# Programming in C++

Training course number : 1722022, 9 – 13 May 2022

Dr. Sandipan Mohanty,
Jülich Supercomputing Centre,
Forschungszentrum Jülich, Germany

May 12, 2023

# Contents

# Chapter 1

# Course introduction

## 1.1 Introduction

C++ is a widely used multi-paradigm programming language, with a very large developer community, a very rich collection of available libraries, and currently, a rather lively update cycle. Being a multi-paradigm language, it allows the programmer to adapt the code to fit the problem at hand, rather than forcing adherence to any specific programming paradigm like object oriented programming or functional programming. The language has grown to be extremely feature rich. One can write perfectly memory managed, elegant, brief, fully amateur-readable programs to perform complex tasks. At the other extreme, an expert programmer might embed assembly language code directly in a C++ source file, if they consider that to be advantageous. One can write highly maintainable elegant code as well as somewhat less readable but faster code using the same programming language. In C++, these two goals are often not mutually exclusive.

I have used C++ for over 25 years for a variety of scientific projects ranging from solid state physics, astrophysics, high energy physics and biological physics. I have overseen a much wider range of scientific and engineering applications of C++. To me, the appeal of this language has always been the ability to create rather elegant abstractions while not sacrificing execution speed. It is a powerful tool, and requires a bit more effort to master than other popular programming languages. To some, it is frustrating that C++ code looks very complicated and the number of concepts you need to know until you are productive is somewhat higher than in "easier" languages. The complexity of appearance quickly diminishes in importance when one grows familiar with the syntax. You can write easily understandable and maintainable code to solve your problems within months of starting to learn the language. As you learn more, you may later revisit your earlier creations and realize that some of those problems can be solved differently for greater clarity or better performance. With C++, it is no longer a story of “learning the language” and then using it exactly as you first learned it, for 25 years. That approach (kind of) works, but it is regressive and hinders further innovation. Many creative people are constantly trying to bring in new ideas to the language. Even the most experienced C++ programmers, continue to learn fundamentally new techniques as long as they like. It is therefore unproductive to wait until you have learned “everything” before you try to use C++. Sure, the code you write now will not be as good as the code you will write a year from now, but that future, in which you are writing high quality code, is less likely if you don’t experiment with what you do know. A good attitude to cultivate when learning a new trick is to ask yourself: “what can I do with this?”

Code written by experienced C++ programmers often looks complicated because the problems we solve are complex. The language is very good at expressing and organizing complex ideas, and we take advantage of those features which allow us to solve complex computing problems to run as fast as possible on modern computing hardware. Simple ideas do translate to simple code in C++. But this programming language has been developed by a community through a peer-review process to solve real world problems, which, unfortunately entails a certain complexity. How often do student pilots feel productive after two flying lessons? C++ is complex. If the few extra key-stokes required for a hello world program compared to another language deters you, this is not the language for you. But I trust that each of you, in your respective scientific or engineering fields, have worked with far more complex ideas than what you will need in this course. If you take the time to learn C++ well, and practice, you can reach targets much harder to reach with simpler tools.

### 1.1.1 C++ standards and this course

The rules of the C++ language are decided by the "C++ standards committee", and codified in an ISO standard. Ideas for improving the language are proposed in papers. They are reviewed by the community in a peer review like process. Merits and demerits of proposals are discussed in standards committee meetings. The committee then votes on whether a proposal should be accepted as a language or library modification. Proposals which are not accepted may sometimes be re-submitted after the objections raised during the review process have been addressed. There is no “benevolent dictator” deciding what is good for everyone, but rather, just a community generating, vetting and adopting new ideas over time. The first proper C++ standard was C++98, published in 1998. This was followed by a minor revision in C++03. The language went through drastic and far reaching changes in the next revision in 2011 with the arrival of C++11. Since then, a new standard has been finalized every three years with the arrival of C++14 and C++17 and C++20. The C++20 standard was approved in September 2020. The next official standard is expected to be C++23, which is now near the end of its review process.

For this course, we will set up our tools to use C++20. Even though it will soon be 3 years since C++20 was approved, it was such a large change in the language that the compiler vendors have not yet been able to implement all the new language and library features. The previous standard, C++17, is very well supported by the compilers, and will be perfectly adequate to solve almost any programming problems you will face. Why, then, do we not stay with C++17? It is because there is little downside to setting up the tools to use C++20. This standard does not throw out everything that came before and invent a new language. Like most C++ standards, C++20 is largely backwards compatible. New features were added to the language and the standard library, but few old features were discarded. In fact, most code written in C++98 will compile and run just fine when using tools set up for C++20. This means, since C++17 was already an elegant, capable and modern language to solve real world programming challenges, partially implemented C++20 is no less. Knowing about important shifts about to happen in the coding best-practices will help you adjust and take advantage of those changes as soon as they become available. Most of the code you learn in this course will also be valid C++17 code, a little less of it will be valid C++14 and so on. If, for some reason, you have to write code in the confines of an older standard, you can do that afterwards, when you are familiar with the core concepts of the language. There are many instances where a newer C++ standard simplifies the written code. Code becomes more readable and easier to check for correctness. Although the same programming problems could of course be solved in older standards, the solutions would often be more verbose, and error prone. I, therefore, see no reason to avoid already available new features in an introductory course.

Besides, like C++11, C++20 is a relatively large structural change of the language. It will fundamentally change how C++ is written in the next 10 – 15 years. Since you are learning C++ now, it would make most sense to start with this standard, as you don’t have to unlearn previously learned bad habits. Our great constraint is that we don’t yet have a complete implementation in any compiler. Fortunately, this is not much of a limitation for an introductory course, since the language is vast enough to present you with new ideas for months. By the time you internalise the core concepts of C++, the implementations of C++20 will be more mature. Sometimes we will have to switch compilers between `g++` and `clang++` because neither covers all of the most exciting new additions.

## 1.2 How to try out the examples in this course

You learn to code by doing it. This course will therefore be full of (not always) short code examples and some exercises for you to solve. Unless you use a modern rolling release Linux distribution (Arch Linux, OpenSuSE Tumbleweed ...) as your operating system, it is unlikely that your computer has what it needs to work with C++20. In section1.3, you will learn how to set up the most important tools to work with the newest C++ standard. That, however, can be a considerable amount of work, depending on the current state of your infrastructure. Do it at your own time, and the effort will pay for itself. But at least to get started, you can try the examples on one of several excellent online compilers available nowadays. I recommend (Obs: these links are clickable!) Wandbox and Coliru. These sites allow you to copy and paste your code into text boxes (with syntax highlighting!) and compile and run your code. The two sites have slightly different ways of specifying the compiler options, which you should figure out by navigating to the sites. Look for ways to change an option looking like `--std=c++17` to `--std=c++2a`. Of course, if you have a quality compiler of your own, you can do everything locally on your own machine. Why not try it out straight away? Take the code below and compile and run on one of these webpages, just to find out how that works!

```cpp
// examples/hello.cc
#include <iostream>
auto main() -> int
{
    std::cout << "Hello world!\n";
}
```

Since copy and paste from PDF readers often creates undesirable artifacts, each of the examples you see in the text will have a comment at the top indicating the name of the source file you should also receive with these lectures. If copy and paste from this document does not look right, open the source files in your preferred editor and copy-and-paste from there instead.

## 1.3 Setting up your own computer for the latest C++ standard

This section describes the steps you need to take to prepare your computer to do the examples locally, without a web based compiler service. As mentioned before, you can do the examples for this course using the online compilers, but in the long run, you might want to set up your own system properly. Choose your own time when you want to go through these installation steps. You can also jump directly to Section 1.4 to dive directly into C++ programming.

### 1.3.1 Compilers

So, does your computer understand C++20? Answer: It doesn't. C++ programs need to be "compiled" to machine code before they can be executed by a computer. The translation process which converts human readable C++ code to a series of instructions for the processor is called compilation, and the tool that does the translation is called a compiler. The compiler is just another program on your computer, nowadays often written in C++. In order to develop in C++, you need a compiler. Whichever operating system you use, there are many excellent compilers for C++ available for you to choose. The first step is to install an acceptable compiler on your own machine. The usual approach is to "ask the administrator", or "use a package manager" to do everything automatically. Do that. But usually the compilers you get in this way are outdated. No compiler released in 2015 can be reasonably expected to understand C++20 syntax, which the C++ community finalized in September 2020! We need compilers released after September 2020. Why bother with installing the older compiler then? Well, we need the old compiler to build a new one. Below, I will show you how.

How much does it cost to get a compiler? Commercial compilers can be extremely expensive. But we don't need those for our purposes. There are two high quality open source compiler suites, GCC (https://gcc.gnu.org) and Clang (https://llvm.org), that you can download and install yourself, without any administrative privileges (This is certainly true for any Linux distributions. I don't know how easy it is for a non-privileged user to install software for their own use, on Windows or Mac OS. Let me know what you find out!) If you have to build the new compiler locally, your computer will scream a bit for a few hours, but apart from the electricity bill, corresponding to about 0.1 kWh, you don't have to pay anything to get two of the best available compilers.

In the following, I outline instructions for Linux based operating systems, to be executed on the command line as a non-privileged user. If you are a Mac OS user, you will recognize the steps, but some commands will not work exactly as I write them. Hopefully you will be able to figure out what works. If you use Windows 10 or 11, you have access to a Windows subsystem for Linux (WSL), where you should be able to run most of these Linux style instructions. If you can not make this work with WSL or Mac OS as it is, you can always install Linux in a virtual machine running inside your primary operating system. Any Linux distribution released in the last year or two should work. So, here is my step by step guide to install a new compiler, as your preparation to learn C++ in 2023.

1. In order to build a cutting-edge compiler, your computer needs to have a (possibly older) compiler on it. Type `which g++` or `which clang++` in a terminal window to see if you have any version of those compilers. Check the version by typing `g++ --version` or `clang++ --version` . If you get 10.0 or a higher number you are off to a great start! Unless you are using Mac OS, in which case, it is more complicated. On Mac OS, what you have in the name of GCC is usually just Clang pretending to be GCC, and that Clang is really “Apple Clang”. Version numbers for Apple

Clang are usually two major versions ahead of Clang, (which may cause the misleading impression that it is more recent than the open source version), although Apple Clang supports much fewer of the newer language features. Compare the columns for Clang and Apple Clang for the supported C++20 features at (clickable link →) en.cppreference.com. To learn and experiment with C++ in 2022, it is far better to simply install the open source Clang or GCC compiler.

If the above commands indicate that you don't have any compiler for C++ installed, use whatever system tools you have at your disposal to install the best compiler it can. It will likely still be too old, but it might give us a better starting point. Warning: do not mess with the system compiler, i.e., the one installed in the path /usr/bin/g++. A lot of your applications are built against the libraries of a specific system compiler version. Changing the system compiler or its libraries may make other programs on your system fail, because they often depend on a specific compiler version. The system compiler needs to exist in order to give us a start. If you use the official tools of your operating system to install a new compiler, the new compiler is usually installed with a different name, e.g., /usr/bin/g++-10 etc., which is safer. My recommendation is to make sure you have a system compiler installed, install it if it is missing, but otherwise leave it alone. Even if it is two years old, just leave it alone. Use the system compiler to build the latest available open source compiler (instructions below), and then use the newer compiler for this course.

2. Install essential build tools like "CMake" and"GNU Make". For CMake, it is better to have as recent a version as you can. This will be described later, but for starters, get the best version your package manager can give you.

3. Apart from a pre-existing compiler, you need packages called "flex" and "bison" for compiling gcc. Install these prerequisites.

Beyond this point, our path splits depending on which compiler you want to install.

#### 1.3.1.1 LLVM/Clang binary downloads

At present, on most systems, llvm/clang often provides a quicker way to a cutting-edge compiler. On Linux (and Windows 10), clang is not the system compiler, so that one can update /usr/bin/clang++ to the latest available version of llvm/clang, and not break any essential functionality. Try searching for llvm or clang in your package manager. If clang-10 or newer is offered as a choice, install it. Make sure you also install clang's own implementation of C++ standard library, libc++, along with clang. Some distributions are very literal about packages, so that if you say "install clang", they just install clang, without the accompanying standard library. Without a matching standard library, you lose about half of C++. So, install llvm/clang, and at the very least, also libc++.

If your package manager does not know about clang, you can download it from the LLVM website. Follow the links to the download page for sources and pre-built binaries. For the latest version available in May 2023, version 16.0.0, a direct link to this downloads page is here. Find the closest match to your OS, e.g., Windows 64 bit, or MacOS, or Ubuntu 22.04, ... Closest match usually works. For instance, if you recently updated to Ubuntu 23.04, you can take the Ubuntu 22.04 version and give it a go. My OS is (RebornOS, based on Arch Linux) is not in the list. Previously, as a user of OpenSuSE 15.3, I downloaded and used the version for SuSE Linux Enterprise Server 12.4 without any problems. If none of these listed options seems promising, jump to section 1.3.1.2.

The downloaded package is an archive containing a plethora of tools from the LLVM suite. The Windows package seems to be an installer (if anyone of you tried to install it, let me know if it works). For Linux and Mac OS, install it as follows:

```
tar -xvJf clang+llvm-16.0.0-x86_64-linux-gnu-ubuntu-22.04.tar.xz
```

Replace the filename above with your downloaded version. When done, move the resulting folder to some convenient location. My suggestion: create a "local" sub directory under your home and organize it so that you can use different versions of various software packages. Like so:

```
mkdir -p ~/local/llvm
mv clang+llvm-16.0.0-x86_64-linux-gnu-ubuntu-22.04 ~/local/llvm/16.0.0
```

### Post installation steps

Now, you should modify your environment variables so that this new compiler can be found from any directory. In Linux, while using the BASH shell, you would do the following.

```
export PATH=$HOME/local/llvm/16.0.0/bin:$PATH
export LD_LIBRARY_PATH=$HOME/local/llvm/16.0.0/lib:$LD_LIBRARY_PATH
```

Now, change directory to any other place, and type "clang++ -v" and you should see Clang version 16.0.0 in the answer. Try to compile the hello.cc program above from the examples directory you received by typing:

```
clang++ -std=c++20 hello.cc -o hello1
```

If everything worked, you should see a new executable file `hello1` in the directory where you executed the compiler command. Irrespective of whether or not it worked, try the following:

```
clang++ -std=c++20 -stdlib=libc++ hello.cc -o hello2
```

Run the following two commands to check that `hello2` is indeed linked against the 16.0 version of Clang's own standard library `libc++`.

```
ldd hello2
```

Among the libraries listed here, you should see an entry like "/something/something/local/llvm/16.0.0/lib/libc++". If this is the case, your installation is complete and can be used. Options we used above will be needed frequently. A nice convenience is to create an alias for the compiler including some desirable options, like this:

```
alias Clang++='clang++ -std=c++20 -stdlib=libc++ -O2 -pedantic -Wall -g'
```

Since this is a C++ course, we stay away from non-standard C++ extensions by using the "pedantic" option. The "-Wall" means "warn all", although it now means something like "warn about a lot of things", but certainly not everything. Clang has "-Weverything" for that! We also enable a fair amount of optimization, choose our desired C++ standard and standard library. We also enabled debug symbols with the '-g'. When this is done, we can call the compiler and run the resulting executable like this:

```
Clang++ hello.cc -o hello
./hello
```

The alias command and the two `export` commands to set the `PATH` and `LD_LIBRARY_PATH` environment variables could be added to the `.bashrc` or a similar shell initialisation file, depending on the SHELL you use, so that those steps happen automatically every time you open a new terminal.

#### 1.3.1.2 LLVM/Clang compile from source

In the download page for sources, binaries etc., for LLVM, scroll down, find and download the source package `llvm-project-16.0.0.src.tar.xz`. You can now unpack, build the new compiler, and install it in the same directory structure as described in the previous section using the following series of commands:

```
tar -xvJf llvm-project-16.0.0.src.tar.xz
cd llvm-project-16.0.0.src
mkdir build
cd build

CC=gcc CXX=g++ cmake -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=~/local/llvm/16.0.0 \
  -DLLVM_ENABLE_PROJECTS="clang;lld;clang-tools-extra"\
  -DLLVM_ENABLE_RUNTIMES="libcxxabi;pstl;libcxx" \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DCLANG_DEFAULT_CXX_STDLIB=libc++ -DCLANG_DEFAULT_LINKER=lld \
  -DLLVM_INSTALL_MODULEMAPS=ON \
  -DLLVM_INSTALL_UTILS=ON -DPSTL_PARALLEL_BACKEND=tbb \
  -DLIBCXX_ENABLE_PARALLEL_ALGORITHMS=ON \
  -DLIBCXX_ENABLE_INCOMPLETE_FEATURES=ON \
  -DGCC_INSTALL_PREFIX=$(dirname $(which gcc))/.. \
  ../llvm
cmake --build . --target all -- -j 4
cmake --build . --target install
```

The second last line, because of the `-j 4` option, asks `cmake` to compile using 4 processes. Adjust it depending on the number of processor cores you have. The build step should take quite a while, and it will make your computer sweat a bit. Depending on what libraries are installed on your system, you may end up with some errors regarding missing libraries or include files. While installing clang 16 on different system, I had to resolve issues regarding missing `valgrind`, missing `libxml2` and `mpfr`. Unlike GCC, Clang does not offer a way to build such dependencies in the process of building Clang. You will have to install those libraries using system tools, or by simply downloading them and going through the `configure`, `make` `make install` step for each of them. For the 3 libraries mentioned above, the process took a few seconds for each.

In the above, I also assume that you have the open source Threading Building Blocks (TBB) library installed in such a way that `cmake` can find it. The current implementation of the parallel algorithms of C++17 in GCC and Clang are based on TBB. In case you want to build without TBB, you can replace the `DPSTL_PARALLEL_BACKEND=tbb` with `DPSTL_PARALLEL_BACKEND=serial`. It is also (unfortunately) common to build `clang` entirely without the parallel algorithms. To do that, you would skip `pstl;` from `LLVM_ENABLE_PROJECTS`.

You may also have to adjust a few other options. If you are not using GCC to build Clang, adjust your `CC` and `CXX` variables. I have assumed we are building for the `x86_64` architecture. If you are compiling for an M1/M2 processor on a Mac, you can change the `LLVM_TARGETS_TO_BUILD` option to include `AArch64`. You can also complete remove that option so that it will build for every architecture it knows, although that will take a lot longer to compile.

But, if all goes well, at this point `~/local/llvm/16.0.0` should now contain a freshly compiled version of clang. Follow the steps in the previous section regarding environment variables and aliases (see section 1.3.1.1). Test the compiler as described there.

**A typical common issue and one solution:** Sometimes, when you install self-compiled clang as described above and try to use it, you get errors about missing indirectly included headers such as `__pstl_memory`. Those headers are actually present, but for some reason clang does not look for them in the location where the installation process puts them. Go to the folder you used for `CMAKE_INSTALL_PREFIX` above. Under the directory `include` you will find a few files with names starting with `__pstl`. There is also a folder `include/pstl`. Soft link these from a folder where clang does look:

```
 cd ~/local/llvm/16.0.0/include/c++/v1
 ln -s ../../__pstl_* ./
 ln -s ../../pstl ./
```

Clang should now be able to compile and use parallel algorithms using `pstl` and `tbb`.

#### 1.3.1.3 GCC

For a long time GCC used to be the only widely used open source compiler suite. Nowadays clang provides a great alternative, but the "competition" has probably helped both compilers to get better. As things stand in May 2023, there are few compelling reasons to install GCC, if you also have Clang 16 on your computer. The GCC version of the standard library does provide a few extra language and library features not yet implemented in Clang and libc++. With respect to some other features of C++20, e.g., `modules`, GCC is much less usable compared to Clang. Besides, having two different compilers when developing code sometimes helps finding obscure bugs.

For installation, GCC provides binary distributions for Windows and Mac OS. On Linux it is a rather straight forward download, build, install process I will outline here. The instructions here are written for GCC 11.2.0. Adjust the version numbers as appropriate if you wish to use the freshly released 11.3.0 or 13.1.0.

```
wget ftp://ftp.gwdg.de/pub/misc/gcc/releases/gcc-11.2.0/gcc-11.2.0.tar.xz

tar -xvJf gcc-11.2.0.tar.xz
cd gcc-11.2.0

./contrib/download_prerequisites

mkdir build
cd build

unset LIBRARY_PATH
pathrm .

../configure --prefix=~/local/gcc/11.2.0 --enable-optimized --disable-multilib \
    --enable-languages=c,c++ --enable-linux-futex

make -j 8
make install
```

Most of the steps above should be self-explanatory for anyone used to compiling and installing software from source files. A few of the steps above require some clarification. `./contrib/download_prerequisites` is a nice script included with the GCC source tree. It allows one to download a few small dependencies, such as `gmp`, `mpfr`, and `mpc`, so that they are built at the same time as GCC. You do not need to have those libraries already installed on your system. The step written above as `pathrm .` is a placeholder. What you need to do is to remove the current working directory `.` from the `PATH` environment variable. This is because, one subdirectory in the GCC source tree is unfortunately called `cp`. During the build process, GCC has to issue the copy command. If the `.` directory is in the `PATH` before the directory providing the `cp` command, the `cp` call may be resolved to `./cp`, because it exists. That leads to the puzzling error message `cp: permission denied`, because the sub-directory called `cp` is not a command! Removing `.` from the `PATH` prevents this. Along with the course material you should have received a few utilities. One of them is called `pathutils.sh`. If you source that file, it does provide you with a `pathrm` command as shown above. For the purpose of GCC compilation, you have to temporarily remove `.` from the `PATH`. It does not matter how you accomplish it.

