. Such machines consist out of two components. One is a machine which can execute a command. One is a storage device, which can hold either of two information. One is data, the other is commands. The process is now to start at the beginning of the storage, and read the first information. If it is a command, it is executed. If it is data, the data is memorized for the next command. It is also possible to write to the storage, and to have a command by which to move some steps forward or backward in the storage.

Modern computers are of the same type, though in detail much more sophisticated. There is still a separation between commands, usually the code, and data[1].

Data is nowadays stored in a hierarchy of storage systems, which are primarily characterized by two features: Capacity and speed. As a general rule, the higher the capacity, the lower the speed. The slowest are harddisks, including SSDs, optical media, and sometimes tapes. The next level is memory, known as RAM, or random-access memory. Random means here that any part of the memory can be directly accessed. In the sense of the von

[1]There are some languages in which the code is in addition also considered as data, and can be modified by the program itself. Such self-modifying code is not without security issues, and whether it is a good idea or not is still not settled. This situation will not occur in this lecture.

Neumann machine this implies that one can jump at every point in the storage directly. The next is then cache, which grows from the slowest level 3 to the fastest level 1 cache. Finally, there are the registers. These registers contain the actual data on which a program operates. Thus, data has to be taken from some higher level of storage and brought into the registers, and obtained data has to be moved back. This is done by intricate specialized code to do this very efficiently. User-provided data, like what is typed on a keyboard, is usually directly transferred to the RAM, and used from there.

All of these issues of where data is stored is mostly transparent to the programmer. The only notable difference will be access to memory and to objects like a harddisk. They differ by the fact whether they are persistent, i. e. whether their content is still available when switching the computer off and on again. RAM and all faster memories loose their content when doing so, again a sacrifice for their speed, while harddisks do not. Thus, in the following writing to harddisks or similar devices for persistent storage or using the memory while doing computation will be the only two different concepts of data storage used.

When one wishes to write efficient code the distinction between the various types of memory becomes again important.

The code is executed by the central-processing unit (CPU). It works, for all intents and purposes, linearly, i. e. executes the code one command at a time[2]. However, the CPU is not able to understand natural language, but requires the commands in its own language, so-called opcodes. These opcodes are essentially numbers, which then the CPU knows how to treat, as it is physically hardwired into it. For a non-expert human, these opcodes care unintelligible. They are actually symbolized in a slightly more abstract notion as so-called assembler code (where each statement can actually be more than one opcode).

Such assembler statements are, e. g. of the type that they command to load two numbers into two registers, then add the two registers and store the result in a third register, and finally write the result from the register again to memory. As is visible, this is not a particular elegant style from a human point of view.

2.2 Compilers and interpreters

Thus, over time, people developed programs which allowed to write code in a more accessible language, so-called high-level languages, which are then translated to assembler and opcodes for the machine automatically. C and C++, to be used in this lecture, are such high-level languages. The programs which translated, e. g., C-code into assembler

[2]Again, this is not exactly true with modern machines, but remains irrelevant for this lecture.

are called a compiler. The one to be used primarily in this lecture is called gcc/g++.

A compiler takes a whole program, and translates it. It is then ready to be executed, i. e. started like a normal program. An alternative are so-called interpreters, which interpret the code in real-time, and therefore the code need even not be finished before starting to execute it.

While interpreters are very convenient for fast or real-time development, they are usually less efficient. On the one hand, their translation in real-time requires resources, and has to be done every time the program is run. Also, the interpreter cannot use the knowledge of the whole program to optimize the code, which modern compilers do to a large extent.

By now, there a literally hundreds of languages, C remaining among the most popular ones, and for many languages both compilers and interpreters are available. These languages provide quite different abstraction levels, and allow to a different extent to interact with the underlying machines. Generally, a more abstract language tends to provide better understandable, manageable and writable code, while at the same time obstructs writing efficient code. E. g. C is less abstract than C++, as will be seen. If efficiency is not an issue, more abstract languages are usually better suited. However, this is often not the case in physics, and C and C++, which are at a lower and higher middle level, respectively, of abstraction, are therefore suitable examples.

2.3 The structure of a program

Since the setting has now been created, the next step is to understand the basic structure of, essentially, any program.

Every program has a single entry point, i. e. a first instruction. These are the lines 1 in listings 1.1 and 1.2. An instruction is sometimes also called a statement.

There is not necessarily any unique second or further statement, as which instruction is executed next can depend on the previous statement. Indeed, even randomness can be included in a program, so it can even become wholly non-deterministic. This aspect will be ignored for the moment, to simplify things. However, even in a deterministic program there may be many final statements. This is not the case in listings 1.1 and 1.2, as these programs are just worked through sequentially, and the last lines are the last statements.

Somehow, a program needs therefore to separate statements, i. e. there must be some kind of delimiter between statements. In C and C++, this will be a semicolon. Thus, a typical part of a C program looks like

```
1 statement 1;
```

```
2 statement 2;
3 ...
4 statement n;
```

That is, there is a sequence of statements, which will be executed one by one and one after each other. In Mathematica, these are so-called cells, which are marked by a bracket in the environment. In listings 1.1 and 1.2 every line is a statement of its own. Would this be C(++)code, each line would needed to be terminated by a semicolon. It should be noted that a statement can involve rather complex actions. This can be compared to an expression in mathematics. Both $1 + 1$ and $\sin(1 + 4/3 - 12\pi)$ are expressions, but of quite different complexity. In this sense, it is often talked about atomic statements, which cannot be decomposed further without becoming trivial. Thus, in the second case $1+4/3$ is an atomic statement, while 1 and $4/3$ are trivial and $1+4/3-12\pi$ can be further decomposed, and thus neither are atomic.

However, as emphasized in the von Neumann machine, it is also necessary to have data. Data is not specified in statements, but otherwise, and again specific to each programming language.

Statements must be understandable for the compiler. This is achieved by having a particular syntax, which defines the rules for creating well-formed statements. In principle, the syntax makes the semantic deterministic, and there is no room for misinterpretation. However, especially for complex languages with rich syntax, it may happen that not every possibility has been taken into account when creating the syntax. However, this will very, very likely not be an issue during this lecture. Usually the compiler will stop compiling or give an error message when encountering malformed statements. This is like in mathematics. $1 + 1$ makes sense, but $11+$ does not.

An important part of the syntax is precedence. Precedence is similar to calculational rules, e. g. that first all multiplications are performed, then afterwards additions. Similar precedence rules exist for languages. They determine in which order instructions are performed, when a statement is a composition of atomic statements. Comparing again to mathematics,

$$3 + 4$$

is an atomic statement, but

$$x = 3 + 4$$

is not, as there are two instructions, addition and assignment. The precedence here tells that the addition is performed first, and the assignment afterwards. But this is a convention. Though languages usually keep the precedence for mathematical operations, it is no

longer obvious how they act when presented with more complicated atomic operations, which have no analogy in mathematics. Therefore, the rules of precedence are part of the syntax.

Parentheses are the usual tool to modify precedence, e. g. in

$$3 + 4 \times 5$$

the multiplication is done first and in

$$(3 + 4) \times 5$$

the addition is done first. Parentheses are used in the same way in programming, though their precedence-altering feature is extended to arbitrary statements, not only mathematical statements.

Chapter 3

Variables

While a program could, in principle, only perform instructions, it is actually most useful if it uses these instructions to manipulate information. These information has to be made accessible to the program. This is done by so-called variables.

3.1 A rudimentary program

To demonstrate how variables operate, it will be necessary to have, at least, a rudimentary program. Though this could also be done in the form of pseudo-code like listings 1.1 and 1.2, there are many subtle differences in how different languages treat the concept of variables. It is therefore useful to start becoming concrete. It will therefore now be switched to explicit C(++) and Mathematica programs, even though it may need some elements which will only become available fully later.

In Mathematica, this is comparatively simple, as a newly created notebook is already a well-defined program frame in itself. In a notebook, any line can be used to start the program, and a line is executed by pressing shift+return.

This is different in C. The most simple program is

```
1 int main(void)
2 {
3  return 0;
4 }
```

which will do nothing. It therefore corresponds to lines 1 and 3 of listing 1.1 only. It is complied by the command g++ filename, where filename is the filename, usually some name with the extension .c or .cpp. The complier creates then an executable called a.out. To give it a different name, use the -o targetname option. It is also recommendable to use

the additional options -pedantic -Wall, which will turn on all possible warning messages from the compiler. This is very helpful when searching for bugs.

The listing contains a number of important concepts, which can help to illustrate what the concept of syntax can encompass. The first is the declaration of a so-called function. This will be discussed in much more detail in chapter 5. However, in C everything is considered as a function, and thus also the program itself.

In general, a function in C is declared by the syntax

```
return_type name(arguments)
{}
```

which has four parts. The name is just the name by which the function is known in the program. It can contain all kinds of letters, numbers, and a few special characters, like underscores. It cannot start with a number. The name is case sensitive, as everything in both Mathematica and C(++) is. The name main is special: It denotes the function which should be executed as the program. I. e., the first line of this function will be the first instruction executed at program start[1].

The arguments, as will be discussed later in chapter 5 in more detail, is information passed to the function. For now, it is only important that the identifier void has the meaning that no information is passed to the function.

The return_type signifies of which type the result of the function is, e. g. an integer number - just like a mathematical function also a function in C(++) can have a result. In general, a function does not need to have a result, and therefore the result type can be void as well[2]. Here, the type is int which signifies that the result is of the type int, essentially an integer number. This will be discussed further in section 3.3.

This result is returned by the statement return 0; in line 3. This instruction terminates the function, and returns the value 0. The return value of main is passed back to the operating system. The value 0 is interpreted by operating systems as a normal program termination. So, this program just terminates itself and telling the operating system that everything is just fine.

As noted above, the semicolon signals in C(++) the end of a statement[3]. After a semicolon the next instruction begins. It therefore separates instructions. Note that not

[1] This is not entirely true, but this will come later.

[2] Functions with no return type are sometimes called procedures to signify the difference. Then the name functions is kept for anything which returns a value, just like a mathematical function. In the context of C(++) this is rarely so, as nothing is usually considered as something as well.

[3] In Mathematica, it has a different meaning. There, it suppresses the output of the result in the notebook.

every instruction needs to be finished on a line, and line breaks can be inserted at the programmer's discretion.

To every function belongs the so-called body of the function. This is the set of instructions, as noted above separated by semicolons, between the curly braces. Thus, the curly braces signal the set of instructions belonging to the function. If a function is executed, the instructions are executed one-by-one in the order they are written in the body, starting with the first.

This completes the basic definition of a C(++) program and a Mathematica program. Note that the Mathematica program needs not to be compiled[4]. Rather, they are interpreted, i. e. executed manually and in real time.

3.2 Basic feedback: Screen output

Compiling and executing the previous program or looking at the Mathematica notebook is not yet useful: Nothing happens. While for the Mathematica notebook this is not surprising, as there is not yet any statements, this is somewhat surprising for the C program. Somehow, some kind of feedback would be desirable. Some kind of output would be interesting, signaling what is going on. For this purpose, here basic output will be added to the C(++) program. As the Mathematica notebook is interpreted, direct feedback will be provided within the notebook. This will be seen in the examples in section 3.3.

Here, for C(++) only one possibility of such output will be discussed. A more extensive discussion will be given below in section 3.10. In fact, the one discussed here is specific to C++, and not usable in C, but it is more convenient.

Actually, screen output is not part of the C(++) language. There is therefore no such instruction as part of the language. But C(++) is accompanied by extensions, called libraries. This concept will be discussed in more detail in chapter 7. For now, it suffices to say that a library provides additional functionality to a language, and that there can be multiple libraries providing different things. Screen output is contained in such a library. How such libraries can be systematically used will be discussed in more detail in section 7.2.

For now, it suffices that it is possible. To make the example particularly convenient, this is actually the only element missing to get listing 1.1 into C++. Doing so yields

Listing 3.1: Hello world

[4]There is support for compiling also Mathematica programs, if need be, but this will not be part of this lecture.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void)
6 {
7  cout<<"Hello world"<<"!"<<endl;
8  return 0;
9 }
```

The include instruction in line 1 instructs C++ to make the library iostream, which contains screen output, available to the program. It is then known for all parts of the program following this instruction. Note that such inclusions are instructions to the compiler, and not part of the program to be executed. This is therefore also called preprocessor, and indicated by the leading hashtag #. It is therefore available even before the first instruction in the function main. More on the preprocessor will be presented in section 7.3. Here, the instruction requires to load a library called iostream. The brackets around the name inform the compiler where to search for this library, and the fact that they are angular brackets indicate that it is a library coming together with the compiler.

The curly braces {} again define the actual statements. Such a collection of statements is also called a block - a set of statements which form a logical unit. Blocks will be used to identify units of statements, which are considered to belonging together, also called local. In the present case, the whole program is a block, which separates the program itself from the administrative part used to instruct the compiler in the include and using statements.

This using statement in line 3 is actually only for convenience. Imagine a situation in which several libraries, programmed by many people, are included in a project. In such a situation it can easily happen that different libraries use the same names. This can give rise to ambiguities. To avoid this, libraries can be given a so-called namespace. Then, every element from the library gets the namespace as a kind of family name on top of its proper name. Using the combination of namespace and proper name is done in C(++) by creating a statement namespace::element, where the double-double dots :: are a so-called scoping operator. The statement in line 3 now instructs the compiler to implicitly assume that the namespace is called std, if not specified otherwise. If this would not be made, the statement in line 7 would need to always use the full name

```
1 std::cout<<"Hello world"<<"!"<<std::endl;
```

as both cout and endl are part of the library iostream. Thus, the using is again an instruction

to the compiler, and not part of the instructions of the program itself, and merely serves the purposes to shorten the instructions inside the program. More on namespaces will be presented in section 7.4.

This leaves the actually interesting instruction in line 7, for which all of this has been done. It involves actually three instructions, which are concatenated in a very peculiar fashion, which will be discussed in greater detail in chapter 5. It is based on an object, defined in the library, called cout. The double left angular bracket << is actually a function of this object. The meaning of this function is that its argument should be send to the screen. Thus, the line instructs to print out "Hello World", and as a second instruction to do so as well with "!". This splitting is arbitrary, and just for illustration purposes. The double apostrophes are used to enclose a sequence of letters, or other printable characters. Such a sequence is called a string, and how to deal with them more generally will be discussed in more detail in section 3.11. The final thing send is a quantity which is a not-printable character, and which is a combination of ending the line and jumping to the beginning of the next line. Since it is not printable, it has received a particular identifier, the name endl. Thus, this statement, which is made out of three atomic instructions, prints "Hello world!" on the screen, and then jumps to the beginning of the next line.

The final line 8 again just finishes the program, and returns as a result the value 0 to the operating system. This is usually done on the command-line, or by clicking the program. 0 is the (almost) universally accepted value for normal program termination, i. e. without error. However, the operating system, without further effort, just disregards this value.

Thus, all of this complexity has only the purpose to print out a single line on the screen. It seems like a lot of effort to achieve a rather simple objective. Indeed, there are simpler ways to achieve the same end in many other languages, and even in C, rather than this C++ example. The same is achieved by the program

```
1 #include <stdio.h>
2
3 int main(void) {
4   printf("Hello world!\n");
5   return 0;
6 }
```

While the program is rather similar in many respects, it differs in the way how it performs the output. It also uses a different library. First, it is only a function, which gets a single argument, so no object involved. It also does not need a namespace. Also, the endl is replaced by \n, which has for the function printf the same meaning as endl for cout.

In Mathematica, any output is created by writing some statement, e. g. “Hello world!”, and then pressing simultaneously the Shift and Return key. The value of the expression will then be printed. To suppress this printout, add a ; to the end of the expression.

3.3 Variables and variable types

While it is nice to output something to the screen, this is little better than a (primitive) electronic book. The real possibility only arises when data is involved. As an example, think again of listing 1.2, which interacted with the user, but also needed to keep data in mind.

But to use data requires to store data. The data storage of the computer is the memory. Chunks of this memory are used to store the data. These chunks are called variables. Variables require two things. One is an identifier by which a chunk of memory is addressed. I. e., some information where to find the chunk of memory in the total memory of the computer. It thus acts like an address, and hence the name. The name of a variable, as for functions, is usually a sequence of letters, numbers, and some permitted special characters. In Mathematica also (fixed) indices are possible (usually input by using ctrl+_). The second part of a variable is the data stored in it.

In this respect it is important to differentiate between so-called strongly-typed languages and weakly-typed languages. Mathematica is an example of a weakly-typed language, as are most interpreted language. A variable is introduced by just using its name. The variable adapts automatically, i. e. it can hold whatever content one desires, and Mathematica takes care of how to store it. E. g. a=2 in Mathematica will create a variable named a and store the value 2 in it.

The situation is substantially different in C(++), which is a strongly-typed language. Here, variables need to be declared before using them. In the declaration the name has to be defined, and also the type of data it should hold. The type can be, say, an integer number or a letter. Only data of this type can be stored in the variable.

C(++) knows a number of primitive, i. e. built-in, data types. The most common ones, and how variables of their types, are declared, are the following

```
1 int a;
2 char i_am_a_variable,metoo;
3 float b;
4 double s12;
```

This chunk of code declares five variables, a..., of the names indicated. The second line

shows that it is possible to declare several variables of the same type just by separating their names by a ,, while a declaration is again finalized by a ;. A variable cannot be used before it is declared. However, declarations can occur at any time during program flow. It is, nonetheless, considered good practice to declare all variables at the beginning of a function, rather only when needed[5].

The variables do not live forever. They only exist within the part of the program which is enclosed by the current block, i. e. set of curly braces. Outside these blocks they simply do not exist. If they should be available in the whole program, so-called global variables, then they must be declared outside all blocks, e. g. after the namespace. Global variables are despised for the purpose of structured programs and security reasons, but may occasionally be necessary to gain efficiency. However, if there is no reasons for a variable to be global, it should not be. This will be discussed in more detail in section 5.3.

The names in front of the variables in 3.3 define what kind of data the variable, and thus the chunk of memory addressed by the name, holds.

The type int is containing an integer number, which can either be positive or negative. This number cannot be arbitrarily large. However, precisely how large is possible depends on the computer and compiler. The actual value can be either obtained by checking documentation or at run-time. To check at runtime, the function sizeof can be used. The command sizeof(a) returns the data size in byte. The range of integers stored for n bytes is then obtained as $-2^{8n-1}-1$,...,2^{8n-1}, where the -1 in the exponents stems from the requirement that half of the numbers is negative and half of the numbers are positive, and the -1 for the negative numbers is used to have space for zero.

The type char is designed to hold a single letter or other character. It has always the size of 1 byte[6]. Internally, characters are represented as numbers, which are mapped by a table to characters. Therefore, a char can be considered also as an int of one byte size.

The types float and double are both real numbers of either sign[7]. They differ only in the size, and thus precision. The type double is more precise, but at the expense of the amount of memory needed for them larger and dealing with them requires usually more computer runtime. Internally, real numbers are always stored as an exponent and a mantissa between zero and one, where the range of the exponent depends on the type.

[5]The compiler takes care of actually creating the variable and data only when actually needed, so this does not waste efficiency.

[6]Occasionally, if Unicode is used, they may be larger. Again, this depends on the configuration in question.

[7]For complex numbers there is also the type complex⟨*subtype*⟩, where *subtype* can be double, float, int, or extended declarations of that. Why and how this works requires the concepts of templates, which will only be discussed in section 10.11, as these are not primitive types.

Also, the mantissa is not exact, but only a finite number of digits are stored. If attempted to store a number with more digits the surplus digits will be dropped or rounded, which again depends on the configuration in question. It should also be noted that internally real numbers are stored in binary format, which also affects this. Numbers which are decimally having only a finite number of digits may have an infinite number in binary, and vice versa, and are thus truncated.

These completes the list of basic (primitive) types in C(++). However, these types can be modified in several ways. The actual precision can be changed by adding the modifiers short or long to the name, yielding, e. g., long double. These reduce or enlarge, respectively, the range the variable can hold. For int even the modification long long int is possible. Again, the actual range can depend on the configuration in question.

In addition, it is possible to change int and char by the qualifier unsigned, which then stores only positive numbers and zero, but doubles the range for them. Note that this does not change the capacity for a char to be interpreted as a character. The usage of unsigned is not only to change the storage capacity. It also tells the compiler what to expect for a number. Therefore, if, e. g., it is attempted to store a negative number in a positive variable or if it is compared to a signed number, the compiler can notify this, which can help in preventing errors.

Note that in C(++) there is also the special type void, which means that something does not have a type. It is therefore not possible to use it to declare variables, but it appears regularly if something should be defined to have no type.

3.4 Constants

C(++) also allows for all variables a further modifier, const. This declares that a variables value is fixed when it is declared. This is done, e. g., as

```
1 const int five=5,six=6,seven=five;
2 const double pi=4.0L*atan(1.L);
```

The value of the variable five is immutable the value 5 throughout the whole block in which it is declared. As is visible, it is possible to create multiple constants in a comma-separated list, which can use priorly defined constants, where assignments in the list are read from left to right, and even mathematical operations.

The second line defines π from a mathematical operation, which will be discussed in more detail in section 3.6.1. This has the advantage to have a value of π as precise as the computer is capable of, which would not be assured by entering a decimal. The L requires

the compiler to evaluate the given number at double precision. Otherwise, it will evaluate 1 as an integer and 1. as a float. This distinction can have unexpected side effects if not used in a controlled way.

The advantages of using constants are twofold. On the one hand, they are faster than variables when accessing their values. This is because the compiler replaces any occurrences of the constant in the code with the value during compilation, rather then first loading the contents of a variable. Especially, it preevaluates any mathematical operation involved, like the inverse tangents in the second line, at compile time. Since anything, except addition, subtraction and multiplication, are very expensive in terms of computing time, this saves a lot of execution time.

The same effect could also be achieved by using at all occurrences of the constant directly the number. In fact, this may seem like a good idea. The downside is that if the value of the constant ever changes, it has to be replaced everywhere in the code, which in larger codes requires a large amount of time. While it can be argued that the value of π is unlikely to change there are many features, especially in scientific programs, which could change. E. g. a new measurement of the neutron life time during the time your code is used shifts the value slightly, but significantly, for your purpose. On the other hand, this value is unlikely to change during program execution. Thus a constant is the best choice to include the value of the neutron life time in your code.

In Mathematica in most cases there is no distinction between a constant and a variable.

Note that some quantities, e. g. π, are already available inside a library or in Mathematica. E. g. the value of π in Mathematica is given by entering π, which is done by the sequence EscpiEsc, Esc is the escape key. Euler's number is obtained from E, and so on. The list is available from the corresponding documentation.

3.5 Access to variables and type casts

Once a variable is defined, it is necessary to give it a value. Several examples are shown in the following code

Listing 3.2: Assignement of variables

```
1 int a,b=3,c;
2 float f=1.5,g=1./2.,h=0.5;
3 c=5;
4 a=b;
5 f=b;
6 a=(int)g;
```

```
7 a=static_cast<int>(g);
```

As shown in lines 1 and 2, it is possible to initialize variables with a value, just like it is possible for constants. Real numbers can be written both as decimal numbers or as (continued) fractions. Because the quantities are cast to binary before the calculation is performed, it may occasionally happen that in situations like for g and h both quantities may not have the same value, but differ, say on a typical machine, by a quantity like 10^{-19}. This may seem not to be a lot, but can make all the difference between equal and not equal.

Line three then shows a typical assignment, using the, so-called, operator =, of a value during run-time, giving c the value of 5. As line four shows, it is possible to copy the contents of one variable to another by again an assignment. It is important to note that only the contents, and not the address, of the variable is copied. Both variable remain distinct, and a subsequent change to b will not change the value of a. It is very important to carefully distinguish between a variable and the contents of a variable.

In a language like C(++), which is strongly typed, the assignment of variables of different types is not an entirely trivial issue. If the variable, which the value is assigned to, is of a type which can hold all values of the type of variables from which it receives the value, it is simple. Then the assignment is done, as in line 5, just as if the variables are of the same type.

If this is not the case, there occurs what is called a type cast, that is a change of the contents of the variable to fit into the new variable. In this process actually only the transferred value is changed, and the contents of the old variable is left untouched. This can be done either implicitly or explicitly. In an implicit typecast the compiler decides how to truncate the value of the variable. How this is done may depend on the compiler. If the implicit typecast is used, the assignment looks like in line 5, e. g. f=b.

If the rules should be set by the program, an explicit type cast is called for. This is done in line 6, where the value contained in `g` is first converted to an integer[8], and then assigned to a. In this case, the value of `a` after this will be 1. Line 7 contains an alternative way to perform the typecast. This so-called static cast operator is much more powerful as it can operate on a much wider range of types, and especially also on derived types, while the previous case is somewhat limited in scope. It is therefore the preferred version to use. However, for the types of section 3.3 either will work.

Note that as long as a variable has not been assigned any value its contents is undefined, and usually has some random value. It particular, this value can be different for each program run, and therefore no assumptions on it should be made. Variables should never

[8]It is changed to the, in absolute value, next smaller integer value for most compilers.

be used without explicit initialization either when declaring them or by initializing them at first use[9].

In Mathematica, all variables are just declared by using them, and all variables can hold any values, and therefore no type casts are necessary. Assignment to variables are performed by using the equal sign. As in C(++), the value of other variables is copied, and the variables remain distinct entities afterwards.

While in Mathematica just stating the name of a variable followed by Shift+Enter is sufficient to print its value, this is a little bit more complicated in C(++). Using the two variants for C++ and C of section 3.2 to print the value of the variable a of listing 3.2 looks like

Listing 3.3: Variable output