Typically my laptop needs about 40-45 minutes to compile GCC with the above configuration. Once done, we can adjust PATH etc. in a similar way to what we did for clang (see section 1.3.1.1).

```
export PATH=$HOME/local/gcc/11.2.0/bin:$PATH
export LD_LIBRARY_PATH=$HOME/local/gcc/11.2.0/lib64:$LD_LIBRARY_PATH
alias G++='g++ -std=c++20 -O2 -g -pedantic -Wall'
```

Compiling with the new GCC can now take place like this:

```
G++ hello.cc -o hello
```

To make the PATH and alias changes "permanent", insert those lines to the end of your ".bashrc" file. An alternative is to use the environment variable management system called "Modules".

### 1.3.2 CMake

CMake is a very popular configuration tool for C++ projects. As we saw in section 1.3.1.2, it is used by LLVM/Clang. The examples and exercises folders given to you in this course are meant to be projects managed by CMake. Most modern Integrated Development Environments offer ways to import source trees with CMake based build systems. The most recent version of CMake available usually has the best support for the latest C++ standard. Almost all modern systems have some version of CMake installed. If your cmake version (find out with "cmake –version") is 3.16 or newer, you probably have a sufficiently new CMake version. If not, it is fairly simple to install. Go to the CMake downloads page. There are binary packages for Windows, Mac and Linux (all distributions). Get the appropriate version, and proceed (Linux) as follows:

```
chmod 700 cmake-3.23.1-Linux-x86_64.sh
mkdir -p ~/local/cmake/3.23.1
./cmake-3.23.1-Linux-x86_64.sh --prefix=~/local/cmake/3.23.1 --exclude-subdir \
    --skip-license
```

Repeat the `PATH` set up commands we used for `GCC` and `Clang` adjusted for your `CMake` installation.

### 1.3.3 A few very useful open source libraries

The purpose of this course is to give you a good introduction to C++ in a way that you might use it in the coming years. This is why we are learning the latest official language standard. Sadly, it has taken the compiler vendors unusually long to provide stable implementations of many C++20 features, such as `modules`, `ranges` and so on. The `modules` feature is a fundamental language level change. But features like `ranges` and `format` are updates to the standard library. These two C++20 features add a lot of value and elegance to C++ code. Neither of the two popular open source compilers, GCC or clang, implements both of these features. Fortunately, these new features are based on pre-existing open source libraries (range-v3 and fmt), and one can benefit from them even before the compilers fully implement them in the standard library. These are open source libraries and can be downloaded and used immediately. If you are already comfortable installing and using third party libraries, feel free to install it in any way you like. If not, I will propose one way to do that which will make your life easier while learning. You can later reorganize based on your needs and preferences.

**range-v3:**

The Range-v3 library brings some nifty conveniences to a C++ project. The C++20 `ranges` feature was proposed based on this library. Although the library has been developed further, `range-v3` maintains a portion that provides exactly the parts which were standardized in C++20 in an appropriately named namespace : `ranges::cpp20`. Here is an example of its usage:

```cpp
// ranges example
std::vector<std::string> L{ "Magpies", "are", "birds", "of", "the", "Corvidae", "family" };

namespace sr = ranges::cpp20; // Assuming range-v3 headers are included
namespace sv = ranges::cpp20::views;
// In case your compiler does have the ranges feature implemented, you can
// instead use
// namespace sr = std::ranges;
// namespace sv = std::views;
// The rest of the code can remain the same

```

```cpp
std::cout << "Top 3 after alphabetical sorting...\n";
sr::sort(L);
for (auto el : L | sv::take(3) ) std::cout << el << "\n";

// No second sorting. Just look at it in the reverse order...
std::cout << "Top 3 after alphabetical sorting in reverse order ...\n";
for (auto el : L | sv::reverse | sv::take(3) ) std::cout << el << "\n";
```

Range-v3 is a header-only library. You can download it from this link. Once you download and unpack it, there is nothing to install. Just include the necessary headers and use it. Since we will be using several such libraries, we can arrange them in a way so as to minimize the required typing when using them.

```
$ wget  https://github.com/ericniebler/range-v3/archive/refs/tags/0.11.0.tar.gz
$ tar -xvzf 0.11.0.tar.gz
$ cd range-v3-0.11.0/
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=~/local/cxx2022 \
    -DRANGES_DEEP_STL_INTEGRATION=ON \
    -DCMAKE_CXX_STANDARD=20 -DRANGE_V3_TESTS=OFF -DRANGE_V3_EXAMPLES=OFF ..
$ make
$ make install
```

See the end of this subsection regarding how to compile code examples using this library.

### CTRE

Standard library of C++ got some regular expression parsing functionality in C++11. It was based on the, by then relatively mature and well known, boost regular expressions library. Since then, there is a growing perception that the interface standardised in C++11 has fallen behind the times. Using the language features of C++17 and C++20, one can write better regular expression parsers with far more elegant syntax, faster compilation as well as execution times. The Compile Time Regular Expressions library (CTRE) is one such library. This functionality provided by this library will not be imminently absorbed into the standard library. But it is a far better example of regular expression parsing in modern C++ compared to the old C++11 style `std::regex` library.

We will explore this header-only library in a later chapter. For now, you can install it as follows:

```
$ wget https://github.com/hanickadot/compile-time-regular-expressions/\
archive/refs/tags/v3.6.tar.gz
$ tar -xvzf v3.6.tar.gz
$ mv compile-time-regular-expressions-3.6 ctre_3.6
$ cd ctre_3.6/
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=~/local/cxx2023/ \
    -DCTRE_BUILD_TESTS=OFF ..
$ make && make install
```

### fmt

The fmt library provides fast and type safe formatted output facilities. The syntax is inspired by the string formatting in Python ( `str.format` ). Parts of the library were standardised in C++20, although the popular open source compilers do not have usable implementations. Unlike the other two libraries listed above, `fmt` is not a header-only library, although it provides a simple way to use as one (use compile time definition `-DFMT_HEADER_ONLY` ). Install it as follows:

```
$ wget https://github.com/fmtlib/fmt/archive/refs/tags/8.1.1.tar.gz
$ tar -xvzf 8.1.1.tar.gz
$ cd fmt-8.1.1/
$ mkdir build
$ cd build
$ CXX=g++ CC=gcc cmake -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=~/local/cxx2023/ \
    -DCMAKE_CXX_STANDARD=20 -DCMAKE_CXX_STANDARD_REQUIRED=ON \
    -DCMAKE_CXX_EXTENSIONS=OFF -DFMT_TEST=OFF ..
$ make -j 2
$ make install
```

The libraries `fmt` and `range-v3` offer important C++20 features when those features are not implemented by the compiler. There is a way to use them such that the compilation will revert to using the native (standard library version) when available but use the external library when not. This is achieved by using a redirecting header file like this stripped down version:

```cpp
// Example redirecting header for c++20 ranges
#include <version>
#ifdef __cpp_lib_ranges
  #include<ranges>
  namespace sr = std::ranges;
  namespace sv = std::views;
#elif __has_include (<range/v3/all.hpp>)
  #include<range/v3/all.hpp>
  namespace sr = ranges::cpp20;
  namespace sv = ranges::cpp20::views;
#else
  #error No suitable header for C++20 ranges was found!
#endif
```

The `version` header contains information on what is available to the compiler. In the above example, we use that information to either include compiler's own version of the `ranges` library or the version from `range-v3`. In both cases we create `namespace` aliases `sr` and `sv` so that those namespaces contain the necessary definitions. One can do something similar for the `fmt` library. Versions of these redirecting headers are available along with the code samples for this book in the folder `util`.

Copy the redirecting headers to a place you will be searching while compiling.

```
$ cp ProgrammingInC++Book/util/cxx20ranges ~/local/cxx2023/include/
$ cp ProgrammingInC++Book/util/cxx20format ~/local/cxx2023/include/
```

With a set up like this, you can compile programs using `ranges` or `format` by using something like this as one of the compiler options: `-I ~/local/cxx2023/include -DFMT_HEADER_ONLY`.

## 1.3.4 Integrated Development Environments

Many people (including yours truly) prefer typing their programs in a light weight text editor like "vim" and perform all configuration, compilation, version control etc. from the command line. The command line is always there, and this way of development work requires the least resources (RAM, network traffic etc.). While the graphical IDEs have many compelling advantages, I have found that the extra bells and whistles sometimes end up wasting a lot of my time. When they work, they can, sometimes, save time through their auto-indent features or customisable keyboard shortcuts for complex sequences of operations. In the following, I will describe how to set up a few IDEs on Linux.

### 1.3.4.1 QtCreator

You can usually install `qtcreator` using your package manager. If you do that, skip the rest of this paragraph. If the version available through the package manager is very old, you can download the open source version from Qt open source downloads page. Download the online installer script and run it. From the list of packages available in the installer, select `qtcreator`. Some dependencies will be auto selected. Probably it is best not to add additional packages at this stage. Select an installation directory under your $HOME, for instance, `~/local/Qt/5.15.2`, and let the installer finish. Set up the PATH environment variable as you learned in the previous sections on compiler installation. Start the IDE.

- Help -> About Plugins: Enable the plugins in the C++ section: Beautifier, Clang code model, ClangFormat, ClassView and CppEditor. In the Code Analyzer section, enable `ClangTools` and `Cppcheck`. In some Linux distributions (like OpenSuSE 15.1), the “Clang Code Model” plugin is simply not listed. Someone maintaining the package for OpenSuSE thinks it is a non-essential unsupported extension. Without this extension, this IDE is not usable for C++ development in 2020s. If your distribution does this, you should uninstall their version. It is useless. Download the open source version from the link above, which retains all essential functionality. Enable “Clang Code Model”.
- Tools -> Options:
  - Under "Build & Run", go to the CMake tab, add a new version of CMake and set the path for the new version of CMake you installed.
  - Under "Kits", go to the compilers tab, and add any new compilers you have installed. You need to set both the C and C++ compilers.
  - Under "Kits", in the Kits tab, create a new Kit with a new name. Choose your CMake version, Compiler versions for C and C++ compilers. Under CMake Configuration you need to set the variables `CMAKE_CXX_COMPILER` and `CMAKE_C_COMPILER` to the paths you chose for the kit. In the environment field, set the environment variables `LD_LIBRARY_PATH` the library path of your compiler, for instance,

    ```
    LD_LIBRARY_PATH=/home/you/local/gcc/11.2.0/lib64
    ```

- Code formatting:
  - Tools -> Options -> Beautifier: In the tab for clang-format, set the path to your preferred (most recent) version of clang-format.
  - Select a code formatting style. I usually prefer “Webkit” for my projects, but that is a matter of taste. Choose yours!
  - Tools -> Options -> Environment: In the Keyboard tab, scroll down to find ClangFormat, and click on it. Set a keyboard shortcut for “Format file”. I set it to `Meta+I`, which does not conflict with any other keyboard shortcuts I use. Any time I have to indent or format code, I just press `Meta+I`.
- Enable STL documentation:
  - Download the Standard library documentation for Qt creator from en.cppreference.com ( Look for Qt Help Book in that page), and unpack the archive.
  - In Qt creator, open the dialog at: Tools -> Options -> Help -> Documentation. Use the “Add” button to add the downloaded archive as a documentation source.
  - Restart Qt creator
  - Any name in your code for which documentation is available will show a little F1 symbol in the tool tip when the mouse hovers over it. Pressing F1 then shows the documentation.
- Using: as you would expect with an IDE.
  - Open a project by clicking on its CMakeLists.txt file in the File -> Open File or Project dialog, or the Open project button in the welcome screen.
  - Build with Ctrl+b. Run with Ctrl+r.
  - Comment out or uncomment a line with Ctrl+/ .
  - Standard input and command line input: Once a project is open, click on the “Projects” link on the left panel. In the “Build and Run” configuration, find your chosen kit and click on “Run”. Click the box “Run in terminal”. This helps with interactive programs which read from the standard input. There is also a text box there to input command line arguments.

#### 1.3.4.2 Visual Studio Code

Cross platform IDE from Microsoft (!), with good defaults. You can download it from its website or a freely licensed community maintained version of the same IDE from the VSCodium github page. VSCode can also be installed using the `snap` universal package manager. VSCodium supports installation through many native Linux package managers such as `yay`, `zypper`, `dnf`, `apt` as well as universal package managers `snap` and `flatpak`. Windows users can choose to install VSCodium using the Windows Package Manager (WinGet), Chocolatey or Scoop. Mac OS users can install VSCodium using `brew`.

**Setup:**

VSCode uses `json` configuration files to store user preferences. Some of the settings are stored on a per-project basis. Some are stored centrally for the user. The local per-project settings are in a hidden folder `.vscode` and can be edited with any editor. I prefer to start VSCode by browsing to the directory containing the source code and typing `code .`, which starts VSCode with the settings I chose for that project. To familiarize yourself with the IDE, first use the top menu to navigate to `Help->Introductory Videos` and `Help->Keyboard Shortcuts`. For C++ development, you should then do the following.

Find the "Extensions" button on the sidebar to the left and install/enable

- C/C++ Intellisense extension (ms-vscode.cpptools)
- "Clang-format" extension (xaver.clang-format)
- CMake extension (twxs.cmake) and CMake Tools (ms-vscode.cmake-tools)

If you now close VSCode, navigate to the `examples` directory for this chapter and start VSCode there by typing `code .`, it should ask your permission to configure Intellisense for the folder. In the bottom bar, you will find clickable buttons showing which compiler is currently in use, which build type is in use etc. You can change these options by clicking on them. You can also add other compiler versions there.

#### 1.3.4.3 CLion

This is an excellent IDE available on Linux, Mac OS and Windows, which can potentially improve your productivity. One feature I particularly liked was how it showed the names of the function parameters when you are trying to use the function. Libraries like blas often have function signatures with over 10 parameters, many of them of the same data type. It is hard to remember what is what. CLion shows you the name of the parameter you are currently typing, which is a great help. Support for the latest C++ features is implemented very well. Two big negatives: it is neither free nor open source, and it is quite resource hungry. You may be able to get a students' license for little or no cost. But I hesitate to recommend it, as it may build a dependency which might end up costing you money in the long run. I recommend you use open source, or at least, non-commercial tools. I also recommend that you learn C++ so well that you are not handicapped by the unavailability of a specific non-essential tool such as an IDE.

## 1.4 C++: the first steps

I will assume that you have some way of compiling and executing C++ code at this point. If not, you should revisit Sections 1.2 and/or 1.3. For now, let's play around with the "hello world" example and discuss some very basic C++ syntax.

### Exercise 1:

```cpp
// examples/hello.cc
#include <iostream>
auto main() -> int
{
    std::cout << "Hello world!\n";
}
```

Compile and run this code in Coliru or Wandbox or on your own computer. What happens if you changed the formatting of the program like this?

```cpp
// examples/hello.cc
#include <iostream>
auto main() -> int { std::cout << "Hello world!\n"; }
```

You will see that, changing the formatting, inserting extra white space between keywords etc. have no effect. C++ is oblivious to the white spaces in your code. Proper formatting is for the human reader, as it is easier to read consistently formatted code. But unlike, say Python, white space does not demarcate blocks or carry any other syntactic significance. The two lines before the `auto main()` above had to be written as separate lines though (more on that soon).

Now, let's break this program in different ways to understand the role of different components. Where can you delete one or more characters and still compile and run the program? What happens if you, for instance remove the (a) `;` (b) `{` (c) `}` (d) `//` (e) `->` (f) `::` (g) `"` (h) `\` (i) `()` (j) `#` (k) `<<`?

Now, in the original code, replace `auto main() -> int` with `int main()` to get another valid and older form of the `main()` function. The form in which I have written it in the boxes above represent a new function syntax introduced in 2011 with C++11. Although it has a few extra characters, it has the advantage of being consistent with the syntactic style of the post-C++11 language.

### 1.4.1 Comments on the hello world code

- The `//` sign, at the start of the program above starts a *comment*. Everything you write after that in that line is ignored by the compiler. That's why we could not write the entire text of the code in one line: it would be ignored because of the `//`. Comments in the code are for the human readers. They are not translated to machine code or executed. You can place them anywhere in the code. But remember, everything that comes after the `//` in a line will be ignored.

- If you want to comment out multiple lines, it is often more convenient to use the alternative notation `/* text text text */`. Everything between the /* and the */ is regarded as a single comment even when spread over multiple lines.

- Think of `std::cout` as a sink which prints what you throw at it on the screen, a bit like the Fortran `write (*,*)` or the Python2 `print` function

- The `std::cout` line is a "statement". All code statements in C++ end with a semi-colon, `;`

- `#include <header>` tells the preprocessor to include the contents of `header` in the current "translation unit". In common tongue, a "preprocessor" is a tool automatically called by your compiler to do some text manipulations before your code is handed over to the C++ language parser. You recognize *preprocessor directives* by the `#` sign at the beginning of the line. The `#include <header>` preprocessor directive asks the preprocessor to, essentially, preprocess the file `header` and then copy and paste the text content at the location of the `#include <header>`. In our code, the `#include <iostream>` asks the preprocessor to insert all the definitions in the header file called `iostream` into our program. `iostream` is where we have the input output facilities of C++, e.g., writing something on the screen. If you don't have it, the `std::cout << "..."` function would not be understandable to the compiler. There is an alternative form `#include "somefile"` which is nearly identical. Conventionally, one uses the `"name"` for headers in your own project and `<name>` notation for standard headers.

## Exercise 2:

Fix the following code by filling in the appropriate C++ "punctuation". If you fix all errors, the programs should compile and run in your IDE or in one of the online compiler sites.

- (a)

```cpp
// examples/hello2.cc // broken code for you to fix!
#include <iostream>
#include <numeric>

auto main() -> int
{
    auto x = 2147483645, y = 2147483641

    std::cout << "Mid-point of " << x << " and " << y << " is "
              << std::midpoint(x, y) << "\n"

    std::cout << "Naive mid-point of " << x << " and " << y << " is "
              << (x + y)/2 << "\n"
}
```

- (b)

```cpp
// examples/hello3.cc // broken code for you to fix!
#include <iostream>
#include <numeric>

auto main() <- int
{
    auto x = 1000000005, y = 2000403030;

    std::cout << "The greatest common divisor of ";
              << x << " and " << y << " is "
              << std::gcd(x, y) << "\n";

}
```

- (c)

```cpp
// examples/hello4.cc // broken code for you to fix!

#include <iostream>
#include 'numeric'
```

```cpp

pkw main()
{
    auto x = 15, y = 24;

    std::cout << "The least common multiple of "
              << x << " and " << y << " is "
              << std::lcm(x, y) << "\n";

}
```

## 1.4.2 Functions and the general structure of a program

`auto main() -> int` or `int main()` starts the definition of a "function" called `main`. Functions are reusable units of code accepting zero or more inputs, to calculate a "result" based on the inputs (e.g., take input 1.5 and calculate exp(1.5)) or to produce an observable side effect (e.g., take input "Earth" and write that on the screen). In general, `auto functionname(expected_inputs) -> output_type` or
`output_type functionname(expected_inputs)`
starts the declaration or definition of a function called `functionname` which "returns" an object of the `output_type` type. This part of the function code, where the function name, the types of expected input parameters and the return value are specified, is sometimes called its header (see below). A function may have zero or more input parameters, and in C++, when it does not need any input parameters, the input parameter parentheses are written as we have done for the `main()` function above, i.e., as empty parentheses.

Functions can be "invoked" or "called" with some input values, and they can "return" an output value. Think of the cosine function in mathematics for orientation: you have an angle 60°, you invoke the cosine function with 60° as the input and receive 0.5 as the answer, or output of the function. The output of a C++ function is called the "return value".

The " body " (see below) of a function usually contains the recipe, i.e., step by step instructions, to compute the return value from the input value, and is enclosed in a pair of braces `{` and `}` following its header. The instructions in the body of a function may contain calls to other functions to perform the calculation. As an example, consider the following function:

```cpp
1 auto tanh(double Koala) -> double
2 {
3     auto pl = std::exp(Koala);
4     auto mi = std::exp(-Koala);
5
6     return (pl - mi) / (pl + mi);
7 }
```

In the above, line 1 is the function header telling us that we are declaring a function called `tanh` which expects a double precision real number as input and returns a value of the same type. In line 2, we have a single opening brace `{` which matches with the closing brace `}` in line 7. Writing the braces enclosing the function body in their own lines is a popular style among C++ programmers, and one that I personally like. But, as hinted before, this is not a syntactic requirement. I have only written the function in a way that makes it easy to read. Line 3, 4 and 6 contain the recipe by which the return value is supposed to be calculated from the input. As you can see, we make use of the input by its name (i.e., `Koala`). This name is up to the person writing the code, and is usually chosen to be meaningful in the context (unlike in this example.[1]). In lines 3 and 4, we call another function, called `std::exp`, which is the exponential function in the standard library. The function body can contain as many lines as it needs, and it can call as many functions as it needs to get its work done.

[1] I chose Koala to avoid giving the incorrect impression that typical function parameter names like `x`, `i` etc. somehow have special hidden meanings

A function header followed by a semi-colon (without the braces containing the body), is called a function prototype. As long as a prototype is visible, it is syntactically valid to use that function. For the whole program to be compiled into an executable though, a real definition of the same function with the recipe will be required.

```cpp
auto file_exists(std::string filename);
auto main(int argc, char* argv[]) -> int
{
    if (argc > 1) {
        std::string flnm{argv[1]};
        if (file_exists(flnm)) {
            std::cout << flnm << " exists!\n";
        }
    }
}
```

The above code contains a *declaration* for the function `file_exists()`, providing only the header, but no actual recipe to do the job. The existence (and visibility) of a prototype for `file_exists()` makes it possible for the compiler to check that the code in `main()`, which uses `file_exists()`, is syntactically valid. The program is still incomplete and can not be converted into an executable. But at least, the correctness of all the places where we are trying to use the function `file_exists()` can be checked. This is why, in C and C++, quite often we make header files containing just these declarations. These header files can then be included in any other source file in our code where we intend to use those functions. The compiler can check that we are using them correctly. Separately, somewhere there is a *definition* of the function, containing the same header but also a body. The compiler would need to convert that to binary code at some point. In the final stage of building an executable out of a program like the above, the compiler invokes a separate program called the linker, which matches up the places where we used a function like `file_exists()` to the actual binary code generated by compiling the definition of that function. These definitions are usually aggregated in shared or static libraries.

Every C++ program [2] must contain one and only one function named `main`. When you run a program, the operating system (OS) calls the `main` function. Then, as usual when a function is called, the code in that function is executed line by line until the end of the function (closing brace `}` of the function body or a `return` statement) and then control is returned to the caller. In the case of `main`, when execution reaches its end, the program ends. Officially in C++, the `main()` function is expected to return an integer to its caller, usually the OS. Its return type must be `int`. If you have read code like `void main()` in some book, stop with that, as that is not standard C++. No decent compiler will accept that.

So, what is this integer value we are supposed to return to the caller of `main()`? A return value 0 means successful completion, and any other value is interpreted as "something went wrong". UNIX based operating systems make use of this. While the function header for `main` must indicate that it returns an `int`, uniquely for `main()`, an explicit `return` statement is not required! We have used this fact in our `hello_world` example code, where the main function ended without a return statement. If you don't write one, a `return 0;` statement will be assumed at the end of `main`. This special treatment is only done for `main`. For all other functions promising to return a value, the compiler will warn if you forget to clearly write what answer it is supposed to return.

[2] Technically, only those in a "hosted implementation", but we can leave that technicality out for this course

All code which translates to something happening (other than some initialisation) is contained either in `main()`, or in functions invoked directly or indirectly from `main()` To see what I mean by this, try this!

## Exercise 3:

Take this tiny program and compile it using your favourite way.

```cpp
#include <iostream>

int main()
{
    double x = 2.1;
    x = x * x * x;
    std::cout << x << "\n";
}
```

Now, try to take one or more lines from inside the body of main to a place outside the main function. (Don't create a new function to put them in!) Which of the lines inside `main()` could be placed outside and still have a syntactically valid program? (It is one of the jobs of the compiler to tell you if your code is not syntactically valid. We ask the compiler by trying to compile the code.)

You will find that the creation of the variable `x` and its initialisation at the start of `main()` can be outside `main()` (which will have implications on its life time, which we will discuss shortly). The rest of the statements performed some action, like calculating some value and changing `x` to the new value, or writing something on the screen. These things must always be in the body of a function, and may only be reached when the said function is called.

So, when you run the program, the OS calls `main()` and the computer starts executing the instructions in the body of `main()`. In the following example with pseudocode, the execution will start inside `main`, write "Hello world" and arrive at line 14. But to execute line 14, it needs `f(7)`. It therefore invokes the function `f` with input 7. This is a valid input, as the function f expects an integer input. Then it executes line 7. For line 8, it needs to calculate `g` using the just calculated value called `intermediate` as the input. So, it invokes the function `g`, and executes line 3. When it runs the `return` statement in line 3, it "returns" to the invocation point, i.e., line 8, and finishes executing it. Then, it reaches line 9 with the `return` statement for the function `f`, and returns to line 14, from where the function `f` was called. Then it comes to line 15, and after that, the implicit return statement of the `main` function.

```cpp
 1  auto g(int x) -> int
 2  {
 3      return x+1;
 4  }
 5  auto f(int x) -> int
 6  {
 7      auto intermediate = 6 * x;
 8      auto result = g(intermediate) + 4;
 9      return result;
10  }
11  auto main() -> int
12  {
13      std::cout << "Hello world!\n";
14      auto res = f(7);
15      std::cout << "result = " << res << "\n";
16  }
```

Every C++ program has this structure [3]. You have an entry point called the `main` function, and while executing the code in it, you call other functions, each of which might call any number of other functions and so on.

Now is a good time to say that the above is just a useful mental model to think about how C++ programs work. Modern compilers perform all kinds of clever optimisations which might re-order or change the instructions in your program for efficiency, if doing so will produce the same observable outcomes.

Modern processors also re-arrange the stream of instructions they receive to improve the speed at which they execute the programs. Do not take the above line-by-line walk through of the example code too literally. The compiler and the processor will execute your code such that they will produce results as if they had followed a process like what is described above. But they will often produce results faster than would be possible with a literal line-by-line execution of the code you write.

### 1.4.3 Interacting with your program when it is running

### Exercise 4:

**Standard input:** This example `examples/math_functions.cc` demonstrates some of the mathematical functions available in the C++ standard library. It's a simple program where a real number is stored in the variable `inp` and then we print different mathematical functions calculated at that value. There are two lines in the code which have been commented out. If you uncomment (remove the `//` at the beginning of ) the lines 8 and 9, the program is supposed to ask for an angle to be entered by the user.

```
1  // examples/math_functions.cc
2  #include <iostream>
3  #include <cmath>
4
5  auto main() -> int
6  {
7      double inp = 3.141592653 / 6.0;
8  //    std::cout << "Enter angle: ";
9  //    std::cin >> inp;
10     std::cout << "sqrt(" << inp << ") = " << std::sqrt(inp) << "\n";
11     std::cout << "cbrt(" << inp << ") = " << std::cbrt(inp) << "\n";
12     std::cout << "cos(" << inp << ") = " << std::cos(inp) << "\n";
13     std::cout << "sin(" << inp << ") = " << std::sin(inp) << "\n";
14     std::cout << "cosh(" << inp << ") = " << std::cosh(inp) << "\n";
15     std::cout << "sinh(" << inp << ") = " << std::sinh(inp) << "\n";
16     std::cout << "exp(" << inp << ") = " << std::exp(inp) << "\n";
17     std::cout << "erf(" << inp << ") = " << std::erf(inp) << "\n";
18     std::cout << "riemann_zeta(" << inp << ") = " << std::riemann_zeta(inp) << "\n";
19 }
```

- **Local build and run:**
  - Uncomment lines 8 and 9.
  - Compile it (Type `g++ math_functions.cc -std=c++20` )
  - Run it (Type `./a.out` in the `examples` folder)
- **You are trying it out remotely on Wandbox**
  - Alternative 1: Leave the comments as they are. If you want to change the value of `inp` , just change it in code and recompile.
  - Alternative 2: On Wandbox, there is a link "Stdin" below the code box. Clicking it opens a text input box, where you can enter what you want to feed the program

[3] Coroutines in C++20 change this a little bit, because they can suspend and resume their execution. Coroutines are not part of this introductory course

when it expects input from the user. Write some value there before pressing the "Run" button.

The reason for the awkwardness of programs with standard input on platforms like Wandbox are as follows. We don't have direct access to the standard input on the system where our code runs. The web browser first collects our code and any other parameters (compiler options, standard input etc.) and then forwards that content to a server, which then executes it, possibly in a Linux container, and sends us the outputs. There is no direct interactive session on the computer where the code actually runs.

Coming back to the code above, observe how the output is chained with successive `<<` operators: we write the text `sqrt(` and then the value of the variable `inp` and then the text `) =` and then the square root of `inp` and then an end-of-line character. [a] The `cmath` header provides a lot of useful mathematical functions. If you want to see what else is available, click here!

[a]In older textbooks, you will often find the use of `std::endl` to indicate the end-of-line. This is now discouraged. `std::endl` means "insert a newline character and then flush the output". Additionally, it was supposed to provide a platform independent end-of-line indicator, i.e., printing a UNIX style end-of-line character on UNIX like systems and a Windows style CR+LF end-of-line character on Windows. This is no longer necessary, as a literal '\n' character is translated in precisely the same way. And flushing output output stream after every line is detrimental to IO performance.

## Exercise 5:

**Command line arguments:** Another way to introduce interactivity is by using command line arguments. Functions can receive input, and since `main` is a function, it can too. In the version we have seen so far, `main` does not expect any arguments when it is called. But there is an alternative form of the main function that looks like this:
`auto main(int argc, char* argv[]) -> int`
or equivalently,
`int main(int argc, char* argv[])`. Like in the other version of `main` we have seen so far, the return type of `main` has to be an integer. But when `main` is defined as shown here, the OS calls it in a special way. It takes the full command you type, starting from the name of the program till the end of the line, and splits it into tokens. The total number of tokens, an integer, is passed as the first argument ( `argc` is "argument count") to `main` and the entire array of tokens is passed as the other argument ( `argv` is "argument vector") of `main`. Since the program name itself is also in the command line when you run it, the integer `argc` received by `main` as the first argument can never be zero, and the name of the program is always the first element of the array of arguments.

```cpp
// examples/math_functions_cmdln.cc
#include <iostream>
#include <cmath>
#include <string>

auto main(int argc, char* argv[]) -> int
{
    double inp = 3.141592653 / 6.0;
    if (argc > 1)
        inp = std::stod(argv[1]);
    std::cout << "sqrt(" << inp << ") = " << std::sqrt(inp) << "\n";
    std::cout << "cbrt(" << inp << ") = " << std::cbrt(inp) << "\n";
    std::cout << "cos(" << inp << ") = " << std::cos(inp) << "\n";
    std::cout << "sin(" << inp << ") = " << std::sin(inp) << "\n";
    std::cout << "cosh(" << inp << ") = " << std::cosh(inp) << "\n";
    std::cout << "sinh(" << inp << ") = " << std::sinh(inp) << "\n";
    std::cout << "exp(" << inp << ") = " << std::exp(inp) << "\n";
    std::cout << "erf(" << inp << ") = " << std::erf(inp) << "\n";
    std::cout << "riemann_zeta(" << inp << ") = " << std::riemann_zeta(inp) << "\n";
}
```

Now compile it locally by typing `g++ -std=c++20 math_functions_cmdln.cc -o mathfunc` to obtain an executable called `mathfunc` .

You can also compile and run it online at Coliru. Coliru shows you an editable field at the bottom of the screen with the command it intends to run. Something like this:

```
g++ -std=c++2a -O2 -Wall -pedantic main.cpp && ./a.out
```

They are chaining two commands into one: By using the `&&` token, they are telling the (most probably Linux) command line that it should compile using `g++` , and if that succeeds, it should run the resulting executable. They save the program you enter in the code box under the name `main.cpp` , which is why you see `main.cpp` in the command. And what is `a.out` ? When we don't tell the compiler any name for the executable it is supposed to generate, conventionally the compiler chooses the name `a.out` automatically. That's why Coliru runs your code by calling `a.out` by default. If you want to make it look like the instructions for the local compilation and execution above, you can modify their command line to this:

```
g++ -std=c++2a -O2 -Wall -pedantic main.cpp -o mathfunc && ./mathfunc
```

When you run the program by typing `./mathfunc` , the `main` function will be called with $argc = 1$ and $argv = [''mathfunc'']$. Note how `argc` , the argument count, is 1, when we pass nothing to `main` , because its name is always in the array of tokens. If now you run it again like this: `./mathfunc 1.5` , the `main` function will be called with $argc = 2$ and $argv = [''mathfunc'', ''1.5'']$. The argument `argv` is received as an array of character strings. Our program converts the relevant token to a real number using the function `std::stod` (string to double) and replaces the value of `inp` with the result. Try it out with a few different inputs on the command line!

# Chapter 2

# Overview of C++ fundamentals

## 2.1 Values, types, objects and variables

In the exercise programs we have seen up to this point, there were statements like this:

```
auto x = 15;
double inp = 3.141592653 / 6.0;
```

In each instance, we create new variables in a program. When we do so in C++, the compiler has to be able to determine the *type* of the variable. In the relatively newer syntax with the `auto` keyword, the type is inferred from the initial value. In the example above, `x` is of type `int` because `15` is an integer. Had we written `auto x = 15.0;` instead, the type of `x` would have been a `double`, because `15.0` is a double precision real number. When the compiler reads this code, there is no ambiguity as to what type of variable `x` is, so that it can generate instructions for the CPU to work with `x`. In the second line above, I show another perfectly valid version of variable declaration, where we are quite explicit: we want a variable of name `inp`, type double and value as given on the right hand side.

C++ is a *statically typed* language, like C, Rust, Fortran, Java and so on, and unlike Javascript, Python or Perl. This means

- before the program ever runs, during the compilation process, the compiler must be able to determine the types of all objects, i.e., inputs to functions, result of functions, variables etc. throughout the program
- the type of those objects are immutable for their entire respective lifetimes.

Since the notion of a "type" is so important in C++, let's discuss it a bit while we are laying our foundations.

The numbers $..., -2, -1, 0, 1, 2, ...$ are values of the integer type. Similarly $0.5, 6.67 \times 10^{-11}, 1.381 \times 10^{-23}$ are values of real numbers. "Monday", "Tuesday" etc. are possible values representing weekdays. Values tend to have meanings attached to them, and we are accustomed to interpreting values of different types in our day to day life, and have a working notion of what different types of values are. Humans seldom get confused about whether 729 is bigger or smaller than "Tuesday". We understand intuitively that such a comparison makes no sense as we are comparing entities of different types. This intuitive idea of categories is a good starting point for our purposes, but it needs to become a bit more well defined within the context of programming languages.

**Let's play a game:** Computers store information in binary bits, which means, to do any useful work with the values we care about, there has to be a mapping between the values and a (possibly unique) binary representation which the computer may manipulate. Imagine that you are playing a game with a very gifted friend who can do arithmetic very quickly. But you are only allowed to communicate with each other by holding out your hands with different numbers of fingers extended. You need to communicate a task, such as adding or subtracting small numbers. Your friend has to have a unique way of interpreting your request, and respond also with this kind of hand gestures. Think about what kind of rules you have to make for this game to work. How would you ask your friend these kinds of questions and interpret the answers uniquely?

| Bits | unsigned | signed(naive) | signed(2') |
|---|---|---|---|
| 000 | 0 | 0 | 0 |
| 001 | 1 | 1 | 1 |
| 010 | 2 | 2 | 2 |
| 011 | 3 | 3 | 3 |
| 100 | 4 | -0 | $0 - 2^{3-1} = -4$ |
| 101 | 5 | -1 | $1 - 2^{3-1} = -3$ |
| 110 | 6 | -2 | -2 |
| 111 | 7 | -3 | -1 |

Table 2.1: 3-bit unsigned and signed integers. The second column gives the interpretation of the bits as a plain 3-bit non-negative binary number. The third gives the interpretation as a signed 3-bit integer, in which we simply treat the left bit as (the presence of) a minus sign. The last column shows signed integers in the two's complement representation.

- $a + b = ?$, where $a$ and $b$ are any two integers between 0 and 127
- $a - b = ?$, where $a$ and $b$ can be positive or negative.

Remember, you can not make a plus with two fingers painting a plus, or minus, and you can not spell it out in a human language. Think about how to represent numbers, how to deal with negative numbers, how to communicate different tasks, and if all this is too easy, how to accommodate real numbers. Now, imagine your friend with lots and lots of hands with 64 fingers in each and you are getting close to something useful! Note: This is a thought exercise. In real life, the finger positions for specific powers of 2 might be deemed offensive.

It is instructive to further discuss integer values, as they are simple, and yet illustrate a few important concepts. Every combination of $N$ bits is a valid $N-$digit binary integer, so that the mapping of an integer to a binary representation is relatively straight forward. For practical, engineering reasons, our CPUs can not contain circuitry for an unlimited number of binary digits, which puts further constraints on the "native" binary representations. We can handle arbitrarily large numbers with hand signs in our game, just by agreeing on start/stop finger configurations and adding as many binary digits as we like. But the largest number of bits we can quickly ("natively") handle is limited by the number of fingers we have. For instance, the processor on my laptop can natively manipulate 64 bit integers. Assuming that we use these 64 bits or, 8 bytes to store integers, we have $2^{64}$ different possible values. If the values we are interested in are in the range 0 to $2^{64} - 1$, we can simply map each combination of these 64 bits to a value. But what if we also want negative numbers? There is no place for us to write a "minus sign", and we must use one of the bits to store that information. Imagine that we used the "left-most" bit as the sign bit. If it is 0, we interpret the number as positive, if it is 1 it is negative. Now, did you realize that we created two representations for 0? That's inconvenient, as now we have to check against two things before being able to answer if some integer value is 0. Another possibility is to use what is known as "two's complement" representation. In this bit representation of integers, we also use the left bit as the "sign bit". There is an $N - 1$ digit binary number left after we have used up the sign bit. Let's call that number $m$, and the original full $N$ bit number $n$. If the sign bit is 0, the number $n$ is positive, and has a value $m$. If the sign bit is 1, $n$ is negative, and has a value $n = m - 2^{N-1}$. For $N = 3$, we can enumerate all the possible signed integers in this representation in table 2.1

Observe that we have no duplicate representation of 0 or any other number in the two's complement representation, but, there are 3 positive numbers and 4 negative numbers. In general, in this representation, there are $2^{N-1}$ negative numbers $\{-2^{N-1}, -(2^{N-1} - 1), ..., -1\}$, and $2^{N-1}$ non-negative numbers, $\{0, 1, 2, ..., 2^{N-1} - 1\}$. Native representations for signed integers, other than this two's complement representation are rare, if present at all.

Observe also that even in this extremely simple case of an integer, the bits do not inherently represent any values. The value, or meaning, a sequence of bits (e.g., 101) has depends on a set of rules we construct to interpret them. This "meaning" is not something arbitrary or philosophical. Integers are expected to *behave* in a certain way. Consider the simple operation of adding two 3-bit integers. If the two numbers we are adding have binary bits 100 and 001, and are integers as we know them from mathematics, the result will be

- 101 if the bits represent unsigned integers ($5 = 4 + 1$)

- 001 if they are signed integers in our naive representation $(1 = (-0) + 1)$
- 101 again if they are two's complement signed integers $(-3 = -4 + 1)$.

Just saying that they are made of 3 bits and are integers does not specify how these bit values are to be combined sensibly to produce new values in a manner that preserves their associated meanings. We thus arrive at a few important definitions.

A value, for us, is a sequence of bits along with an associated set of rules to interpret them. A type consists of a specification of a bit representation and a concrete set of fundamental operations on one or more values, which are necessary to ensure that the set of values behaves like the abstract idea it is meant to represent. The type determines all possible values. A concrete instance of some bytes, somewhere in the memory of a computer, holding a value of a certain type, is an "object". An object with a name is a variable. A statement introducing a variable to a program is a "variable declaration". The two statements at the start of this section were variable declarations.

`int x = 3;` creates an object

- which will be referred to by the name `x` throughout its lifetime
- which has the type `int`, meaning that it is (normally) 32-bit long, and its bits are to be interpreted as signed integers in the two's complement representation, and that it is to be used with a set of rules designed to closely mimic the *behaviour* of integers in mathematics)
- holding a certain bit pattern, which when interpreted by the rules of the type (integers in the two's complement representation), translates to a value 3

What possible types could variables have? Like in many programming languages, in C++, the programmer can add new types representing important ideas in their project and then declare and use variables of that type. So, there is an infinite number of possible types. But there are a few types which are called built-in types, such as `int`, `long`, `float`, `double`, `char`, and `bool`. We have seen `int` and `double` before, which represent integers (`int` is usually the fastest kind of integers supported by the hardware) and double precision floating point numbers respectively. `long` represents large integers, often 64-bit on today's computers. `float` are 32-bit real numbers. `char` is a single character, or a one byte integer. A variable of type `bool` can only have two values, `true` or `false`, a very important type for logical operations. Values of these types and operations on them correspond directly to the capabilities on the processor. Much of the design of C++ is about allowing good programmers to create user defined types which can be as efficient as built in types, with as little overhead as possible.