```
1 cout<<a<<endl;
2 printf("%i\n",a);
```

For cout the variable name is used. The handling in printf is more involved. In the text a placeholder for the value of a variable is need. This contains two elements. The first is a %-sign to signify that a variable starts[10]. This is followed by a label to identify the type of variable. Here, i stands for an integer. The documentation of printf provides all possibilities. Afterwards, the additional arguments of printf provide the variables containing the values to be inserted, in the same order as they should be used. In the end, both operations do the same thing: Since a contains the value 0, they print a line containing only 0.

3.6 Basic operations on variables

While it is nice to have variables and store data, it is useful to manipulate and, especially, combine the contents of variables. There are two broad categories of doing so, without invoking the functions of chapter 5. Note that a few more basic operations on variables exist, e. g. the possibilities to manipulate the individual bits of a variable. Though they are not fundamentally different from the operations discussed here, they are somewhat exotic in the sense that they are rarely needed. These will not be discussed here, but more details can be found in the documentation of the languages.

[9] Unnecessary additional initializations are inefficient, and should be avoided.

[10] To output a single %, it is necessary to write %% instead.

3.6.1 Mathematical operations

Mathematical operations are just that: Basic mathematical operations. In Mathematica, they are simply performed by typing them, e. g. a=b-c^2. Note that a space between two variables is interpreted as a multiplication.

In C(++), this is a little bit more involved. Consider the following code, which gives you the most basic ones.

Listing 3.4: Mathematical operations

```
1 a=b+2-3;
2 c=b*a+2;
3 a=(2+3)*a/b;
4 d=sin(c*d)+d;
5 e=log(x)+exp(y)+pow(f,r)-sqrt(a);
```

As is seen, all basic operations are available. Furthermore, certain non-trivial operations, like trigonometric ones or transcendental ones, are also existing, and called in the usual way. However, it requires to include the library cmath by #include <cmath>, and thus like iostream. There is a number of build-in such functions, though there are many libraries available which substantially enlarge them. Which are actually available can be found in the documentation, but these standard operations are always available. Note that there is no dedicated square operator, this is covered by the operation pow(·,2). It is important to note that the fact that the variable a appears on both sides in line 3 is not a problem. Before the value of the variable on the left-hand-side is altered first the whole right-hand side is evaluated. There are also operations to determine the modulus and remain of an integer division, signified by % and ^, respectively. There are also two quite handy operations to increase a value or decrease an integer value by 1, called ++ and −−, so-called unary operators[11]. They can only be applied to an integer quantity. For reasons of mathematics, they have the same precedence as taking the negative of a value, and as a mathematical operation, it does not matter whether they are appended or written in front of the expression to be changed[12]. Note in particular, that these are actually evaluated as a=a+1; i. e. even without assignment they increase the value. Thus the statement a++; increases the value of a by one, even without assignment.

[11] Writing -a to obtain the negative of variable is also considered as a unary operator, though it is in principle just a shorthand for -1*a.

[12] Under certain circumstances it makes a difference for efficiency, as the order of getting the value at the memory position signified by the variable and the actual calculation slightly differ. If such things become important, optimization beyond the level of this lecture are called for. It also plays a role for flow control, see section 4.1.

There are two important caveats here.

One applies again to the type of variables involved. Mathematica always adapts the variables such that they can accommodate any outcome[13]. In C++, this is not the case. Usually, the final variable will determine what is stored, i. e. if in line 4 c is double, but d is float, then any surplus digits will be dropped. This is of particular importance when somewhere integers occur.

To asses the effects, it is also important to understand the precedence of operations. Mathematica and C(++) follow both the mathematical precedence rules for mathematical operations. They also adhere to the rules for parentheses. Therefore, in line 3 the result is 5a/b rather than 2+3a/b. In the same way also the conversions are performed, i. e. a result is always converted to the output required for the next step. Thus, if a is int than b will be converted to int in the first line, as integer numbers are added and the final result is also assigned to an int. A constant can be forced to be evaluated as a float by adding a point at its end, e. g. 2., and as a double by appending an L, e. g. 2L or 2.L, to force conversions to a particular type. Note that even with precedence rules in place always the right-hand side is first completely evaluated before the value of the variable on the left-hand side is changed, e. g. in line 4. This also implies that the left-hand side can never be an expression, but only a variable. All such mathematical operations are not equations, but merely assignments. It is important to understand this difference, even if the assignments look like equations.

This is particular important when it comes to efficient coding and thus the second caveat. Operations involving integer numbers or low-precision floating-point numbers rather than (long) double precision are faster, and should therefore be preferred, if possible. Also, addition, subtraction and multiplication are faster than division. Non-integer powers, including square-roots, trigonometric or transcendental operations are even slower, as internally these are performed using series[14]. Therefore, great care should be taken when performing such operations such that they are not used unnecessarily often.

C(++) offers also a short-hand notation to abbreviate the basic four operations, as shown in listing 3.5.

Listing 3.5: Short-hand notation for mathematical operations

```
1 a=a+2;
2 e+=2;
```

[13]Note that also in Mathematica calculations are done using a fixed (decimal) precision. However, this can be changed arbitrarily using the SetPrecision and N commands.

[14]Usually for different ranges of values different series representations are used, optimized for quick convergence in the corresponding range.

```
3 b=b/(c*d);
4 f/=(c*d);
```

The operations in lines 2 and 4 have the same impact on the variables e and f as the operations in line 1 and 3 on a and b, i. e. adding 2 and dividing by the product of c and d, respectively. This reduces the code length (but not necessarily increases readability), as such situations where the content of a variable is overwritten by the content of the variable itself after performing some mathematical operations by it occur frequently.

Note that a mathematical operation which is nonsensical, as dividing by zero, does not make a program crash if performed on non-integer numbers. Rather, the variable is assigned the special value nan, meaning not a number. Likewise, if a result cannot be stored because it is too large for the variable, since the exponent has a finite maximum size, there appears a so-called overflow, signaled by inf, again a special value. Similarly, an underflow happens if the result is too small to be stored, yielding zero.

3.6.2 Logical operations

An important question in science is if a statement is true or false. Replicating this mechanism in C(++) and Mathematica leads to the so-called logical operations. They also help to formulate now equations, rather than assignments, which can be checked. The basic operations are shown in listing 3.6.

Listing 3.6: Logical operations and the Boolean data type.

```
1 int a=1,b=2;
2 bool c=true,d=false;
3 c=(a+1)==b;
4 c=(c&d)||(!(a!=b));
5 d=(a>b)&&(c^d);
```

The first important new addition is the new primitive variable type bool, which can hold either of two, so-called boolean, values: true and false as logical possibilities[15]. Also Mathematica knows boolean types and, as always, automatically accommodates such values for variables. The values true and false (im Mathematica True and False) are built-in constants, just like, e. g., the number 1.

[15] Actually also integers will be interpreted by C++ as boolean values if used like a boolean variable, where 0 signifies false and all other values true. In fact, boolean variables are internally nothing but usually variables of type char or int, depending on the implementation.

Besides actual boolean values, boolean results can be obtained from equations. Such equations have a left-hand side and a right hand-side, and a comparison operator. During runtime, both sides will be evaluated, and, depending on the comparison operator, the result of this operation is either true or false. In line 3 the comparison operator is the equality operator ==, and therefore the equation evaluates to `true`. This value is then stored in the variable c by the usage of the assignment operator =. Both should not be confused, as == does not perform any assignments. This is also the reason why in line 3 it is possible to have a mathematical operation on the left-hand-side, in stark contrast to the assignments of the previous section. Alternative comparison operators are shown in the following lines, including not equal (!=), greater or smaller (> and <), and greater/smaller or equal (>= and <=). All of these comparison operators are also available in Mathematica.

Logical statements can also be combined by the usual logical and, or, and exclusive or operations, signified in C(++) by &, |, and ^, and in Mathematica by &&, ||, and Xor[*operand* 1,*operand* 2], respectively. Parentheses can be used to arrange the precedence of the operations. Otherwise, these operations do not have any precedence, just like addition and subtraction.

3.7 Pointers

As discussed in section 3.3, the name of variables are actually references to some place in the memory where the contents of the variable is stored. While in Mathematica, or usually in functional languages, this is a hidden technical detail, this is different in C(++). There, it is possible to interact with this information directly. While an advanced topic, understanding, but not expertise, of pointers will be very useful in next section 3.8.2, and they are thus introduced rather early in this lecture.

The information where the storage of some variable in memory is located can be obtained. The answer to this question is obtained in the form of a so-called pointer. In the sense of a von Neumann machine, the memory is represented by a list enumerated bytes, starting with byte zero[16]. The information, at which place a chunk of data is stored is given in C(++) by the & operator, i. e. &a provides a so-called pointer to the memory location referenced by the variable a. As this pointer is some integer number, this pointer

[16]In actuality, the number is not necessarily in a one-to-one mapping to the physical memory installed in a computer. In fact, for many reasons, most notably security issues, this is rarely the case nowadays. However, from the point of view inside a program, and for the purpose of this lecture, the memory available to a program is indeed enumerated in this way.

is usually of size an integer large enough for the system used, e. g. 8 byte on a 64 bit machine. Note that is indeed the same symbol as the logical and of section 3.6.2. The context unambiguously defines how it is interpreted. That has to be kept in mind when reading programs.

Such pointers can themselves be variables, e. g. the declaration int* b implies that b is variable of type 'pointer to an integer', so there is also strict typing of pointers, even though all of them are integer. Thus, b holds a pointer to an integer. To indicate that a pointer is not valid, e. g. because it has not been initialized, the special value[17] NULL exists. Thus, to check whether a pointer contains an address or not, it is possible to check b==NULL. However, this does not mean the address is valid, i. e. the chunk of memory at this address is used. It just tells that this is an address.

Given a pointer, it is possible to read the data at the position pointed to. This is obtained by the * operator, the so-called dereferencing operator. Again, the context and/or the usage of parentheses, will make the interpretation of * as either a multiplication operator or as the dereferencing operator unambiguously.

I. e., given e. g. int *b and if b is not NULL, the operation *b will provide the data located at the memory position referenced by the pointer b. This is also the reason why b needs to have a type, rather than just being a pointer: To read the data, it is necessary to know how many bytes are there, and this is determined by the type. Likewise, it is possible to write to a memory location. E. g. *b=25; will store the value 25 at the memory location referenced by the pointer b.

Pointers are an extremely powerful tool to write highly efficient programs. The reason is that if the memory layout is known, which is, e. g., the case for arrays to be discussed in section 3.8, they can be handled like real integers. Especially, it is possible to perform arithmetic operations on them to calculate other memory locations based on the application logic.

It is also this fact that makes pointers an extremely dangerous feature. One which leads to making them unavailable, or at least quite safeguarded, in many programming languages. The reason is that there is no guarantee that a pointer is actually referencing anything. Even if it is not NULL, it references some location in memory, but holds no information on what is stored there. Even the assigned type, if it is a pointer variable, is only giving an information about how much should be accessed during the dereferencing, but not at all whether this is what is actually stored there is of this type. In particular, in

[17] In more recent versions of C(++), there is also the constant null with the same functionality. There is a subtle difference, NULL is actually the integer value zero, while null does not have such an equivalence, which is important for conceptual reasons. If available, null is preferred over NULL.

chapter 6 means will be introduced to create or remove variables at run-time, making this information even during program runtime volatile. Reading from a location can therefore produce any kind of garbage. Even worse, writing to a memory location using a pointer could scramble whatever data was there[18]. Finding errors due to such a behavior is very complicated, as such an assignment may happen at a completely different, and otherwise unrelated, part of the code or, even worse, may change after every compilation, or even execution of the code.

It is therefore highly advisable to use pointers only if there is a very clear and present purpose and justification to do so.

3.8 Arrays and lists

A situation appearing very often in programming is that a collection of items is required, e. g. a set of numbers. These are treated rather differently in Mathematica and C(++), and called lists and arrays.

It should be noted that arrays and lists are sometimes referred to also by other names, and that, on a formal level, they are not synonymous. Also, other languages may use the names in quite a different way. In day-to-day language, and depending on context, they may also be used (subtly) different. When in doubt, verify what is meant by the respective names.

3.8.1 Mathematica

In Mathematica, a set is called a list, and is defined as {a,b,c,...,d}, i. e. start and end of a list are marked by { and }, and the arbitrary number of elements in the list are separated by ,. Since in Mathematica variables can contain anything, a list can be assigned to a variable. E. g. a={2,4} lets the variable a hold the two-element list {2,4}. There is no limit to the size of the list, except the amount of available memory.

To access and manipulate the elements of a list, it is possible to use a double angular bracket. E. g. in the previous example a[[1]] has the value 2, as it returns the first list element. List elements can also be accessed using variables. If i=1 then a[[i]] gives again

[18]Since even code is stored in memory, it is in principle possible to alter code at run-time, intentionally or unintentionally, by writing to it using pointers. If not done very carefully, this usually results in a crash of the program, or some unexpected behavior. However, for security reasons modern operating systems forbid the change of code during runtime by such an operation, and thus such an attempt usually results in a crash. Modern codes, which should adapt during runtime, will do this not by writing to memory, but by having a build-in compiler, creating executables at runtime, and thus do not need such operations.

2. Also arithmetic expressions can be used to reference elements of a list. The value of list elements can be set in the same way. E. g. a[[1]]=3 will change the list hold by a to {3,4}. To append to a list use Append. E. g. Append[a,5] yields that a contains {3,4,5}. The operation Join[] combines two lists. E. g. Join[a,{6,7}] will then make a hold {3,4,5,6,7}.

Mathematica has a large number of possible ways to manipulate lists. Particular useful is to obtain parts of a list, using the operation ;;. E. g., the operation {1,2,3,4,5}[[2;;4]] yields {2,3,4}. Using [[All]] will return the whole list. Other operations yield the first or last part of a list, or check particular parts of a list.

So far, lists where just list of numbers. However, in Mathematica it does not matter, what is stored in a list. Also, elements do not need to be of the same type, and can even be functions. Thus, a list like {a, "test", x^2} is perfectly valid in Mathematica.

A useful tool to create lists is the build-in function Table. To create a list of the real numbers from 1 to 10 will be obtained from Table[i,{i,1,10}]. The first appearance of i is the actual expressions, which should be put into the list elements. To get, e. g., the squares of the first ten integer numbers, this would be replaced by i^2. The second part is the iterator, i. e. over which range the variable i should run. The syntax is {Iterator name,iterator start,iterator finish,iterator increment (optional)}. The iterator increment permits to change not by the default of +1. E. g., for half-integer steps, this would be Table[i,{i,1,10,1/2}]. The concept of iterator is actually far more general, and will reappear again in section 4.3.

A very powerful feature is that lists can be nested, i. e. any element of a list can itself be a list. Since elements need not to be of the same type, these lists do not need to be of the same size. As these are just ordinary lists, they can again contain lists, and so forth. A valid list would be {1,{{2,3},4},{5,6,7}}. This list has differently sized list elements, including nested lists. Note that an element is not a one-element list. E. g. 1 is not the same as {1}, and as a consequence {1}[[1]] works while 1[[1]] will yield an error message.

To access nested elements the , operator is used. E. g. {{1,2},3}[[1,2]] yields 2. This can be continued for deeper nested lists. Note that this operation can be combined with the ;; and All operations from above. Also Table operations can be nested to create lists. Either this can be achieved by nesting table calls, or by using nested lists in the argument of Table. In addition, Table can use multiple iterators to produce multidimensional lists.

There are many powerful possibilities to sort and search through lists, as well as many other possibilities to work with lists. These will be explored more in section 8.4.

3.8.2 C++

Lists in C(++) is a very different from Mathematica. The first, probably most important, difference is that lists, called arrays in C(++), are strongly typed, i. e. all elements of a

list need to be of the same type. In addition, lists in Mathematica can exist on their own. A statement like {1,2} makes sense. In C(++) an array must always be associated with a variable, which is used to find its place in memory.

As with all variables, a variable holding an array needs to be declared before use. Some possible declarations are shown in listing 3.7.

Listing 3.7: Array declaration and use.

```
1  int a[5];
2  double d[2]={1,2};
3  int f=1;
4  const int b=2;
5  double c[b];
6  a[0]=2;
7  c[1]=1+a[0];
8  a[f]=3;
9  double e[b][3];
10 e[f][2]=c[1]-1;
```

In line 1 of the listing a basic array is defined as a list of 5 integers. The number of elements of an array is given in the angular brackets. The list elements are always counted in steps of one, starting from 0, i. e. the indices of the list elements of a are 0, 1, 2, 3, and 4. There is no alternative to it.

The elements of an array are not having any definite value after the array has been declared. In the end, just like with normal variables, the elements of an array are just the size of the array's memory locations, and their initial value is what every was at that memory location before it was assigned to hold the values of the array. Thus arrays should be initialized before use. An example of how to do so is given in line 2. Here, the array is initialized by giving every array element a value, by listing them on the right-hand side. Note that they are always assigned starting with element 0. Thus, if not enough values are provided, only the elements from zero onwards and up to the end of initialization are assigned values, and any element with larger index is not initialized.

Arrays cannot be declared with a variable size, i. e. it is not possible, using the declaration of line 3, to declare an array `int g[f];`. This will lead to an error during compilation[19]. There are ways to define the size of arrays at run-time, but this needs a different approach to be introduced in chapter 6. Also, once declared, the size of an array is fixed forever, in

[19]Note that this changed over various versions of C(++), and some compilers allow to do such things even if they are conforming to the standard.

contrast to the lists of Mathematica.

It is, however, possible to declare arrays using named constants, as is done in line 4 and 5, as the size of the arrays is then still known during compilation.

To access an element of an array again [] is used, as shown in lines 6 and 7. The value passed gives the index of the element to be used. This can be done either to assign it a value or to use the value, depending on where it is used. Note that it is not possible to assign multiple values simultaneously, like in line 2, after declaration. A statement like a={2,3}; will lead to a compilation error. It is also not possible to assign arrays to each other like d=c;, not even if they have the same dimensionality. All of this is in sharp contrast to Mathematica. Arrays are much more rigid in C(++).

What is possible is to have also multidimensional arrays in C(++), but again all elements have to be of the same type. This is illustrated in lines 9 and 10. Also, the different arrays do not need to have the same size, as the declaration gives an array of 2 elements of type a 3-element array of integers. An initialization is also possible using a nested version of line 2. Accessing elements then requires a sequence of [], as is shown in line 10, where explicit numbers and variables as indices can be mixed. Such arrays are called multidimensional. Note that it is also not possible to assign subarrays to each other, or assign multiple values at a time. The nesting depth of arrays is again arbitrary.

While arrays seem to be thus much more restricted in C(++), there is also an aspect which makes them quite powerful. Though it is not possible to assign something to an array, the name of an array, e. g. a after the declarations of listing 3.7, does have a particular meaning: It is a pointer, as discussed in section 3.7 of type, in this case, int. It contains as value the location where the array in memory starts. In C(++), the arrays are stored in memory starting from element zero onward. In fact, an expression like (a+1)[0] is the same as a[1]. Thus, it is possible to use arithmetics to move inside an array. In multidimensional arrays, the memory allocation is that the right-most index runs fastest, i. e. after the declarations in listing 3.7 the memory layout is e[0][0], e[0][1], e[0][2], e[1][0], and so on. Since pointer arithmetics is not verified, and often not verifiable, by the compiler, it should only be used if one knows what one does. It is easy to create almost undetectable errors. However, knowledge of the memory layout of arrays is extremely important to access multidimensional arrays efficiently. Thus, once one needs to write high-performance code involving arrays, it is necessary to take a closer look at these items.

3.9 Composite variables

While the basic type of variables are quite often covering all needs, it sometimes occurs that a simple type or an array does not quite catch the needs. Especially, it may be desirable to somehow group some information of different types together.

In Mathematica, this can be done by using the lists of section 3.8.1 with multiple entries for this purpose. This is possible by nesting and the use of arbitrary data in a very direct way. This is not as simple in C(++).

To achieve the same purpose, C(++) has a construction, which is called structures[20]. A structure is, essentially, a way of defining a new data type as a composition of primitive data types. The listing 3.8 shows how this is done.

Listing 3.8: Declaration of structs and their use.

```
1  struct complex {
2    long double real;
3    long double imaginary;
4    bool isreal;
5  };
6  complex a,b,c={1,1,false};
7  a.real=0; b.real=1;
8  a.imaginary=a.real+b.real;
9  a=b;
10 complex d[10];
```

In lines 1-4 the composite structure is declared, in this case to represent a complex number, with a real and imaginary part, both being of type long double. There is also a boolean quantity of name *isreal*. After this declaration the new type can be used in essentially the same way as ordinary built-in types, as line 6 with the declaration of two such complex numbers shows. To access the elements of the structure the . operator is used, as lines 7 and 8 shows. It also possible to assign such variables in the same way as for ordinary variables, as line 9 shows. If this is done, every part of the struct, called fields, is assigned the value of the other variable.

Note that, like ordinary variables, composite variables can also be initialized. However, an initialization, like shown in line 6, requires a proper initialization with the correct types.

[20] In fact, this comes actually from C, and in C++ should be superseded by classes to be introduced in chapter 10. However, structures are often a more efficient solution, as they have less overhead than classes. The downside is that they can only store data, but not functionality. Though there are subtleties to the latter statement, which will not be dwelt upon here.

Otherwise, the code will not compile. It is also possible to declare arrays of such composite types, as is shown in line 10. They can be used as ordinary arrays, but need then nested initialization, if they should be initialized.

There are, however, a few things which do not work. Many built-in operations, which work on all of the elementary types, will not work on a structure. This is as the compiler cannot know how to interpret this. E. g., given the declarations of listing 3.8, a statement like *a=b+c;* will lead to an error upon compilation. While it should seem that it is 'natural' how two complex numbers should be added, the compiler does not recognize them as complex numbers. Thus, it does not know how to compute the information, e. g., for the field isreal, as it does not understand the meaning of it, while the human does. In fact, it would not even know that it should add the two fields real and imaginary separately. And before doing something wrong it rather does not do anything at all, and tells the programmer to do it her/himself. How to do so, except by doing it step by step like in line 8, and alleviate the problem will be discussed in section 5.8.

It should be noted that the elements of a struct are ordered in memory as declared. It is therefore possible to use a pointer and pointer arithmetic to work with them. This is not advised as long as you do not know what you are doing[21].

3.10 Persistent variables and (disk) I/O

An important question is how to to communicate with a program and how to store permanently data, as everything created so far is lost after the computer is switched off. In Mathematica, all outputs are actually stored in the notebook file itself (as long as the output is not suppressed by appending a ; to the statement). Thus, here the results are persistent, and because of the interpreting nature input is also straightforward.

This is quite different in C(++).

For the output to screen the operations on cout were introduced in section 3.2. It was somewhat mysterious how it actually operated. This will be expanded on here[22].

[21] E. g., for reasons of efficiency, in some cases some bytes of memory are left empty in between to obtain addresses for every element of a struct to start at an addresses divisible, e. g., by 8. Such aligned memory address are increasing the speed of transfers between the different levels of memory on some machines. Without knowing this, accessing some locations may erroneously lead to such an empty space, which at best contains garbage.

[22] There is actually a second set of operations which perform the same purpose, extending the function printf, also introduced in section 3.2. These functions are inherited from the language C, on which C++ is based, and are in contrast not object-oriented. They will not be discussed here, but are well-documented, and have many similarities to the concepts introduced here, but have also (subtle) differences.

This concept is known as streams. A stream is quite similar to the storage of a Turing machine: It is essentially something into which data is put or taken from. The stream cout is a write-only stream, i. e. it only accepts, using the function <<, data to be put into the stream. The fact that the stuff put into the stream is then printed to the screen is conceptually different from the stream itself. In actuality, what happens conceptually is that the contents of the cout stream is send to a stream which corresponds to the screen, and by this stream ultimately printed. In C++, all streams are descendants of a class in an object-oriented framework. As this is not yet covered, here only how to use such streams will be discussed.

A second important stream is the opposite to cout, the keyboard input stream cin. It is a read-only stream, i. e. it only provides data taken from the keyboard. To signal this, the corresponding function to take data is >>, followed by the name of a variable into which the data should be stored. If there is a variable declared as int a, then cint>>a will read an integer from the keyboard, and the end of the input is marked if the enter key has been used, which therefore plays the same role as endl for cout. Note that hitting enter without inputing anything will result in repeating the request for input.

How does cin knows that indeed an integer has been entered? The answer is, it does not. It just tries to interpret the input as the type of the variable to which the input should be stored. If the actual data input is of a different type, this will in the best of cases result in a non-nonsensical content of the variable, and in the worst case crash the program, as, e. g. data is written somewhere where it does not belong, as cin just keeps on writing into the memory as much data as it gets from the starting position of the variable. If it gets too much then, just as with invalid indices for arrays, it will still keep on writing. It is the job of the programmer to make sure that such problems are taken care of, e. g. by using some type of variable large enough to contain the input.

Knowing now this, it is finally possible to write a C++ version of listing 1.2, which is provided in listing 3.9.

Listing 3.9: "Real code user interaction"

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void)
6 {
7  long double a,b,c;
8  cout<<"Give me two real numbers"<<endl;
```

```
 9   cin>>a;
10   cin>>b;
11   c=a+b;
12   cout<<"The result is "<<c<<endl;
13   return 0;
14 }
```

Note that here no check is made for the input, to avoid a rather lengthy code. The program just trusts the user.

The streams cout and cin were one-way streams, as the things they represent, monitor and keyboard, only have one way for their data. It becomes more interesting when considering files on disk. Files are used to store data beyond the point where the program finishes and/or the computer is switched off, and also to transfer data from one computer to another. They are therefore the central element for the persistent storage of variables[23].

These operations are performed using filestreams. A filestream is used as shown in listing 3.10.

Listing 3.10: Usage of filestreams.

```
1  #include <fstream>
2  ...
3  fstream fout("data",ios::out);
4  fout<<2<<" "<<5<<endl;
5  fout.flush();
6  fout<<"Hello"<<endl;
7  fout.close();
8  fstream fin("data",ios::in);
9  int a,b;
10 fin>>a>>b;
11 fin.close();
12 ...
```

Line 1 includes the necessary library to use filestreams. The dots in line 2 here and hereafter indicate that some code follows, in this case the rest of the start-up of the

[23]Even though many modern devices act like programs are never shutdown, this is internally also managed by having a kind of persistent memory, on which they act. However, this kind of memory is at the current time unsuitable for the necessities of many applications in physics. And it still does not solve the problem of transferring data. The latter could be solved by networks, but this is again not a sufficiently efficient choice yet. And also in its complexity beyond the scope of this lecture.

program, and then the following fragment comes. A new filestream is then declared in line 3. It is created by a special function, a so-called constructor, which will be discussed more in chapter 10. This function takes two arguments. The first is the name of the file, which requires a fully qualified path if not within the same directory. The second is, how the file should be accessed. This is a particular constant, where ios:: is again a namespace qualifier, to be discussed in more detail in section 7.4, but conceptually identical to std of section 3.1. The second part out is determining that it should be written to the file, and the file should be created, if it does not exist. If it should be opened for reading, the qualifier must be ios::in. If both should be performed, it is possible to use[24] ios::in|ios::out. Note that for ios::in it is required that the file exists. To check whether the filed could be opened, e. g. to check whether it exists, fout.good() can be called - this will return 0 (false) if the opening was not successful and 1 (true) otherwise.