In C++, the type of a variable can not change during its lifetime. It is not possible to redeclare a variable, with equal or different type, while the variable is still alive.

```cpp
auto x = 4;
auto y = x * x;
auto x = y * y * y; // Not allowed. x is already defined in this context!
std::string x = std::to_string(x);  // wrong on multiple levels!
```

Only the *value* of an existing variable may be read or updated. The type is immutable. The above duplicate definitions of `x` may look preposterous to programmers who previously worked with Fortran or C or other statically typed languages, but a python programmer might be tempted to think that `auto x = 4;` is just like `x = 4` in python, and therefore in the spirit of the python

```python
# Python code
L = [ 1, 2, 3, 4, 5]
L = np.array(L)
```

one might be tempted to write

```cpp
std::list L { 1, 2, 3, 4, 5 }; // This is fine
L = std::valarray(L);      // This is not!
```

This is ill-formed in C++, because assignment does not change the type of a variable. Even if we had written a function `to_valarray` taking a list as an argument, and used that instead of `std::valarray(L)` above, `L` would remain an `std::list<int>`, and therefore the assignment will fail. Using an `auto` at the beginning of that statement will not help because we can not create a variable of the same name as an existing and visible variable[1].

In the following code, we create two variables of the types *int* and *float*, representing 32-bit signed integers and 32-bit real numbers respectively. We then assign a simple integer 1 to both of them.

```
1   int v1 { 0 };
2   float v2 { 0. };
3   v1 = 1;
4   v2 = 1;
```

After we have assigned 1 to $v1$ and $v2$, what are the types of the two variables? They remain, as before, `int` and `float` respectively. Does it matter? "since they both just hold 1"? To us, frequently not, but to the computer, a great deal! This is because we don't see the actual bits like the computer does. What is stored in memory in the bits holding `v1` and `v2`? On my computer, $v1$ : 00000000000000000000000000000001 and $v2$ : 00111111100000000000000000000000. You see, the rules of interpretation of the bits differ a lot between the so called "floating point" numbers and integers, so that both the above bit representations actually hold the value 1, as in "the number of stars in the solar system". If the computer is going to do any meaningful operations on `v2`, and get the correct answers, it needs to use the correct set of rules to interpret the bits of `v2`. Those rules, for the floating point type will be discussed in a later section. For now, let's just note that they are very different than the rules for integers. For instance, in C++, when we write `1 / 2`, the answer is 0, because `1` and `2` are governed by the rules of integer arithmetic, and integer division of 1 by 2 yields 0[2]. When we assign the literal value 1 to `v2` (declared as a float) in line 4, the integer literal 1 is first converted into a bit pattern storing the same value but following the "floating point" rules rather than the integer rules. Then those bits are stored in `v2`. In C++, the rules state that while a variable lives, the interpretation of its bits as well as the operations done to it follow the rules defined by its type.

### 2.1.1 Representation of real numbers

Real life computing hardware is subject to those pesky laws of physics and practical engineering considerations arising from those laws of physics. We can not use an infinite number of bits inside the fundamental units of computation in our processors. Remember how we had to settle for a native bit-width for the representation of the integers? One can string together many of those native bit-width integers to create arbitrarily large numbers, but calculations using such composites will be sophisticated multi-step operations requiring comparatively more time. By comparison, any values you can express using a native representation can be used directly by the processor for single step computations.

This limits those native calculations to $2^N$ different integer values, and we can choose them to be $\{0, 1, 2, ..., 2^N - 1\}$, by using unsigned integers or $\{-2^{N-1}, -2^{N-1} + 1, ..., -1, 0, 1, ...2^{N-1} - 1\}$, by using signed integers. Typically nowadays, $N = 32$ when we choose 32-bit versions of such integers by using `int` or `unsigned int`, or $N = 64$ when using 64-bit versions of integers `long` or `unsigned long`. The number of different values is always $2^N$. As long as we are dealing with integer values smaller than what can be represented in `int`, `long` etc., our computations can be exact. To compare two integer values, the processor can safely compare their individual bits, and see if the bit sequences for the two integers are in fact identical.

Real numbers are quite different. Given any two different real numbers, $x_1$ and $x_2$, there are an infinite number of real numbers between them. There are an infinite number of real numbers between 1.00001 and 1.000011. There is simply no hope of storing them in their full glory using a finite number of bits.

[1]existing variable declared in the same scope. See the discussion of "scope" below.

[2]Note for python programmers: arithmetic with integers follows integer rules, and therefore `1 / 2` is 0. If you want real number arithmetic, use at least one real number in one side of the arithmetic operator: `1.0 / 2.0`, `1. / 2.`, `1. / 2` and `1 / 2.` will all result in a double precision floating point result 0.5.

If we were trying to represent the space of real numbers between, say, 0 and $2\pi$ as accurately as possible using 64-bits, we could divide the interval into $2^{64}$ equal parts, like bins in a simple histogram. A real number in that pre-defined interval can be represented by the integer bin index $b$, i.e., as $r = \frac{2\pi b}{2^{64}}$. All the real numbers which fall in a certain bin will have to be treated as equivalent, because there would be no way to distinguish between them in our representation. But that's probably going to be OK, as the bins will have a width of about $3.4 \times 10^{-19}$. This is about as well as we can do to resolve very close real numbers in a pre-defined interval. But this is not a practical representation for most tasks we might want to do with real numbers. What if we wanted to, say, add two of them!

If real numbers are to be used to represent quantities of things, adding or multiplying them will be necessary, and their values won't remain bounded in a pre-defined range. If, instead of $2\pi$, we stretch our pre-defined interval to a very large range, say, 0 to $10^{100}$, we will, for many cases, successfully fix the "overflow" problem. But then, we would have stretched our bin size to $5.4 \times 10^{80}$, so that we will lose all ability to distinguish between very real life numbers such as 1.0, 0.0001 and 2000000.0. What we need is a system in which we can distinguish between any two closely separated numbers, and represent a very large range of values using the same representation. You might be thinking, that's not possible, because there just isn't enough information in 64-bits. You would be right.

The solution used for real numbers by almost all computers today is an internationally agreed standard, called the IEEE 754 floating point representation. This is a *variable resolution representation*, and it is important to understand this aspect to write robust code for science and engineering projects.

What we use is based on the "scientific notation" of real numbers, as in, $6.023 \times 10^{23}$. The 6.023 part is referred to as the mantissa, 10 is the base and 23 is called the exponent. Imagine that you have exactly 10 characters to write your decimal real numbers. You can create a convention about where to store what information, but all you write down are the digits. No decimal point, no minus sign. Pause here and think how you will do that, before continuing to read how it is done.

If we denote your digits as $d_0, d_1, d_2, ..., d_9$, you may have come to the idea that you will use one place, $d_0$, to store 0 or 1 for the minus sign, a certain number of digits (say 3) as the exponent (including a possible minus sign for the exponent!) and the rest to store the mantissa. Your number $d_0d_1d_2d_3d_4d_5d_6d_7d_8d_9$ will represent the value $(-1)^{d_0} d_4.d_5d_6d_7d_8d_9 \times 10^{(-1)^{d_1} d_2d_3}$, where $d_0$ and $d_1$ only take values 0 and 1 for the two possible minus signs. There is a compromise on how many digits you want to keep for the mantissa and how great a range of values you can represent. Notice also that there is a fixed number of different possible mantissa values determined by how many digits we decide to give the mantissa. Given a fixed exponent, that's how many different values we can have. The mantissa values are always of the order 1, and the exponent scales that mantissa to a larger or smaller value. Given any exponent, we have an equal number of representable values. Between 1 and 10 in the above representation, we have $10^6$ different possible values. All numbers in this range have the exponent 0. So, the variation comes entirely down to the mantissa digits. Between 100000 and 1000000, all numbers will have an exponent 5, and therefore, we can represent, again, exactly $10^6$ different values! Between 0.001 and 0.01 we will again have exactly $10^6$ different values. As should be clear, by now, not only can we not represent every possible real number, but the space of real numbers will be divided in a non-uniform albeit systematic manner. If we count only numbers we can represent in our system, there will be more real numbers close to 0 than close to larger and larger values. Perhaps you are thinking whether you would ever notice this non-uniformity. Let's take this example: Think how you will represent the number 1 million in the above scheme. Now, add 1 to 1 million, and write the result again in that representation. You will find that they will be the same. In our crude 10 place system, the representable real numbers become sparser and sparser as we go to bigger values, and become so sparse at around 1 million that adding 1 to that number does not change it at all! Now, suppose you added 1.1 and then again 0.5 to that number, and then subtracted the 1 million. You will be left with 0, rather than 1.6.

The above little exercise with a few decimal digits is actually quite similar to the way real number arithmetic works natively in computers. Instead of decimal digits, we use a certain number of binary bits to store our real numbers in a sign-exponent-mantissa scheme, and we have a compromise between how many bits of mantissa we want to store and what total range of values we want to cover. There is however one peculiarity of the binary numbers in the scientific notation, that offers us an opportunity to steal one extra bit of mantissa. In the scientific notation with decimal numbers, the mantissa $d_4.d_5d_6d_7d_8d_9$ could have at position $d_4$, any non-zero decimal digit 1...9. If $d_4$ is 0, we would just move the decimal point right and adjust the exponent until we have something non-zero at $d_4$. If our digits are binary, there is only one possible non-zero binary "digit". It's always 1! If it is always 1, we don't need to store it in our mantissa, and we can use that one extra bit to store an extra bit of mantissa at the right. This makes our binary representation something like $s(e_se_1e_2e_3...)(m_0m_1m_2...)$ for the number $(-1)^s 1.m_0m_1m_2... \times 2^{e_se_1e_2\cdots}$.

Our attempt to squeeze in one extra bit of mantissa has cost us something very important though: the number 0 is now gone! That is the one number for which we can not shift the decimal point left or right to bring a non-zero digit before the decimal point. In our above binary representation, the number 1.0 would have the sign bit $s = 0$, all the exponent bits 0 and the stored mantissa bits also all 0. So all bits in fact would be 0. If we wanted to save 0, we would have the exact same thing! They will differ only in a place we decided not to explicitly store. We get around this by adopting a convention. our exponent bits are stored with a shift encoding. If we had 8 bits for the exponent, they can represent 256 consecutive integer values. But, we can choose where to put our "origin" in this 0 to 255 range. If we say that 127 encodes 0 in this range, 128 encodes 1 and so on, while 126 encodes $-1$ etc., we have one way of representing all our exponents. If we do that, 1 in our binary representation will have an exponent 0 represented by 01111.... The mantissa and sign bits will still be all zero, but at least, it creates an opportunity for us to solve our problem. By convention, we say 0 is represented by a number with all bits 0. 1 now is different, because the exponent 0 is now represented by the integer $2^{n_e-1} - 1$, where $n_e$ is the number of bits in the exponent.

For 32-bit floating point numbers, the computers use 1 sign bit, 8 bits for the exponent with a shift-127 encoding, and the remaining 23 bits as mantissa, with an "implicit 1" on the left of the decimal point. 64-bit, double precision floating point numbers, as represented by the type `double` in C++, are similar except that they store a sign bit, 11 bits of exponent and 52 (fraction) bits of mantissa.

### Exercise 6:

To help familiarize you with real numbers in your computer, as represented by IEEE 754 floating point numbers, you will find a program called `examples/binform.cc`. The syntax in the code, apart from the `main()` function is not of interest at the moment. Observe how we print the binary forms of various numbers, when they are of types `double`, `float` `unsigned long`, `int` etc. Put a few other numbers of your choice in the code, and observe the output, and try to understand how these numbers work.

Up to this point, we have used an important concept, *lifetime*, without explaining. By lifetime of a variable, we mean the parts of the code where the variable is available for use, e.g., for reading its value ($\equiv$ interpreting its bits according to the rules of its type) or saving a new value ($\equiv$ setting bits representing the variable in such a way that when interpreted by the rules of the type, the desired new value results). A variable is not available for use before it is declared. This means, at the point where the compiler is translating a statement using the name of a variable, the compiler must already have encountered its declaration, and therefore know about its type.

## 2.1.2 Blocks

When does a variable cease to exist? The notion of a "block" is useful for answering that. In C++, you can group a few statements into a block by putting them within a pair of braces as follows.

```
statement1;
{ // block 1 starts here
    statement2;
    statement3;
    statement4;
    { // block 2 starts here
        statement5;
        statement6;
    } // block 2 ends here
} // block 1 ends here
statement7;
```

Statements 5 and 6 constitute a "block". This block is contained inside another block, which also contains statements 2, 3 and 4 before it. Blocks can be nested to arbitrary depths to impart a tree like structure to the code. They are used to generally group statements together which belong together. Where can we create blocks? Wherever some "action" is happening, in other words, inside function bodies. Function bodies are themselves blocks containing some statements and blocks. If we need to do 3 things when a condition is satisfied, we put the three things in a block:

```
if (condition) {
    do_thing_1;
    do_thing_2;
    do_thing_3;
}
```

The block containing the three `do_thing__` statements above, bounded by the two `braces`, behaves like a compound statement for the `if` construct. Similarly, we use blocks to bundle multiple statements to be repeated in loops. Anywhere in a function body, if we need to or want to isolate or group together a number of statements, we can enclose them in a pair of braces `{` and `}` and create a block. But just like you couldn't place an orphaned executable statement outside all functions, you can not create a block outside all functions. Function bodies, in this sense, are top level blocks.

Blocks have the important property that variables which are declared within a block cease to exist when the code in the block finishes executing. When the code execution reaches the closing `}` of the block, every variable declared within the block is "destructed". Once a variable is destructed, its name can be re-used to create (construct) another variable of the same or a different type.

```
void example()
{
    double x { 1.0 }; // these are not the braces you are looking for

    {
        // block 1
        std::cout << N << "\n"; // this is an error, because N is not yet defined
        int N { 500 };
        double y = std::exp(N * log(x));
    } // variables N and y are destructed here, so that the names N and y become "undefined"

    std::cout << N << "\n"; // this is an error, because N is undefined at this point.

    {
        // block 2
        std::string N { "Jupiter" }; // this is OK!
        std::cout << N << "\t" << y << "\n"; // this is not ok.
    }
}
```

Notice that the variable `x` is used inside the block 1 above, although it is not declared inside that block. Variables declared in a block **A** are visible at all statements or blocks which are parts of the block **A**, starting immediately after the declaration, until the end of the block **A**. And when a new block starts, the visibility of all the variables at the point of the beginning of the block is carried over into the block. But variables are not visible outside the block where they are declared.

This last aspect is somewhat of a surprise for experienced python programmers when beginning to learn C++. In the following badly written python program, some variables defined inside of `if` blocks are used outside those blocks.

```python
# Python code (pyscope2.py) illustrating different scoping rules
import sys
if __name__ == "__main__":
    if len(sys.argv) > 1:
        N = int(sys.argv[1])
    else:
        N = 5

def fact(n):
    if n > 1:
        return n * fact(n-1)
```

```
12        return 1
13    while N > 0:
14        print(fact(N))
15        if N % 4 == 0:
16            fact = N * (N - 1) / 2
17        N = N -1
18
```

Above, the variable `N` is initialised inside an `if` and `else` block, two layers deep, but is available for use later. The variable `N` in lines 13 – 17 is the same variable as the one defined in lines 5 and 6. Running the above program with, for instance, a command line argument of 7, prints

```
5040
720
120
24
Traceback (most recent call last):
  File "pyscope2.py", line 13, in <module>
    print(fact(N))
TypeError: 'int' object is not callable
```

So, in the above python program, the loop body executes correctly for $N = 7$, $N = 6$, $N = 5$ and $N = 4$, but then for $N = 3$ it runs into an error, namely `int` is not callable. `fact` is declared as a function in line 9, but once we reach line 16, `fact` becomes an integer! This does not happen until the loop body executes with $N = 4$ and we reach line 16. Everything seems fine in the beginning, and if $N$ never got to the value of 4, the program would run through, and we wouldn't notice the error. But during the course of program execution, a situation arises which changes a function into an integer! To reason about the expected behaviour associated with the name `fact` at one specific point, we can not rely on its definition as a function, but we must understand how we got to that point, i.e., what statements were executed before the point of usage (in this case: whether or not line 16 has been executed before our attempt to use `fact`). Since loops and branch statements exist, the sequence of statements leading up to our usage point might actually be located later in the program source code. Had we changed the `fact` in another way, so that the program did not crash but simply changed behaviour, e.g., replacing line 16 above with `fact = lambda x : x + 1`, we would have a program that does not crash, produces the correct results initially, but changes its nature during the execution of the program, such that regions of code which previously produced good results would start producing rubbish. This is undesirable in large complex projects where such errors would be extremely hard to notice and fix. This specific kind of error, which can happen in python, can not happen in statically typed languages, such as C++.

The design goals for different programming languages are different, and our goal here is not to discuss python, and its best practices, but to understand the reasoning behind some of the complexity in C++ syntax. Python associates unchangeable types with values, such as 3.14 or "pi", but its variables can change type freely at execution time. In C++, both values and variables have strict unchangeable types, and we must tell the compiler in some way, what type is bound to a name we introduce into the program. The manner in which we tell the compiler the specific immutable type associated with a symbol may be verbose and explicit, e.g., `unsigned long it = vec.size();`, or the more compact and modern `auto it = vec.size();`, but in both cases we convey the complete type information to the compiler. `auto` does not imply that we leave the type of `it` undetermined until the function `M.size()` runs and then look at the return value and decide what type `it` is. Return types of functions are also immutable, so that before the program ever runs, the compiler can check the declaration for `size()` and without running anything, determine that we must mean `it` to be of the `unsigned long` type. `auto` is just a typographic convenience, and it has no ability to postpone determination of the type of a variable till execution time.

Similarly, python does not impose the hierarchical lifetimes in nested blocks, but C++ does. In C++, a variable declared inside a block (easily recognizable by enclosing braces `{` and `}`), are not available outside. They live separate lives compared to other variables of the same name appearing outside the block. Life time of a variable is easily recognized by looking at the code lines where it is declared and the surrounding pair of braces. Types don't change during the lifetime of a variable. To understand how a name appearing in one line of code might behave, we have to see the scope where that variable or function was declared, irrespective of the sequence of lines executed until we reach that point. These rules impose a certain structure into our C++ programs which sometimes do not look particularly "easy" to the beginner or "pythonic". The benefit is stricter guarantees on run time behaviour which is extremely valuable for large projects.

Let's continue the discussion of variable lifetimes in C++. From the above discussion, we reemphasize that in any given line of C++ source code, you can tell the type of all variables involved without considering the code paths taken at execution time to reach that line. It follows that for each declared variable, there exists a range of source lines where that variable is available for use or modification. Throughout this range, we have access to the value it stores, but over this entire range, its type remains as it was first declared. This range of visibility for a variable is called its *scope*. The scope of a variable in C++ is a purely "spatial" concept in the landscape of the source lines, not a space-time concept like it can be in dynamically typed languages like python. We need to ask "where is it" and not "what happened along the (not necessarily linear) path the particular run of the program took to get to that point in code".

Function bodies, as we noted earlier, are top level blocks. They define a top level block scope for variables. Formal parameters to a function, defined in the function header, are variables whose scope begins and ends with the function body. Their scope extends throughout the function body. We will see later that certain other control flow constructs such as `if` statements, and range based `for` loops, may introduce new variables in their respective "headers" whose scopes extend over the blocks under their control. Variables can also be declared anywhere inside a block.

The *scope* of a variable defined inside a block begins at the point of its declaration, and ends at the end of the block. It is however possible to make a variable "invisible" for a part of its scope. In the following code we have two variables called `name`, with their visibilities colour coded. The first `name`, defined in the outer block would have been visible over the entire inner block, but something hides it for a few code lines...

```
1  void example()
2  {
3      std::string moon{"Titan"};
4      std::string name = moon;
5      std::cout << name;
6      {
7          std::cout << name;
8          int name{10};
9          name = name - 3;
10         std::cout << name;
11     }
12     std::cout << name ;
13 }
```

The inner block creates another variable called `name`, which *shadows* the previous definition of `name` in the outer block. This is allowed. Since we recognize the variables declared in line 4 and 8 as separate entities, there can be no ambiguity about the nature of `name` in any given line of code. Nothing we do in line 9 and 10 will be able to change the fact that `name` refers to a `string` on line 12. This can be thought of as some form of fine grained control over variable visibility in C++, but in the opinion of many, including me, this leads more often to subtle bugs. This happens, for instance, when the types of the inner and outer scope variables are the same, as illustrated in the listing below.

```
auto add_till(unsigned int n) -> unsigned int
{
    auto tot = 0U;
    if (n % 2 == 0) {
        auto tot = func1(n);
        tot = tot + func2(n);
        // std::cout << tot << "\n";
    }
    return tot;
}
```

The above function always returns 0 irrespective of the value of `n` and irrespective of what functions `func1` or `func2` return. We intended the variable `tot` declared in line 3 to hold the calculated answer in all circumstances. For odd inputs, the intended output is 0, and everything goes to plan. For even inputs, the intention might have been to set `tot` to the sum of calls to various functions like `func1` and `func2`. But in line 5, we accidentally created a new variable called `tot` which shadows the one created in line 3. Value assignment in line 5 and the value change in line 6 then happen to the newly created variable, leaving the variable created in line 3 untouched. The accidentally created variable `tot` in line 5 expires in line 8, so that the original `tot` declared in line 3, which has remained unchanged since declaration, becomes visible again. That's what we return in line 9. Although this listing is an artificial, purposely idiotic, demonstrative code, such errors do happen in the wild. The "debugging" attempt in line 7, will also fail, because the value printed there will always be the correct calculated value, whereas the value returned by the function will be always 0. Such situations are best avoided.

C++ being a compiled language, we can ask the compiler to help us detect such "shadowing" situations. Most modern compilers can warn you if one of your declarations shadows a previous declaration. The exact compiler option might vary, but it is something one should find out. For `g++`, one could proceed as illustrated below:

```
1   // examples/blockscope.cc
2
3  #include <iostream>
4  #include <string>
5
6  int main()
7  {
8      std::string x{"three"};
9      {
10         std::cout << x << "\n";
11         double x = 488332;
12         std::cout << x << "\n";
13         {
14             x = x - 1;
15             {
16                 x = x / 3;
17                 std::cout << x << "\n";
18             }
19             {
20                 std::string x{"four"};
21                 std::cout << x << "\n";
22             }
23         }
24     }
25     std::cout << x << "\n";
26 }
27
```

```
g++ -Wshadow blockscope.cc -o blockscopedemo

blockscope.cc: In function 'int main()':
blockscope.cc:11:16: warning: declaration of 'x' shadows a previous local [-Wshadow]
   11 |         double x = 488332;
      |                ^
blockscope.cc:8:17: note: shadowed declaration is here
    8 |     std::string x{"three"};
      |                 ^
blockscope.cc:20:29: warning: declaration of 'x' shadows a previous local [-Wshadow]
   20 |                 std::string x{"four"};
      |                             ^
blockscope.cc:11:16: note: shadowed declaration is here
   11 |         double x = 488332;
      |                ^
```

In your own code, avoid shadowing as much as possible. You should be aware of this possibility when debugging unexpected behaviour from code you did not write. Use the relevant compiler flags to detect instances of shadowing.

Some people draw the wrong lesson from the above discussion on shadowing: “I will put a localized variable declaration area at the top of my source file and never declare variables anywhere else in the code. This way can easily ensure that I am using unique names for my variables”. Doing so (perfectly) will avoid shadowing, but make reasoning about the code a lot harder. The larger the scope of a variable, the more code lines you have to examine to know its current value. This also makes the code more inflexible, because any changes you make to the variable in one line might be incompatible to the way that variable is used in another part of the code not immediately visible. In multi-threaded programs, a great deal of extra effort would be needed to ensure that such wide visibility variables are not changed by another thread in some unexpected manner. Shadowing can lead to real problems, but we have tools to detect and handle them during program compilation. Existence of shadowing should not lead you to abandon one of the best features you have at your disposal: variables with limited scope. The other approach, gathering declarations of parameters in one place, has some more merit for constants, especially those whose value would remain the same for all parts of a program.

Since C++ allows creation of variables inside blocks, which can be nested in a neat tree structure, and variables visible at the beginning of a (sub-) block remain visible (unless shadowed) throughout the block, one simple and effective recommendation presents itself: one should define variables in the smallest scope where they are used. If a variable is only used in a block of code spanning a few consecutive lines somewhere, it should be defined in that block, instead of its parent, grandparent or some other ancestor block. It is easier to reason about such variables, and accidental assignment of the wrong values can not occur out of sight in an ancestor block. So, although the compiler will accept it, please do not declare all your (mutable) variables at the start of a function, as was common practice in the “C” language before C99. Declare it when it is needed, and keep its scope as small as possible.

### Exercise 7:

The program `examples/blockscope.cc` demonstrates the possibly non-contiguous visibility of block scope C++ variables. We have several variables by the same name defined in a nested set of blocks. Observe how shadowing affects the visibility of different variables. Notice also that both `g++` and `clang++` are able to correctly show us where such shadowing happens if we use the compiler option `-Wshadow`.

As you saw in the Exercise 1.4.2, variables may also be defined outside the top level scope of functions or classes. Such variables are global variables, and their lifetime is the lifetime of the program. In section 2.1.4, you will see how to organize such variables in a hierarchy of namespaces. Each additional mutable global variable used by a section of code makes it harder to reason about the state of the program using the local context of code lines. They should only ever be used when no better way can be found. Your first choice for mutable (non-constant) variables should always be to put them in the narrowest possible block.

## 2.1.3 Syntax of variable declaration and initialisation

As we have seen, there are several equivalent ways of introducing a new variable of the name `variablename` and with the given initial value.

1. `auto variablename { initializer };`
2. `auto variablename = initializer;`
3. `decltype(onevar) anothervar { different_initializer };`
4. `SomeTypeName variablename { initializer };`
5. `SomeTypeName variablename = initializer;`

In the first two forms, the type of the variable to be created is the type of the initializer *expression* (more about expressions later). If the initializer is another variable, then the type and value of the newly created variable are the same as the other variable. So, `int y{0}; auto x{y};` will create `x` with the same type as `y`, and an initial value equal to the value of `y` at the time of creation of `x`. If the initializer is a literal value, like 5 or 2.71, the type is deduced based on the type of the literal. For instance, `auto i{0};` will deduce i to be a signed integer with initial value 0. This is because literal integral numbers in your program text, like, `0`, `1`, `2` ... are interpreted as signed integers. Literal representation of unsigned integers contains a suffix `U`. For instance, `auto x{0U};` will create `x` as an unsigned integer (no sign bit, modular arithmetic ...). The size of an integer variable (the number of bits it uses in the computer memory) is usually 32 bits. There is another integer type called `long` which, nowadays, is usually 64 bits long (although it is only guaranteed to be at least as big as an `int`). Literal values of the long type take the suffix `L`. To create an unsigned long integer, with initial value 0, one can use `auto i { 0UL };`. Literal real numbers like `2.71` in program text are interpreted as being of type `double`, which is a 64 bit representation of real numbers. To create 32 bit real numbers, called `float`, from literal values, one can use the suffix `F` for the literal, e.g., `auto f32 = 2.71F;`. Character string literals, as in `"Cologne"` are treated as C-style character strings. C++ inherits these from C and can work with such strings. But generally use of C style strings in C++ programs leads to much needless suffering. Usually, we are much better served with C++ standard library `string` or `string_view`. We will discuss them in detail later. For now, let's just note how one might create them from literals.

```cpp
#include <string>
void somefunc()
{
    using namespace std::string_literals;
    auto city { "Cologne"s };
}
```

The little 's' at the end of the initializer makes it a C++ standard library string literal rather than a C-style string. So, `city` here will be created with the type `std::string` and value `"Cologne"`. Literals with single quotes, e.g., 'y', 'n' etc. are character literals. They can be used to create variables of type `char`.

C++ allows programmers to create literal representations for types they create. For instance, by the end of this course, it will be easy for you to create a type for distance measurements, so that you can create variables of the distance type. You could then imbue your `Distance` type with a literal representation, so that in your program you could do things like this:

```cpp
void somefunc()
{
    auto intercity { 100.0_km };
    auto height { 75.0_inches };
    if ( intercity > 50'000 * height ) do_something();
}
```

Such extended use of literal values in code is also seen in many places in the C++ standard library. For instance, one can create complex numbers like this:

```cpp
#include <iostream>
#include <complex>

auto main() -> int
{
    using namespace std::complex_literals;
    auto Z{ 1.22 + 3.991i };
    std::cout << Z << "\n";
}
```

The same goes for literal values related to date time measurements:

```cpp
#include <iostream>
#include <chrono>

auto main() -> int
{
    using namespace std::chrono;
    using namespace std::chrono_literals;
    year_month_weekday xmas { 2022y / December / 25d };
    if (xmas.weekday() == Sunday) {
        std::cout << "X-Mas on a Sunday!\n";
    }
}
```

The third form of variable declaration presented in section 2.1.3, with `decltype`, creates the new variable with type information from a pre-existing variable, but initial value from a different initializer which may or may not have the same type. In the third, fourth and fifth versions of the variable declarations at the start of this section, the initial value is, technically, optional, and one can create a variable with `SomeTypeName variablename;`. It is good practice however, to always provide sensible initial values. There are some situations where explicit spelling out of the types, as in version 4 and 5 in the list at the beginning of section 2.1.3, are valuable. There are many situations where we intend the newly created variable to be of a different type than the type of the literal value used in the initializer. One example is in the above code snippet, where the variable `xmas` was declared explicitly with the type `year_month_weekday`. In our discussions of the C++ standard library, you will learn that the initializer `2022y / December / 25d` has the type `year_month_day` rather than `year_month_weekday`. We wanted to make use of information regarding weekdays, so we created the variable with an explicit type which knows about weekdays. Another example: we may want to create *a list of integers* starting with an initial list with only the integer 7. Writing `auto L{7};` will not have the intended effect, as it will only give us an integer with a current value 7. We could say `std::list L{ 7 }`, (pre-C++17 we had to say `std::list<int> L{ 7 }`) so that the compiler knows that the type of the new variable is as given on the left, and not something to infer from the initializer. For the case of collections of objects, like lists, even finer grained control of initialisation is desired. What if we meant a list containing 7 elements each initialized to 0 rather than a list of one element initialized to 7? For this case, we can use `std::list L(7UL, 0);`, i.e., with round brackets or parentheses instead of curly brackets. Occasionally it could make it easier to read code if we write the types like `int` or `double` explicitly. But C++ type names can become very long. Given an associative container called `themap` with `std::string` as key type and `unsigned long` as value type, few people will argue that
`std::map<std::string, unsigned long>::const_iterator it = themap.begin();`
is easier to read than
`auto it = themap.begin();`

There is often a choice regarding initialisation between the `= initializer` version and the `{ initializer }` version. Both can get the job done, but one should prefer the `{ initializer }` version to initialise. This form is called uniform initialisation, and has somewhat stricter rules on the kind of transformations it can do to initialise. For example, `int i = 1.2;` will be accepted and the integer `i` will be created with a value `1`. Most likely, when someone types such a thing, they intended the variable `i` to not be an integer. The `{}` syntax does not accept this. `int i{ 1.2 };` will cause a compiler error telling us that we are "narrowing" the specified initial value for storage in the integer. Narrowing conversions are not allowed in the uniform initialisation syntax, and such a compiler error draws our attention to something that might be a mistake. Besides, the `{}` syntax allows us to initialise also containers like `std::list`, `std::vector`, etc. with a sequence of elements. We will learn how to enable such syntax for user defined types later in our discussion of classes.

### 2.1.4 Namespaces

So, what's with all these `::` signs in C++ programs? These relate to the concept of *namespaces* in C++. It is a way to organize names of variables, functions, classes and other entities in a hierarchical manner. The concept is actually quite easy to understand, because it is easily translated to everyday life.

Consider the word glass. It could mean a drink ware, as in, “may I have a glass of water please?” It could also mean the material, like in “the house has big glass windows”. It could also mean eye-glasses. We are very good at intuitively understanding what is meant when a sentence includes more of these different uses of the word, e.g., “I didn’t realize this glass is not in fact made of glass!”. When we hear that, we internally translate it to something like, “I didn’t realize this (drink ware) glass is not in fact made of (material) glass”. That is essentially what we are trying to do with the `::` sign. It is called the scope resolution operator. Using the C++ scope resolution operator in the above sentence, we will get “I didn’t realize this drinkware::glass is not in fact made of material::glass.” Namespaces create a context for a name. Let’s see how to create and use them:

```cpp
// examples/ns1.cc
#include <iostream>

namespace cxx_course {
    unsigned int participant_count{0};
    void greet()
    {
        std::cout << "Study and practice. Years of it!\n";
    }
}
namespace gardening_course {
    unsigned int participant_count{0};
    void greet()
    {
        std::cout << "You reap what you sow.\n";
    }
}
auto main() -> int
{
    cxx_course::greet();
}
```

To add something to a namespace, we do `namespace namespacename { declarations...; }`. Any number of declarations of variables, functions or other things can be put inside the beginning `{` and ending `}` of a namespace. The variables declared can be initialized, but no statement which does not bring a new name into existence can occur freely inside a namespace. In our example above, the statements containing `std::cout` are typographically within the opening and closing of the namespace. But they occur inside functions. As we stated before, any statement which does anything other than declaring a new name, must be in a function. To use a function or a variable belonging to a namespace, we can use its full name, like `cxx_course::greet()` above. There are two functions by the name `greet`, neither requires any inputs. By putting them in different namespaces, we make it possible to precisely specify which one we want to use, i.e., to avoid ambiguity.

Creating namespaces is not simply a glorified way to add prefixes to names. We could create names including some of their context, such as, `std_vector`, `std_cout` and so on. Such would even have less characters! But each such name would stand on its own. We could not choose to use short versions `vector` and `cout` in a tightly controlled context. A real `namespace` allows that. If we need to use one or more symbols from a `namespace` many times in a block of code, we can express that intent in our code:

```cpp
// examples/ns2.cc
#include <iostream>

namespace cxx_course {
    unsigned int participant_count{0};
    void greet()
    {
        std::cout << "Study and practice. Years of it!\n";
    }
}
namespace gardening_course {
    unsigned int participant_count{0};
    void greet()
14         {
15             std::cout << "You reap what you sow.\n";
16         }
17     }
18     auto main() -> int
19     {
20         using namespace cxx_course;
21         greet();
22         std::cout << "The number of participants is "
23                   << participant_count << "\n";
24     }
```

In line 20 above, we state that we are using the namespace `cxx_course`, i.e., we will be using the names defined in that namespace without qualifying them with `cxx_course::`. So, `greet()` in line 21 is a call to the function `greet()` defined in the namespace `cxx_course`, and not the function of the same name in the `gardening_course` namespace. Such a `using` directive can be placed in a block so that all names in the used namespace become available in that block after that directive. The visibility of the borrowed names ends at the end of the block. Writing `seconds` is easier than always writing `std::chrono::seconds`. If we are working on time measurements in a block of code, we can say `using namespace std::chrono` in that block and simply write `seconds` inside that block. If we are programming about angle measurements, in that context we can put a suitable, `using namespace madeup::angle_measurements` statement, and proceed to use `seconds` to mean a tiny angle instead. If we need both tiny angles and time in some context, we use namespace qualified names explicitly for disambiguation.

## Exercise 8:

The program `examples/ns3.cc` illustrates how one can use different namespaces in different parts of a single function. It uses the number of command line arguments to decide whether to use the `cxx_course` namespace or the `hpc_course` namespace. Observe how it works!

## Exercise 9:

In the previous examples, we have used some names from the C++ standard library, which are defined in the namespace `std`. Therefore we used `std::cout`, `std::cos` etc. Modify a few of them by adding a suitable `using` directive in the narrowest possible context.

`using` directives can also be placed in a namespace.

```
namespace cxx_course {
    // declarations of a,b,c
}
namespace hpc_course {
    using namespace cxx_course;
    // declarations of d, e, f
}
```

When we place a `using` directive in a namespace as above, the names from the used namespace `cxx_course` will be made available inside the namespace `hpc_course`, as if they were declared there, i.e., `hpc_course::a` will come to mean `cxx_course::a` and so on.

Namespaces can also be nested to arbitrary depth, in other words, a namespace can contain other namespaces. When referring to a symbol in an inner namespace we use a chain of namespace names separated by `::` till the desired name.

```cpp
namespace MC {
    // declarations of FunctionType, DerivativeType
    namespace utils {
        namespace math {
            auto derivative(FunctionType f) -> DerivativeType;
        }
    }
}
int main()
{
    MC::FunctionType f{};
    // ...
    auto d = MC::utils::math::derivative(f);
}
```

Notice how we referred to the `derivative` function as `MC::utils::math::derivative`. It vaguely resembles a filesystem path in structure, with `::` instead of `/` separating its components. It even shares some properties with a filesystem path. In a function declared in the namespace `utils` above, we could refer to the `derivative` function in `MC::utils::math` as simply `math::derivative`. Just like we could provide an absolute path, necessarily starting from the root of the filesystem instead of the current working directory, by prepending our path with a `/`, we could anchor our qualified names relative to the global namespace by starting them with a `::`. To do that, we would refer to our derivative function above as `::MC::utils::math::derivative`. This is often done to avoid any ambiguities.

The analogy with the filesystem should not be taken too far. First of all, there is no equivalent of the `..` notation to refer to the parent namespace, like we can for the parent directory. The method of determining what a name refers to is also quite different. When we refer to a variable or function name somewhere, the compiler tries to understand what we mean. At first, it checks if that name is one of the names directly visible in the current scope for having been declared there or in an ancestor block. The name may have been imported by a `using` directive. If not, the compiler tries to interpret the name with respect to the current namespace. If that fails it goes one level higher until it finds it at some level or reaches the global namespace. If it is still not found, it is reported as an error. If any intermediate namespace defines a different undesirable object by the name we are searching, but we want to refer to the name in the global context instead, we use the "absolute path" method, and start with a `::`.

## Exercise 10:

The program `examples/ns4.cc` contains a somewhat longer example with lots of alternative definitions of a variable called `c` in different namespaces. Alternative ways to use the name `c` lead to different symbols being found from the different namespaces. The code comments explain how a particular way to use `c` locates a version of it. Study it. Uncomment the alternative versions of the `return` statement and run the program, and compare the results with the explanations given in the comments.

`namespace ABC { declarations ... }` opens a namespace called `ABC` to add declarations into it. If `ABC` doesn't exist, it is created. If we have previously defined the namespace and have some symbols in it, we simply reopen the same namespace and add more things into it. There are no restrictions on the number of namespaces defined in a header or source file, and no automatic connection between the source file names and the namespace names. Similarly the file hierarchy in a source tree is unrelated to the namespace hierarchy.

Long namespace names can be shortened by creating aliases. For instance,

```cpp
namespace sr = std::ranges;
```

will let us refer to `std::ranges::transform` as simply `sr::transform`.

It is also possible to open an inner namespace directly for appending like this:

```
namespace MC::utils::math {
    // new declarations...
}
```

Namespaces can not be declared in block scope. Since the body of a function is a block, namespaces and all their content live outside function bodies. Variables defined in namespaces can be accessed from any function, using a fully qualified name. Whereas variables declared inside a block come into existence when that block of code is executed and are destructed when the block ends, variables in namespaces live for the entire duration of the program. They are, therefore, global variables. Namespaces give us a way to organize them and access them in a fine grained manner, but that does not change anything about the lifetime of the variables in a namespace. As a general rule, one should avoid defining variables inside namespaces, unless they are constants, or there is a good reason justifying having a global variable. Our familiar `std::cout` is in fact, a global variable defined in the namespace `std`! There is, after all, only one standard output, and having that as a global variable has certain benefits. But, global variables have tremendous potential for making our lives miserable with unforeseen consequences. A good rule of palm I use is this: every time I am tempted to use a (non-constant) global variable in my professional life, I go to my fridge, take out an ice cube and hold it tightly in a fist. The price for using a mutable global variable is enduring the cold ice until it melts completely in the hand. If all other solutions I can imagine are more painful than that, sure, I go ahead and use a global. This cost-benefit analysis has largely steered me towards the right decisions.

Namespaces should be used to organize immutable constant variables, functions, classes and concept definitions in a project. Despite the syntactic convenience in modern C++ for creating and aliasing nested namespaces, their raison d'être is avoiding name clashes, and not a full taxonomy of different kind of entities in the project. That's why we have `std::vector`, and not `std::containers::contiguous::vector`, `std::map` and not `std::containers::associative::map` and so on. On the other hand, where a name clash might be otherwise unavoidable, one can use namespaces for disambiguation, as for example done in `std::transform` and `std::ranges::transform`.

## 2.2 Expressions

Expressions consist of a sequence of operators with their respective arguments, and specify some form of computation to be performed. For instance, there are algebraic expressions such as, $13(x^3 - 4x) + \frac{12.3}{x}$, written in C++ as, $13 * (x * x * x - 4 * x) + 12.3/x$, which are evaluated following the usual rules governing parentheses and the arithmetic operators. The above expression would be interpreted as $(13 * (((x * x) * x) - (4 * x))) + (12.3/x)$, and evaluated outwards starting from the smallest parentheses.

Every expression has a type and a value, corresponding to the result of the computation it specifies. The above expression would have the type `double` if the type of `x` is one of the common numeric types. A boolean expression is an expression which results in a `bool` value (`true` or `false`): `x * x < N` is a boolean expression whose value depends on the values of `x` and `N`. The name of a variable or a literal value is also a trivial expression, with an obvious type and value.