In lines 4 and 6 text is written to the file[25]. The endl creates a new line in the file. The command in line 7 closes the file, and no further output is possible.

Line 5 is a very particular command, and has something to do how modern machines operate when working with files. As mass storage is notoriously the slowest part of the storage chain, the operating system often makes a copy of a file in memory, and works on it, to accelerate access to it: It caches it. Unfortunately, memory is not permanent, and its content can be lost if the program or the machine is stopped unexpectedly. In this case, all contents of the file only existing in the cache is lost. For scientific purposes, this is a serious problem, as programs often run for very long times. Thus, to avoid such problems, the command flush forces the operating system to write all cached changes of the file to the persistent storage, avoiding such problems. It is therefore highly recommendable to do, if it can be foreseen that the file is not changed for a while and/or critical or valuable information has just been written to the file. For the same reason, a file should always be closed using the close command, as this will also commence all changes to permanent storage, though this will also be done automatically at program termination.

To access the content of the file is essentially like the case of cout and cin. This is shown in lines 8-11. The file is opened again, and the first line is read again, where the numbers 2 and 5 are thereby stored in a and b, respectively. The intermediate space or a endl will be ignored. However, fin does not check the type, and just assumes that everything is fine. If therefore data is read and attempted to store in a variable of wrong type, it will do so. In the best of cases, this will lead to a non-nonsensical content of the variables. In

[24]The operation | is a so-called bit-wise or, and combines logically the zeros and ones which makeup the two integer values.

[25]The data is actually stored as text. If one wants to save the data as the binary data it is, add |ios::binary when opening the file.

the worst case, it will crash the program. Similarly, it should be checked, whether the file still contains data. This can be checked using fin.eof(), which will return true or false, depending on whether something is left or not. Attempting to read beyond the end of a file can also lead to a crash.

It is often useful to skip ahead or move back inside a file. This is achieved by the seekg function. Its argument is the absolute position in bytes where the stream should move to inside the file. Note that this requires knowledge of the structure of the file to make reasonable use of. To do the same while writing to a file, use seekp.

Also in Mathematica it is sometimes useful to write and read from different files than the notebook. In physics, this happens very often if the output of some other program or experiment should be further processed with Mathematica. This can be done using the functions Import[$filename$], yielding the contents of a file, and Export[$filename$,$data$], where the argument is the (qualified) filename in "". The *data* is, what ever should be written as content of the file. Usually, this will be a list of some sort. Mathematica attempts to interpret the contents of a file correctly. In doubt, it will create a list of the elements found in the file. It is possible to force Mathematica to use a particular, predefined structure when interpreting files. These built-in possibilities are listed in the documentation of Import and Export.

3.11 Characters and strings

In section 3.3 already a type for a single letter, decimal, mark, or the like, has been introduced[26], the character char. However, it can only store a single character, but throughout the previous sections many cases of sentences have appeared, e. g. in line 7 of listing 3.1.

In Mathematica, such a sentence is a so-called string, an arbitrarily long sequence of characters. To identify such a string, it is enclosed in "". As always in Mathematica, it can be used like any other type of variables.

The situation is somewhat different in C(++). There are actually two options.

One is inherited from C. There, a sequence of characters is nothing but an array of characters. In particular, it has a fixed length, even if its contents changes. Therefore, there exists a special end character, usually written as \n, to signal that the remainder of the array is not part of the sentence. Thus, adjusting the meaning of it being a sentence is part of the program logic provided by the programmer, and not inherent to the way it is treated by the language. Still, many methods are available to manipulate arrays of

[26]Modern systems often use substantially enlarged alphabets, with up to 65535 different characters, so-called Unicode. This has implications for the size of this variable type, therefore check if in doubt.

characters, but they are oblivious to the fact that it is actually a sentence. For their purpose, it could as well be an array of numbers. The consequence of this is that all caveats relevant to arrays also apply. All in all, it is a very efficient, but error prone and tedious, way of handling sentences.

The alternative is a special type string. It is actually a class, as discussed in chapter 10, but this is hidden for most purposes. Internally, it implementation works also with a suitable array of characters, but it acts like a string in Mathematica, and can be used like an elementary type. This is demonstrated in listing 3.11.

Listing 3.11: Usage of strings.

```
1 #include <string>...
2 string a="Hello",b;
3 b="World!";
4 a=a+" "+b;
```

After including the necessary library in line 1, two string variables are defined in line 2. The variable a is initialized. As in Mathematica, the value of a string is given by what is within the "", while the delimiters are not part of the string[27]. The second variable then gets a value in line 3. Line 4 demonstrates that it is possible to add (concatenate) strings with the +, and that a is not restricted to the length it had at initialization. It also shows how constants can be added. The final contents of a is Hello World!. The class string contains many more ways to manipulate the contents or get information about it, e. g. whether it contains certain substrings. It is also possible to get a copy of the internal characters of the string with the function c_str, e. g. a.c_str(), as this is sometimes necessary.

The string class can be used like any ordinary other variable, i. e. it can be used in file operations or arrays can be created of it.

3.12 Type casts

In Mathematica it is always possible to transfer the contents of a variable at any time to any other variable. Mathematica will then adjust the type of variable dynamically to accommodate its new content.

This is not so in a strongly typed language like C(++). It is, e. g., not possible to assign an integer to a string or a real to an integer. However, it is possible to recast a

[27] Use \" to have a " in a string. The preset \ always indicates a special character. E. g. a line-ending would (also) be \n.

type, i. e. to reinterpret its content. If the type to which a variable should be assigned to is a superset, this does not need any particular effort. It is always possible to assign an int to a double, or a double to a long double. If information is modified or cannot be copied verbatim, it is necessary to provide an explicit typecast. In this case, the variable is reinterpreted as if.

There are again two possible ways of how to do so. Consider, e. g., the need to assign a double to an int. Mathematically, this requires to deal with the fractional part. Given that the double is variable b and the int is variable a, this can be achieved by either a=(int)b or by a=static_cast<int>(b). In both cases, the fractional part is just cut off[28]. The former method is inherited from C. While it works for basic types, it does not offer a very flexible way of extending it to objects and classes. The second option stems from the concepts of templates, which will be discussed in section 10.11. It is more flexible, and the preferred version to use in C++, but yields the same results when operated on trivial data types.

It is strongly recommended to always perform explicit type casts, even if implicit typecasts in the given case work, as then the compiler can make some amount of sanity checks. Since the compiler otherwise blindly trusts the programmer, a problematic typecast can otherwise yield hard to detect serious problems.

Note that it is possible to also typecast the pointers of section 3.7. This changes how data is read from and written to the memory location when dereferencing the pointer. If the type of data at the location is not of the assumed type, this will yield garbage at best and a crash at worst.

3.13 Documentation

Now, there are many variables with names and properties. How to find a way among them?

The probably most neglected part of programming is documentation. Any programming language offers, in various degrees of sophistication, the possibility to document the code within the code itself. I. e. there are statements which are ignored upon execution, and therefore only serves the reader of a code, but not the user.

It is commonly assumed that in a well-documented code is about a third to a half documentation.

This documentation serves the purpose to make the code understandable not only to other people (a situation very often happening in physics with the high rate of new

[28]The function round performs mathematical rounding, and there are also other ways to deal with the fractional part, like determining the closest larger or smaller integer.

master and PhD students becoming involved with the code), but also to the programmer herself/himself. After a couple of years understanding one's own code can become as challenging as understanding the code of others.

In fact, the pseudo-code in the listings 1.1 and 1.2 can already be considered as a first step towards a documentation of the final codes.

Besides the documentation of the code inside the code itself, there should also be two other documents describing the code. One should document the definitions and interfaces of the code, and should be aimed at other programmers, who will reuse the code and continue developing it. The other should be discussing how to use the code for its intended purpose, i. e. how to apply it. Today, many tools are available which can extract suitable marked parts of the documentation inside the code to build at least a skeleton of these two documents.

Chapter 4

Flow control

So far, the programs started from a starting point, being that initial values for variables or user input from file and keyboard, to some output. However, it was not possible to react on what is going on, it just continued the program without alternatives. That is, of course, unsatisfactory. A program should be able to react differently on different inputs. This is achieved by the so-called flow control. There are various possibilities, which will be introduced in turn.

Flow control works rather similar in both Mathematica and C(++), and is especially conceptually of the same mind-set.

Generically, all kinds of flow control are based on logical statements, i. e. statements which can be either true or false. To formulate them uses heavily the logical operations of section 3.6.2: If something is true do this, and if it is false do something else.

4.1 Binary if

The simplest kind of flow control is a binary decision: If yes do something. If no do nothing or something else. The basic structure of this binary if in C(++) is

$$\mathsf{if}(\mathit{condition})\{\mathit{statements}\}\ \mathsf{else}\ \{\mathit{statements}\}; \tag{4.1}$$

How this works is exemplified in listing 4.1. Assume for it that the variables a and b are ordinary int, and have been initialized.

Listing 4.1: Binary if.

```
1 if(a==2) cout<<c<<endl;
2 if((a>2)&&(b==3)) cout<<b<<endl;
3 if((a==2)||(b==3)) {
```

```
4   int d=a+b;
5   cout<<d<<endl;
6   d=a-b;
7   cout<<d<<endl;
8  };
9  if(a==2) { a++; cout<<a<<endl; } else { a--; cout<<a<<endl; };
10 a=2;
11 if((++a==2)&&(b++==3)) cout<<a<<endl;
```

Much of the general form is actually optional. The simplest case is shown in line 1. Here, the condition is formulated as the logical test a==2, i. e., whether the value of a at the moment is 2. It does not matter what the value of a is before or after this. If this is true, the following statement, putting the value of c to screen, is executed. If not, nothing happens. Note that what is done is free, and does not need to have anything to do with the condition. This shows that the else part of the definition (4.1) is optional. It is not necessary to provide an alternative reaction, the default is simply to do nothing if the condition is not fulfilled. Also, a single statement does not need an enclosing block. It is not possible to leave out the first statement. Rather, if it is necessary to not do anything if something is fulfilled, but otherwise do something, the test should be on the negated condition using the logical not operator !.

Line 2 of the listing shows that the conditions can be quite involved, but as long as they are delivering ultimately either true or false, this can be as complicated as desired or required.

Lines 3 to 8 show that also more than just one statement can be conditionally executed. However, in this case they need to be enclosed as a block, similar to section 3.3. It is then, as done in line 4, even possible to define new variables, which will only exist inside this block, but can be used freely within. I. e., it is no longer possible to access the variable d after line 8. There are also issues with this, also applying to the following sections of this chapter, to be detailed in section 5.3.

As line 9 shows, it does not matter how the lines are arranged in a block, and all statements can be written in one line, or many. The additional new thing in line 9 is the else-branch: If the condition is true, the block before the else is executed. If it is not true, the block after the else is executed. In this case, if a is two, a is increased before output, and otherwise decreased.

Line 10 and 11 show an important trap. Conditions can also involve statements which alter variables. However, the evaluation of the condition of the if is not continued, for efficiency reasons, if it cannot become true. It is now important how the ++ operation

works, which was introduced in section 3.6.1. If it is before the variable, the value of the variable is first increased, and then provided. If it is appended, the variable is first evaluated, and then increased, and likewise for the -- operator. Thus, in line 10 the content of a is first increased, and then used to evaluate the condition. Thus, the condition fails, as it can no longer evaluate to true. As a consequence, b++ is not executed, and therefore the value of b is not changed. If, on the other hand, the statement would be a++, the value of a would first be used to evaluate the condition, and then the value of a would be changed. Thus, the first part would be true, and b would be changed, and then evaluated to test the condition.

This is an example of a so-called side-effect. Such side-effects can be used to write extremely efficient code. However, it is very hard to avoid mistakes, and therefore only recommended, if you know what you do.

It is an important feature of a conceptually pure functional language that such side-effects are not possible. Mathematica is not so conceptually pure. Its version of if

If[*condition*,*statement*,*else statement*]

works in the same way as the one in C(++), except for the little bit different syntax. Note that variables in Mathematica stay once used, and thus if a variable is declared inside the statements, it is afterwards still available. This is the major difference to the C(++) case. Furthermore, if evaluated, the value of this If statement in Mathematica equals the last result it produces inside its statements, acting like a mathematical statement. Thus, If[a >2,3,c=4] will yield 3 if a is greater than two, and otherwise set c to 4, declaring it if it does not yet exist, and will yield 4. To use multiple statements in the block of the Mathematica If use ; to separate them. Thus, the use of , and ; is here reversed between C++ and Mathematica, though such problems are syntax problems and therefore immediately recognized by either the compiler or Mathematica.

The ordinary if of C(++) does not yield a result. However, there is a version, which does so, a so-called trinary operator (in contrast to binary, like +). It has the syntax

(*condition*)?(*statement*):(*else statement*)

As an example, a=(b>3)?(0):(1) whill give a the value 0 if b is greater than 3, and otherwise 1. Again, the rules for evaluation order apply.

It should be noted that if statements can be arbitrarily nested. E. g. in a condition for a C(++) if trinary ifs can be used, though note that still the rules for evaluation ordering apply. Also, in all blocks an arbitrary nesting of if statements are possible. This is true, both for C(++) and for Mathematica.

4.2 Switch

It happens quite often that it is necessary to not only decide between two possibilities, but between many. It is possible to do so with a suitable chosen sequence of nested if statements. However, because it happens quite often, an additional construct exist to simplify some of these cases[1]. This is the so-called switch statement. An example of how it works is given in listing 4.2.

Listing 4.2: Switch statements.

```
1  const int b=1;
2  switch(a) {
3   case 0: cout<<a; break;
4   case b: cout<<a; cout<<endl; break;
5   case 2: cout<<(a+1)<<endl;
6   case 12: cout<<1<<endl; break;
7   case -3: case -1: cout<<endl; break;
8   case 4...11: cout<<"All"<<break;
9   default: cout<<"None"<<endl;
10 };
```

The starting point is line 2. The switch statement requires an integer variable on which to operate, whose name is given in the parentheses. After this, the block has a fixed structure, it is a sequence of case statements, followed by an (optional) default statement, after which the block is closed. Thus, the switch statement provides the possibility to do something depending on the contents of integer variables, which are then enumerated, though not necessarily in a particular order. As line 4 shows, also constants can be used for this, but not variables[2]. Every case consists out of one (line 3) or more (line 4) statements, which are terminated by a break command. If such a break does not exist, the switch will continue to execute. I. e., if a=2, not only will line 5 be executed, but afterwards also line 6, even if a is different from 12. This will be continued until either a break is encountered (here in line 6) or until the end of the switch statement is reached (here in line 10). It is possible to give several values the same code, which is done in line 7, where two different values do the same thing, or in line 8, where are range is used[3], i. e. the same will be done for

[1] In actuality, it also makes these cases substantially more efficient. This is also the reasons for the restrictions applying to it, as such an increase in efficiency would otherwise not be possible.

[2] To give the constants a useful framework there exists also a so-called enum type, which is essentially an array of mappings of names to constant numbers.

[3] Ranges are common, but not part of all versions of C++. Some compilers may therefore reject them.

a containing any value between, and including, 4 and 11. If no case matches, the default will be executed, if it exist. Otherwise, nothing happens.

There are a few caveats. A case cannot be doubled, also not inside a range. While the value of the variable, in this case a, can be manipulated in a case, this will not change anything, as the destination for the switch is chosen before the statements are performed. The switch can be performed also based on a numerical expression. Note that also logical expressions are interpreted internally as integer, and could be used. But in this cases an if is the recommend procedure. It is possible to construct unreachable code. If, e. g. a is an unsigned integer, the code in line 7 will never be executed[4].

Also Mathematica has a switch operation, which has the syntax

Switch[*condition,value,statement,value,block,...*]

Where condition is evaluated, and then the statement is performed which follows the first value fitting the condition. Note that it is more flexible, as both the condition and the value can take any form. This indicates that in Mathematica this is rather for convenience to avoid nested ifs, while its performance aspect is much more relevant in C(++), as well as the strong typing.

4.3 Loops

Another kind of flow control is a conditional repetition of some instruction(s). The simplest possibility is, e. g., to add all numbers up to a given value, or to calculate a faculty. However, loops play a very important role for much more complicated behaviors. If, e. g., your computer is expecting your input, it is in truth somewhere deep down some loop, checking the keyboard or touchscreen, whether something has happened, until something happens.

For this purpose three kinds of loops exist. While it is (almost[5]) possible to write any loop with any of the three constructions, this is not the intention. Each of these loops replicates some particular kind of problem, and by using the most appropriate construction the program becomes easier readable, as the intention of the programmer can already be gleaned from her or his choice of loop construction.

[4]Note that compiling using the -Wall and -pedantic option would point out this as a warning to you, but otherwise not.

[5]Actually always if the possibilities of 4.4 are used.

4.3.1 For loops

The idea of a for loop is essentially to replicate the mathematical summation (or product) construction, though its possibilities are far more powerful. Its syntax is

$$\mathsf{for}(initialization;condition;modification)\ block;$$

Thus, there are four relevant parts. The simplest is the block: This is either a single statement or some collection of statements collected as a block, which then requires a set of {} to open and close the block, so very similar to the if case in section 4.1.

This block is repeated until the condition is fulfilled. Since the condition is evaluated before the block is executed for the first time, it can happen that the block is not executed at all. The condition is a logical statement, and has to evaluate to either true or false, but can otherwise be any kind of arbitrary complex logical expression, and can e. g. even involve trinary ifs. Before this condition is evaluated for the first time, the initialization is performed, which can be any statement, as long as it is a single statement. It is even, and in fact quite common, to declare a variable in there. This variable is then known in both the condition and the modification, as well as in the block. The modification is finally a statement, which is executed after the end of the block, but before the condition is evaluated. As an example, listing 4.3 shows how to calculate the sum up to a given number and the faculty and the odd faculty of the same number.

Listing 4.3: Examples of the for loop.

```
1 int n=5;
2 int sum=0,fac=1,fac2=1;
3 for(int i=1;i<=n;++i) sum+=i;
4 for(int i=n;i>1;--i) fac*=i;
5 for(double i=1;i<=n;i+=2) fac2*=i;
```

This shows that the variable i, the so-called counting variable, can be reused and redefined in the different loops. Lines 3 and 4 illustrate that the modification can go any way, and line 5 that it is not necessary to just do a simple increment, even though this is by far the most common case. As is also visible, the counting variable is available within the block of the for loop. It would not be available outside the for loop.

It should be noted that the for instruction part can be modified in the block. E. g., it would be possible to write

```
for(int i=0;i<n;++i) { sum+=i; n-=2; };
```

In this case the condition for the termination of the loop would be changed during the execution of the loop. It is also possible to manipulate the counting variable. E. g.

```
for(int i=0;i<n;++i) { sum+=i; i=(i>5)?(n):(i); };
```

would terminate the loop if i should become greater than 5, even if n is greater than 5. As a consequence, parts of the for control can be left empty, e. g. a for loop without modification or initialization is possible. Also the condition can be empty, though this implies that the for loop runs forever, except by forceful interruptions to be introduced in section 4.4.

This possibility to manipulate the behavior of a for loop in the block provides both tremendous flexibility and, at the same time, a huge potential for unintended side-effects, which are notoriously hard to track. Especially, even though this is not the case in the examples in listing 4.3, the blocks can become quite lengthy, and especially in combinations with the functions of chapter 5 quite involved. It is therefore recommended to not manipulate the control part of a for loop as long as one does not has a very precise idea of why one is doing it, and no alternative way is practical.

Again, also Mathematica provides a very similar for loop with the syntax

For[*initialization*,*condition*,*modification*,*block*]

It operates just like the C(++) loop, but for the increase of flexibility in all parts as with the Mathematica versions of If and Switch.

4.3.2 Do and while loops

It happens that a counting variable is not necessary when performing a loop. Rather, it is sufficient to perform a loop until some condition is met. For this purpose C(++) offers two possibilities,

while(*condition*)*block*;
do{*block*}while(*condition*);

The only difference between both constructs is that in the first case the condition is checked before the first execution of the block. Therefore, the block may never be executed. In the second case, the block is executed at least once, because the execution comes before checking the condition. Thus, the do version could be considered to just help in structuring programs, but may also be useful to optimize the last bit of efficiency. Still, it is highly recommend to use the kind of loop which actually suits the purpose, instead of just using always the same type of loop.

Note that in both kinds of loops it is necessary that the block influences the condition, as it otherwise will remain forever either true or false, and thus the block is repeated always or never. Thus, the flow control of these kind of loops is conceptually different than for for loops, where the flow control is intended to be outside the block, and not inside.

Mathematica has also both a Do and a While loop, but they work a little bit differently than in C(++).

The Do loop has a number of alternative syntaxes

Do[*block*,*number*]

Do[*block*,*counting list*]

In the first case the block is done number times. In the second case, this is steered by one or more comma-separated lists. Each list has as a first entry the name of a counting variable. If there is a single second entry, e. g. $\{i,n\}$, the counting variable will take values from 1 to n in steps of 1, and every time the block will be executed. In case $\{i,n1,n2\}$, it starts from $n1$ and goes to $n2$ in units of 1. In case $\{i,n1,n2,step\}$ in does so in steps of size *step*. This is thus rather like a For loop. Finally, it can be $\{i,list\}$, and then i takes the values given in the list in the order of the list, when every time the block is executed. Thus, the Mathematica version of the Do loop does not check for a condition, but works through a predetermined range.

The While loop has the syntax

While[*condition*,*block*],

and tests for the condition in the beginning, and repeats the block until the condition is no longer true. Is is thus very similar to the do loop of C(++). In contrast to the For and Do loops it is in Mathematica only the While loop where the flow control is intended to reside inside the block.

4.4 Flow interruptions

As already indicated in section 4.1 there is a possibility to alter the flow control of loops, and actually also blocks, in C(++). There are four such possibilities.

The simplest one is continue;. It can be used inside of any of the loops and makes the program skip the remainder of the block and let it continue at the next test. It is thus a valid way to, e. g., avoid an if statement, or a nesting of if statements inside a loop.

A more drastic operation is break;. It already appeared in the switch statement of section 4.2. It can also be used inside a loop. Its effect is that the loop is terminated, i. e.

the remainder of the block is skipped, and the program will continue after the loop. It has thus the same effect as a continue where afterwards the termination condition is found to be true. Note, however, that the condition is not necessarily true; the program just acts as if it were true.

The next one is the return statement, which has syntax

return *optional value*;

It leaves the current function, a concept to be introduced in chapter 5, which are essentially subprograms. They are thus terminated by this statement. If it is used inside the main part of the program, like in section 3.1, it will terminate the program. It can be optionally passed a value, which, however, is restricted by the (sub)program it is returning from, as discussed in chapter 5. If this is the main program, this must be of the same type as written in front of main. E. g. in line 1 of listing 3.1 this is int, and thus the value must be an integer. The only alternative for main is void, which requires no return value at all. The returned value in the main program is given to the operating system. Usually a value of 0 indicates a normal program termination, and any other value a problem. This depends on the operating system. If the return value is void the operating system will always assume a normal program termination.

Finally, there is the command

exit(*integer value*);

which will terminate the program immediately, and pass the provided value to the operating system, which will interpret it as above for return. The command exit is usually not used for a controlled end of the program, but rather a panic shutdown because the program found itself in a situation precluding normal operation. The intended end of the program is reaching the end of the main program. However, this is convention, but considered good style[6].

4.5 Debugging

With these new possibilities come also new problems.

Just like calculations, codes are rarely correct in the first go, especially if they are a little bit more complex. The process of correcting errors is called debugging.

[6]Concerning style: In some languages there exist possibilities to actually jump directly to a particular line of code during program execution. These so-called goto statements are considered to be not only bad style, but, in fact, dangerous, and should be avoided.

The basic paradigm to avoid errors as good as possible is “Test early and test often”. Testing means here to use the program, or some chunk of it, feed it with input for which the answer is known, run it, and verify whether the result is indeed the one expected. In addition, it should be feed with random input or even intentionally invalid data to see, whether it deals with potentially problems satisfactorily. Ideally, this is automatized and repeated whenever there has been a change to the code.

If an error shows up, locating it is probably the most annoying task there is. The main reason for this is that errors usually do not arise close to where they surface. Furthermore, as will be seen, there a plenty of possibilities that an erroneous behavior observed in one part of the program is generated by an error in an entirely different part of code, which then sets a chain reaction into motion culminating in this erroneous behavior. The main reason for this is that the data is shared among the whole of the program. Unintentional changes to the data in one part of the code can therefore entail problems elsewhere.

There is a number of preemptive possibilities to avoid these kind of problems. One is to code as restrictive as possible. As an example, it will be possible to assign a chunk of data the type of data it can hold, e. g. an integer number with or without sign. Choosing the most restrictive possibility still compatible with the purpose will reduce the likelihood of having an error because the data is erroneously set to a value not sensible for the purpose.

Also, most systems for coding offer the possibility to check code automatically at various stages. It is useful to set the warning level for these automatized checks to the most sensitive ones, even though this will force one to deal with a great number of constraints and especially demands quite a lot of discipline during coding. Taking all warnings serious helps to avoid errors before they arise.

Finally, always code cleanly. Avoid hasty solutions. Time saved in coding is usually time lost doubly in debugging.

Chapter 5

Functions

In principle, everything is now available to write a program. However, the resulting code would be a very lengthy thing. Especially, if parts of the code is needed in different places of the program, they would be needed to be repeated at every place. All of this leads to a structure which is, derisively, called 'spaghetti code'. While its aesthetic value may be discussed, it is definitely very hard to maintain. It is also prone to errors, since if in a repeated part of the code an error occurs, it is necessary to correct it everywhere.

To avoid these problems, a whole sequence of programming paradigms has been developed, of which the first is introduced in this chapter, and other follow in later chapters. The one to be treated here is called function-oriented programming.

The idea behind it is that code which is reused should exist only once. Reused can be here twofold. One is actually identical code. The other case is code where the operations are the same, but the affected variables are different. Think of addition. E. g. $a+b$ and $c+d$ are the same operations, but work on different variables.

This is achieved by putting reused code into subprograms, the functions already allured to in section 3.1. To achieve the ability to operate on different variables, these functions will have the possibility to work on passed variables, like the arguments of the addition above.

5.1 Imperative languages and code reuse

All of this can be put into a more formal context.

The basic program structure exhibited in section 3.1 is what is known as an imperative language: It executes statement after statement. While this seems to be the logical thing to do to solve a problem, this has one serious drawback.

Consider the situation that to solve a problem, you have to do things often. Consider

for example that you have some function, which is not known in closed form, e. g. the Γ function, but only in a series representation, which can be used to approximate it by a finite sum. If there is now a problem in which this function has to be evaluated multiple times, the simple-minded approach of statement after statement runs into trouble: Should it really be necessary to replicate the statements needed to calculate the sum multiple times?

In fact, in the first imperative languages, this was the case. Fortunately, it was early on discovered that this would be a tremendous problem, and that it would be much better to create some way how to reuse the code, instead of rewriting it. This was achieved by creating functions. Functions are chunks of code, which can be executed using variables, and which pass their result back. In a bit of pseudocode, this looks like

```
1 calculate_sum(argument)
2   do the sum for argument;
3   return the value
4 end of function
5
6 statement ...;
7 a=calculate_function(2);
8 b=calculate_function(3);
9 statement ...;
```

In this example, a function has been defined, which does the sum. It takes an argument, e. g. the value for which the Γ-function should be calculated for, does this, and returns the value, which is here symbolically given by assigning the function to a variable. This is reminiscent to a a formulation in mathematics, which also explains the name 'function' for this action. This function is then called during normal program flow twice for different arguments, without needing to replicate the code. This is the basic starting point of code reuse. It transforms also the language from an imperative language to a function-oriented language.

Code reuse has become one of the major topics in programming, and efficient code reuse is at the heart of essentially all computing today. Without code reuse, modern technology would be impossible.

5.2 Basic functions

Start once more with C(++). So far, all code had to be within the block after main, as introduced in section 3.1. Functions are not, but they are outside, and before, main in the program. In fact, they need to be written before they are used for the first time in the code[1]. The syntax is

$$\textit{return-type}\ \textit{name}(\textit{arguments})\ \{\textit{block}\}$$

There are therefore four relevant parts.

The simplest is the block. This is just an arbitrary sequence of statements. However, as always, any variable declared inside this block is only valid inside the block. Importantly, if a variable is given a value inside the block, it will not keep this value after the function ends. This will be discussed in more detail in section 5.3.

The name, which has to adhere to the same restrictions as variables in section 3.3, needs to be both unique and to be different from any other function's name. If there are variables known in the same context, or scope, as the function, the name of the function may also not be the same as that of any of these variables. Other then this, the name can be chosen freely. However, it is strongly advised to have a name which somehow represents the purpose of the function. Also, if there exists multiple functions all associated with a similar purpose, it is also good practice to make this clear by choosing some part of the name similarly. E. g., if there is a function to add two matrices and one to multiply, a possibility would be matrix_add and matrix_multiply.

The return type is a possibility to yield a result, which can then be used. This will be explained in more detail in section 5.5. If the function should not return any value, the return type needs to be void.

Similarly, the arguments is a list of variables, which can be passed to the function to work with. This is will be discussed in more detail in section 5.4. As before, if no variables should be passed, this must be declared void. I. e. a function which neither returns a value nor needs input would be declared as void do_something(void) {...}.

To use a function, it is used in the program. Consider listing 5.1.

Listing 5.1: Example for a function.

```
1 void print_hello(void) {
2  cout<<"Hello"<<endl;
3 }
4
```

[1] There is a possibility to circumvent this, as described in section 5.7.

```
5 int main(void) {
6  for(int i=0;i<5;++i) print_hello();
7 }
```

Here a function is used to print five times hello to the screen. The function is called by its name, and since it has no parameters, it is followed by ().

Functions in Mathematica, as a partly functional language, are a somewhat more involved topic.

Purely mathematical functions can be defined as

$$name[v1\_,v2\_,...]=expression(v1,v2,...)$$

The name is again a valid name, following the same rules as for variables. The list of arguments are then arbitrary variable names, where each name is followed by _. The expression is then an arbitrary mathematical or other expression, which involves the variables, but here used without the _. The function is called as $name[x1,x2,...]$, where the variables can now have arbitrary names. Consider as an example

$$f[x\_]=1+\sqrt{x}$$
$$f[a]$$

where the first line is the definition, and the execution in the second line will yield $1+\sqrt{a}$, while f[2] will yield $1+\sqrt{2}$. This is very useful to define more complex mathematical functions. This will be explored in more detail in section 8.1.

It is also possible to define functions in the same form as in C++. The syntax is

$$\mathsf{Function}[arguments, body]$$

where multiple arguments are passed as a list, and the body can use the name of the variables in the list (and of all variables declared before this). However, to use the function later, the result of this operation has to be assigned to a variable, which then formally keeps the definition of the function as a value. This is the difference between function-oriented programming in C(++), where functions are independent entities, and a functional language like Mathematica, where functions are the result of an expression, which can be assigned to a variable, or just executed. Such a function is also called a pure function.

To call such a function, it is necessary to use the variable, providing the argument list for the case of a delayed executing. E. g.

$$c=1$$
$$a=\mathsf{Function}[\{x,y\},x+y+c]$$
$$a[d,b]$$

will yield d+b+1. If d and/or b have actual values, it will yield the actual value. It is also possible to evaluate the function directly, by appending the arguments list, just like for the variable, to the function definition. This is a consequence of the idea that functions are just expressions.

It is possible to also define functions in another way, but to the same effect. It has the form

$$(body(\#1, \#2, ...))\&$$

I. e., the body is written without an explicit parameter lists, and the parameters are implicitly labeled in their order by #1... E. g. a=(#1+#2+c)& would do the same as the declaration above, and would be executed in the same way.

5.3 Local and global variables

As noted already in chapter 4, variables in C(++) are not eternal, but are only valid within the block, i. e. set of braces {}, within which they are declared. They are therefore called local variables.

This has several important consequences.

First, they cannot be used outside their block, in which they are declared. But if a block is nested inside a block, e. g. by two if statements, they can be used inside this nested block. Hierarchically, they are available within every block, which is declared inside the block they are declared within.

This 'inside' is with respect to the position of the braces quite literally. A variable declared is not usable inside a function called in the same block. The reason is that the block of the function is not declared inside this block, except if the function declaration itself appears within the block, which is in principle possible, but usually considered bad style[2].

Second, if a variable is declared inside a block, and a variable of the same name exists outside the block, the variable inside the block overrules the one outside the block, it shadows it. Thus, when accessing a variable of such a doubly used name, the innermost variable is used, and the outer variable(s) remain unchanged, and not directly accessible. While this is a logical necessity to make meaningful programming in large projects possible, it is still error prone. Therefore, if possible, multiple uses of the same name in nested blocks should be avoided.

[2]Exceptions are anonymous functions, i. e. function declared without name, but these are quite advanced constructs to be skipped here.

There is also the possibility to declare a variable outside of all blocks. Such a variable is called a global variable. Its validity is thus being available everywhere, unless not shadowed by a variable in a local block. Global variables are usually considered bad style. However, as their position and existence in memory is not subject to exiting or leaving blocks access to them is very quick. Thus, when it comes to efficiency, global variables are useful. However, because of their unlimited visibility, they can be modified in very different parts of the code. This can lead to very hard to track errors.

In listing 5.2 all of these visibility issues are exemplified.

Listing 5.2: Visibility of variables.

```
1  int a; //Global variable, version 1
2
3  void fun(void) {
4   //Here, version 1 is visible
5   int a; //Version 2 of the variable, local to this function
6   //Here always version 2 of a is visible
7   for(int i=1;i<2;++i) {
8    int a; //Version 3
9    //Here always version 3 is visible
10  };
11  //Again version 2
12 }
13
14 int main(void) {
15  //Here version 1 is still visible
16  int a; //Version 4
17  //Here version 4 is visible
18  for(int i=0;i<5;++i) {
19   int a; //Version 5
20   //Here only version 5 is visible
21   fun(); //Inside the function again version 2 and 3 are visible
22   if(i==1) {
23    int a;//Version 6
24    //Here only version 6 is visible
25   };
26   //Again version 5
27  }
```

```
28  //Again version 4
29 }
```

Everything following a // in a line is a comment, and ignored by the compiler. It is only for the reader of the program. The comments show where which variable of the same name is where visible. If this appears confusing this illustrates well why the naming of the variables should be done such that it is always clear which variable is referred to. Especially, the same name should not be used within nested blocks. However, using in consecutive blocks the same name is very common. Otherwise, every for loop would need a new name for the counting variable, which would quickly lead to a proliferation of variable names.

Note that all what has been said for variables also applies to constants.

5.4 Pass-by-reference, pass-by-value, defaults, and variable lists

In many cases it is useful for a function to have arguments passed to it, and on which it can operate. The arguments are a ,-separated list of the following structure

const *type name=value*

The const modifier and the value are optional.

Consider first the non-optional part. This is just like a variable declaration and a name for the variable. It is by this name by which the variable (or the value) is known inside the function. The type is not restricted, but arrays need to be of fixed size, as discussed below[3]. It can also be some struct, as long as it contains only fixed-size array. Thus, an argument is treated inside the function like a locally declared variable with an initial value which is set by the function's argument upon calling. As it is treated as a local variable, its contents is lost at the end of the function, even if the content is modified. This is called pass-by-value.

Consider the example in listing 5.3.

Listing 5.3: Example of arguments of a function.

```
1 void fun(int a,int b) {
2  int d=1;
3  a=b+d;
```

[3]Again, many compilers are somewhat flexible here, and this issue has changed over time.

```
4   b=1;
5  }
6  [...]
7  int c=3;
8  fun(1,c);
9  [...]
```

The function *fun* has two arguments of type int. When it is called in line 8, it get passed a value for the argument *a*. It also gets passed for the argument *b* as content the value of the variable *c*. At line 2, *a* has the value 1 and *b* the value 3. However, inside the function *a* and *b* are treated like locally defined variables, and can be modified. Thus, the values passed along act as initialization for these variables. Hence, it is also possible to pass the same variable for multiple arguments, e. g. fun(*c*,*c*) works as well, as do entire statements, which yield a result of the correct type. Thus, even though *b* gets assigned a new value in line 4, the variable *c* will still have the value 3 in line 9.

It should be noted that it is always necessary to pass a value, explicit or in form of a variable, for each argument when calling a function. It is, however, possible to perform type casts when doing so, as described in section 3.12.

The optional const modifier can be used to declare a variable as unchangeable inside the function. As with normal constants this allows for optimization by the compiler. It should therefore always be done, when it is possible. However, the argument can still receive a variable upon calling for the purpose of initialization, i. e. line 8 in listing 5.3 is still valid even if *b* in line 1 would have been declared as const. Line 4, however, not anymore.

The second optional part is a default initialization with = and a value. This is an exception to the need to always pass along a value for each argument. If an argument has such a so-called default value, no value needs to be passed on. However, to avoid ambiguities, every argument after the first argument having a default value needs also to have a default value. Since the order of arguments is chosen by the programmer, a corresponding reordering is always possible, and therefore does not pose a problem.

Note that it is possible to shadow also arguments, which should be avoided.

It is often inconvenient that the results cannot be passed back, except when using a return type as discussed in section 5.5. There are two possibilities how to do so[4]. The first is inherited from C, and makes use of the pointers of section 3.7. While it is true that the argument cannot be changed, it is possible to pass along a pointer to a variable, which can

[4]A third possibility is the usage of global variables, which is not recommended except when necessary for critical performance reasons.

then be used to change the contents of a variable by dereferencing. This is exemplified in listing 5.4

Listing 5.4: Example of arguments of a function.

```
1 void fun(int *a) {
2  *a=2+*a;
3 }
4 [...]
5 int c=3;
6 fun(&c);
7 [...]
```

This function will increase the value of the passed variable by 2, and thus c will have the value 5 in line 7. This works as follows. The function has as an argument a pointer to an integer. The value of this pointer is provided with the address of the variable c using the address operator in line 6. In line 2 the dereferencing operator is then used twice. On the right-hand side it is used to read the current value stored at the location pointed to by the pointer in the variable a. Afterwards, the assignment stores the value at the address provided again within the variable a. Thus, the contents of the variable a does not change, but the contents of the variable c.

While actually all implementations of manipulating the contents of variables inside functions work in this way behind the scenes, it is for a programmer still error-prone, as working with pointers always is. As a consequence, C++ offers an alternative syntax of this procedure, which is called pass-by-reference[5]. By declaring an argument like *type* &*name*, the whole referencing and dereferencing is now taken care of by the compiler. I. e., changes to the variable passed as argument inside the function will affect the variable outside. Thus, the function fun of listing 5.4 could be written with the same effect as void fun(int &a) { a=2+a;}.

The advantage of this form of variable passing is that no error can be made in terms of referencing and dereferencing. The disadvantage is that, especially with multiple nested functions, it is possible to loose track of which modification of variables will have which effect.

Above, arrays have been mentioned as a possible type of argument. However, this is problematic. The reason is that arrays become quickly large, and pass-by-value becomes inefficient in terms of memory and speed, as the arrays need to be copied. Therefore, it is recommend that arrays are always passed by reference. It is important for this to note

[5]Actually, so is the pointer version.

that arrays in C and C++ are actually a pointer to a chunk of memory containing the data. Having declared int a[5]; the variable a is a pointer of type int. Thus, to have a function taking an array of integer could be declared as void fun(int *a). This immediately highlights a problem: How does the function know about the array structure? The answer is: It does not. Thus, the function is entirely responsible to make sure it does the right thing - or the programmer calling the function. As a mnemonic, it is possible to also declare void fun(int a[]) or void fun(int a[10]). However, in neither case will the compiler check what is done inside the function, and internally there is no difference on how the three cases are handled. In particular, the size of the array in the last declaration will be ignored inside the function when accessing its elements. Thus, passing arrays as function arguments is potentially very dangerous. This is best avoided by object-orientation, as will be discussed in chapter 10. Multidimensional arrays are even more cumbersome, as a pointer-on-pointer not necessarily uses a continuous chunk of memory, while a multidimensional array does. Thus, for the purpose of passing variables, a pointer-on-pointer is considered to be different than a multidimensional array, and therefore explicit type casts may be necessary.

It should be noted that constant values or expressions cannot be passed-by-reference, and will yield a syntax error.

As discussed in section 5.3, Mathematica does not distinguish these cases, and therefore all of this plays no role there.

Finally, it is possible to create functions of the same name and return type, but with different arguments, provided the lists are truly distinct. This is called function overloading. Each such function must be defined independently, i. e. has its own body. Depending on the passed parameters, the compiler will then select the correct version. If certain parameters are automatically type casted, it may be necessary to add explicit type casts. In dubious cases, some compilers may also stop compilation, if a unique selection is not possible.

5.5 Functions with return type

The far most common case is that a function returns none or only a single quantity. While these cases can be captured using arguments, there is also an alternative way to do so, the return type. So far, the functions had been declared as void fun(). However, it is alternatively possible to declare them as *type* fun(), where *type* is an arbitrary elementary type, including any kind of pointers, or a struct, but not an array except in form of a pointer to its first element.

If a function has a return type, the value returned is determined by the statement

return *value*; at any arbitrary point inside the function. If this statement is reached, the function terminates, and returns a value. This value can be used like any value. Consider the following listing

Listing 5.5: Example of return types of a function.

```
1 int fun(int a) {
2  if(a==0) return -1;
3  return a+1;
4  return a-1;
5 }
6 [...]
7 c=d+fun(g);
```

The function fun of listing 5.5 returns an integer value. Thus, when evaluated in line 7, the variable c gets assigned the sum of d and whatever fun returns for the value of a. Inside the definition of fun, the function returns the value of its argument increased by one in line 3. However, the return statement can be called at arbitrary points in the function. This is exemplified by line 2: If a has the value 0, the function will return -1 instead of 1. The function stops there. Correspondingly, line 4 will never be reached, and a pedantic compiler will issue a warning because of this.

Note that it is in addition possible to have arguments for such a function, which can be modified as well.

The true reason for adding a second possibility to return a single value in this form is that this allows to use functions in (mathematical) expressions. E. g. something like $fun(\mathsf{a})+fun(\mathsf{1}+fun(\mathsf{a}))$ is well possible. The possibility to perform nested calls of functions is helpful in constructing more complicated expressions. There is no limit to the number of nestings, and of course different functions can be used.

Note that here the order is uniquely fixed - first all calls for the arguments will be performed before the function itself will be performed. With multiple nesting levels, this will occur from innermost to outermost and (usually) from left to right. This is very important to keep in mind to avoid any unwanted side-effects. The latter is also the reason why one should be very careful with such constructions. Of course, it is not possible to perform a pass-by-reference in this way.

5.6 Recursive functions and the heap

In section 5.5 it was shown that return values of functions can be used as arguments. Indeed, it is even possible for a function to call itself. Consider the example int fac(unsigned int n) { return $(n > 1)$?(n*fac(n-1)):(1); }. This function calculates the faculty of a number. It does so by a so-called recursion, i. e. by calling itself, using a modified argument. While such a recursion can always be constructed entirely with loops, a recursive construction is often the most elegant form.

However, how does the function knows where it is, and what are the corresponding variables?

The answer is that all functions get a local chunk of memory when they are called, the so-called heap. It contains all the memory for all variables of the function. It also contains an information where the function returns to, once finished. Such a heap is created each time a function is called, and thus the program follows from one heap to the next the way back.

This already indicates a problem with recursive functions: Each time the function is called, another chunk of memory is allocated. Though the administrative part is not too large, substantial amount of data can let the memory required quickly grow. Thus, deeply recursive functions can easily use up all memory of a machine, and thus crash. Thus, while elegant, the maximum depth of a recursive function, together with the amount of memory required should always be kept in mind.

Furthermore, creating the heap and only filling it with just the administrative function requires time. The loop version res=n; for(int i=(n-1);i >0;$--i$) res*=i; is much faster. Thus, recursive functions are usually not a good choice if efficiency is most important. However, flattening the code, as putting a recursive function into a loop structure is called, can create also extremely hard to read and maintain code, and also sometimes very lengthy one. Thus, efficiency, maintainability, and memory consumption have to be carefully considered before deciding whether a loop structure or a recursive structure is more appropriate. Still, in physics mostly the flat version wins.

5.7 Forward declarations

It can happen occasionally that two functions need each other. This can pose a serious problem, as usually only things are known which are declared in the code before first use.

The solution to this problem are forward declarations. A forward declaration of a function is done by giving the name of the function, as well as its return-type and arguments,

but without a body, and a final semicolon ;, just like a statement. This can be done wherever otherwise a function can be defined. The function can then be used afterwards.

However, the remainder of the function has to be supplied later in the code, but somewhere in the same block, or in one of the enclosing blocks, or in the body of the main program.

5.8 Operator overloading

While using names for functions usually does the job, it is not always convenient for readability. Consider, e. g. the idea of adding two matrices. It is possible to write a function matrix $madd$(matrix $m1$,matrix $m2$), where it was assumed that matrix is a suitably defined new type, e. g. using a struct. For readability it would, however, be helpful if this could be written as $m1+m2$. This is possible by operator overloading. This is a special case of the concept of function overloading of section 5.4.

To get a + which operates on matrices in the same way as the function before would be declared as matrix operator+(matrix $m1$,matrix $m2$), and would have the same body as the function before. In the remainder of the code it would be possible to write res=$m1$+$m2$; rather than res=madd($m1$,$m2$). Note that the number of arguments, and whether something is returned, must match the number of arguments of the original operator. However, it is not necessary that all are of the same type. If there would be also a type vector, matrix-vector multiplication could be realized as vector operator*(matrix m,vector v). Note that operator precedence remains as it is for the standard versions of the operator, i. e. e. g. * has a higher precedence than +.

In a similar way many of the other operators can be overloaded, especially all mathematical operations, but even operations like []. This is helpful to implement, e. g., more comfortable versions of arrays.

Other than that, operator overloading is essentially just a very special way of functions with a return value.

While this is a very powerful possibility, it can also quickly become confusing. Just think of the possibility to overload the operator + to have a function actually subtracting something. Thus, operator overloading requires very careful design decisions, and should normally be only used to replicate behaviors known to readers and users of the code (not the program!) from the corresponding context, e. g. above linear algebra. That these operators can then have also many more types as arguments can also make it necessary to ensure by explicit typecasts that always the right context is used.

5.9 Function pointer

It happens occasionally that it would be useful to choose different functions when executing something. It appears at first a simple problem. Just define all functions, and use if or switch.

However, in larger projects not all possibilities are known from the outset, or some functionality should only be defined much later. This situation can be dealt with by so-called function pointers. Or, if it should be changed at runtime. A common situation where this arises are callbacks, i. e. that some part of the code would like to be informed, if somewhere else something is is happening.

In essence, function pointers are not different from the pointers to data of section 3.7. The only difference is that they not locate chunks of data in memory, but a function in the code[6].

However, just like a pointer needs a type[7], also a function pointer needs a definition of the function. It needs the type. This is done in almost the same way as for a forward declaration, except that the name needs not been known yet. Thus, to have a function pointer to point later at a function defined as

$$\mathsf{int}\ fun(\mathsf{double}\ a,\ \mathsf{int}\ b)$$

requires to declare a variable as

$$\mathsf{void}\ (\mathsf{*}name)(\mathsf{double,int}),$$

where *name* is the name of the variable. Any variable of this type can be assigned a function by using the address operator &. E. g., using the declarations of listing 5.5, it is possible to write

$$\mathsf{void}\ (\mathsf{*}fp)(\mathsf{int})=\&\mathsf{fun}; \tag{5.1}$$

Then, the variable fp contains the address of the function fun. To use the function, it is possible to just write $fp(variable)$, where *variable* is the variable of type int required by the function fun in listing 5.5. If an integer variable of name a exists, this would lead to $fp(a)$. In this case, the explicit dereferencing operator * is not needed, though using it as $(\mathsf{*}fp)(a)$ will work as well. The compiler interprets this correctly. Such statements can

[6]Actually, where the function is in the memory. After all, the code of functions is stored somewhere, and there is therefore an address where the function starts. Internally, it is just this address which is passed around, and therefore internally function pointers are represented in the same way as ordinary pointers, and only interpreted differently by the compiler.

[7]There is actually also a non-typed pointer, void*. However, to use it for anything but an address requires a type cast.

then be used everywhere where ordinary functions could be used, including the evaluation of the return value of the function.

5.10 Programming styles and philosophy

With these possibilities at hand, it is useful to discuss a few more elements of how to deal with programming.

There is, unfortunately, no best way how to program. Just like a calculation, it is for some people better to just jump into the fray, while others first create all auxiliary things before attacking the real problem. However, just like with a calculation one should never trust one owns skill without questioning. Best is if somebody else double checks what one is doing. Since this kind of resource is not always available, the content of section 4.5 show always be kept in mind.

Irrespective of the verification of the written code, there are still very different ways of how to actually organize writing the code. The most extreme versions are probably to write the full code in one go from start to finish, and then for the first time deploy it for its purpose. The other extreme is to divide the code in as many small pieces as possible. These are written and then deployed individually to solve only their particular subproblem. These chunks are then integrated over time to create the full program. There is also the possibility to deploy only a rudimentary code at first, and then add features over time, which is called continuous deployment.

The kind of teams which should develop the code are also subject to various different philosophies. This starts from one person-one task over the idea of having two persons to always code together to provide check-and-balances to larger subgroups, which meet often to coherently create parts of the program.

Inevitably, no matter how one programs, there will be errors.

Chapter 6

Dynamical variables

In section 5.3 it was discussed that variables can exist within different blocks. In chapter 3, the declaration of variables, including arrays, was presented.

However, these are all static definitions. Even if a variable only lives within a block, it does quite statically so. If it exists or does not exist depends only on whether the program is currently operating within the block. Such variables are therefore called static.

This is very often not sufficient. The reasons for this are manifold. One is that it may not be known at program begin how many information has to be stored, and thus how many variables are needed. Another is that sizes of arrays are not yet known. And there are many more. It is therefore often necessary to create variables at run-time to variable extent. Such variables created according to need and not declared at compile time are called dynamic variables.

There are two caveats to note here.

One is that many language consider all variables as to be 'created on first use'. This is particularly true for interpreted languages, but also applies to Mathematica. Therefore, in these languages no distinction is necessary, and the problem does not arise. Thus, the following only applies to C and C++. Such languages also have automatized functionality to deal with all issues concerning memory management as discussed in section 6.2. This is quite convenient, but leads also directly to the second caveat.

Because dynamical variables are not known at compile time, there are much less possibilities to optimize access to them. Thus, dynamical variables usually mean a loss in efficiency. Thus, if not necessary, they should not be used[1].

[1]Also, dynamical variables are more error prone than static variables, as they involve at one level or another pointers.

6.1 Memory allocation

A computer requires precise information. It is therefore not possible to reserve 'some memory'. Rather, it is required to reserve a definite amount. Furthermore, in a strongly-typed language like C(++), it is advisable to declare also the type of data to be stored in memory. The side-effect is that it is not necessary to know how much memory is needed, but only how much entities of a certain type need to be stored.

The simplest possibility is to create memory according to an elementary type (or struct). To memorize where the data is located in memory requires a pointer. Thus, if an int should be dynamically allocated requires to first declare a pointer to an integer

int *a;

There is not yet any memory associated with this variable. In fact, the variable declared does not store the data. It only stores the address where there data will be stored.

To actually reserve a chunk of memory requires an explicit command, which is called[2] new. Getting the memory is done by

a=new int;

The memory located at the address stored in a is now reserved to keep a variable of type int. Its contents can now be set by using, e. g., *a=2, and be read also by using *a, i. e. by using the dereferencing operator.