There are many kinds of expressions in C++, but for now we need to know about only a few of them. An expression as a whole has a value and type which are the value and type of the object resulting because of the evaluation of the expression. If `x` is a `double`, the expression `x * x + 1.` has a double type, as its value can be any real number greater than 1.0, and `x * x > 5.0` has a `bool` type, as its value can only be `true` or `false`. As in many programming languages, in C++, if we want to compare two values to be equal, we use the equality comparison operator, `==`, with two equal sign. A single equal sign is interpreted as "assignment". So, if `A` is an `int`, `A = 5` sets the value of `A` to `5`, where as `A == 5` compares `A` with `5` and is either `true` or `false`. Boolean expressions can be combined using boolean operators like `and`, `or`, `not` etc. For example, `A >= 0 and A < 10` is an expression that is true if the value of `A` is between `0` and `9` including both ends. The operators can be written spelt out as `and`, `or` etc. or as `&&`, `||` etc. One peculiarity of boolean operators is that they are evaluated left to right only to the point necessary to establish their truth value. In boolean algebra, an expression ($A$ *or* $B$) can never be false if $A$ is true, and ($A$ *and* $B$) can never be true if $A$ is false. So, when we have an expression like `i < N and a[i] > 0`, the second part `a[i] > 0` is only evaluated when `i < N`. If `i < N` is false, the truth value of the boolean expression is already known, and the rest of the expression does not need to be evaluated and is (officially) skipped.

Assignment expressions, like `resultvar = 4 * x * x + 3 * x + 9`, contain (often, but not always) a variable name on the left hand side and an expression on the right hand side of the `=` sign. The expression on the right of the `=` sign is evaluated, and the resultant value is stored in memory as determined by the left hand side. Entities which may appear on the left hand side of an assignment expression are called "L-values". L-values, like a plain variable name like `resultvar`, have a writable memory location associated with them where the result of the right hand side expression is written. The assignment expression as a whole has a value: it is the assigned value. So, it is possible to chain assignments like this: `x = y = z = 0.0`. A chain of assignment expressions is evaluated from right to left. Here, `z = 0.0` stores 0.0 in `z`, and the expression `z = 0.0` has a value 0.0, so that `y` is effectively being assigned to 0.0, and then `x`.

Expressions may compute a value like the above, and some expressions may produce a side effect, like showing something on the screen. To evaluate an expression, it is parsed into a tree of sub-expressions. The sub-expressions are then evaluated and the results of the sub-expressions are combined to create a value for the expression. Sub-expressions are themselves expressions. If an expression in this tree is a simple variable name or a literal value like 5.0, it is no longer sub-divided, but simply substituted by its value. Similarly, if the (sub-)expression is a function call, it is replaced by the result of the function call. Sometimes, evaluation of an expression can have side-effects, and that has an important consequence regarding evaluation of expression trees, which I would like to point out right from the start.

```
1  // examples/eval_order.cc
2  #include <iostream>
3
4  namespace mystuff {
5      int counter{42};
6  }
7
8  auto f(int input) -> int
9  {
10     auto tmp = mystuff::counter;
11     mystuff::counter = mystuff::counter + 1;
12     return input + tmp;
13 }
14
15 auto g(int p1, int p2, int p3)
16 {
17     std::cout << "Received parameters " << p1 << ", " << p2 << ", " << p3 << "\n";
18 }
19
20 auto main() -> int
21 {
22     g( f(1), f(1), f(1) );
23     std::cout << f(1) << ", " << f(1) << ", " << f(1) << "\n";
24 }
```

The function `f()` in listing above changes the global variable `mystuff::counter`. This is a "side-effect". The function changes something outside itself besides calculating its result. In this case, the result of the function will depend not just on the input we provide, but also on how often the function has been called before. Same inputs can therefore give us different outputs in such a function. A function call expression like `f(1)` with that kind of a function is then an expression which has a side effect. Using an expression with such a side effect more than once in an expression can lead to unexpected results. There are two examples of such usage in the listing above, in lines 22 and 23. In line 22, we are calling it 3 times to generate the 3 inputs to the function `g()` and in line 23, we are just writing the results of 3 successive calls to `f(1)`. When evaluating the function call expression for `g()` in line 22, we have to evaluate the 3 sub-expressions corresponding to the arguments to `g()`. We have written all of them as `f(1)`, and we have to replace each argument by the result of `f(1)`. But we know that the result of `f(1)` will depend on how often we have called the function before. We need 3 calls. In what order do we execute them in order to generate the first, second and third argument for `g()`? Similarly, in line 23, we have 3 calls to `f(1)` to be written out to standard output. Let's see what happens:

```
$ g++ -std=c++20 eval_order.cc -o eval_order.gcc
$ ./eval_order.gcc
Received parameters 44, 43, 42
45, 46, 47
$ clang++ -std=c++20 eval_order.cc -o eval_order.clg
$ ./eval_order.clg
Received parameters 42, 43, 44
45, 46, 47
```

Notice how the different compilers passed different sets of parameters to `g()`. It turns out that both are within their rights to do what they did. The code above invokes what is known as undefined behaviour! The order in which sub-expressions are evaluated is governed by a set of well defined, but not necessarily obvious rules, described in this link. The relevant part to understand what is happening in our example above is that the sub-expressions corresponding to the different arguments to a function (`g()` in this example) are "indeterminately sequenced" relative to each other. Since it is not prescribed by the standard, compilers may (and do) evaluate those sub-expressions in different orders. This would not be a problem if `f()` did not have any side-effects, but in our example, it does. One recommendation suggests itself: if an expression has side-effects, do not use it in multiple sub-expressions inside an expression. More strictly, different sub-expressions should not modify the same global state (like the variable `mystuff::counter` above).

## 2.3 Basic control flow regulation

### 2.3.1 Branches

For our purposes, when a function is called, the instructions in its body are executed from the top, one by one, until a `return` statement is encountered. As an example, let's consider the following simple program to solve a quadratic equation by taking the coefficients from the three expected parameters. The solution here is taken from the numerical recipes series of books:

$$\begin{aligned}
ax^2 + bx + c &= 0 \\
q &= -\frac{1}{2}\left[b + sgn(b)\sqrt{b^2 - 4ac}\right] \\
x_1 &= \frac{q}{a} \\
x_2 &= \frac{c}{q}
\end{aligned}$$

The pair of roots $(x_1, x_2)$ is written in terms of a convenient intermediate value $q$. This form of writing the solution is more stable for small values of parameters $a$ and $c$. Here is a simple C++ implementation.

```
auto solve_quadratic(double a, double b, double c) -> std::pair<double, double>
{
    auto D = b * b - 4 * a * c;
    auto q = -0.5 * ( b + std::copysign(std::sqrt(D), b) );
    return { q / a, c / q };
}
```

It's fairly straight forward: we calculate `D`, use it to calculate `q` and then use that to calculate and return the pair of roots as an `std::pair` of two doubles. The function `copysign(A, B)` returns a number with the magnitude of `A` and sign of `B`.

### Exercise 11:

The file `examples/quadratic0.cc` contains the above function and a `main` program to test it. The program expects three command line arguments taking the three coefficients $a$, $b$ and $c$ and prints a message with the equation and the solutions. Compile and run with a few example coefficients for the equation. For example,

```
g++ -std=c++2a -O2 quadratic0.cc -o quadratic0
./quadratic0 1.0 4.0 0.25
```

If you tried it out with a few different sets of values, you would quickly have discovered that

1. If you forget to provide all 3 arguments, or give too few arguments, the program simply unceremoniously crashes! It would be better if it told us what went wrong.
2. If $b^2 - 4ac < 0$, the square root calculation runs into trouble. In this case there are no solutions in the space of real numbers. Perhaps it should somehow tell us that the equation does not have any real solutions.

We have to do different things based on conditions arising during the program execution. We want to say something like "if AConditionIsTrue, then do something". In C++ syntax this is written as `if (condition) statement_or_block`, where `condition` is a boolean expression. For instance,

```
 if (argc < 4)
     std::cerr << "Not enough command line arguments!\n";
```

The body of an `if` statement can be a simple statement as above, or a block.

```
 if (argc < 4) {
     std::cerr << "Not enough command line arguments!\n";
     return 1; // Remember: Returning 1 from main indicates an error
 }
```

Similarly, for the function `solve_quadratic`, we have the situation that it can only sometimes return an answer. One can say that the roots, that it is supposed to return, are optional. They are returned only when possible. See if you find the following code readable...

```
auto solve_quadratic(double a, double b, double c)
    -> std::optional < std::pair<double, double> >
{
    using namespace std;
    optional<pair<double, double>> solution;
    auto D = b * b - 4 * a * c;
    if (D >= 0.) {
        auto q = -0.5 * ( b + std::copysign(std::sqrt(D), b) );
        solution = make_pair(q / a, c / q);
    }
    return solution;
}
```

The two lines calculating `q` and the two roots are only executed if `D` is calculated to be non-negative. The `std::optional` construct is one of the many utilities available to handle errors. The function now has a return type such that it optionally holds a pair of doubles. An `std::optional` object can be thought of as a little box which may or may not hold an entity of a given type. You can create it as an empty box with no value. If you assign a value, that's value the box holds. The value can be retrieved from the box when needed ( `opt_object.value()` or `*opt_object` ). That type is specified inside those angular brackets (in this case, `std::pair<double, double>` ). If you test an `optional` object like a condition, it returns `true` if and only if it (the box) currently holds a value. `if (optionalobject) do_something` only does that something if the `optionalobject` actually holds a value. In our example above, if `D` in fact is negative, we would never assign a pair of doubles to the solution, and so the `optional` will not contain any values.

Outside of `solve_quadratic`, when we use it, we can test whether we got a solution or not, and act accordingly. Taking these elements together, we have an improved version of our quadratic equation program in `quadratic1.cc`.

## Exercise 12:

Try out `examples/quadratic1.cc`. Remember we said earlier that variables should be declared as locally as possible? The variable `D` is not really needed outside the `if` statement and its block. We can not put it in the block, because it is used in the condition. Now try this: cut and paste that entire line where `D` is being defined and initialised, moving it to the highlighted location in this example if statement (trailing semi-colon and all):

```
if (     condition)
    do_something;
```

What does your compiler say about that?

As you may have discovered in doing the above exercise, it is possible to declare a variable in an `if` statement, just before the condition itself. The scope of such a variable consists of the condition expression itself as well as the blocks of code belonging to the different branches created by the `if`. The full syntax of the `if` statement is as follows:

```
if (variable_declaration; boolean_expression)
    statement_or_block
else
    statement_or_block
```

And since `if` is a statement, one can use it as the statement to be executed in the `else` case, so that we get a chained version ...

```
if (variable_declarations; boolean_expression)
    statement_or_block
else if (another_boolean_expression)
    statement_or_block
else if (you_know_what)
    statement_or_block
else
    statement_or_block
```

Each `if` in such a chain can have a variable declaration. But remember that the variable declared in one such `if` header, remains visible for the rest of the chain. So, declaring another variable with the same name anywhere later in the chain amounts to shadowing.

Chained `if ... else if ... else if ... else ...` statements can be used to trigger multiple actions such as a rudimentary "menu", where different things happen when an integer has different values. In the example below, we have a chain of such `if` statements. The program takes a single integer as a command line argument and chooses an action based on its value. The action used here for demo is just tell us what option we chose, but it could be any block of code.

```

// examples/switchdemo.cc
```

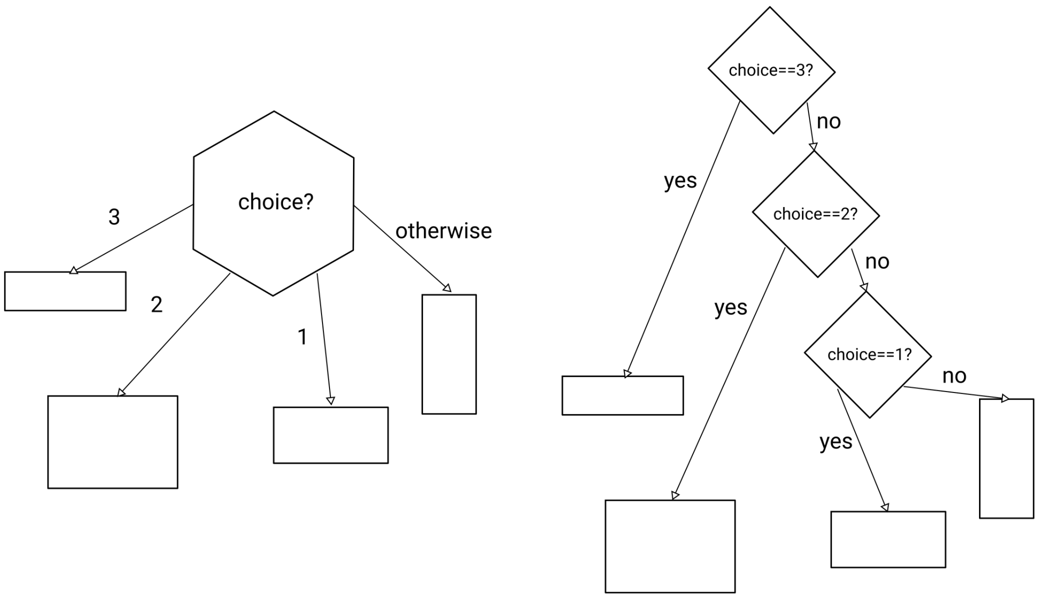

Figure 2.1: Converting a multi-way selection into a series of yes-no questions for `if` statements.

```cpp
#include <iostream>

auto main(int argc, char *argv[]) -> int
{
    if (argc > 1) {
        auto choice = std::stoi(argv[1]);
        if (choice == 3) {
                std::cout << "You chose option 3\n";
            } else if (choice == 2) {
                std::cout << "You chose option 2\n";
            } else if (choice == 1) {
                std::cout << "You chose option 1\n";
            } else {
                std::cout << "You have to choose an option between 1-3\n";
            }
        }
    } else std::cerr << "Needs an integer as a command line argument\n";
}
```

In the above, we express that depending on the value of a single object, `choice`, we want to trigger various different actions. But since the `if` statement is a yes-no split, we express our intent by converting a multi-way decision into many yes-no questions, as in Fig. 2.1. There is however another construct in C++ that matches our intent more directly: the `switch` statement. The `switch` statement works with any integer like expression, and selects an action based on its value. The different actions are organized as "cases". Here is the above example written in terms of the `switch` statement.

```cpp
// examples/switchdemo.cc
#include <iostream>

auto main(int argc, char *argv[]) -> int
{
    if (argc > 1) {
        auto choice = std::stoi(argv[1]);
        switch (choice) {
            case 3: {
                std::cout << "You chose option 3\n";
                break;
            }
            case 2: {
                std::cout << "You chose option 2\n";
                break;
            }
            case 1: {
                std::cout << "You chose option 1\n";
                break;
            }
            case 0:
            default: {
                std::cout << "You have to choose an option between 1-3\n";
            }
        };
    }
}
```

Like the `if` statement, `switch` can define variables before the selection expression: `switch (variable_definition; selection_expression) { cases };`. When executed, `switch` jumps to the `case` matching the value of the selection expression, and executes the code starting from there, until it reaches the end of the `switch` statement or a `break;`. This means when jumping to one case, `switch` would continue with the code in all cases that follow! The `break` statements at the end of a case block prevent this. Execute the above code, and try deleting or commenting out, say, the break statement in case 2. You will see that if you choose option 2, both the code in case 2 and case 1 will be executed. If you choose 1, only the code in case 1 will be executed. The `default` option at the end represents "in all cases not covered by the explicit case blocks".

### 2.3.2 Repeated actions with loops

One of the most useful tools in the arsenal of a programmer in imperative programming (as opposed to functional programming) is the ability to explicitly specify repeated execution of the some code lines. This is accomplished using *loop* constructs.

The `while` loop has the form `while (something_is_true) do_something`, and simply repeats a statement or block while given a boolean expression remains true. It should be immediately obvious that there should be some way for the boolean condition to become false while the repetitions are taking place, or else the loop will continue for ever! Here is an example:

```
1 int N = 5;
2 while (N > 0) {
3     std::cout << N << "\n";
4     N = N - 1;
5 }
6 // code after the loop
```

We will reach the line 2 in the code above, with $N == 5$. So, $N > 0$ will be true, and the body of the loop, in the block bounded by the bold `{}` will be executed. 5 will be printed, and then $N$ will become 4. Then the body of the loop is completed, we return to check the loop condition. $N > 0$ is still true, so the body will be executed again. And so on. We will see 4, 3, 2, and 1 printed. But after printing 1, $N$ will be changed to 0. The next time we check the repetition condition, it will fail, and the loop body will not be executed, and instead we jump to the code coming after the loop.

It is possible to break the loop at any point using a `break` statement. For instance, in the following, we count down from some given number till 0, but stop prematurely at a number $n$, if the total number of 1 bits in the binary representation of $n$ is 3.

```
// examples/while_with_break.cc
void countdown(unsigned long N)
{
    while (N > 0) {
        std::cout << N << "\n";
        if (std::popcount(N) == 3) {
            std::cout << "Reached exceptional termination with N = " << N << "\n";
            break; // this breaks the while loop
        }
        N = N -1;
    }
}
```

There is a slightly different kind of loop called the `do ... while;` loop. In this loop, the loop body is executed at least once. As an example, imagine that we need to find the first integer such that the sum of logarithms of all positive integers up to it ≥ 100. We can do this by following this procedure:

1. Initialise the sum to real number zero, and a "current" integer to integer zero.
2. Increment current integer by 1
3. Add the logarithm of the current integer to the sum
4. If sum is still smaller than the limit, repeat, starting from step 2

This can be done in many ways to C++, but we will demonstrate with the `do...while` loop.

```
auto func(double limit) -> unsigned int
{
    auto current = 0U;
    auto sum = 0.; // Because we will be summing logarithms
    do {
        current = current + 1;
        sum = sum + std::log(current);
    } while (sum < limit);
    return current;
}
```

The `do loop_body while (boolean_expression);` loop repeats the code in its body (which may be a single statement or a block) as long as a given condition remains true. The loop body is executed before checking the condition, and therefore runs at least once.

The syntax of the two kinds of `while` loops does not highlight how many times the loop body executes. Two other forms of loops exist which are designed to be a bit more easy to understand in terms of how often they will repeat the body. First let's see the "for each" loop. In C++, it is called the "range based for loop". The syntax is
`for (auto i : range) do_something`. As with other loops, the body can be a single statement or a block. What exactly is a `range` here? By a range here we mean anything which has a `begin` and and `end`. The precise definitions will be given when appropriate, right now let's just use the loop! Here are a few examples...

```
// examples/range_for.cc

std::array A { 1, 2, 3, 4, 5 };
for (auto a : A) std::cout << a << "\n";

for (auto partialsum = 0; auto a : A) {
    partialsum = partialsum + a;
    std::cout << a << "\t" << partialsum << "\n";
}

for (auto day : {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"}) {
    std::cout << day << "\n";
}

namespace sv = std::views;
for (auto i : sv::iota(71)) {
    std::cout << i << "\n";
    if (std::popcount(i) == 7) break;
}
```

Our `std::array` in the above example, as well as the `initializer_list` containing the week day names are entities with whose `begin` and `end` are known to the compiler. It is therefore possible to iterate over the elements in the range with this kind of a loop. The loop variables, `a` in the first two cases and `day` in the third, take successive values from a given sequence in the body of the loop. In the last example, we show a loop over an open-ended range starting from 71. The loop variable takes values 71, 72, 73 ... and that loop has no specified end. We therefore provide a `break` statement to terminate it at some point. `std::views::iota(n)` creates an unlimited sequence of integers starting from `n`. It is not stored anywhere like our array example. The integers are simply available when they are needed. We will learn more about these `views` in a later chapter.

Finally, we have the `for` loop. The most common form of this loop is `for (int i = 0; i < N; i = i + 1) do_something`. The syntax can be described as `for (initialization; continuation_condition; increment) do_something`

Typically, in the initialisation part, we initialise a loop counter. The continuation condition is usually a boolean expression involving the counter, and the increment part should change the loop counter in some way. The loop is conceptually interpreted in this way:

1. Perform initialisation
2. Evaluate continuation condition. If false end loop.
3. Execute loop body.
4. Execute increment part.
5. Go back to step 2.

In its most commonly found form, e.g., `for (int i=0; i<10; i = i+1) ...` the loop body runs exactly 10 times, and the loop counter takes values in the *half open range* $[0, 10)$. This means that the lower limit is included in the iterations, but the upper end is not. So, the loop variable takes the values 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. `i = i+1` is almost always shortened to `++i` or `i++`. The pre-increment form `++i` should be preferred. For integer counters there is no difference in practice, but the post increment form in this context is actually sloppy in stating our intent. It is better to develop a habit from the beginning, of using the pre-increment operator in the `for` loops as a default.

Because of the simplicity of this loop, it tends to be somewhat overused by inexperienced programmers. I deliberately delayed its introduction to avoid giving it too central a position in your mind. We will see that the `algorithm` header in the standard library gives us extremely powerful tools to perform a lot of very common tasks. Whenever possible, one should prefer an algorithms based solution. In case no suitable algorithm exists, we have low level tools like the basic loop structures presented here. Let's close this section with an example where we calculate the total gravitational potential energy between $N$ objects.