If no new has been performed, the quantity hold by a is not a valid address. Dereferencing it will yield some garbage at best. Writing to it will possibly result in a crash of the program. Again, it is not checked by C(++) if the dereferencing is done on a valid address, this is up to the programmer. It is therefore easy to play havoc with the contents.

If there is a second variable declared of the same type, say b, then b=a does not copy the data located at the memory, but the address of the memory. Thereby, a and b now reference the same location, and *a and *b will yield the same result. It is just two pointers pointing at the same information in memory. This is useful, as it allows, e. g., to allocate memory in a function, and keep the memory allocated and accessible by passing the pointer to it to an argument of the function[3]. To actually transfer the contents of the memory requires *b=*a. Note that this requires that b holds an address to a part of

[2]There is also an older set of operations centered around the command malloc originating from C. This will not be used here, but may be necessary in certain environments.

[3]Note that to do that, the variable passed to the function must be declared as either int** or int*&, as it is necessary to change the value of a pointer

memory previously acquired using new. It is not automatically acquired. This has to be done by the programmer.

If there is a third variable declared as int c, then c can be assigned the value stored at the allocated memory location by c=*a. Note that this does not allocate memory. For the variable c, this is done automatically by the declaration for c by the compiler. Note also that the data is copied, and not that the memory allocated for c is now in any way connected to the memory allocated for a, i. e. &c does not equal a, while c equals *a. Later changes of either a or c will not affect the other variable, nor its content.

It is sometimes useful to dynamically allocate the memory to store a pointer. To have a corresponding amount of memory, this requires to declare the variable as int **d - thus the * stack. The memory is allocated as d=new int*. Alternatively, *d=a would copy the address to the previously allocated memory chunk. To access the data would then be done by dereferencing twice, **d, as well as for changes. This can actually be stacked arbitrarily by adding more *s, to get more indirections.

This may seem like an odd thing to do. While there are cases where it is necessary to have these kind of indirections, the more common use is when thinking about arrays.

Going back to section 3.8, the basic concept of an array was to have a fixed number of consecutive spaces in memory to hold the same type of values. Thus, a variable declared as int a[5] could be regarded as a pointer which pointed to the first element of the array. Thus, a[0] and *a yielded the same result, as did a[1] and *(a+1). However, so far it was not possible to get the number of elements selected at run-time. This is now possible using dynamical allocation. If there is a variable, say n of type int n with the number of elements of type double an array should hold, it could be created as

int *ar;

i. e. in the same way as a single pointer to an int! Why is this so? Because there is no conceptual internal difference. Whether the pointer points at a single int or a sequence of int is the same. It just points to an int.

The difference arises when the memory is allocated. Here, it does make a difference if space for a single or more int is allocated. This is done by

ar=new int[n];

This allocates the necessary space to hold whatever value n holds. Note that a nonsensical value, e. g. a negative one, will usually lead to a crash sooner or later. After this, the variable ar can be used like an ordinary array, i. e. like ar[0] to access the first element

and so on. Still, the memory so allocated can again be passed and kept in a different variable by assigning the address, rather than the contents[4].

It is now that multiple indirections come into play. In section 3.8 the case of multi-dimensional arrays was discussed. A multi-dimensional array is allocated as shown in listing 6.1

Listing 6.1: Allocating a two-dimensional integer array dynamically.

```
1 n=5; m=6;
2 int **a;
3 a=new int*[n];
4 for(int i=0;i<n;++i) a[i]=new int[m];
```

In line 1 the variables n and m are set to the size of the different dimensions of the array. Then a pointer to a pointer to an integer is declared in line 2. This pointer is set to a newly allocated array of pointers in line 3. Each of the pointers in this array is now set to a newly allocated array of integers in line 4. The integer elements of the final array can now be accessed as in an ordinary array, i. e. e. g. by a[1][2]. Using only a[1] will be a pointer to an array of (in this case) 6 integers. Note that because the arrays are created consecutively it is not guaranteed that they are lying in a single chunk of memory like in section 3.8. Using pointer arithmetic may therefore not work with them. For this only statically allocated arrays should be used. This can be seen by the example of allocating multiple arrays in different pointers. If their creation is interleaved, the memory is not dealt out for each array consecutively, but consecutively in the order of requests. Thus, the different arrays have non-continuous memory over which they are distributed. This also implies that accessing dynamically created arrays can be slower than statically allocated ones, as different parts may be located in quite different memory locations, and thus disrupt cache coherency or just take longer to look up. Also, the subarrays can be of different size. Therefore, whenever possible, static allocation should be preferred. But this is only possible, if the (maximum) size of an array is known beforehand. That is often not the case.

Of course, this can be continued arbitrarily to higher-dimensional arrays by creating pointer to pointer to pointer and so on.

Note that if the requirement for new memory cannot be fulfilled, an error may occur, or the pointer returned may just be NULL.

[4]In principle, this could be done also with a static array, as the variable still is just a pointer to an array of integer. However, since the memory becomes no longer available once the program no longer needs the array, this can easily lead to problems, and should be avoided.

6.2 Memory management

Memory is a finite resource. For many programs, this is actually not a very important issue individually. However, if multiple programs are running or one program does require very much memory, its scarceness becomes felt even on modern machines. While automatically the hard disks will be used as a memory extension by most modern operating systems, so-called swapping, this slows down the system dramatically, making numerical computations essentially impossible.

Thus, memory should only be used as little as reasonable possible. In particular, on multiuser machines nobody can tell in advance how much memory someone else needs.

While the memory of statically allocated variables is managed by the compiler, this is not the case with dynamically allocated memory. This has to be done by the programmer[5].

To free up memory, which had been previously allocated by the operator new, the operator delete can be used. Thus, after a=new int; the call delete a; will free the memory. Afterwards the pointer stored in a is useless - whatever is now at the memory location pointed to is as random as when choosing some random location in memory and reading from there. To free up space allocated for arrays, the modification delete[] ar; is used. Note that it is not necessary to provide the information how large the array allocated was. If a multidimensional array should be freed, every subarray must be freed. Thus, in listing 6.1 this would be done by

$$\text{for(int } i\text{=0;}i < n\text{;++}i\text{) delete[] } a[i]\text{; delete[] } a\text{;}$$

to free up all allocated memory.

It should be noted that it is necessary to know the pointer to free the memory. Thus, if in a function memory is allocated, but the pointer is not communicated to the outside, it will be impossible to free the memory.

Taking care of memory is also known as memory management.

If memory is not freed, or cannot be freed, this is called a memory leak. This happens if a pointer is overwritten or not returned from a function, before the memory is freed. While individually usually not harmful, it can happen that a program exhausts the memory, and this leads to a crash of the machine. Thus, great care should be administered when doing memory management.

[5]In fact, some languages, e. g. Java, also automatically take care of dynamically allocated memory. This is called garbage collection. There are some libraries for C(++) which also provide garbage collection tools, though then requiring to use their own versions of new to allocate memory. However, while modern versions are pretty efficient, there is still some loss and friction incurred. Also, quite often this requires to use OOP, with all its drawbacks when it comes to being efficient.

Note that memory management can also become intricate in itself. By allocating and freeing often large amounts of memory it can happen that the available memory becomes fragmented, i. e. the memory is a patchwork of free and used memory chunks. At some point, arrays fit no longer in the free patches, creating problems. Also, the efficiency may fall if the memory become strongly fragmented. Also this is a reason to avoid dynamically allocated memory if possible.

6.3 Linked lists

As an example of a pattern in connection to dynamical memory consider the following, common problem. An array of elements should be kept in ordered form, but elements should be added or removed quite often. Using arrays for this purpose has the consequence that they may need resizing and sorting. Sorting is something which can be done quite efficiently[6], but an array's size cannot be changed. The only option is to create a new array, and copy everything. This is very inefficient.

A solution to this problem is the pattern of a (doubly) linked list. Consider the case where the relevant information is an integer. Then the following listing shows everything needed to work with a so-called linked list.

Listing 6.2: Worked with a linked list.

```
1  struct ll {
2  int a;
3  ll* next;
4  ll* prev;
5  };
6
7  ll* find(ll *i,ll *s) {
8   if((i->a<=s->a)&&((i->next==NULL)||(i->next->a>s->a))) return i;
9    else return (i->next!=NULL)?(find(i->next,s)):(i);
10 }
11
12 ll* add(ll *i,ll *toadd) {
13   if(i->a>toadd->a) {
14     ll* p=find(i,toadd);
15     toadd->next=p->next;
```

[6]Sorting is a problem which occurs very often, and thus quite powerful patterns are available for it.

```
16      toadd->prev=p;
17      p->next=toadd;
18      if(toadd->next!=NULL) toadd->next->prev=toadd;
19        else toadd->next=NULL;
20      return i;
21    };
22    i->prev=toadd;
23    toadd->next=i;
24    toadd->prev=NULL;
25    return toadd;
26  }
27
28  ll* remove(ll *i,ll *toremove) {
29    ll* p=find(i,toadd);
30    if(p!=i) {
31      if(p->a==toremove->a) {
32        p->prev->next=p->next;
33        if(p->next!=NULL) p->next->prev=p->prev;
34      };
35      return i;
36    };
37    i->next->prev=NULL;
38    return i->next;
39  }
40
41  ...
42
43  ll *start=new ll;
44  start->a=1; start->next=NULL; start->prev=NULL;
45
46  ll *next=new ll;
47  next->a=2;
48
49  ll *temp=add(start,next);
50  if(temp!=start) start=temp;
```

This is a rather lengthy example, and it demonstrates a lot of techniques and how to work

with memory management.

To start out, in lines 1 to 5 a struct is declared which will become a link of the linked list. The basic idea is that the result should be sorted in order of the integer value a. Of course, in real applications the struct will have usually a much larger payload. There are then two pointers of the type of the struct. Note that it is possible to use them already in the declaration itself. It would not be possible to use variables of this types, as this would immediately yield an infinite recursion. But pointers do not require now to be filled, and are therefore fine. They will be used to form the links from every element of a chain to the next and previous one.

Now jump to the actual program to follow the flow. In line 43 a variable *start* is created. Its initialization in line 44 shows that it is possible to write instead of (**start*).*a* also *start*-> *a*, without explicit dereferencing. It is again a (very common) shorthand notation. Note that the initial element has not yet any neighbors, and thus the pointers to the next and previous element are set to NULL.

In line 46 another element is created. In line 47 only its payload is assigned. Its links are not yet created. This is done in line 49, where the function *add* is used for this purpose. Note that a variable is initialized with the return value of this function. No memory is allocated here.

Jumping to the function *add* means going back to line 12. The idea of the linked list is that the values of *a* increases from the beginning of the list, but allows elements to have the same value. Then, the element is repeated. It is checked in line 13 if the element, which should be added should be behind the element which has been passed is already right. It is here assumed that *i* has the value of the beginning of the list when called first - this is not checked, and therefore part of the implicit information on the function. This needs to be described in the manuals. Also, there could be more lists, and the function cannot know, which is the current one. Thus, calling the function in the correct order is the responsibility of the user.

If the element is not yet in order, the function continues in line 14, using the function *find*. This will return the correct element after which the new element must be placed. This will be discussed below. The element is now included in the list in lines 15-19, by adjusting the pointers *next* and *prev* of previous and following list elements accordingly, checking carefully whether there is a next list element. It then ends by returning the list element which is considered to be the start of the list. If it turns out that line 13 evaluates to false, the element to add must be the next beginning of the list. This is arranged for in lines 22-24. Returning then the new beginning of the list implies that the return value of function *add* is the beginning of the list.

Since the caller does not know, whether the list has a new beginning, he compares this return value with the original start of the list, and sets the variable *start* to the new beginning of the list. This could have been made shorter by just writing *start*=add(*start*,*next*);. Note that though there is no pointer left to *start* in the variable itself, if it changes, the elements of the list still have this information.

Returning now to the finding of the correct list element, consider the function $find$ starting in line 7. The way how it is used in line 14 implies that it should return the element preceding the element with the value s-> a. This is done using recursion. In line 8, it checks first, whether the passed element i fits the bill. To add the element at the end of a sequence of repetitions, it furthermore checks, whether the following element is actually not of the same size. Note that by ordering the comparisons in the if, it is made sure that the last test is only performed if i-> $next$ is not NULL, and therefore no crash or meaningless dereferencing can happen. If the conditions are fulfilled, or the element i is the last in the list, it is returned. Otherwise, if the next element is not NULL, the function is called recursively with the next element of the list. If it is the last element of the list, it is returned anyways. Note the trinary if is used to write this compactly.

The function to remove the element follows the same logic in lines 28-39. It is a useful test of the reader's skill to understand it.

This small code fragment has been using a lot of the structuring which was available so far: It used multiple functions to distribute different functionality. It used recursive functions, trinary if, and ordered tests to avoid nested loops and conditions. It also used repeatedly tests for NULL to check the validity of pointers. However, this required to careful manage that all pointers have been set to NULL, where needed. Still, to optimize the code, the elements *prev* and *next* of *next* in line 46 and 47 have not been set, as the function *add* does this as well. Together, it represents a typical piece of code using pointers.

Chapter 7

Structuring programs

With the example of section 6.3 a quite intricate structure has already been reached. At such a point, it becomes necessary to structure the program better. This will be discussed in the following. Mathematica is somewhat special, and will thus only be briefly commented on at the end of this chapter in section 7.6.

7.1 Libraries, APIs, and patterns

As has been seen in the concept of functions, one of the central ideas of the development of programming has been code reuse.

This idea is carried further by not just thinking of the reuse of single functions (or later objects), but by collections of them. Such a collection is called a library. There are both highly specialized libraries aiming at very narrow problems, or libraries which collect any kind of useful things. The probably most important library during this lecture is the library (actually, a collection of libraries) which is supplied together with C and C++. In fact, many of the commands used are actually not part of the language itself, but are from the libraries.

The way how a function looks like, e. g. in the example in section 5.1 it has single real variable and returns also such a number, is called the interface of the function. Later, similarly classes have interfaces. Or, more aptly, are interfaces. The collection of all interfaces of a library is called the application programming interface (API) of the library.

Libraries are code, and therefore tied to a particular programming language[1]. The next level of abstraction beyond libraries will be patterns. Patterns are problem solution recipes, which are specified not in a particular language, but sometimes in a meta language

[1]Most languages nowadays offer possibilities to call functions from libraries written in different languages. This will not be detailed here, but should be kept in mind.

or in a natural language, and describe how problems are solved efficiently which occur again and again in programming. These patterns are collected in various repositories, and thus in form of (collection of) libraries. It is highly recommendable to search such pattern libraries before solving a problem which appears to be somehow something many people could have come across.

7.2 Multiple files and linking

The first possibility to structure the code is to divide the code in files, each file containing part of the program logic. Especially, every file, or set of files, should be created to cover a particular, well-defined purpose. Otherwise, there is a good chance that total chaos reigns. Thus, partitioning into different files should be planned ahead.

For this purpose, C(++) has a twofold concept: Code files (ending usually as .c for C and .cpp for C++) and header files (.h and .hpp, respectively). Header files contain declaration and code. Code files contain only code. There can be header files without a code file. However, only the code file containing main can be useful without a header file.

A header file can contain any functions and constants. For functions it is also possible to provide only the forward declarations of section 5.7 in the header, and the remainder of the function in the code file. In fact, this is more the rule than the exception. It cannot contain global variables. Rather, if a global variable should be declared, it needs to be done so in the form of an extern variable, e. g.

extern int *name*;

It is then necessary that in the code file of the same name the declaration int *name*; appears at the global level, as the memory is allocated by the code file.

The other difference between the header file and the code file is that whatever is written in the header file can be used by another code or header file. This is done by an include statement, e. g.

#include "somecode.h"

This is exactly the same type of #include as was already used from the beginning since section 3.2. The usage of "" instead of <> originates from the fact that the brackets instruct the system to use build-in header files of the compiler, while the quotation mark allows to add user-provided headers. Not that the name of the header files have to provided either as relative or absolute paths[2], following the conventions of Unix, i. e. different

[2]Some compilers provide the possibilities to search some directories automatically. For g++, this is done using the -I option.

directories are separated by /.

All functions, constants, and global variables of the included file can afterwards be used in the code file which included the header just as it would have been declared where the #include statement is. Note that #include statements can appear anywhere in the code. It is, however, conventional to make all #include statements at the beginning of a code or header file.

However, conflicts can arise if a header, which is included, is also included by a second header, which is also included in a file. In this case, the code would be repeated, and the compiler does not know how to handle the situation, as it cannot distinguish whether this is twice the same code, and whether there are interdependencies, which need to be resolved. It will therefore just report an error and stop compiling. To deal with this problem the preprocessor exists.

7.3 Preprocessor

In fact, the preprocessor is much more than just a mean to resolve such conflicts. The preprocessor is a possibility to provide instructions to the compiler during compile time. All preprocessor instructions start with a #. Indeed, as may now be guessed, already the #include statement is a preprocessor statement. After all, it instructs the compiler, and thus not the program, to do something: To put here the contents of a different file.

The preprocessor is a quite powerful tool, and a full appreciation of its possibilities would go far too deep. Here, besides #include, only two more preprocessor structures will be discussed in more detail.

The first is #define. It can be used to define a name, e. g.

#define *name*

Note that no ; is needed. The object name is now visible throughout the code, but only for the preprocessor. However, in this form it will only be useful for the preprocessor, as it is a name, and not a variable. However, it is possible to use the preprocessor to define constants

```
#define two 2
```

Afterwards, the compiler will replace all occurrences of two within the code before compiling with 2. This is the important feature of the preprocessor: Its statements are executed before compiling. Hence its name.

It is also possible to provide one or more parameters for definitions. E. g.

```
#define sqr(x) ((x)*(x))
```

will replace everywhere in the code sqr(x) by ((x)*(x)). While it appears tempting to use such so-called macros for many purposes, it has the disadvantage that the compiler will only check the created code, but not the macro - after all it is the preprocessor which replaces the macro by code, not the compiler. It my therefore become quickly involved to track problems, and is therefore generally not recommended.

The idea to use macros instead of functions to improve efficiency is also not an argument. For this purpose C(++) offers the inline declaration, which can be put in front of a function. It recommends[3] the compiler to replace every call to the function by the code of the function, just as a macro would do, but by the compiler with all corresponding checks.

The more important consequence of a defined name is that the preprocessor has flow control. If declared like

#ifdef *name*
code
#endif

the *code* part will only be included in the code passed to the compiler if *name* has been defined in a previous #define statement. There is also a test if a name is not defined, with the preprocessor statement #ifndef. This possibility to include or exclude code from compilation has a twofold advantage.

One is the resolution of the problem of section 7.2. To avoid duplication of the code inside a header, it is sufficient to write

#ifndef *name*
#define *name*
code
#endif (7.1)

and to put everything of the code inside this the *code* block. If the *name* is not defined, it will then be defined, and therefore when entering the header a second time, the block

[3]Compilers may not follow this suggestion. Also, when passing the optimization options (-O1 to -O3 with increase in aggressiveness of the optimization while at the same time requiring more careful and precise programs to permit various assumptions of the compiler), the compiler may expand also functions not declared inline. The same is true for the option register for variables, put before the type. This recommends the compiler to keep the content of a variable in a CPU's register rather than in memory. Again, the compiler may or may not adhere to this recommendation. It should be noted that being better in optimization with such tricks than the compiler requires an enormous amount of experience and knowledge of the inner workings of the compiler and the specific computer architecture on which the program should be executed.

will be skipped during compilation. Thus, a header should always have such an embracing preprocessor directive.

The second advantage is that parts of the code can be included or excluded. E. g. parts of the code which are only useful during development, e. g. for debugging purposes, can be dropped in the production version, where it may incur performance penalties, just by defining or not defining a name[4]. This avoids altering the code from the development to the production version, making it easier to have correct code. Alternatively, this can be used to create different versions of a program using the same code base, e. g. if there are different optimizations for some different sets of parameters in a simulation. After all, it is always better to optimize a program for a purpose during compile time than by using variables, costly to access, and flow control, even more costly, to select between different variants at runtime. Still, the code is cohesive, and not different variants, so-called forks, have to be maintained.

Some compilers also offer the possibility to define names as options passed to the compiler[5], so that they may not even be set in the code. The details of this depend on the compiler.

7.4 Namespaces

Even after using the preprocessor there is a potential for problems with doubled definitions. The reason is that two different header files may contain the same name, even if used for entirely different purposes. This is less likely to happen in a single project, but often enough external code is included in a program, possibly from many different sources, and this then creates the problem.

This is solved in C(++) by the use of namespaces. A namespace adds to every name an additional name, to which scope the name belongs. If there exists a variable bar in the name space foo, its fully qualified name is foo::bar, where :: is called the scoping operator.

A namespace is declared as

namespace *name* { *code* }

Everything which is declared inside the block is now within the namespace, and has to be used using its fully qualified name, that is with *name*:: attached in front of it. It is not necessary to declare everything belonging to the same namespace within the same block.

[4] A preprocessor statement can be commented out using the usual possibilities.

[5] This is somewhat misleading. E. g. g++ is a single program, but which combines both the preprocessor and the compiler, one running after the other.

It is always possible to add in another block, declared in the same way, further members of a namespace. Note that namespaces can be nested, requiring strings of scoping using the names of the namespaces.

To avoid the tedious use of the fully qualified name, it is possible to push anything within a namespace to the current namespace. This is done with the using operator. E. g. cout is declared inside the namespace std included with C(++). Thus, it would usually be needed to be fully qualified as std::cout when used in the code. By using std::cout; it is possible to refer to it as cout only within the current block and after the using statement. This can happen outside functions. By using *namespace* it is possible to do this simultaneously for anything declared in a given namespace, as was done for the namespace std already in section 3.1 in the most basic program.

This can also be done for multiple namespaces simultaneously. However, if the same name is used within multiple namespaces, C(++) will usually use the latest one included by using. Just as with the shadowing of global variables by local variables in blocks, this can lead to hard to identify problems. Thus, it is better to use using sparingly if the elements of a namespace are not used extremely often.

7.5 Libraries, and dynamical and static linking

A set of headers and associated code files, which are intended for use by other programs and not by a specific program only, are called a library. The set of all declarations inside the headers are then usable by other programs, and therefore form the application programming interface, the API, of the library. In the end, all libraries are of this type. To use a library requires thus the headers, and they need to be included in the code.

Code files of a library can be compiled separately from the code involving the main function by a compiler option (-c for g++). The resulting files are then containing the code, and often referred to as object files (a different object than in OOP), and usually have the file ending .o. They do only form part of a program, and cannot be run alone.

The reason for this is twofold. On the one hand, this allows separation of the compilation of libraries from those of the main code. For extensive libraries which are not changed during developments, a quite common situation, this can save hours or days of compile time. They are just once compiled. This is also useful if a company does want to sell libraries, but does not wish to provide also the code, and then will just provide the headers and object files[6].

The process of combining these objects with the part of the program containing main

[6]There are also more possibilities to package libraries, but they will not be considered here.

is called linking. For this purpose still the header files are required, such that the compiler knows the declarations, so-called interfaces, of the functions and/or variables in the libraries. This usually does only require to tell the compiler where to find the object files (for g++ this is done using the -l (l) option for the location of the libraries and -I (I) for the location of the header files), but does not require any other compiler switch.

There are now two possibilities how this linking can be done, static and dynamic linking. In the case of static linking, which is usually the default, the object files and the compiled code are combined into a single file, which can then be executed. In case of a dynamical linking this is not the case, and the object files are transfered into so-called library files (sometimes just the original object files), and need to be kept together with the program itself.

Dynamical linking has the advantage that it is possible that different programs use the same library files. This guarantees that all programs use the same version of a library. As some libraries can become rather large, this also saves space[7]. To some extent, it is also possible to replace these libraries after compilation, so libraries can be updated without recompiling the program. This has its advantages during deployment of code. However, the details of this requires in-depth knowledge beyond the scope of this lecture. Thus, for the remainder of this lecture always static linking will be assumed.

7.6 Mathematica

Also Mathematica knows the concept of libraries. On the one hand, in every instance of Mathematica, i. e. during program runtime, every open notebook lives in the same environment, and definitions in one notebook will also be available in another notebook. That can also easily lead to confusion.

The other form is that of packages, which also know concepts like namespaces. Such packages can be included using <<packagename‘ (note that this is the reverse high comma!). Package development is much more involved than just having code somewhere else as in C(++). Thus, it will not be detailed further during this lecture. However, there are many powerful libraries available for Mathematica, which can be included in this way.

[7] At first, this does not seem to be an issue, as really much code would be necessary to get large programs. It is, however, possible to embed also data like pictures into libraries, so-called resource libraries, and they then quickly grow very large.

7.7 Programming and project management

Now that with libraries a possibility exists to structure programs, it is necessary to consider a few more issues when it comes to larger projects.

Just like with a calculation, a simple program is done quickly. But like a complicated and lengthy calculation, which may require several auxiliary calculations and dragging on for weeks and months, a more powerful program is not simple. It becomes a project. Therefore, writing any reasonable complicated program requires management.

It is therefore important to plan a program, i. e. before implementing it and writing actual code. The first stage is answering the questions posed in section 1.1. However, it then requires to structure the development process by answering the following questions

- When should it be complete?
- Are there optional parts?
- Can the program be decomposed in subunits, which work independently? If yes, in which order are they required?
- How should everything be organized?
- How can various versions of the program be separated?
- How can be ensured that the code is readable? How can be ensured that it can be understood in a month? A year? A decade? How can be ensured that somebody else can understand it?
- Are and, if yes, how are other people involved?
- Where and how is everything stored (and backuped)?

This list of questions can be extended almost indefinitely. The bottom line is, it is not only necessary to plan what the program should do. It is also necessary to plan how the programmer(s) can create the program. Think ahead. Plan from the end, not from the start. Plan before you start to do anything. Keep the schedule. Use milestones, i. e. intermediate goals, to structure the temporal evolution. Check that you meet your milestones. If you do not meet your milestones, understand why your fail and how you can avoid it. Implement whatever changes necessary to avoid failing a second milestone. Do not change your project and your plan without necessity. If you have to change it, assess what this implies and how it changes your plan. Do never assume that a change to the program does not entail a change to the plan. Include buffer time and never expect that everything goes smoothly. Murphy is real.

Chapter 8

Functional programming

The following will discuss a number of concepts and techniques for functional programming with Mathematica. Therefore, much of the following is not available in C++. It is not an exhaustive list of possibilities, and can give only a slight flavor of how functional programming is done in Mathematica. Moreover, Mathematica is, as noted, not a pure functional language. Therefore, there is not always a strict separation.

The basic tenant of functional programming can be summarized in two basic statements:

- Everything is a (mathematical) function yielding a result
- Functions do not have side-effects

The first statement requires that every activity has a return value. This is, e. g., the reason for Mathematica's If of section 4.1 to have a return value.

In particular, functions themselves can again be the result of a different function, using the concept of pure functions from section 5.2 - functions without a name. Thus, this generalizes the idea from values to entities to operate on. Conceptually, this kind of generalization also starts out like OOP. But while OOP approaches more the real-world situation with entities, functional programming marches towards the formally more pure concept of mathematics. Hence, in a functional language the idea is not to have a persistent set of data, but map the problem solution to a chain of evaluation of (mathematical) expressions, from the beginning to the end. In that sense, functional languages are also known as declarative languages rather than imperative ones.

Little surprising, functional languages are very well suited to mathematical problems, as their structure is very close to mathematics. However especially large programs tend to become quite unwieldy to read without experience. Though it is possible to essentially cover all problems also in functional language, today most functional languages have also

aspects of imperative languages, while imperative languages also have some aspects of functional languages, e. g. by being able to chain together evaluations of (mathematical) functions or the use of functions as return types.

8.1 Placeholders and delayed evaluation

In this context, it is important to understand how mathematical functions are defined in Mathematica.

This requires to understand the concept of placeholders. A placeholder in Mathematica is a name followed by an underscore, _, like a_ in the place where a function is defined. It can then be used where the function is defined to use as a placeholder, where the actual value is then put in when the function is used. E. g.

```
a[x_]=x+1
```

defines a function, which will evaluate to 3 if called as a[2]. Without the placeholder, i. e. as a[x]=x+1, a[2] would evaluate to a[2], because no evaluation is performed. With placeholder, the argument can now be used also for another variable, e. g. a[y] will evaluate to y+1. Note that a function can also be undefined using =., e. g. a[x_]=. will make a[x_] again unknown, and thus a[2] would afterwards again evaluate to a[2]. Note that the variable a is different from a[x_], and in particular not a function. However, there can only be one object of name a at any time. Leaving out the argument is therefore not possible in Mathematica. After the function has been defined, it can be used as any mathematical function in Mathematica. If the function has multiple arguments, they appeared separated by commas[1], e. g. y[x_,y_]=x+y.

It often happens that a function should be defined using a previous calculation. For this purpose the % operator is useful - it has the value of the last previous result, i. e. of the last evaluation of Mathematica after pressing[2] shift+return. Multiple % refer to the previous %%, or prior-to-previous %%% and so on, results. Note that results are denoted by a number (written as Out[*number*]), and can therefore be directly accessed, e. g. %12 will yield the result denoted as Out[12].

Consider listing 8.1.

[1]Note that this not a list. To map a list to a set of arguments use Apply[Sequence,*list*], e. g. for the example y[Apply[Sequence,{1,2}]].

[2]Note that cells, the Mathematica slang for statement blocks, where statements are separated by ,, ;, or return, can be set to automatic evaluation upon notebook opening. If no manual evaluation has been performed after opening a notebook, the operator % always refers to these automatic evaluations.

Listing 8.1: Usage of % in Mathematica.

```
1  b=2+x
2  c[x_]=%
3  c[3]
4  d[x_]:=%+x+3
5  c[5]
6  d[x]
7  c[x]
8  d[x]
9  c[2]
10 d[2]
```

In line one the variable b is assigned the function x+2. Note that b is not a function itself, it is just a variable containing a function. In line 2, c[x] is assigned the previous value, which is x+2. Thus, c[x] is now the function x+2. Consequently, evaluating line 3 will yield 5.

This is not always, what is needed. For this purpose, there exists the possibility of a delayed evaluation, created using := rather than =. This requires Mathematica to only at time of execution evaluate the contents of the function. This seems to be not different from before. In fact, a[x_]=x+2 and a[x_]:=x+2 will act in the same way, because everything on the right-hand-side is immutable. This changes, however, as soon as non-mathematical operations like % come into play.

Consider such a delayed execution as defined in line 4. Because of line 5, the operator % would yield 7. Thus, line 6 evaluates to %+x+3, and thus yields 10+x. Likewise, line 8 after executing line 7 will evaluate to 5+2x, and line 10 after executing line 9 to 9, as now the placeholder x in d receives the value 2.

Delayed assignment is particular useful to define recursive functions. E. g., the Fibonacci sequence can be defined by f[1]=1; f[2]=2; f[n_]:=f[n-1]+f[n-2]. Any attempt to do this with ordinary assignment using = would yield an error, as Mathematica attempts to assign the structure immediately, and therefore would create an infinite loop.

Another interesting effect is seen when attempting to apply this formula for very large numbers. Because of the recursion it becomes very slow. The reason is that Mathematica uses for every step again a full recursion. To avoid this, it is useful to save the results. This can be done by a combination of delayed and direct evaluation by f[n_]:=f[n]=f[n-1]+f[n-2]. Then, when any value is first evaluated, it is stored as a value of f[n]. Since this explicit value has higher precedence over the delayed evaluation afterwards always this value will be used. On the other hand, the delayed evaluation beforehand avoids that Mathematica

attempts to get all results immediately after initialization. This explicitly demonstrates how a combination of delayed and direct assignments can be used to speed up calculations while at the same time avoiding problems[3].

Delayed assignment can have hard to predict consequences and side effects, if employed unwisely, and should therefore be used with care.

8.2 Pattern matching

A very powerful possibility in Mathematica is pattern matching. The idea is that some pattern in an expression is replaced by another pattern. This is done using the /. operator. Consider

```
x+2/.x->3
```

This will yield 5, as any occurrence of x is replaced by 3. This seems to be much the same as evaluating a function. However, in practice often sequences of expressions are evaluated, where several replacements are done, rather than to define in every step a new function. The assignment -> is called the replacement rule, and it can also be applied delayed, just like assignment, by using :> instead. This is also the reason why Mathematica formulates solutions in terms of such rules: After all, a solution to an equation is obtained when something is replaced with a certain value.

However, it becomes much more powerful, if combined with the placeholders of section 8.1. Consider

```
x+2+Sin[x+2]/.Sin[c_]->Sin[2c]
```

This will yield x+2+Sin[2(2+x)], as the placeholder is replaced accordingly. Note that Mathematica also inserts automatically parentheses to keep mathematical precedence. Also, it interprets the context. It only replaces the placeholder, if it is inside a Sin, as otherwise it would need to also replace the first expression. This is important. E. g. using just c_ instead of Sin[c_] would yield Sin[2(2+x)+Sin[2+x]], as here the context is a complete statement. Likewise the replacement pattern c_+2->Sin[2c] would yield Sin[2(x+Sin[2+x])], as again the context would be the whole expression. To replace both occurrences of x+2 with 2(x+2) would be achieved by the rule x_->2(x+2)-2.

It should be noted that pattern matching has a low precedence. To have the desired effect, it may be needed to use parentheses. E. g. 2*(a+2)/.a->3 will first evaluate the complete mathematical expression, while in 2*((a+2)/.a->3) the multiplication by two

[3]To see which values are already stored, and which are newly calculated the command Trace can be used.

will be done after pattern matching. This is also important when performing multiple pattern matchings inside a single statement.

It also possible to pass to /. a list with pattern matching rules, i. e. lists of replacements with ->, e. g. {a->2,b->3}. Note that also here the order can yield different results. E. g. {a->b,b->3} will yield something differing from {b->3,a->b}.

To enhance pattern matching, the placeholder _ can also be made more flexible. So far, it is used to be identified if there is an exact match. Using it twice, __ requires one or more equalities, and thrice ___ even allows for zero or more agreement. E. g. 3 will match to any of the three types of placeholder, but not match to any {_}, as 3 is never a list.

Note that as always such pattern matching can be assigned to variables. E. g. a=x_y_->x+y would yield for cb/.a just c+b. Furthermore, by using //. the rule will be applied until no matching pattern remains. Thus cdefgh/.a yields c+defgh while cdefgh//.a yields c+d+e+f+g+h.

Pattern matching can be further extended by the use of conditional patterns. E. g. it is possible to qualify patterns with a condition by using _?, followed by the test. For example, to replace all integers in a list, this would be possible by *list*/._?IntegerQ->x. This will replace any integer inside the list by x. To replace all quantities bigger than 0 a pure function[4] can be used, *list*/._?((#1>0)&)->x and of course any other test, as long as it yields a boolean value.

Since tests are quite common conditions, another way of formulating them is available with /;. E. g. the test for larger than zero could have also been written as *list*/.a_/;a>0->x to the same effect.

These tests can also be used to define functions without using If. The following two statements are equivalent

```
h[x_]=If[x>0,1,0]
h[x_/;x>0]=1; h[x_]=0
```

and there are several other possibilities how to formulate the same statement.

8.3 Useful functions in Mathematica

Mathematica has a wide variety of built-in functions, a library, many of them available at start. Additional ones can be included using the Needs[*name*] command, where *name* is the name of an additional package, and operates similarly to the #include directive of C(++). As with this case, it is necessary to know the appropriate name. Mathematica

[4]For the sake of the example. This can also be done with the build-in function Positive[].

has built-in methods for almost any mathematical functions, including special functions like the Γ-function or hypergeometric ones.

Moreover, it supports many mathematical procedures in an analytical way. The most important ones are

- N[*expression*] gives a numerical rather than an analytical value of *expression*, to the extent possible
- Sum[*expression*,{*index*,*start*,*finish*}] allows to implement a sum with terms given by *expressions*, with the *index* of the sum ranging from *start* to *finish*. An additional step size is also possible, as are multiple indices
- Series[*function*,{*variable*,*start*,*order*}], which gives the Taylor series of the *function* in *variable* around the expansion point *start* up to *order*. E. g. Series[Sin[x/2],{x,0,2}] yields x/2+O[x^3]
- D[*function*,{*variable*,*order*}] gives the *order*th derivative of *function* in the *variable*. E. g. D[Sin[x],{x,5}] yields Cos[x]
- Integrate[*function*,{*variable*,*low*,*high*}] attempts to integrate *function* over *variable* analytically from *low* to *high*, as an indefinite integral if no limits are given. This is not always possible, and also not for all analytically integrable functions the solution is known. Thus, this may not always work. Note that the limits can also be names or variables
- Simplify[*expression*] attempts to simplify mathematically *expression*, if possible. There is an extension FullSimplify[*expression*] which knows more relations which can be used to simplify an expression, but is also considerably slower
- Expand[*expression*] and TrigExpand[*expression*] expand *expression* algebraically and using trigonometric identities to a sum with as simple terms as possible, which can be inverted using (Full)Simplify
- Solve[*equation*,*variable*] tries to solve *equation* for the *variable* analytically. Both arguments can be lists, where it is then attempted to solve the equation as good as possible for the variables, even if the number does not match.
- DSolve[*equation*,*function*,*variable*] attempts to solve the differential *equation* for the *function* of *variable*. E. g. DSolve[D[g[x],{x,2}]==a g[x],g[x],x] yields {{g[x]->E^(Sqrt[a] x)C[1]+E^(-sqrt[a]x) C[2]}} where C[*i*] are the integration constants

- FindFit[*data*,*expression*,*parameter*,*variable*] attempts to find values of the list of *parameters* of the *expression*, which is a function of the *variable* such that it represents the *data*, which is a list of variable-value pairs, as good as possible. Optionally, a first-guess value of the *parameters* can be provided by giving a two-element list with the parameter and the guess

Of many of the above listed analytical functions exists also numerical versions, with a prepended N. Furthermore, many of them support as additional arguments various (and many) options, e. g. for numerical integration NIntegrate the way how to select the integration points, or initial guesses for the variables in NSolve. The full listing of all options is very extensive, and therefore should be obtained from the documentation.

It is also possible to use as an additional argument Assumptions-> *list*, where *list* is a list of assumptions, like, e. g. {a>0}, which assumes for the operation that a is greater than zero. There is a wide array of possibilities for assumptions.

As it is often necessary to formulate lists for these functions, the command Thread[*op*[*list*1,*list*2]] is quite helpful, as it recombines the list elements such that they become a single list, pairing the elements as arguments of *op*. E. g., Thread[Equal[{a,b},{c,d}]] yields {a==c,b==d}. The same effect could be obtained by {a,b}=={c,d}//Thread.

8.4 Rule-based programming

The operator -> of section 8.2 is actually a far more powerful concept as it first appears. It is generically called a rule. In section 8.2 this was used to provide a rule how to replace expressions. There is much more to it, especially when further combined with the placeholders of section 8.1.

Consider first the very general Select command[5]. Its application has to do with the often appearing question of how to find some elements in a list which match particular criteria. It has generally the syntax Select[*list*,*condition*]. While the *list* is just a standard list, the second part is more involved.

First, the *condition* ends with a &. The next is that the element to be worked upon is referenced to be #. The list is traversed, and one-by-one every element is assigned to #. As it is a variable, it can be acted upon as normal. Finally, the total condition must evaluate to either true or false, to give a criterion whether the element should be included

[5]Note that there are specialized versions of many of the following statements, which reduce the amount of specifications needed to operate them compared to the general ones. These can be found in the documentation.

in the result or not. E. g.

Select[{1,23,6,9},#-1>5&]

will yield all elements of the list, which, after subtraction of 1, are still larger than 5, and therefore yields {23,9}. If the list contains list, it possible to access various elements. E. g.

Select[{{1,2},{2,1}},#[[1]]>#[[2]]&]

yields {{2,1}}, as only here the condition is true. Note that the first index is already set implicit, i. e. #[[1]] operates on the ith list element like[6] #[[i,1]].

Such constructions appear at many places, emphasizing the importance of the construction of rules. E. g., the Sort function to sort a list works similarly. However, since here comparisons have to be made, it is necessary to refer to two different elements. This is achieved by appending a number 1 and 2 to #. E. g.

Sort[{{1,1},{4,-1},{2,π/3}},Sin[#1[[2]]]>Sin[[#2[[2]]]&]

yields {{2,π/3},{1,1},{4,-1}} as it sorts the list depending on the relative size of the sines of the second element of the lists being the elements of the list.

8.5 Predicate functions

Because Mathematica does not have strict typing, it is often necessary to test what the type of the contents of a variable is. For that purpose, a large number of so-called predicate functions exist. Predicate functions, in general, are just functions which return a boolean value depending on whether its arguments evaluate to true or false.

In Mathematica predicate functions exist to test for the type of the contents of a variable. E. g. IntegerQ[*variable*] will return true if *variable* is an integer and otherwise false. There are many other such functions, usually ending in Q to test for properties. Because of the importance of lists there exist also tests on list properties. E. g. MemberQ[*list*,*element*] is true if *element* occurs in *list*. Note that e. g. == and && as logical operations yielding boolean values are also considered to be predicate functions.

8.6 Maps and apply

Because of the mathematical structure of functional programming the concept of a mathematical map finds itself also in Mathematica. The command Map[*function*,*expression*]

[6]Therefore, an error will be encountered if it is attempted to access a list element, which does not exist.

applies *function* on *expression*, yielding e. g. for Map[f,{a,b,c}] {f[a],f[b],f[c]} and for Map[f,a+b+c] f[a]+f[b]+f[c].

This shows that the *function* is applied to a certain organizational level of the *expression*, especially the basic structure is bypassed, and the *function* is applied to the first non-trivial level. In the first example, the basic structure is a list and in the second a sum. The first non-trivial level is then the list elements or the terms of the sum. If the structure is deeper nested, it is possible to require Map to be applied to elements at every nesting level, or just to (a) particular nesting level(s). By giving as a third argument a number, the function will be applied to all levels up to this level, by giving a list of numbers to the levels in the list only.

The basic structure can also be replaced by Apply[*function*,*expression*]. E. g. Apply[f,{a,b,c}] yields f[a,b,c], Apply[f,a+b+c] f[a,b,c], and Apply[f,g[a,b,c]] f[a,b,c].

Both functions therefore allow to manipulate the structure of an expression. This is something which is not possible[7] in C(++), and a genuine part of functional programming.

Note that Map[f,g] can be written as f/@g and Apply[f,g] as f@@g.

8.7 Trees and hashes

While not formally a central part of Mathematica, there are two concepts, which play an important role in the context of functional programming, actually in programming in general.

In a very general sense, a mathematical map can be thought of as a (lookup) table, as it gives for any input a predefined value. The concept of such tables is always very prevalent in programming, and the input is often called a key which maps to a value. Thus, frequently key-value pairs need to be organized.

However, quite frequently the key is far from a simple expression, and also tables can become very quickly very large. Both effects make lookups quite slow. This lead to various solutions.

The simplest one is if the key has some kind of ordering principle, such that statements that one key is larger, in a very generalized sense, than another key remain true[8]. If this is the case, the table can be ordered, and thus it is sufficient to start at the beginning and then traverse the table in ascending order until the searched-for item is found.

[7]To some extent this can be attempted to be replicated using function pointers and/or classes, but neither are intended for this purpose, and this is therefore awkward.

[8]Note that the following also works if the ordering relation is more complicated, e. g. because the comparison has some finite number n of outcomes, as long as it can order in some sense. It is only for simplicity of presentation that only the two-possibility case is considered.

The drawback is that for large tables this becomes quickly very slow, as for every element a comparison is needed. To speed-up this process so-called trees can be used.

A tree is constructed as follows. It is made up of nodes, and every node can at least store a key-value pair as well as being able to point to two more nodes, like in the linked list of section 6.3. The tree is build in the following way. There is a root node, which contains a key-value pair. The next key-value pair is then assigned to one of the connections, if the key is larger than the key of the node and to the other one, if it is smaller. Equality is assigned to one of the cases by convention, though it is often useful to avoid an ordering principle allowing equality, if possible. If the next node to be added would have to be added to the same connection, it does not replace the connection. Rather, it is checked whether its key is larger or smaller than the key of the connected node. Then, following the same convention which connection to chose, it is attached to that node, rather than to the root node. In this way, the tree is build with the key-value pairs to be stored.

To search if some key-value pair is stored in a tree, it is started at the root. The key is then compared, and depending on the outcome, either the result is already found, or the tree is traversed by either of the two connections. This is repeated until a fitting key is found, or when arriving at a node without suitable connections. In the latter case, the tree cannot contain the searched-for value.

Of courses, nothing guarantees that the tree is balanced, and thus not as bad as a list. However, it is always as least as good as a list for searching, and the more balanced, the quicker. So it is on the average certainly better, and actually reduces search time on the average from order N to order $\log_2 N$.

If there are more than two possibilities, just more nodes are attached at every node.

If a tree becomes unbalanced, it may also be useful to reorder it, and make it balanced again.

Trees have been developed to a very substantial degree, and for special cases much more optimized versions of trees are available. For ordered data sets they are often the best choice. It is therefore useful to search for patterns on the subject if some ordered structure needs storage.

It is often the case that an exact comparison may actually be quite costly in terms of computing time. In such cases the concept of hashes is useful. A hash is essentially a map from some information to another quantity, obeying the following principles:

- A comparison between two hashes is (much) faster than between the original data
- A hash does not need too much memory
- Two different sets of original data may have the same hash, but this should be

unlikely in practice

As is visible, these conditions are all soft in the sense that there is no mathematical formulation of them, and therefore a solution will strongly depend on the problem in question. In particular, the third requirement may be hard to satisfy in practice.

A, quite trivial, example of a hash is a crosssum, or checksum for numerical data. While it usually satisfies conditions one and two quite well, it is only in special circumstances effective at condition three.

Chapter 9

Graphical output

9.1 Generalities

Graphical output is a very involved and extensive topic. Two concepts have to be distinguished here quite carefully, though they often go hand-in-hand in practice.

One is the actual graphical output to the screen, or a generalized screen for, e. g., a VR glass. So far, this can be considered to be a two-dimensional picture. This picture is decomposed in practice in discrete units, so-called pixels, which have unique two-dimensional coordinates, and which can be assigned a color[1]. Thus, in the end, all graphical output is somehow determined by providing color values for all pixels, and animations by a sequence of how the colors of every pixel changes as a function of time.

Therefore, at the lowest level, the operating system provides the program by an API call with a chunk of memory, usually an array, which will take the color values for the pixel, and another API call will then switch to this new output[2]. The programmer is responsible to create graphical shapes, by mapping their exact mathematical form to the pixels. Fortunately, for many purposes powerful libraries exist to create and manipulate many graphical objects, usually using an extended set of classes. This is quite specific to the various libraries, and often OS, and can therefore hardly be generalized.

Note that even output to the command line, like `cout`, is doing somewhere on a very basic level nothing but representing every character on the screen as a set of pixels which

[1]Most systems define a color by a mixture of some numbers. Probably best known is RGB, which defines a color as a combination of the amount of red, green, and blue components, which each range between a minimum and maximum value. All three at maximum is white, all three at minimum is black. But there are many other options.

[2]For an OS using windows, this can also be only part of the screen, e. g. the window or inner part of a window of a program

have to be white (or green or amber or...) on a black (or other colored) background. This necessity is hidden to the programmer in the usage of cout. In fact, even the class cout does not do this directly, but in turn uses the API of an even more low-level library all the way down to the most inner parts of the OS.

The second concept is how to provide three-dimensional pictures to the screen. For this purpose, the procedure is always to have a three-dimensional representation of the picture. For this, there is a third coordinate, a depth, which is usually also discrete. This world volume is then filled. After it is completed, it is projected to the screen by considering a viewpoint, the observer, somewhere behind the screen. The remainder is a geometrical operation to figure out, what can be seen by the observer, and what is hidden behind other objects.

Such calculations, especially the projection, is highly non-trivial, and quite costly in terms of computing time. This is the main reason for the necessity of dedicated graphical processors in computers[3]. Such calculations, as everything, are simplified by libraries, which are again not standardized. An in-depth discussion is therefore also not possible within the scope of this lecture.

However, in both cases in the end the purpose is to fill an array representing the screen, which is then transferred to the screen, and then presented to the user.

9.2 Mathematica

Mathematica also allows access to some direct graphical output using the function Graphics, which also offers the possibility to draw things like lines, circles, etc., so-called graphical primitives to create a more involved picture from it.

However, what makes Mathematica particularly useful in physics (and related areas) is its abilities to draw mathematical data, especially functions.

The simplest possibility is Plot[$function$,{$variable$,$start$,end}], which draws the $function$ of $variable$ in the interval $[start, end]$. Mathematica adapts the output and introduces suitable coordinate axes. However, often the presentation is not entirely as wished for, and a multitude of options exist to adapt the graphical presentation to one's needs and aesthetics. Probably the most important one is AxesLabel, which will provide labels to the axes from the list provided to it, e. g. AxesLabel->{x,y} to provide according standard

[3]Most of the necessary operations is actually linear algebra. Therefore, these processors are just highly optimized and specialized linear-algebra machines. This has led to a widespread use of them also as numerical auxiliary processors, often much faster than the CPU, for numerics, as most numerical procedure are at their heart linear algebra. This is not entirely trivial, as the specialization of these processors also induces complications. This goes far beyond the scope of this lecture.

labels. Of course, any string can be used. Note that the *function* can be either an explicit expression like x+1 or a function f[x], which can be constructed at will. However, it needs to deliver a numerical value for every value of the *variable* inside the range.

There is an important caveat with this. Of course, what is actually drawn is not a function. Rather, it is points, which are drawn, which corresponds to pairs of values for the *variable* and the value of *function* for the value of the *variable*. Never a really continuous function is plotted. Rather a number of values are calculated, and then the pixels in the display are filled by joining these points by lines, in the simplest case straight lines, in more involved cases some complicated spline interpolation. Because of this, fine details of a function can get lost in the process. Thus, it is possible to manipulate by various options the way how the points for which the function should be evaluated are chosen as well as how the points should be interpolated to fill pixels. This may also be useful if the function to be evaluated is very expensive, and more than the usual share should be interpolated.

There are many derivatives of Plot for a particular purpose, like e. g. LogLogPlot to create a double-logarithmic plot or ImplicitPlot to plot a function which is known only implicitly[4], and ParametricPlot which traces a curve in two-dimensional space, and many other. Two are of particular importance.

One is the possibility to plot functions of more than one variable. A plot of a function of two variables is Plot3D, which works as Plot, but now with two variables. This will provide a surface. There is also the possibility to plot equipotential surfaces of a function of three variables, using ContourPlot3D. Similar purposes can be pursued using ParametricPlot3D.

A probably even more important tool in physics is the possible to plot discrete data, i. e. two-dimensional or three-dimensional lists. In the two-dimensional case this is achieved by[5] ListPlot[*list*], where the *list* must be of type {{$x1$,$y1$},{$x2$,$y2$},...}. This is a typical outcome of an experiment: A set of input values and output values. Often, the latter also have an error, which can be accommodated by[6] ErrorListPlot. The required list needs then a third element giving a (symmetric) error or an error bar created using the function ErrorBar, which also allows for asymmetric errors. For the case without error bars there is also a method ListPlot3D, which allows to have a three-element list for the points to specify their coordinates in a three-dimensional space. As with all plot methods, a multitude of options allows to tailor the appearance to most requirements.

All of these methods support to plot multiple input in the same plot. For this, it is only necessary to provide a list. E. g. Plot[{x,x^2},{x,0,2}] will plot x and x^2 over the

[4]Which Mathematica deals with by using NSolve to find the necessary results. This also needs to include the corresponding library by <<Graphics‘ImplictPlot‘.

[5]If the points should be joined by a line, use ListLinePlot.

[6]After loading the package with Needs["ErrorBarPlot“].

indicated range. Mathematica will automatically assign different plot styles to make the graphs distinguishable. Also how this done can be modified by options.