## Exercise 13:

Write a program that calculates $x^n$ where $n$ is an integer, using the loop and branching structures you have learned so far. The program could ask for inputs, and read them one by one using `std::cout` and `std::cin`, or it could take the inputs as two arguments passed on the command line. Intended behaviour:

```
yourprogram 3.0 4
81
yourprogram 3.0 -2
0.111111
```

**Hint:** To convert a character string `s` into a double, use `std::stod(s)`, to convert it to an integer, use `std::stoi(s)`

**Exercise 14:**

The program `examples/G_potential.cc` contains a rudimentary function to calculate the total gravitational potential in a system of $N$ objects. Two `for` loops are required, they are not written. Fill in the missing loops and finish the program.

Use the following contracted forms of algebraic operations:

- `X += y` means `X = X + y` , `X -= y` means `X = X -y` and so on for `*` , `/` ..., and the logical operators `&&` , `||` etc.
- `++i` for an integer variable `i` means increase `i` by 1 and then use the incremented value.

## 2.4 Lambda expressions

We have briefly encountered the basic syntax of a function in C++. Functions in C++ can not be defined inside a block, e.g., within the body of another function. Yet, we often require entities which behave a bit like functions, but which can be defined inside a block and interact with existing block scope variables. We will discuss these so called lambda expressions in detail, after we have built up the necessary background. Since they share a lot of characteristics of functions, it would be useful to be introduced to the concept of lambda expressions, along with the building blocks of the language. Let's motivate them using an example.

Let there be a sequence of numbers $L = \{l_0, l_1, l_2 ... l_{n-1}\}$. Suppose we want to create another sequence where each element is obtained by multiplying the corresponding element of $L$ by itself. We could think of it as going over the sequence and applying some operation on each element and creating a new sequence out of the results. What kind of an action is it? In this case, the action is some kind of entity which when applied on a number produces its square. The mapping $x \mapsto x^2$ expresses our meaning. The mapping can be applied to a concrete number to produce a concrete result: $(x \mapsto x^2)(3.0) = 9$. Our operation to create the new sequence out of $L$ is then $L2 = \text{apply\_to\_all}(L, x \mapsto x^2)$. Perhaps we want to filter out all elements of $L$ which are less than 3.0. This operation can be regarded abstractly as $L3 = \text{select}(L, x \mapsto (x < 3.0))$.

Clearly the operations we want to perform on the individual elements can always be implemented as functions. But, functions have names, and often our needs are so local that it becomes unnecessarily cumbersome to dedicate mental resources to finding a suitable name for an action we may only need at one place in the code. What we need is a way to create anonymous function like objects, which can be used like functions. This kind of entities are called lambda functions in C++. Lambda functions are created using lambda expressions, which look like this:

```
[](function parameters) -> return_type { function_body }
```

There is a lot more to them than what we have in the above line, but this should get us started. You will see that it is very similar to the function syntax I have been using since the beginning. We don't have the initial `auto` and we don't have a name. Instead we have a peculiar pair of square brackets at the beginning. But otherwise, they are written just like functions.

Our mapping $x \mapsto x^2$ translates to `[](double x) -> double { return x * x; }`. As with all C++ functions using the trailing return type syntax, the return type specification can be skipped if it can be unambiguously inferred from the function body. So, we could write $x \mapsto x^2$ as simply `[](double x) { return x * x; }`. All functions are mappings from the input parameters to the output types. In C++, the "result" of a mapping encoded in a function is its return value. The job of a function is to perform all necessary calculations so that the input parameters are mapped to the output in such a way that the intended mapping is realised for all valid inputs. The mapping $x \mapsto (x < 3.0)$ translates to the lambda `[](double x) { return x < 3.0; }`. A lambda for an identity mapping, $x \mapsto x$ would be `[](auto x) { return x;}`, where I have used the `auto` keyword to make it a generic lambda. Generic lambdas can be applied on any type of input, and the actual type of the input parameters is inferred based on the types of the values it operates on. A lambda can represent any mapping, not necessarily something to do with numbers! For instance, we could imagine,

$IntegerAsText \mapsto Integer$, to map a given string representation of an integer into an actual number in a program. In C++, that will be, `[](std::string text) { return std::stoi(text); }`.

## Exercise 15:

The program `examples/lambda0.cc` illustrates the use of lambda functions to perform the transformation and filter operation described above. Here is the code, apart from the includes and some output lines.

```cpp
// examples/lambda0.cc

auto main(int argc, char *argv[]) -> int
{
    std::vector v { 9., 2., 3., 8., 2., 1., 0., 5., 7., 6., 1., 2. };

    decltype(v) w{};
    // Create w with the type of v, but initialize as an empty container.
    namespace sr = std::ranges;

    sr::transform(v, std::back_inserter(w), [](auto x) { return x * x; });

    decltype(v) x;
    sr::copy_if(v, std::back_inserter(x), [](auto x) { return x < 3.0; });
    //...
}

```

Identify the lambda functions!

Functions like `std::ranges::transform`, `std::ranges::copy_if` are “higher order functions”. They expect, among other arguments, an operation that they need to perform as an argument. This can be given as a named function, but it is most often more convenient to use a lambda. `copy_if` copies elements in an input range into an output location if the element satisfies a condition. It is the C++ version of a "filter". The condition is given above as a lambda function. Similarly `transform` applies an operation to an input sequence and writes the output into a supplied output location.

For the output location in the example above, we have used `std::back_inserter(container)`. Details of how it works will be explained later, but for now, just remember that if you try to write something there, it appends that something to the end of the container it is attached to.

Lambda functions together with algorithms defined in the `algorithm` header give us a set of very powerful tools to do lots of interesting operations on sequences. Let’s end with an exercise.

## Exercise 16:

The following program creates an array of complex numbers. We want to sort them. The problem is, complex numbers don’t have an obvious way to compare and say what comes after what. For instance, is $1 + 2i$ bigger or smaller than $2 + 1i$? Sorting requires being able to tell what goes before what. To sort a set of entities, we need an operation comparing two of these entities, $x$ and $y$ and telling us whether the first should be considered less than the second for the purpose of sorting. Something like $x, y \mapsto true/false$. The standard library `sort` functions expect such a sorting criterion to be provided. We are using the function `std::ranges::sort` for this example. Fill in a suitable lambda function in the indicated area to sort by the real parts of the complex numbers, and then by the absolute values.

- Hint: Given a complex number `z`, its real part is obtained by `std::real(z)`.
- Hint: Given a complex number `z`, its absolute value is obtained by `std::abs(z)`.

```
// examples/lambda1.cc
#include <iostream>
#include <string>
#include <algorithm>
#include <ranges>
#include <complex>
#include <vector>

auto main(int argc, char *argv[]) -> int
{
    namespace sr = std::ranges;
    using namespace std::complex_literals;
    std::vector nums{ 1.+2.i, 0.11+3.229i, 3.1+0.5i, 2.001+1.5i };

    sr::sort(nums, _________ ); // Fill in an appropriate lambda expression!

    std::cout << "Sorted list of complex numbers, using the given "
              << "comparison operation...\n";
    for (auto num : nums) std::cout << num << "\n";
}
```

## 2.5 Constants: various degrees of const-ness

Certain quantities involved in a calculation should have unchangeable values in order to serve their intended roles. Imagine that you are in a self driving car, driving along on the autobahn. Your life might depend on the program operating the car being able to keep certain variables constant, for instance, a variable storing the acceleration due to gravity on the surface of earth. Accidental assignments such as $g = 100000.0$ can lead to actual accidents in the real, physical world. Similarly, being able to change the value of constants such as $\pi$ or the speed of light does not enhance their usability. We need some way to create objects whose values, in addition to their types, remain constant in our programs.

In C++, when declaring a variable, we can "qualify" it to be a constant by attaching the `const` qualifier to the declaration. E.g.,

```
const auto pi { std::acos(-1) };
// alternatively,
const double pi { std::acos(-1) };
```

The variable thus created becomes a *constant* object in the program. Constant variables can be given an initial value when they are created, but that value can not be changed thereafter.

```
 const auto pi { std::acos(-1) };
 for (auto i = 0UL; i < npoints; ++pi) { // Compiler error!
 // Error, because we tried to change a constant object pi
 // after it was initialised.
 }
```

For those of you who have previously programmed in C, this is not just another "spelling" for the C macro definitions:

```
#define pi 3.14159
```

Macros are text substitution recipes, and are handled by the preprocessor, before the compiler ever examines the code. The macro definition above would replace every occurrence of the name pi with the literal 3.14159 in the code. If we wrote it as

```
/* macropi.c: Don't do this in C++! */
#define pi calc_pi()

```

```
int main()
{
    int i, j, k;
    i = pi;
    j = pi;
    k = pi;
    return i;
}
```

every occurrence of pi will be replaced by a call to `calc_pi()`. You can verify that by running the pre-processor command directly yourself, instead of letting the compiler do that behind the scenes:

```
cpp macropi.c
```

`cpp` in the above command stands for "C pre-processor" and not "C plus plus". It will print out the source code after the pre-processor has done its substitutions. You will see that a separate call to `calc_pi()` is inserted for the assignment to `i`, `j`, and `k`. Technically, that function might be asking the user politely to enter the value of `pi`, reading the user's response from the standard input, and returning that value. In the text substitution approach, every occurrence of the "constant" `pi` defined in this way will be replaced by this call, i.e., possibly asking for user input.

The C++ `const` qualifier is something quite different. If you write

```
const auto pi = calc_pi();
//...
auto i = pi;
auto j = pi;
auto k = pi;
```

a real variable `pi` is created, and is initialised with the return value of `calc_pi()` in line 1 above, possibly also asking the user politely to enter the value of `pi`. In lines 3, 4 and 5, the previously created variable is used, without a new call to the initialising function `calc_pi()`. `pi` has a value and a type, like any other variable, so that it can be used to create new variables or be examined by debugging tools. Note that `const` does not mean "known at compile time" or "substituted by the compiler". The value can be set by the compiler, if it is known at compilation time. But that's not what the `const` qualifier says. The only difference from the a variable declaration without that `const` qualifier is that the compiler won't accept code that changes the value after initialisation. `const` is a promise by the programmer: "I solemnly swear that I shall not change the value of this here variable, from the point of its creation with an initial value till the end of its life". Compiler holds you to this promise. If you (accidentally) write code that tries to change that value, the compiler complains loudly.

For anyone who at least occasionally enjoyed sporting activities, it is helpful to think of programs with many variables as games. The variables are the players, and they have "states", i.e., the values they currently hold. They can interact with other variables in pre-defined ways and change their states. To reason about the program, you have to check what states different variables have as the program proceeds. The state a variable ends up in, depends not just on a single expression attached to it, but on the states of all other variables involved in that expression. Just like the state of a goalkeeper (horizontal / vertical / earthbound / ...) at any given point in time depends on the state of the opposition strikers. We are used to reasoning about changing states in complex systems with multiple interacting entities. We even enjoy it: sports exists! The only difference is that source code, in contrast to a football field, appears static. In languages like C++, a sequence of changes to the state of variables stored in memory produces a result we seek. It is however useful to have a few entities who, like line umpires in tennis, stay fixed, so that the rest of the game is easier to follow. "Programs in which every variable can influence everything going on everywhere" is a pernicious parody of how imperative programming languages such as C, C++, Fortran, Python, Java etc. work. Every programming language offers tools to manage the factors influencing the states of variables in any segment of code. We have so far seen two of the main tools in C++: variable scopes which controls what is visible where, and now, the `const` promise[3].

[3] We are discussing mutability of the values of variables here, not their types. If names were allowed to change type, it would be a game where, e.g., depending on what happens, the chair umpire (tennis) might jump in to return a serve.

For every variable we create, we should ask whether it is meant to carry a state that changes in a manner that is helpful towards achieving our goals, or to represent a fixed entity we need during our computations. Having fewer changeable variables often makes it easier to predict their behaviour, and write correct code. Beyond a certain point though, it is a matter of diminishing returns to try to reduce mutable variables in C++ code. We can have both kinds. We don't need to reach zero mutable variables. It is a good idea to default to make your variables `const`. When `const` is not an option because of the role of the variable, it can be declared without the `const`. For instance, the loop counter is useless as a `const`. But, `pi` or `speed_of_light` should usually be programmed as `const`. Some in the C++ community argue that it would have been better to make all variables automatically `const` qualified, and only make them "mutable" when we need them using an expressive keyword. But not everybody agrees, and that is not the language we have. For now, a useful habit is to always try to make variables `const` first, and only relax this requirement when there is a good reason.

There are some other keywords related to `const`: `constexpr`, `constinit` and `consteval`. We will learn to use them in some detail in a later chapter. The only other kind of `const` like keyword you might want to use at this stage is `constexpr`. `constexpr` declaration specifier does indeed indicate that the variable should be initialised during compilation. Variables storing literal values like $\pi$ are excellent candidates. This is a different kind of guarantee we give to the compiler than the promise "I shall not change this hereafter". A `const` variable may be initialised with a value that the compiler could not possibly know about (e.g., from the return value of a function which asks user input and returns it). If a variable is declared with `constexpr`, that sort of initialisation is not possible. The compiler will try to evaluate its fixed value. Since the compiler knows the value of a `constexpr` variable, it may simplify expressions in your program by already using that value during compilation.

`constexpr` can also be attached to a function, e.g.,

```cpp
constexpr auto sqr(double x) -> double
{
    return x * x;
}
constexpr auto pi { 3.1415926 };
constexpr auto pi2 { sqr(pi) };
```

A function declared as `constexpr` is available for use to initialise `constexpr` values at compile time, like the function `sqr` above. While a great many functions in the C++ standard library are `constexpr` functions in C++20, the standard inverse cosine function `std::acos`, and other similar mathematical functions declared in the `<cmath>` header, are not `constexpr` functions. We can not therefore use them to initialise `pi` if we declare `pi` as `constexpr`.

A `constexpr` function can call other `constexpr` functions in its body, in addition to declaring local variables, using loop and branch constructs etc. With C++20, almost everything one can do in a normal function can be done in a `constexpr` function. If the arguments to the function are compile time constants, the function is evaluated and its result is made available for `constexpr` variable initialisation. The body of such a function therefore can not rely on another function that can not be evaluated at compilation time. If the arguments to a `constexpr` function are not constants known to the compiler, the function can not be evaluated by the compiler, and we can not use it for `constexpr` variable initialisation. There is no problem using a `constexpr` function normally with non-constant variables, as it remains available for run-time use. We just can't use it for initialisation in a `constexpr` variable declaration in that situation.

The `consteval` keyword can be applied to functions to make them into "immediate functions". These functions go one step further than `constexpr` functions regarding their availability at compilation time: they can only be used with arguments known at compilation time. If we want to initialise a variable, and want to guarantee that the variable is initalised during the compilation process, we could make the initialisation function a `consteval` function.

Explicit specification of different degrees of constantness of our objects also contributes to the perceived complexity of C++ programs. However, not being able to protect variables such as $\pi$ or $c$ against unintentional modifications is an unacceptable compromise.

```python
# python example to illustrate the need of constant
# qualifiers

import numpy as np
def circle_area(r):
    return np.pi * r * r

# later
def func1():
    np.pi = 0.

# still later, after calling func1...
surf_to_cover = circle_area(10.0) # returns 0
```

When anything of importance needs to be calculated, is the price of having to type or read a few extra keywords (describing immutability) really too much? Remember also that in C++, the variables qualified as `const` or `constexpr` do not have to be trivial entities like a number or a character array. One can create a hash-table, a red-black tree or just about any data structure, and make it `const` within a program. One can pass an otherwise mutable object as a constant reference to a function, so that it has to be treated as an immutable object within that function. With C++20, one could make compile time constant objects of the `std::string` or `std::vector` types. Constants simplify programs by reducing its total number of possible states. Please do not ignore or avoid these immutability regulators, but rather learn to use them to write programs whose validity is easier to deduce.

## 2.6 Pointers, low level arrays and references

Up until this point, we have discussed topics which are safe and elegant. We will now take up one area of C++ which is often regarded as dangerous or difficult. Pointers. As a concept they are not difficult to grasp, and in my opinion, you can not really be a C++ programmer without understanding pointers and references. They underpin a lot of the elegant abstractions we are able to use, like `std::vector`. Someone had to program with pointers so that we don't need to do the same all the time. Knowing how such things work is an integral part of an introductory C++ course. Learning about pointers will also help you rationally decide when not to use them and when doing so may in fact be the best solution.

### 2.6.1 Pointers

Remember that we defined an object as "a concrete instance of some bytes, somewhere in the memory of a computer, holding a value of a certain type". An object therefore lives somewhere in the memory of the computer. As an abstract model, one can think of the memory to consist of numbered locations, like houses in a long street. Imagine that the numbering is done in metres, and the house number for any given house is the distance in metres between the leading edge of the house and the beginning of the street. If I know the house number for a house, I can find it quickly, observe its properties, and depending on circumstances, modify its properties. Since a variable lives somewhere in the memory, it has an address. Pointers are a general type of types which store addresses of objects.

Given a variable `var` of type `Type`, its address can be retrieved as `std::addressof(var)` or `&var`. This address is of the type `Type*`.

```cpp
int i{ 9 };
int* ip{ &i };
```

`int*` above is a "pointer to an `int`". Similarly there are pointers to `double`, `float` ... for any type for which there are objects. Pointers to different types are different from each other: e.g., `int*` and `double*` are different types, even if they are both locations in the memory. A pointer holding the address of an object, like how `ip` is holding the address of `i` above, is said to be pointing at that object. If a pointer is pointing at an object, the object can be accessed through the pointer. In the above, `*ip` would be the same object as the one described by the variable `i`.

```cpp
int i{ 9 };
int* ip{ &i };

std::cout << *ip << "\n"; // prints 9.
*ip = 4; // Write access to the pointed object
std::cout << i << "\n"; // prints 4.
```

For a pointer `ip`, the expression `*ip` is called dereferencing the pointer. `*ip` is a way to access the same object as `i`. We see above, that it is possible to change the value of `i` indirectly, by assigning to the dereferenced pointer. One interesting question is what happens to our promises regarding `const`. If `i` is a `const` object, we should not be able to change its value after we initialize it. Can one sneakily change the `const` object using a pointer? It turns out that it is not easy to do.

```cpp
const double pi{3.14};
double* ptr { & pi }; // Error!
const double* cptr {& pi }; //OK
*cptr = 4; // Error!
```

The address of a `const double` is a `const double*`. So, we can not store it in a `double*` variable. Pointers to objects of different type are different! We can store the address of a `const double` object in a `const double*` variable. But then, dereferencing a `const double*` object gives us a `const double`, which we can not overwrite! Allowing the creation of a `double*` from a `const double*` address would let us break our promise. It is therefore not allowed. If we tried to create a `const double*` from the address of a non-`const` object, it is harmless. Because dereferencing the pointer would give us a `const` object which we can not change. The pointer is more restrictive than the original address. It is therefore allowed.

"Dereferencing" with a prefix `*` operator should feel somewhat familiar to those who have programmed in C. Pointers were present in C, and C++ inherited this aspect from C when it started. The dereferencing syntax remains the same between the two languages. This syntax has also been co-opted for other types of indirect access to an object. We saw one example with the `std::optional` in the quadratic equations example in section 2.3.1. The "iterators" used in the C++ standard template library are also deliberately designed so that accessing elements of a container through an iterator looks like indirect access through a pointer.

A pointer pointing at one object can be reassigned so as to point at another object.

```cpp
1  const double d{8.}, e{3.};
2  const double * p { &d };
3  std::cout << *p << "\n"; // prints 8.
4  p = &e;
5  std::cout << *p << "\n"; // prints 3.
```

Above, we reassign a pointer declared as `const double *`. This exposes a sublety you should be aware of. The pointer variable itself is an object. Its value is the memory address it is pointing to. Had the pointer been a `const` object, we would not have been able to change its value and make it point to a different location as we do in line 4 above. But what we have here is an ordinary non-`const` pointer to a `const double`. The objects it can point to are `const`. But it can point to any `const double` variable in the program. What if we wanted a pointer that points to a fixed location, i.e., a pointer which itself is a `const` object. We will have to put the `const` qualifier to the right of the `*` to indicate that intent. A good way to remember this is "`const` binds towards its left". The `const` keyword sticks to the entity to its left. If there is nothing there, like in `const double*` it takes the object to its right. Some developers, therefore, prefer always writing `const` to the right of the thing they want to keep constant, e.g., `double const x` instead of `const double x`, even for ordinary variables. In this, "east-`const`" style, the placement of `const` is more consistent whether or not pointers are involved. Native speakers of languages where adjectives normally come after nouns find it natural, while others, like me, think "a constant number" rather than "un numero costante", and may have contributed to keeping this unnecessary syntactic diversity in the language. We therefore have an entire zoo of possible combinations of `const` placement for a pointer to a double, which are listed below along with their meanings...

```cpp
double X{1.0}, X2{11.0}; // plain, mutable doubles
const double Y{2.0}, Y2{22.0}; // constant doubles.
double const Z{3.0}, Z2{33.0}; // Exactly the same as the line above.

double * p{&X}; // a non-constant pointer to a non-constant object

double const * q2{&Y}; // a non-constant pointer to constant object
const double * q{&Y}; // Same as above

double * const r{&X}; // a constant pointer to a specific non-const object

const double * const s{&Y}; // a constant pointer to a specific constant object
double const * const t{&Y}; // Same as above
```

Test yourself: which of the pointers above can be reassigned to point to `X2`? Which of them can be reassigned to `Y2`? Which pointers can you use to change the value of X or Y?