However, this does not allow to plot, e. g., a function and a list simultaneously. This is done using the command Show, which requires a list of plots. As Mathematica is a functional language, any evaluation of plot functions is not just creating the plot. Rather it creates a return result which is the plot, which is then used by Mathematica to make the graphical output. But as such, it could be stored, e. g., in a variable, or can be passed to other functions, in this case Show. Show will then create, as good as possible, a single graph with one common set of axises, to display both results. For this purpose, both plots must be of the same dimensionality.

E. g.

```
Show[ParametricPlot[{u+1,u-1},{u,0,3}],Plot[x^2,{x,0,3}],PlotRange->All]
```

shows a parametric plot and an ordinary plot together in a single graph. The option PlotRange rescales the plot such that both graphs are completely shown. Otherwise only the range of the fist graph would be shown. Also, Show does not adjust the plot style of the graphs, so to optically separate both would require to provide corresponding options to the two plot functions.

There are many more options like legends, titles, multiple axises etc. This includes the possibility to save figures in several of the more common picture formats.

9.3 C and other high languages

Languages like C(++) provide multiple layers for dealing with graphical output.

The lowest layer is the API of the underlying operating system. This usually depends on the operating system, and is therefore not portable. Depending on the operating system, it can provide raw access to the memory array representing the screen up to rather high-level constructs like windows and user-interface elements. All of these are provided by libraries and corresponding header files. Most modern operating system do so usually in forms of classes and OOP, but not necessarily so.

Above this level exist, more or less, platform-independent graphic libraries. These provide the same interface on several or most operating systems. They also exists from rather direct access to graphics, like OpenGL, to highly abstracted levels which provide things like windows and user interface elements, like Qt for C(++). Some of these are part of libraries belonging to certain languages, like AWT and Swing of Java, some are, to some degrees, language-independent.

Finally, there are also special-purpose libraries, which may or may not be platform-dependent. They are tailored for a particular purpose, and provide thus often not full support of everything a graphical system can do. An example is the ROOT library, which was developed by CERN for the particular needs of particle physicists, and provides, like Mathematica, extensive possibilities to draw data from typical physics situations. It also offers rudimentary support for other purposes. At the same time, as with many such specialized libraries, it has extensive non-graphical features, in this case the ability to manage and manipulate (large) amounts of data.

This short, non-exhaustive lists, gives an idea of the multitude of possibilities to do graphical output in C(++). It is extremely flexible by the library concept. However, the more efficient it needs to be, the mores specific it will be. The choice of approach should therefore always be governed by the purpose. This purpose must also respect questions like maintainability and a reasonable ratio between effort and effect.

Chapter 10

Objects and classes

Within the previous chapters all necessary information to program using functions have been established. Here now the necessary additions for doing OOP will be introduced. OOP is actually a far wider field than programming with functions, and thus it is only possible to scratch the surface.

Mathematica, as a primary functional language, does not have objects in the same sense as C++. Here, expressions are the basic entities. Thus, also this chapter applies (almost) only to C++.

10.1 The object-oriented paradigm

The idea of the object-oriented (OO) paradigm is that the basic elements to work on are entities called objects, which contain all relevant data and functions to work with the entity. The structure of such entities are created from blueprints, called classes. Thus, just like int is the blueprint for a variable containing an integer, a variable declared as int is the entity to work with. In a sense, classes are thus struct augmented by functions.

If this would be all, little more would be needed to be said about OOP. But there are two more important concepts associated with OOP, encapsulation and inheritance.

Encapsulation is associated with the idea that it is not necessary to know how an objects does what it does internally. Even more, it could be harmful to interfere with it. As a consequence, a class should only offer an interface to an object, but should not provide all details of it. Thus, a class can have parts, which are not visible to the outside world, and what the class offers is only an interface to the entity, but not the entity as a whole. Pushing the idea further, the possibility of direct manipulation of the data of an object from the outside is often able to interfere with its inner workings. This would require more overhead to ensure that all variables contain sensible information. Rather

than doing so, it is better to not provide direct access to any variables, but only functions. Then, so-called getter and setter methods replace direct access to variables, and contain all the necessary logic to avoid non sensible states of the objects. This will be discussed in more detail in section 10.8. The object being derived from such a class is called an instance of a class, and this will be discussed in more detail in section 10.4.

Inheritance is based on the recognition that objects can offer a new possibility for more ease of maintenance. Because often only slight changes to a class are necessary to adapt it for a more specialized or different purpose. To support this, it is possible to derive a class from a class. In this process, a class inherits, thus the name, all of the interface of its parent class, and can extend it further. It is possible for a class to have multiple children[1]. This will be discussed in more detail in section 10.7. One advantage of this is that it is possible to select the actual instance during runtime of the program. This is called polymorphism, and will be discussed in more detail in section 10.9.

10.2 The object paradigm

While functions are already a big step forward, they specialize on instructions only. However, from a logical perspective, it would be best to combine somehow functions with data. Think, e. g., about drawing squares on the screen. If you want to draw multiple squares, using functions to reuse the code for drawing makes a lot of sense.

But now every square has a size and a position. It seems hardly logical to separate the ability to draw a square and its position and size. This issue is captured in so-called object-oriented languages, leading to object-oriented programming (OOP), which is arguably the most common programming style today. C++ is one such language.

The idea is to define objects as a collection of data and functions operating on the data. Then different objects have the same functions, i. e. abilities, but differ in their data contents. Since both are connected, it is never a problem of not having sensible functions operating on the data available. This encapsulation can also be used to avoid manipulation of the data. E. g., if instead of being able to directly change the size of the square, there is a function to do it, the function can check the new value to be positive, and ignore the change if it is negative. This avoids having non-sensible data for the next draw command. This helps very much in creating safe code, i. e. code without errors.

Also, it is no longer necessary to keep the code and the data separately, and thereby risking not having one or the other available. Everything is in one place. In fact, in many

[1]It is also possible for a class to have multiple parents, so-called multiple inheritance. This is a quite involved topic, as this gives rises to many potential conflicts. It is thus beyond the scope of this lecture.

more modern languages, e. g. Java, the concept of data has been completely replaced by that of objects, i. e. every chunk of data can always carry functions with it.

The downside of objects is that they are essentially always associated with additional overhead - if the data is moved around in memory, so have the functions to be. This can reduce the efficiency of programs. Especially in physics, where large problems need to be solved requiring sometimes billions of hours of runtime, it always needs careful consideration, whether the structural advantages of objects compensates the cost in computing time. Hence, in physics nowadays a mixture of object-oriented and function-oriented code exists. Languages like C++, which support both possibilities in the same code, are therefore particularly suited for these problems. Also, very careful considerations of what aspects of object orientation is included and very efficient compilers can make some purely object-oriented languages, like Java, almost, but not quite, as fast as C. But this requires a very good understanding of the inner workings of an object-oriented language.

Definitions of objects are also highly reusable. The definition of an object is also called a class, and the actual use of them, the objects, are called instances of the classes. Thus, objects have data, classes declare that they have data. Functions of objects can be used, functions in classes are declarations of what could be done with an object.

10.3 Classes

The first step is thus to define such a blueprint. An example is provided in listing 10.1

Listing 10.1: Defining a class.

```
1 class rect {
2  public:
3   double x;
4   double y;
5   void Draw(void);
6   double Area(void) { return x*y; };
7   void Reset(void) { x=0; y=0; };
8   bool AreaGreaterZero(void) { return Area()>0; };
9 };
```

Note the ; at the end in line 9, compared to function declarations.

A class can be declared, wherever the struct of section 3.9 can be declared. However, it is common to declare a class in a header file, and quite often every class in a separate header file, even if its declaration is quite short. For ease of use, a single header file for a

collection of classes can be then declared, which only contains a list of includes for every file.

A class is defined as a block started by class followed by the *name* the class should carry, in this case rect, followed by a block of its content. Thus, the declaration is, apart from a different keyword, similar to a struct. Inside the block in line 2 the so-called access modifier public appears. Access modifiers will be discussed in more detail in section 10.8. For now it is sufficient that this gives access to the following declarations from outside the class.

After that follows two variables in lines 3 and 4, declared in the same ways as in a struct. After that the functions follow in lines 5 to 8. Conceptually, the variables will hold the extent in x and y directions of the rectangle.

Note that the body of a function inside a class can be given in two different ways. One is directly inside the class, as is shown in line 6, and can include initializations. It also shows how inside such a function access to the variables work: They are just accessed like arguments. Thus, in a sense, all variables of a class are passed as arguments to any function of a class in a call-by-reference fashion. Thus, they can be changed inside the function. This is exemplified in line 7. Calling Reset will set the variables x and y in the given instances of a class to zero. Note that they will not set the values of all instances to zero, only of the instance on which Reset is called.

If it is desired that all instances share a variable, then it must be declared static. Then all instances work on the same variables. Note that it can otherwise be used as every other of the variables. It is also possible to define variables as const. This is often encountered together with static, to obtain class-specific constants[2]. Note that memory must be allocated for a static variable. This needs to be done in a code file, not a header file, just as with global variables. It must also occur in the same scope as the class has been defined. This is done by adding the line *type classname*::*variable*;, and may have an initialization. This must be done for all static variables.

The function Draw in line 5 is not declared inside the class definition. It needs to be declared somewhere else[3]. To do so requires to notify the compiler that it belongs to this

[2]Note that static variables can be given a value without having an instance of the class, but this requires a corresponding variable. This requires various considerations, and will not be discussed in detail.

[3]It is possible to define a function without giving its details. In such a case the class is called abstract, and no instances can be created from it. This is useful if eventually multiple children of the class should be derived, and they should have different implementations. Abstract classes also play a role in multiple inheritance. Note that as an abstract class essentially only defines an interface, abstract classes are sometimes just called interface.

class. This is done using the scoping operator,

```
void rect::Draw(void) {...
```

and the body follows as usual. Note that even in this case the variables of the class are passed-by-reference to the function. So, except for the different location, there is no difference of whether the function is declared inside the class body or outside. It is usually done to separate declaration from implementation, which can have maintenance reasons or simply that only the interface should be available to someone, but not the code, e. g. for commercial libraries.

Other functions of a class can be called inside any function of a class without further qualification. This is exemplified in line 8. Again, the functions are called which belong to a given instance of a class.

Also functions can be declared as static, and can then even be called without an instance. However, static functions can only access static variables. Given a static function void bar(void) in a class foo, it would be called as foo::bar(); and thus again with the scoping operator.

The variables and functions of a class are called members of the class.

10.4 Instances

After a class has been created, an instance can be created. Obtaining an instance of the class declared in listing 10.1 is done as

```
rect box;
```

similar to struct.

The elements of the object can be accessed using the . operator. Thus, box.x gives access to the variable x, which can be used as any other variable. Note that this always gives the variable associated with the given instance. If there are two instances, e. g. a second one called box2, then box.x and box2.x are two different variables. In the same way, functions can be called, e. g. box.Draw() - again, the function uses the variables of the given instance.

It is also possible to create an instance dynamically, e. g. rect *box=new rect;. Consequently, the memory can again be freed by delete box;. To access it members now, as for a struct in section 6.3, the access operator is -> for both variables and functions, e. g. box->y. It is not necessary to dynamically allocate memory for the variables of an object - as for a struct they are allocated in one go by allocating the memory for the object. It may

seem strange to allocate memory for an object as it contains functions. What actually happens is the memory is allocated for function pointers of section 5.9, pointing to the functions of the class declaration. This will play an important role in section 10.9.

Note that, as for struct, the comparison of two objects will not compare the variables, but actually whether the two instances are the same. Thus, two instances which have the same values for all variables are still not equal. Such a test would require that the class provides a corresponding function.

As for structs, it is also possible to declare arrays of instances of a class, e. g. rect boxes[5].

10.5 This

It is sometimes necessary for an object to have access to its own instance. This is realized by the this pointer. Every object has a pointer to its own instance. Thus, if an instance is declared as rect box; this contains the same information as &box. The this pointer is available in all non-static functions, and will always point to the instance for which a function is called. It cannot be accessed from outside the class - there &box needs to be used.

Consider, e. g., the following two functions to be added to the example of listing 10.1.

Listing 10.2: Using this.

```
1  bool Compare(rect *box1,rect *box2) {
2    return ((box1->x==box2->x)&&(box1->y==box2->y));
3  };
4  bool CompareTo(rect *box) { return Compare(this,box); };
```

The function Compare allows to check if the two arguments describe a rect of the same size. The function CompareTo allows to compare some rect to the given instance. Internally, rather than copying the necessary code from Compare, and thus requiring to change both codes if something changes, this is done by calling Compare with one argument being the instance itself, as signaled by the this pointer as an argument.

10.6 Constructor and destructor

It happens regularly that a newly created instance needs to initialize itself. Also, especially when it allocated memory, it needs to do cleanup before it ends its life. In principle, this can be done by providing two functions, say Init and CleanUp, which needs to be called

after the object is created and before it is freed. However, it then depends on the user of the class to actually do so. This is rather unreliable, especially when it comes to memory management. As a possibility to force initialization and clean up there exists the concept of constructor and destructor.

Constructors and destructors are functions. However, they do not have a return type, as they are not called explicitly, but implicitly during the creation and destruction of an object. E. g., for the class of listing 10.1, it could be added the following parts in listing 10.3.

Listing 10.3: Constructor and destructor.

```
1  static unsigned int rectcount;
2  rect(void) {
3   x=0; y=0;
4   rectcount++;
5  };
6  ~rect() {
7   rectcount--;
8  };
9  rect(int nx,int ny) {
10  x=nx; y=ny;
11  rectcount++;
12 };
```

It is assumed that in the memory allocation for the static variable staticount somewhere the initialization is int rect::rectcount=0; such that it contains the value zero at program start.

Lines 2-5 define the constructor. As is seen, a constructor has to have the same name as the class. It also has no return type. Here, it also has no arguments. This can be changed, as will be discussed below. It is used here to give x and y sensible initial values, and to increase the static variable rectcount - by this the number of instances can be tracked.

For the purpose that this really works, the variables needs to be reduced by one if an instance is removed. This is cared for by the destructor in lines 6-8. The name of a destructor needs to be the same as that of the class, but with a leading $\sim$. It can have no return type, and has always an empty argument list, not even void. The reason is that objects may be destroyed when leaving a block. Therefore, the destructor is called from the compiler, and therefore needs to be unique. The destructor will thus always be called whenever an object is destroyed, either when leaving a block, or by an explicit call using

delete.

A destructor is automatically called whenever an object is created, thus

```
rect box1,box2(),*box3=new rect();
```
(10.1)

will all call the constructor rect(void). However, constructors can be overloaded. This is done in lines 9-12. Here, an additional constructor is declared, which takes initialization values for the box size. Declaring rect box4(1,1); will use this constructor. For the overloading of constructors all applies what has been said for the overloading of functions in section 5.4.

Note that constructors or destructors cannot be used as ordinary functions.

10.7 Inheritance

Arguably the most powerful feature of OOP is inheritance. Inheritance allows to create a class as a modification of another class. The idea behind inheritance, and which thus guides how it works, is that a child derived from a parent class is more specialized than its ancestor. It is also possible to derive a class from a class, which has been derived from another class, which is therefore even more specialized[4]. This idea, a top-down approach, should also be behind the design of such class trees. The least specialized class is at the root, and from this unfolds ever more and more differentiated and specialized classes.

Note that if two classes are derived from the same parent class this implies no further relation between these two classes. However, they will still use both the same static members defined in the parent class. This can yield unexpected side effects, one reason why static members should only be employed with great care.

For that purpose, the new class starts out with everything the old class has, and can add things. To create a colored rectangle as a derivative of the class rect, this would be done as in listing 10.4.

Listing 10.4: Inheritance.

```
1 class crect:rect {
2  public:
3   int color;
4   crect(void) {
5    color=0;
```

[4]It is possible to derive a class from multiple classes. This multiple inheritance has unique issues of its own. While pretty standard in modern languages, the difficulties involved moves this topic beyond the scope of this lecture.

```
6     };
7     void Reset(void) { x=0; y=0; color=0; };
8  };
```

Using a parent class is done by just adding a : after the class name and giving the parent class, as done in line 1. Then, without explicit declaration, the new class has immediately all variables and functions of the old class. In addition, it can add to it, as is done in line 3, where an additional member of type color is added.

If no constructor or destructor is provided in the child class, the constructor or destructor of the parent class is used. If one is declared, as here in lines 4-6, it is not called alone. Rather, all constructors, which have been overloaded, in an inheritance chain are called, starting with the class which has no ancestor, and then following the inheritance path. Thus, first rect would be called, and then crect would be called. This cannot be avoided, as the idea behind inheritance is that the descendants are specializations of their ancestors. Therefore, whatever they do should also be done for any descendants.

The only thing what can happen is the following. The derived class does not have a constructor with two integer arguments. As a consequence, this is interpreted as that the specialization no longer provides this functionality. Therefore, a declaration like crect box(1,1); will not work.

The same applies to destructors. However, in this case, the order of execution is reversed, starting with the last descendant, therefore having the most specialized destructor, and working then down the chain of ancestors.

The new class can alter the behavior of functions existing previously, as is demonstrated in line 8, where a new version of the function Reset is added, now also affecting the color. After a declaration like crect cbox; a call cbox.Reset(); will now call this new version. This is generically true: If something of the same name is declared, it will always overwrite what has been there in (any of) the ancestors of the same name. However, it is possible to still access them using the scoping operator and the name of the desired ancestor class. E. g., an alternative way to implement the new version of Reset would be

```
void Reset(void) { rect::Reset(); color=0; };
```

In fact, from the point of view of clean design, this version is preferred over the version in listing 10.4, as functionality is not duplicated. This avoids inconsistencies, if anything in the ancestor's version of Reset would be changed, which would then needed to be changed also in all descendant versions. This follows the line of thought that a child class only adds functionality to the parent class, but never takes functionality away. The later would be achieved by writing an own version of Reset, which would do something entirely different.

10.8 Private, public, and protected

It is now time to come back to the access modifier public which appears, e. g., in line 2 of listing 10.4. As noted, one of the aims is to encapsulate data and functionality inside a class. The problem is now the following. In the current version of the class rect it is possible to modify the values of the variables x and y. The class cannot detect any changes, and therefore cannot ensure they make sense. Thus, if somebody would do on an instance box the sequence box.x=-1; box.y=1; the function Area in line 6 of listing 10.1 would yield a negative area. Also, Draw may no longer work properly.

The obvious question is, why would any programmer want to do so, as she or he should obviously know that this makes no sense. However, in general, there are several possibilities how this can happen. A bug. Or, more bugs somewhere else in the code could lead to this, and thus creating chain reactions whose source cannot easily be traced. Indeed malevolent behavior, which is unfortunately not rare, as the problems of viruses, Trojans and the like illustrate. Situations where user input is directly piped to the object. That the values are results of involved, possibly non-deterministic, calculations, which are not easy to check for validity - after all, not everything is as simple as this example. Or, especially if the class is part of a library, the programmer in fact cannot know what admissible values for a variable is, as this follows from some very involved procedure.

No matter the origin of the problem, this leads to the idea to make encapsulation explicit. For this purpose, the access modifiers have been created. The access modifier public: declares that all following members are accessible as in the way discussed before. The alternative is the access modifier private: which declares that all members following the declaration are only visible from inside the class.

Thus, a solution to the problems would be a modification of the class as shown in listing 10.5.

Listing 10.5: Usage of access modifiers.

```
1 class rect {
2  private:
3   double x;
4   double y;
5  public:
6   void SetX(double X) { x=(X>=0)?(X):(x); };
7   double GetX(void) { return X; };
8   void SetY(double Y) { y=(Y>=0)?(Y):(y); };
9   double GetY(void) { return Y; };
```

```
10    rect(int nx,int ny) {
11     x=0; y=0;
12     SetX(nx); SetY(ny);
13     rectcount++;
14    };
15   ...
```

In this version of rect the variables are now private, and can therefore be not accessed from the outside. To modify them, the function SetX and SetY need to be used, which are declared in lines 6 and 8. They only accept positive values, and therefore ensure that the rectangle is always well defined. The constructor in lines 10-14 demonstrates that even in the public section of a class, the class itself can access all its private members, as is shown in line 11. It also shows that if it receives new data for its variables as happens for this constructor it also uses the same functions to set the values in line 12, after creating a sensible initial state. The idea behind this is that this avoids to replicate the logic to check the validity of new values for the variables. Thus, if this logic should change, there is only a single place in the code where changes need to be made. The private: modifier not only forbids to write to variables, but hides them altogether. Therefore, if variables needs to be read, also suitable member functions for this purpose are required, which are here provided in lines 7 and 9. Of course, the present example is quite simple, but in general this approach is followed even for far more complicated situations.

It should be noted that also functions can be declared private. Then they are only accessible from inside the class. Also variables are not needed to be made accessible by functions from the outside by functions to set and get them (so-called setters and getters), and can only be known inside. This is a design decision, which should be guided by the need-to-know paradigm: If something needs not be known to use the class in its intended purpose, it should not be visible outside. In fact, making all variables only accessible by getters and/or setters is often considered good style, and some languages have additional facilities to make this more elegant. Furthermore, functions setting or getting variables can have further consequences. E. g., in the present case void SetX(double X) { if(X>=0) { x=X; Draw(); }; }; would make a lot of sense, as when the rectangle changes, it may need to be redrawn.

Note that there is no need to have only a single public and a single private section in every class. The keywords can be repeated as often as one needs, and every time the access for the following members conforms with the last previous access modifier. If no access modifier is used, as a consequence of the problems noted above, the default is private.

Private is in fact a very harsh restriction, as also child classes of a class cannot access

private members. To make this possible, but restrict access only to descendants of a class, a third access modifier can be used, protected: which works otherwise in the same way. Again, the design rule is to make as much as possible private, but as much as necessary protected. Note that it is not even possible using the scoping operator to reach a private member of an ancestor class.

Access modifiers can also be used when declaring from which class a class inherits, and this changes the access modifier to the more restrictive version of either the inheritance modifier or the one in the base class. E. g. a declaration like class crect:private rect... would make all public members of rect private in crect. However, a declaration like class crect:public rect... does not make any private members of rect public in crect. The default is no change. Thus, it appears that public has no effect, but this is not true as discussed in section 10.9.

10.9 Polymorphism, early and late binding, and virtual functions

Inheritance makes classes and objects already an extremely versatile concept. An even stronger boost is obtained by the concept of polymorphism. So far, objects where considered to be strongly type cast, i. e. every object must be of the type it declares. Polymorphism makes this more flexible. A pointer to an object is now allowed to also point to any of its descendants. To make this possible requires that the class from which it is derived is derived publicly, i. e. the access modifier for the parent class must be public.

If this is the case, it is possible to declare

```
rect *box=new csquare();
```

This may seem a bit strange at first. Why should this be done? The answer is that if there are multiple descendants of rect, e. g. an array of rect could hold any of these.

It appears still to be not useful. E. g. if calling box->Reset(), this would execute the Reset version of rect, instead of crect. To change this the concept of virtual functions come into play. Using the keyword virtual in front of a function declaration in the parent class instructs the compiler to use the version of the function belonging to the actual instance, rather than of the class as which the pointer is declared. Thus, if line 7 of listing 10.1 would be changed to virtual void Reset(void) { x=0; y=0; };, the appropriate version would be called. Note that this is an alteration to the base class, not to the class of which the actual instance is derived of. It has therefore to be incorporated in the most ancestral class declaration from which onwards this should be possible. Note that it is not necessary

to declare the corresponding functions in the descendant classes also as virtual. Once the function has been declared as virtual, it remains so in all descendants. However, keeping the virtual modifier is good style to highlight this feature. This feature is particularly important for destructors. Since the appropriate cleanup for the instance, rather than for the parent class, should be done, they should usually be made virtual.

The actual contents of a pointer could change at runtime. Consider, e. g., the situation that there is a second child of rect called square. Then a valid declaration would be

```
rect *box=(besquare)?(new square()):(new csquare());
```

Which version of Reset should now be called depends on whatever value besquare has at runtime. If this changes because of user input, there is no way the compiler can predict this. However, because Reset is defined virtual, always the version appropriate to the instance is called. This is the true power of polymorphism: It allows to change the things upon which is operated on at run-time, rather than at compile time, still ensuring consistent behavior. Because this decision is made upon run-time, this is also called late binding. In contrast, if the function is not virtual, the compiler can implement it already during compile time. This is called therefore early binding. Of course, without dynamical allocation the declared class and actual class coincide, and thus always early binding is done.

Technically, this is realized by the program by checking at run time[5], essentially with a type of if, what the actual content of the pointer is, and then selecting the corresponding function, based on the result[6]. Thus, the use of virtual functions always comes with an additional overhead, and thus at a price in performance. It should therefore be avoided, if performance is of central importance.

It should be noted that constructors cannot be declared virtual but this is also not necessary. After all, the new statement needs to already reference the actual class to be used for the instance. It is sometimes necessary to assign an instance referenced by a pointer to an ancestor class to a pointer of its own class, or some class in the hierarchy between the ancestor and its own class. This can be done using dynamic_cast< *targetclass* >, a generalization of the case of the static_cast of section 3.5. Note that a dynamical type cast is done on faith by the compiler. As it occurs during run-time, it is not possible to check its validity during compile time. It is up to the programmer to ensure that the cast makes sense.

[5]This so-called run-time type info is also available to the programmer using the typeinfo functionality. However, it is considered bad design if this is necessary, even though there exist circumstances where it is not avoidable. This is beyond the scope of this lecture.

[6]This is usually realized by a lookup table, which needs then to be added with all possibilities to the pointer.

10.10 Abstract classes

To improve design the concept of abstract classes has been developed. An abstract class is a class, which defines an interface, i. e. set of functions, but does not implement them. This can be either achieved by not providing a block for them, or, more explicitly, by

virtual void Do(void)=0;

and thus making them explicitly purely abstract. Note that abstract functions necessarily need to be virtual to make sense.

No instances of an abstract class can be created, as otherwise undefined functionality would exist. While it is technically possible to add data and implemented, and non-virtual, functions to an abstract class, this is not their intention. An abstract class should define an interface and set of functionality, and not functions.

10.11 Templates

Another possibility to increase flexibility are templates. The idea behind templates is that certain groups of types, and also classes, have common features. If some functionality only requires this feature, it should not be necessary to create the code once for every type.

This is possible for functions. As an example, consider listing 10.6.

Listing 10.6: Use of template functions.

```
1  template <typename T> T DoubleSubtract(T x,T y) {
2   return x-2*y;
3  };
4
5  template <typename T,typename R> T DoubleSubtractMixed(T x,R y) {
6   return x-2*y;
7  };
8  ...
9  int a=3,b=4,c;
10 double c=5,d=6,e;
11 c=DoubleSubtract(a,b);
12 e=DoubleSubtract(c,d);
13 a=DoubleSubtractMixed(a,d);
```

A function with the desired feature needs to have the leading template declaration. Afterwards, in the <> angles the placeholders for the types are given, lead by the keyword

typename. This is exemplified in line 1. The placeholder for the type is in this case is called T. As can be seen in the rest of the definition in lines 1-3, it is used as any other type.

It is then used with different types[7] in line 11 and 12, yielding corresponding results. However, there are two restrictions. One is that all functionality used inside the templated function are provided by the types actually used, in the form of (overloaded) functions. In this case, this is subtraction and multiplication. Thus, calling the function with two, say, filestreams, would yield a compiler error. Secondly, the types are strongly checked. Thus, a call like DoubleSubtract(a,c) would yield a compiler error. To make such a call possible would require a different function, as defined e. g. in lines 5-7. Note however that the strong typing also persists to the return type. Thus, even now a=DoubleSubtractMixed(c,b); would not work, but e=DoubleSubtractMixed(c,b); does. What is still possible is to act with a type-cast to perform conversions of any of the parameters.

Of course, templated functions can be used for classes as normal members. It is, however, possible to also create templated classes. Consider a templated version of the rect class in listing 10.7.

Listing 10.7: Use of template classes.

```
1 template <typename len,typename area> class rect {
2  private:
3   len x,y;
4  public:
5   area Area(void) { return static_cast<area>(len*len) };
6   ...
```

Here, the class is templated with two types, which have the suggestive names area and len to indicate their purpose. The data members are now of this type, while the function Area now returns the corresponding type. However, because such operations are strongly typecasted, an explicit type cast is done in line 5, even though it is not strictly necessary, as line 6 in listing 10.6 shows. Of course, all functions have then to be typecasted, including constructors. To get an instance using, say, float for the coordinates and double for the area would be require a declaration like

```
rect<float,double> box;
```
(10.2)

[7]Note that in this case also the build-in type-casting would do the corresponding job, but any example requiring this would be too complicated to exhibit here. So assume for a second that automatic typecasting between int and double would not work.

10.12 Exceptions

So far, error handling was a quite obnoxious procedure. In principle, everything which could create an error needs to be wrapped in if statements, to avoid errors before they happen. Even under the best of circumstances, not every error can be foreseen. Also, the proliferation of ifs makes the code hard to read and hard to maintain. Therefore a possibility to separate error handling and code has been devised, the so-called exceptions.

As the name already indicates, it is not really errors which are the only purpose for them, though it is arguably the most common one. It is intended as a possibility to deal with exceptional circumstances, of which the occurrence of an error is only one.

The basic structure of how to handle exceptions is given in listing 10.8

Listing 10.8: Exception handling.

```
1  try {
2   //Doing something
3   ...
4   //Logical error detection
5   if(oops) throw -1;
6  }
7  catch(int exc) {
8   if(exc==-1) cout<<"Something went wrong"<<endl;
9  }
10 catch(exception &e) {
11  cout<<"Caught standard exception"<<endl; delete e;
12 }
13 catch(...) {
14  cout<<"Something unforeseen happened"<<endl;
15 }
```

The basic starting point is the try block, which is initiated by a try keyword in line 1, followed by the block from line 1 to 6. In this block is the code which should be supervised by the exception handling.

To this end, various possible exceptions are caught in the three catch blocks in lines 7-15. These catch blocks have to immediately follow the try block. There can be as many as wanted, and each starts with the keyword catch and is followed by a block. Just like in a switch statement, they are transversed until a fitting block is encountered. If none is found, i. e. the exception is not handled, the exception is considered to be a show stopper, and the program terminates. It is possible to nest try blocks, and then the exception

is escalated to the enclosing try block. But again, if unhandled, eventually the program terminates. Note that an exception can also escalate to outside a function in which it occurs, and can then be caught by the code calling the function, if the function call occurs inside a try block.

If no exception is encountered, i. e. the last line of the try block is executed without problems, the program will resume operations after the end of the catch blocks, and thus none is executed. If an exception occurs, the try block will be left at this point, and the corresponding catch block will be executed. After this, the program will continue after the catch block.

To avoid crashing, a possibility is to declare a catch-all catch block. This is done in lines 13-15, and it is declared by the (...) after the keyword catch. However, for such a catch to make sense, it needs to guarantee that after its execution the program is in a viable state to commence operation. Just noting a problem, as is done here as the only treatment in line 14, is usually not a wise course of action.

An exception handler can get information from the try block, and needs to declare this information as an argument. This is shown in lines 7-9, where the information is passed on as an integer. This integer can then be used as an ordinary variable in line 8. Note, however, that its lifetime ends after the catch block. The only exception is, if the passed variable is a pointer to data. This data needs to be dynamically allocated inside or before the try block, and needs to be freed afterwards, either in the catch block, which is error prone, especially if there is more than one catch block, since it then needs to be freed everywhere, or after the catch block.

The variable passed can be a class instance. This happens in line 10-12, where a pointer to an instance of the class exception, which is declared inside the header exception coming with C++, is provided. Alternatively, this can be any arbitrary class, but standard is a class inheriting from the class exception. As is seen, it frees up the corresponding memory in line 11. While this gives the possibility to provide more complex information on the error, the real advantage arises when combining this feature with the polymorphism of section 10.9. Then, the pointer can contain some derived class, and using virtual functions can provide much more specific information. In fact, testing the class type and performing a dynamical type cast can give access to much more detailed information. In fact, the library on exceptions inside the header exception provides a class hierarchy for the various exceptions thrown by library functions, e. g. the class bad_alloc is used in an exception in many library functions if allocating memory fails[8].

[8]It is not new which throws this exception. The operator new just returns NULL. By testing on NULL the library functions decide whether to throw an exception.

How this is actually done is demonstrated in line 5. It is possible to trigger the exception handling also inside the code, and not only by relying on something else. In line 5 an exception is created explicitly by the keyword throw, because something went wrong, signaled by the Boolean test on the variable oops. This is done utilizing an integer variable, and therefore will be captured by the block in lines 7-9. This allows to provide ones own exception handling mechanism. Note that the try block is then left at this point, and after finishing the catch block the code continues after the last line of the catches. To reach the handler in lines 10-12, an alternative would have been throw new exception();, which also would have allocated the required memory for the class, which is freed in line 11.

As with library functions, using throw is also possible without a try block. This then requires whatever code calls the function to provide exception-handling abilities, or otherwise this will also lead to a termination of the program. Thus, the ability to throw exceptions should be documented[9].

10.13 The standard template library

C++ comes with a large range of classes for many different purposes, such as resizable arrays, sorting, maps, file and screen input and output, string manipulation and many, many other purposes. All of these are subsumed in the so-called standard-template library (STL), which can be included as usual, having the namespace std. As the name suggests, the library makes heavy use of templates.

It would be by far too repetitive and lengthy to just only list everything contained in the STL. It is highly recommendable to have a look at the corresponding documentation in case something is needed, as there is a good chance that a solution is provided in the STL.

10.14 Refactoring

Now that much more powerful paradigms are available, it is time to take a step back and reconsider what has been programmed before.

Consider, e. g., the examples 1.1 and 1.2, which are in pseudo-code. They will have

[9]Earlier versions of C++ and other languages provide keywords for this for the function declaration. In C++, this is now assumed to be the normal case, and therefore no keywords are provided. However, it is good documentation to provide information about all possible exceptions a function could throw on its own, besides any which may originate from other circumstances.

to be still fundamentally changed to actually work. This is not rare. Such a significant change in code, which is essentially a complete rewriting, is called refactoring.

A need for this arises when more features are added to programs, and they thereby grow beyond their original purpose. Then it often happens that the original design of the code, i. e. the structure derived from the questions of section 1.1, becomes incapable of sustaining efficiently the new purpose. At this point, it becomes necessary to change something, or otherwise the code may become no longer maintainable or extensible. This is the occasion calling for refactoring.

When refactoring code, nothing is changed in its functionality. In fact, when refactoring any changes need to be avoided. The only changes are made to the underlying structures. The design, or architecture, is changed to be more flexible and easier to maintain. Often, this does not affect the whole program, and outer layers, e. g. user interaction, can be kept. Only how the core is organized is changing, though this often requires recoding of substantial parts of the internal structure.

Refactoring is, in a sense, an emergency break. It is necessary to make a program become able to grow beyond its original purpose. It is something happening in physics not too rarely. It is also not a contradiction to section 1.1. Section 1.1 is about how to start with the information present and known at the beginning. These may change over time. It is often (much) simpler too change the program to adapt, rather than to restart every time something new comes up.

Chapter 11

Some useful patterns

Just as in physics the same equations have the same solutions, no matter from where they arise, also in programming the same problems have the same solutions. Solutions to problems appearing in multiple contexts are known as patterns. Here, a few patterns will be discussed, which are quite useful. Such patterns are solutions to problems. They are therefore not specific to a programming language, here Mathematica or C(++), and are therefore not necessarily formulated in a specific language. In particular, every pattern can be realized in every language. But it may happen that they are much simpler to realize in some languages than in others.

11.1 Sorting

One very often appearing problem is sorting, i. e. ordering sets on which a (binary) ordering principle is defined. Simple solutions scale badly with the number of elements to be sorted, usually at least as bad as the number of elements squared.

But, sorting can be done such that it only scales (on the average) like the number of elements times the logarithm of the number of elements. This is done using the so-called quicksort algorithm. The idea is again based on a divide-and-conquer strategy, and follows a similar idea as the tree structure of section 8.7. It works by reducing the problem to the problem of sorting two elements.

The idea works as follows: Select an (arbitrary) element of the list of elements[1]. Then put all elements larger (or equal) that this in a new list, and the others in another new list. After this, repeat for the new lists. This works better the more equal the number of

[1] Actually, cases can be constructed where a bad choice will slow down this algorithm such that it takes again as long as naive sorting. There are algorithms, like heapsort, which have no worst case, but have on the average a larger coefficient.

elements larger or smaller than the chosen element is. In the end, combine the list in the order of the splitting elements again into a single list, which is now sorted.

This procedure can be implemented most easily using a recursive routine. It is easier to implement when using a second list to hold the results, but can also be created by swapping elements if memory is insufficient, but this is somewhat slower.

11.2 Event-Listener

A situation arising in interactive environments is often that a reaction is necessary if some external input takes place. This may be the pressing of a key on the keyboard, an information coming from a server or the output of another program.

To deal with this problem the event-listener pattern is suited. It is usually implemented in OOP, but could also be realized using the function pointers of section 5.9. It usually involves two to three classes. One class represents the input type, e. g. the keyboard. A second class, usually an abstract class of section 10.10, is a so-called listener class. From this ancestor class classes are derived which implement the possibility to interact with classes interested in the information.

To realize this relies heavily on polymorphism. The class representing the input provides the possibility to add to it a polymorphic pointer to the listener class, and thus to its childes. The listener class has a virtual function member, which is called for all added listeners by the instance of the input class whenever an event occurs. The descendant of the listener class created for the purpose of the class which wants to have the input that uses whatever device it has to push the information through to the interested class. This is usually done by the listener descendant by having a pointer to an instance of the interested class, and which then can call a function member of this class.

Using here polymorphism allows the input class to have no clue about who will be ultimately interested in the information. This makes the pattern so flexible.

11.3 Data-Observer

A very similar concept to the event-listener pattern of section 11.2 is the data-observer pattern.

The idea is that whenever data is changed, e. g. an element of an array, somebody maybe interested in this. If the data is encapsulated in a class, this can be realized by considering this to be an external event, even if the source is within the data itself. Thus,

it can be realized in the same way as with the Event-Listener pattern of section 11.2: A listener class is used to notify interested parties.

This exhibits a particular feature of patterns: Often, there is a very similar solution for two, at first sight, very different situations. One is a trigger from outside the program, created by the user of the program. The other is something entirely internal to the program, as the data is part of it. The solution for both is still the same.

11.4 Streams

The console and the file input and output structure of C++, as discussed in sections 3.2 and 3.10, realize a pattern known as streams.

The idea behind streams is that often data needs to be moved to somewhere or is coming from somewhere. What this somewhere is does not matter - that is a specialization. The stream pattern deals with this situation in the following way: It defines that a stream has certain properties - it is available to take or provide data. It has the feature to accept data of certain kinds. It can be activated or deactivated. It has information whether additional data is available. And so on. If necessary, it is also possible to realize streams just for input and just for output. The most important element is that it has the possibility to put a chunk of data into the stream or take it out of the stream, and to be connected to a stream.

All of these features can be realized as members of a class. Special types of streams are then realized by the descendants of this, usually abstract, class. In C++, cout and cint as well as fstream are all descendants of such an abstract stream class, and represent the screen, the keyboard, and files, respectively. Note that they are specialization not only in the type of device. The class of which cout is an instance has, abstractly, an input stream which is always closed, and can never be opened. The class fstream, on the other hand, provides a two-way stream.

A particularly interesting possibility is to combine the stream pattern with the event-listener pattern of section 11.2, as events can be considered to become a stream. Thus a class providing events to listeners can have an internal stream of events, which are then distributed to the listeners.

11.5 Serializable

It is often necessary that some or all of the data of a class is kept safe, even if an instance is destroyed. Think of a class which represents the contact data of someone. It should be

kept after the program finishes.

This situation is solved by the serializable pattern. The idea is that the data of the instance is encoded into a series of bytes, or other primitive types, in a sense 'flattened'. A member function would take care of this. Another member function would then be able to reconstruct the state of a class from this series of bytes. Thus the class becomes serializable, i. e. a list of bytes, which contains all the information.

This pattern can be easily combined with a stream of section 11.4 to avoid making the serializing process to dependent of how it is realized. In this case, the member function to (de)serialize the object just receives a stream class, and by use of polymorphism serialization can occur to any medium without the class to actually know where she is serialized to - a file, the screen, a server, or whatever.

This shows on the one hand the power of polymorphism. On the other hand, if a class implements the serializable pattern, it becomes very simple to save the state of a program to, say, disk. If being serializable is a feature of the ancestor class of all relevant objects, it is just needed to go through all of them one by one and serialize them.

Deserializing is not entirely trivial, if the program is dynamic. After all, it is needed to know what kind of class comes next from the stream. For this purpose, a factory pattern helps - a class checks the stream which class comes next. This is encoded during the serialization in the stream, usually at the beginning. It then creates an instance of the class, and deserializes it from the stream. This is repeated until the stream provides no more objects. Of course, for this the factory class has to have knowledge of all possible classes in the stream.

11.6 Data integrity

Streams are not always reliable. The same is true for event sources. No matter the origin of this, data integrity is always the relevant question.

At the simplest level, this will be just the necessity to note if the stream read is identical to the original stream. The simplest possibility is to use a hash function, like discussed in section 8.7. When data is written to the stream, e. g. when serializing an object, a hash is calculated of it. In the simplest case, this can be the cross-sum or the number of bits being one, the later known as a CRC code. This value is then also written in the stream. When reading the data from the stream, the hash is again calculated, and compared to the one in the stream. If it does not match, something went wrong. Of course, since the hash is always a reduction of information, there are errors, which cannot be detected by it. But there exist no perfect system. This concept can be enhanced in two ways.

One is that the hash allows to correct for at least some possible errors. So-called error-correcting check sums are able to correct an error of a single bit, and detect errors in two bits. This can be further enhanced, but always at the expanse of the amount of data to stream. After all, this is always redundant data in the sense of the final purpose, and only serves to detect the errors.

The other is that the hash acts as an integrity check. If someone wants to change the data without the recipient noticing a hash like a crosssum is easy to trick. It just needs to recalculate it, and the data appears still to be intact. If the hash is not known to the recipient via a different channel, s/he is not able to notice that a change has occurred. To avoid this requires either that nobody than the one reading and writing to the stream is able to correctly calculate the hash, either because no one else knows how to do this[2] or is able to do this. The latter can be realized, e. g., by public-key cryptography to be discussed in section 11.7.

11.7 Public-key cryptography

To solve the problem of being trustworthy is very hard. In fact, no perfect solution is known.

One good solution is so-called public-key cryptography. The basic idea is that to make something to be identifiably from a particular person requires two elements. One is an element known to belonging to this person, the so-called public key. The person also has a private key only s/he knows. If a message is sent by the person, s/he uses her/is private key to sign it, e. g. by calculating a hash of it. It requires the message, the hash, and the public key to check that the hash of the message is valid. Thus, nobody else can create the correct hash, as nobody else has access to the private key. Of course, this requires that the way how everything is mixed is so expensive that not somebody can solve the problem just by trying every possible values for the private key[3]. This can also be used to ensure the data integrity of section 11.6 against manipulation.

Conversely, this can also be used to send a message to a person, which only the person can read. The message to the person is encrypted using her/is public key such that it is only (feasible) possible to decrypt it with her/is private key.

The decisive problem, aside from the actual algorithm to perform the en/decryption

[2]So-called security-by-obscurity. Usually not the best choice, as experience tells that social engineering sooner or later removes the obscurity, or reverse engineering does.

[3]This is actually the weak point of most ways of mixing it - it is usually not possible to disprove that there is some yet unknown fast way to solve the problem in much shorter time. It is also often calculational quite expensive to make the keys bigger to make brute force less likely to succeed.

is how to be sure that a public key actually belongs to the person in question. The best possibility is to get it in person, but this is not always possible. How to build a so-called web-of-trust without personal meetings is a central, and not satisfactorily solved, problem.

Chapter 12

Programming by contract

Probably the most important question of programming is how to write a 'correct' program, i. e. a program what does what it is supposed to do.

However, often it turns out that a program is correct, but the question of what it is supposed to do has not been correctly answered, or even posed. This is an issue of design, and actually not only applies to programming but to engineering in a very broad sense. It is therefore too abstract for here. Thus, in the following it is assumed that what the program is supposed to do is perfectly known, and the only question is how to ensure that the implementation of the program is behaving in this way.

12.1 Formal proofs and the science of programming

While most of this lecture concentrates on actual programming, there is far more to it. Since programs are structured, it is possible to map them on mathematics, even if some amount of randomness is included. As a consequence, the whole apparatus of mathematics is available to treat programs and algorithms.

Especially, it is possible to mathematical proof if some piece of code, given input, is formally correct, i. e. has a well-defined output. This is especially important if the piece of software is crucial, if, e. g., human lives are at stake. Think about the software involved in controlling a nuclear power plant. Such software must be formally verified and proven to be correct.

The whole formalism turns programming into a hard science. Besides correctness also issues like performance or usage of resources can be treated in a mathematical well-defined way. While this will not be subject of this lecture, it is important to know about this possibility. Especially, if there is a persistent bug, this may be the only way to find it.

12.2 Contract

To achieve this goal, a good approach is programming by contract. The idea is that there is a well-defined set of conditions under which a program provides a certain outcome. This is essentially a contract.

The important insight is that a contract requires two parties: Somebody who gives an order, and somebody who performs accordingly. As in a real contract, both sides have obligations. The one giving the order has the obligation to provide input in a particular way. The one performing the order is required to guarantee a certain output, given the input.

There are two particularities involved. One is that input is not necessarily exactly defined. E. g., if considering performing an addition as a contract, the input can be, say any real number. It is part of the contract that these are real numbers. Including also complex numbers would be a different contract. The other party then guarantees an output in a certain range, e. g. again the real numbers. This is different from ordinary contracts, which have often a one-to-one character. Rather, it is more like a service agreement.

The second particularity is that there is a particular purpose. In the example above, this is that an addition should be performed. This is an additional information. So far, the contract was just to have a mapping of two real to one real numbers. Thus, purpose needs to be added.

12.3 Preconditions and postconditions

To ensure this contract leads to the idea of preconditions and postconditions. The idea is that a part of code, usually a function, requires certain conditions, i. e. preconditions, and then guarantees that something is achieved, i. e. postconditions.

In the example of addition above, part of this can in C(++) already be implemented using the strict typecasting. If the return value and the parameters are already real numbers, float or double, this is ensured, and thus implements these precondition and postcondition. An additional postcondition can then be formulated as a mathematical expression, that the output is the sum of the two inputs[1]. This last part can often involve statements which are almost as complicated as the function itself, but should still not be done sloppily. A mathematical formulation is needed to ascertain a precise statement.

[1]There is, of course, the subtlety that this is not exactly true, as even long double variables have a finite range, and thus the strict mathematical statement is only valid up to rounding errors. Including this is an important, though often implicit, part of the conditions.

While the previous example of addition is almost trivial, there are much more complicated cases, especially if user input is involved. Consider the case that an input is a user-provided email address, and the output is that an email to the user has been send. The preconditions now decide how the function needs to be written. Is it ascertained that the input is a well-formed string containing a valid email address? Then the function does not need to check it. If it is only guaranteed that a well-formed string is provided, it is necessary for the function to check that it is valid email address, and have a postcondition which specifies the behavior of the function for the case that it is not. If it is not even guaranteed that the string is well-formed[2], even this has to be checked by the function. Thus, the preconditions will decide a lot about what the function has to do.

Since preconditions and postconditions are so far just statements, there is a danger that they can be ignored. To avoid this, libraries exist which force the test of preconditions and/or postconditions at run-time, throwing exceptions in case of violations. Such constructs are often created by heavy use of the preprocessor of section 7.3. Also, there exists special languages, e. g. Eiffel, which include preconditions and postconditions as part of the language, making it even mandatory to state the absence of conditions if none are needed.

Preconditions and postconditions are a valuable tool to enforce contracts. Of course, they cost computing time. Depending on the purpose, convenient or critical, it may be useful to enforce conditions only during testing and programming or also in the final version of a program.

12.4 Invariants

Besides conditions on the execution of a part of a program, there are also so-called invariants. These are conditions, which are guaranteed to hold at every stage of a program.

One example of such an invariant is that the static variable rectcount of the class rect in chapter 10 on classes is always non-negative. Such an invariant could in this case be guaranteed by the strong typing of C(++) declaring it to be an unsigned int. Another example could be that the area of every instance is always non-negative. The later would require to ensure that the product of the members x and y is non-negative. It should be noted that this invariant does not imply that x and y are separately non-negative. This would be two more invariants, which imply the area invariant. Such invariants, which are defined based on a class, are also called class invariants. There could also be global

[2]It could contain, e. g., wrong information about its length, a trick quite often used in buffer-overflow attacks.

invariants, which pertain to the whole program.

Again, ensuring the invariants beyond stating them requires code. In the example of the length of the object, this is achieved by the getters and setters of section 10.8 and by making the members private. For rectcount this is done by its type.

As for preconditions and postconditions, some libraries, or even languages, support their enforcement. However, ensuring them is even more expensive, as whenever something changes, which is involved with a given invariant, it has to be checked. E. g. for the rect class the area would have to be calculated whenever x or y are changed, if only the non-negative area class invariant would have been formulated.

12.5 Formal proofs

If postconditions and class invariants can be formulated in mathematical terms, it is actually possible to provide formal proofs of whether they could be guaranteed by the post condition.

The basic idea is to start at a given postcondition, and then follow the code backwards. Every statement, which affects any element of the postcondition or invariants[3], is then used to reformulate the postcondition. If the code is formally correct, then the postcondition should have turned into the precondition in this way. At the same time, at every step the (class) invariants need to be satisfied. This constitutes a formal proof of the correctness of the code[4].

Ideally, every code should have explicit preconditions, postconditions, and invariants, and they should imply the satisfaction of the contract. In practice, this is not always possible, or, in comparison to the consequences of a failure of the code, not always justified. It remains a decision of the project manager to identify to which extent code should be tested formally. In the end, this decision has also ethical aspects, leaving the scope of this lecture, but should be aware for any programmer. This reaches from the fact that literally lives may depend on the code, e. g. control software for an airplane, down to questions whether it is acceptable that the customer's code crashes every once in a while.

[3]Other statements should, strictly speaking, not exist, because either they have no effect or the post-condition and invariants have not been formulated strongly enough, as not all actions of the code have been taken into account.

[4]Provided no error is made when performing the proof. There are also tools to automatically proof code, at least to some extent.

Chapter 13

Parallel programming

So far, code was executed in a serial way, i. e. one statement after another. In practice, this is often inefficient. In science, the reason for the inefficiency is often that a code would run far too long. In commercial applications, this may be a problem as this would block the computer to the user while something is done. Also, it is technically easier to build slower but many cores, rather than a single very fast core.

All of these issues lead to the development of parallel programs, i. e. programs in which multiple statements are executed in parallel. Thereby, they can utilize multiple cores, or even computers, so-called clusters. Also, one part of the program can do some calculation, while another part interacts with the user. Finally, it is possible to user many slower cores to get the same effect as a single fast core.

How this parallelization is actually done is something which is quite involved. Thus, normally, this is hidden from the programmer. Rather, the programmer works on an abstract level, saying that n things should be done in parallel, and an underlying library, or the operating system, takes care of doing so. The one notable exception is if performance it the most important concept. Here, the actually distribution over the available resources is decisive, and the programmer/scientists, has to take care of this. This is, however, somewhat advanced, and beyond the scope of this lecture.

It should be noted that parallelization is not necessarily the best solution. The time needed to do a program in n parallel parts behaves, roughly, as

$$t = t_0 + \frac{a}{n} + bn^2,$$

what is called Amdahl's law. There is always a part which takes the same time no matter how much the code is parallelized. E. g. initialization or opening files. Then there is a part which is determined by calculation, and will ideally by n times quicker if the code is distributed over n cores. But then the code needs usually to combine somehow the results

of the different cores, requiring communication. This usually scales like n^2. Thus, there is a sweet spot of n where the time is smallest. If more cores are added, the performance degrades again. Where this spot is is determined by the coefficients a and b, which depend on the problem at hand. If b is small enough, the performance of a program will increase proportional to n. If this the case, it is said that the performance scales.

Parallelizing programs is highly non-trivial, and substantially more complicated to do than serial programs, especially if b should be made small. Still, there are some common elements to all parallel programs.