It is possible to create a pointer of a given type, say, `double*`, but not initialize it to the address of any actual object. One valid way of doing that is to initialise it to `nullptr`. Such a pointer evaluated as a boolean expression will evaluate to `false`, and this fact can be used to know that the pointer is in fact not pointing at anything useful. Dereferencing a `nullptr`, or a pointer initialised to `nullptr` is undefined behaviour. This means anything may happen beyond that point in your code. If you are lucky, the program will merely crash.

Fortunately, one can easily check that we are not dereferencing `nullptr`:

```cpp
auto somefunc(double* inp) -> double
{
    if (inp != nullptr) {
        do_something_with(*inp);
    }
}
```

If a pointer does not hold the address of a valid object, but is also not `nullptr`, we would not be able to check the validity of its value as above. Really bad things can happen when we dereference such a pointer to read or modify the "pointed to" object. It could after all, be the address of another valid object in your program, or be a memory location outside the allowed memory for your program. In this course, we have, from the beginning always initialised variables when we declare them. That habit is always important, and you should most certainly continue doing that for pointers. There is another pattern you need to recognise to work safely with pointers.

```cpp
1  double * dp{nullptr};
2  if (something) {
3      double sum{ 0. };
4      dp = &sum;
5      // work
6  }
7  // more work
8  if (dp != nullptr)
9      std::cout << "sum = " << *dp << "\n";
```

Here, we initialised the pointer `dp` to `nullptr`, and then later assigned it to the address of a valid object, `sum`. Yet, the last line in the above code invokes undefined behaviour. The reason is that the scope of the variable `sum` ends at line 6. The pointer `dp` though, still contains the address of what used to be `sum`, because we didn't assign a new value to `dp`. As soon as the scope of `sum` ends, that address can be reused for something else. Therefore, our pointer `dp` is pointing to invalid memory. There used to be a house at that address, in which there lived a person you once knew, but there is a railway track running through that area now. Operations which were sensible at that address may not be sensible or safe any more. This kind of a pointer, which once pointed to a valid object, but continues to hold the same address after the object itself has expired, is called a "dangling pointer". Such situations have to be avoided with careful programming. If we are storing the address of any object in a pointer, we have to be careful when dereferencing the pointer that the object itself is still in scope. For instance, you should never return the address of a local temporary variable from a function.

## 2.6.2 Raw arrays and pointer arithmetic

Just like we can create one variable of a certain type using the syntax `TypeName variablename`, we can create multiple instances in a raw array, e.g., `TypeName v[52]`. In this case, an array of 52 objects of type `TypeName` will be created. The first object can be accessed as `v[0]`, the second `v[1]` and so on. This is like a line of houses of identical design in our fictitious street. The address of the first house is, `&v[0]`, address of the second is `&v[1]` and so on. In an array, the subsequent elements are right next to each other in memory, with no gaps. Since the addresses here are sequentially numbered locations in memory, the difference of the address values `&v[1]` and `&v[2]`, measured in bytes, must be the size of one element of the array. For instance, if `double v[10]`, the separation of the addresses of two consecutive elements of the array `v` is 8 bytes. This helps understand what is referred to as "pointer arithmetic".

Given `TypeName* ptr`, and `int dist`,

- Two pointers to the same type are equal if they point to the same object.
- In the expression `ptr + dist`, the value of the pointer in bytes jumps a total of `dist` units of `sizeof(TypeName)`. If it was `BBB` before, it will be `BBB + dist * sizeof(TypeName)` afterwards. For a pointer to `double`, it means that adding 1 will shift its value by 8 bytes. Because of this property, if we have a pointer to one element of the array, we can make the pointer point to the next element by adding 1 to it.
- Difference `p1 - p2` of two pointers `p1` and `p2` is a value of a signed integral type (typically `long` these days) `diff`, such that `p2 + diff == p1`.
- Addition, multiplication or any other type of arithmetic operations with two pointers are meaningless.
- Only allowed operations involving a pointer and an integral number are addition and subtraction, with the effect being a shift in the address stored.
- Pointer arithmetic for pointers not pointing at a valid array is meaningless.

The rules of pointer arithmetic are made to work with arrays. If `ptr` points to one element, `ptr + 1` points to the next, `ptr + 2` points to the one following and so on. Therefore, `*(ptr + 1)` is the object at the next array location, i.e., the next object. In fact, the next array element can also be retrieved as `ptr[1]`.

```cpp
int v[5]{10, 11, 12, 13, 14};
int *ptr = &v[0];
std::cout << *ptr << "\n"; // 10
++ptr;
std::cout << *ptr << "\n"; // 11
std::cout << ptr[1] << "\n"; // 12. Same as v[2]
std::cout << ptr[2] << "\n"; // 13
```

As seen above, `ptr`, which points to a position somewhere in the middle of the array `v`, can itself be used as an array. `ptr[i]` is just another notation for `*(ptr+i)`. If `ptr` is set to point to the first element of the array, `ptr[0]`, `ptr[1]` ... would be identical to `v[0]`, `v[1]`.... as if `ptr`, the pointer was another name for `v`, the array. To make this symmetry more complete, name of an array (raw, low level built-in array), e.g., `v` can be thought of as a pointer to its first element, i.e., `*v` is the same as `v[0]`.

Despite the array like syntax that pointers also have, it is useful to think of arrays and pointers as different kinds of things. An array is intended to house multiple elements of the same type. It is a location in memory with multiple objects stored consecutively. A pointer could point into an array, but it could also just point to one isolated object.

A peculiar thing happens when we pass arrays as function arguments. Consider this program:

```cpp
 1  // examples/array_or_pointer.cc
 2 #include <iostream>
 3
 4 void f(double arr[4])
 5 {
 6     std::cout << "size of function argument in bytes = " << sizeof(decltype(arr)) << "\n";
 7 }
 8
 9 auto main() -> int
10 {
11     double v[4]{3, 2, 3, 4};
12     std::cout << "Elements of raw array...\n";
13     for (auto a : v) std::cout << a << "\n";
14
15     std::cout << "size of raw array in bytes = " << sizeof(decltype(v)) << "\n";
16     f(v);
17 }
```

We create an array of size 4 in the `main` function. We can even loop over this array using our range based `for` loops. Clearly it has 4 elements. When we print its size in memory, using the `sizeof` operator in line 15, we get 32 bytes, as expected. We try desperately to indicate that our function `f` expects an array of 4 doubles. Yet, if you compile and run this program, you will see that the size of the function argument will be printed as 8 bytes. When a raw array is passed as an argument to a function, it loses any identity as an array. The function thinks of it as a pointer. The array, kind of, “decays” into a pointer. The same loop we use above, in the `main` function does not work if we try to iterate over the input parameter for the function. This pointer that the function receives, refers to the same memory location, and therefore does indeed point to the beginning of our array. If we tell the function the extent of the array, it can process it correctly.

```cpp
void f(double arr[], unsigned length)
{
    for (unsigned i=0UL; i < length; ++i) {
        std::cout << arr[i] << "\n";
    }
}
```

Raw arrays are not aware of their own size. In the context that they are declared, the compiler might be able to see the extent, and use that information, for instance in range based for loops. But when passed to other functions, that information must be passed along separately. C++ containers which are similar to arrays, like `std::vector`, `std::array` etc. are aware of their extent. It is therefore possible to pass them to functions alone, without an accompanying length argument. `std::vector`, `std::array` etc. are real containers, meaning they store and manage the lifetime of the contained elements. There is another C++ way of writing the function `f` above, which can be used as an adaptor around functions in C libraries: using `std::span`. `std::span` is a “view” of an array-like container. It does not own any of its elements, and does not manage their lifetimes. It just provides a C++ style interface around a bare pointer and an extent at no extra cost compared to passing those two parameters to the function.

In the above example, we use `double arr[]` in the function signature instead of `double arr[4]`. This indicates that `arr` is an array of unknown length (as far as the function is concerned). Writing `arr[4]` there is misleading, as it indicates that somehow `arr` contains some information about its size, which it can’t. If a pointer is meant to hold an array, we can declare it with the `[]` syntax as above. If we initialize it with an explicit list of values, the size is determined by the compiler.

```
double arr[]{5, 4, 3, 2, 1}; // arr is double[5]
```

Irrespective of whether we state the size ourselves or the compiler counts the number of items in the initializer list, the size of a built-in array must be a compile time constant. In standard C++, it is not possible to do this:

```
void f(unsigned N)
{
    double v[N]; // Not allowed.
    for (unsigned i = 0U; i < N; ++i) v[i] = 0;
    // ...
}
```

The C99 standard of the C language allows such variable length arrays. But it was once proposed and rejected as a feature of C++. Of course, C++ has multiple ways of creating variable length array like objects such as `std::vector`, `std::valarray` ..., so that feature is not really necessary.

For pointers, like for integers, operators `++` and `--` are defined to add or subtract 1 in the pointer arithmetic sense. Given `Type* p=&v[0]`, `++p` will change the value of `p` so that it points to `v[1]`, and then `--p` will bring it back to `v[0]`. This allows code like this:

```
void mycopy_n(unsigned howmany, int* source, int* destination)
{
    for (unsigned i=0U; i<howmany; ++i) {
        destination[i] = source[i];
    }
}
void mycopy(int* start, int* end,
            int* destination)
{
    while (start != end) *destination++ = *start++;
}
```

The first of these is fairly straight forward. Presumably the `source` and `destination` pointers point to some location inside an array. Our code simply copies a certain number of elements from the source to the destination using a `for` loop. It is entirely up to the user of the function to make sure that the argument `howmany` does not lie about the true extent of the arrays `source` and `destination`.

To understand the second function, we need to know a little bit more about the use of `++` to perform increments. There are two ways of using it to increment an integer or a pointer. `++var` and `var++`. At the end of each of these, the variable `var` would have been incremented by 1. But, this is a "side-effect" of that expression. Just like expressions like `var + 5`, `var * var + 2`, the expressions `++var` and `var++` have values as well. For `++var` the value is the incremented result. For `var++` it is the value it had before the increment. The following code listing should help:

```
auto var = 0, foo = 0;
foo = ++var; // var == 1 and foo == 1
foo = var++; // var == 2 and foo == 1
foo = ++var; // var == 3 and foo == 3
foo = var++; // var == 4 and foo == 3
```

The compact line `*destination++ = *start++` therefore says the following:

- Evaluate the values of the left and right hand sides of the assignment expression: LHS = original location pointed at by `destination`. RHS = original location pointed at by `start`.
- Perform the assignment
- Implement the side effects of the sub-expressions, i.e., increment `destination` and `start` to the respective next positions.

### Exercise 17:

The program `examples/copyptr.cc` contains the above code and a `main` program to use these functions. Convince yourself that the program does what we just discussed. What happens if you give an incorrect size to the `mycopy_n` function, like 100? What happens if you create the two arrays with unequal sizes? This program has no more library features than our `hello_world` program, and is therefore a good toy to familiarize yourselves with some kinds of errors.

### Exercise 18:

In the program `examples/copyptr1.cc` we make two tests in two blocks inside `main`. In both, we create two arrays `A` and `B`, initialize them with different values and pass them to two functions, along with an extent argument. In each test function, we equalize the two arrays. In test 1, we do this by calling the `mycopy` function we saw in the previous example. In test 2, we simply write something like `B = A;`. After this, we change one of the elements of the first array, and print the contents of both. Run the program. Do you understand the output? Can you explain the difference of the output coming from the two tests?

### 2.6.3 Built-in arrays and pointers forever?

One of the things you might have noticed in the two exercise programs above is how little of the standard library you had to use. Nothing beyond what you needed for `hello_world`. To some people, this has some kind of value. I don't have to learn about vectors and spans and so on. I don't want to think about all those fancy modern stuff. To me, any sentence that starts with "I don't want to think," rarely leads to anything intelligent. Let's dismiss "I don't want to think" and "I don't want to learn" as reasonable excuses to not use C++ more completely.

I do however sometimes see performance as an excuse to stick with pointer based low level syntax for everything. Since we only use low level constructs, it must be "light weight" and fast, right? There is the expectation that there must be a huge cost for writing elegant code using the standard library or other libraries which leverage recent language changes to bring us more elegant usage, more maintainable code. Done perfectly, it is indeed possible to write extremely fast code with direct pointer manipulation. It is also possible to make hard-to-detect mistakes which might crash the program, run with incorrect results, run slower than they need to, run with *almost always* correct results... Until you become proficient in C++, and have developed several proper projects in this language, you are not likely to beat the automatic optimisations that modern compilers can do.

### Exercise 19:

To help you decide whether to go it alone with bare metal tools like pointers for all tasks or higher level facilities of modern C++, consider the following two functions:

```cpp
#include <span>
#include <ranges>
#include <algorithm>
void mycopy(int* start, int* end, int* destination)
{
    while (start != end) *destination++ = *start++;
}
void mycopy2(std::span<int> source, std::span<int> destination)
{
    std::ranges::copy(source, destination.begin());
}
```

They do the same thing, except we use more human readable syntax in the second instance. `std::span` refers to a sequence of consecutively stored elements in memory, like a section in an array. But it knows its extent. We use a simple "copy source to destination, starting at the beginning of destination" kind of function call to achieve this. Copy the code into the code window of the Compiler explorer. On the right hand side, choose GCC trunk or Clang trunk as the compiler. In the box to the right of the compiler choice, you can enter compiler options. Enter at least `-std=c++20` there. Compiler explorer shows the assembler code that the compiler might generate for your code. Most of the time, less assembler code means a faster program, or a cleaner translation. You will see the assembler version of the two versions of the copy function above. Set the compiler optimisation level to different levels, `-O1`, `-O2`, `-O3`. How do the two versions compare?

The reason why we are learning about pointers is not that we will abandon our `vector`, `array`, `span` etc., but rather that we will understand how they work, so that we don't misuse them. Another reason is to be able to read code written by the previous generations of programmers, so that you can update them to more reliable, more readable, and perhaps faster code.

### 2.6.4 Dynamic memory

We learned before that sizes of C++ (built-in) arrays must be compile time constants. For instance, the following will not work:

```cpp
#include <iostream>
auto main() -> int
{
    std::cout << "Please enter a positive integer: ";
    auto num = 0U;
    std::cin >> num;
    double X[num]; // Not standard C++, as the value of num
                   // is not a compile time constant
    for (unsigned i=0U; i < num; ++i) {
        X[i] = 0.;
    }
    // More work...
}
```

Of course, we often need arrays whose size are dependent on something happening when the program is running, i.e., can not be known at compile time. If your program is meant to make a catalogue of books the user has, you don't know how big the array of book names needs to be when you are compiling the program. And of course, when you need such runtime sized entities, you should use `std::vector`, `std::string` and similar standard library facilities.

```cpp
#include <iostream>
#include <vector>
auto main() -> int
{
    std::cout << "Please enter a positive integer: ";
    auto num = 0U;
    std::cin >> num;
    std::vector<double> X(num, 0.); // OK. Vector was built for this
    // More work...
}
```

But, how does the `std::vector` do it? Just to understand how standard library containers solve these problems and to know their potential performance implications, let's explore the details, by means of a little analogy.

#### 2.6.4.1 Whiteboard stack

Imagine that you have a few white boards and you want to do some calculation. You start by, for example, writing $N = 4$, $c = 1$, $x = 1.008$ etc. and then numerically calculating something using those values. The numbers you need are right in front of you and you are cruising along at a good speed. Then you reach a point where you need the result of a function $f(x)$ to proceed. $f(x)$ is in itself a sophisticated multi-step calculation. So, you mark the point you have already reached in your original calculation, copy the value of $x$ to the next white board, and move over to the next white board to do the steps needed to evaluate $f(x)$. You finish calculating $f(x)$, and copy the result from the second board to the place on the first board where the value of $f(x)$ was required. Now you have finished doing what you wanted to do with the second board, so you wipe that board clean, and resume your work on the first board. Next time you need to calculate some non-trivial function again, you copy the inputs from the first board to the second. Go through the steps required for the other function on the second board. Perhaps those steps will require calculation of yet another complicated function so that you move to a third board. Everytime you finish calculating a function, you copy the result back to the previous board and wipe the board you used to calculate that function. This is roughly how function call hierarchies work. The white boards we are using for calculating a function and storing any intermediate variables form a “stack”. The first thing you can remove from this stack is the last thing you added to it. If you are currently using 3 boards, because you went from your main calculation to calculating $f(x)$ and then from within $f(x)$ to calculating $g(x)$, the first board you will wipe clear is the one for $g(x)$.

The white boards, your stack memory, are frequently reused small areas of storage, so that you have an overview of everything on it and can access everything quickly. But the white board is unsuitable for storing a hundred thousand numbers. Sometimes you need to work with lots of data, which do not fit in your stack. In the library of your institution, they have a humongous white board, which we are going to call the heap. When you need a large amount of space to store the data, you can ask the library heap space manager to mark out a place of the required size on that humongous white board for your use. They take a marker, find an unused area large enough for your requirement, and put a red line around it. They note that you reserved it, and then give you a number uniquely identifying where it is on the heap. A little later you ask them for another chunk, and they give you another number identifying the space they reserved for your second request. You can do calculations with the data stored in these bigger chunks if you are mindful and store the identification numbers they give you with each of your requests. Notice that they are not returning you the actual newly reserved storage area. That would be pointless because the whole reason for asking them was that you could not store all that data on your white board. They are just telling you where it is. You can save the *address* of one such allocated block in a variable, say $P$ on your white board. You can always ask for the $100000th$ number stored at location identified by $P$. Imagine that you finish your function for the current white board and you wipe it clean. You have thereby also wiped clean where you previously wrote something like $P = 82293221$ to note where the library allocated something for you. There is no way for you to see what you stored there. There is no way to write some new data into that area. Worse, in the record of the library heap space manager, that area of the heap is still in use, and they keep guarding it, so that nothing else can overwrite your precious calculations. If you keep doing this, the entire humongous white board at your library will be full of red blocks you reserved, but can’t use any more. Of course, when you leave, they notice that you are not doing anything with any of the blocks you reserved, so the remove all the red lines they drew for you and make it available to other people. But the right thing to do is this: as soon as you are done using a block of memory you reserved, you send them another request saying that they can free up the area you reserved with identifier 82293221. That block is then cleaned up, and made available for future use. Notice that the life time of the entity you reserve started at the time of its allocation on the heap, and ends when you made a deallocation request. The space with your information on the big white board in the library does not automatically get released when you clean up your local white board. Therefore, before your “pointer” to that storage, $P$, expires, you should send a deallocation request with the address it stores, or copy the address of the allocated resource to another pointer so that you can continue using it or free it later.

Now let’s say that again in C++ vernacular. Every time you call a function, the C++ runtime creates a “stack frame”: an area in memory given to you to evaluate that function. Inputs to the function are initialised on the stack frame from their values at the call site. All block scope variables you create inside your function are written in this stack frame. When you finish the function, the result is copied back, the stack frame is erased and the execution continues in the parent stack frame. The same bytes in memory which constituted the stack frame may be used to create a new stack frame to calculate the next function, potentially a completely different calculation. Stack frame is a “call frame” rather than a “function frame”. There is not a different white board for $sin(x)$, $cos(x)$ etc., but rather a few that you use, erase, reuse as you need. Stack frames, being frequently reused bytes in memory, are usually “cached” by the CPU and can be accessed very very quickly. In C++, all your local variables are created on the stack, even if they are non-trivial user defined types. This was always the case in C++. But the stack frames are small, and can not store large amounts of data. There is another part of the memory, which is not used to implement the stack, and it can store as much data as your installed RAM allows. That is called the heap. To request a certain region to be reserved on the heap for use as a single integer, you have to do this: `int* p = new int`. If you need 2000000 doubles, you would do this: `double* p = new double[2000000]`. The `operator new` is your way of requesting that a certain chunk of memory be reserved for your use. The runtime reserves that memory and returns to you the address of the reserved location. If you do not note down this address, like we do in the pointer variables here, there is no way to refer to the reserved locations. The opposite of `new` is called `delete`. To clear the "in-use" status of a block of memory you previously got from `new`, use e.g., `delete p`, if the space is for a single object, and `delete[] p` if it is for an array. Heap allocation/deallocation is used in C++ when we need to work with a large amount of data which does not fit in the stack frame, or when we simply don't know at compile time how many bytes are needed to store a variable. In C++ the compiler makes a precise plan for every stack frame it might need to create, with exactly where on the stack frame each variable is going to be written and how many bytes will be used to store the variables. This reduces the number of operations the runtime has to do to perform its tasks. But, suppose you wanted to store someone's name in a variable. How long should the character array be? 20? 30? 200? If you set it to 30, thinking who has a name that has more than 30 characters, you should ask a friend from Spain or from southern India, whether they think 30 characters are enough to store the name of a person. If you set it to 200, you might be fine, but most often that is a lot of wasted space. That's why, for situations where we can not decide on the exact number of bytes at the compile time, we use blocks of memory allocated on the heap, and work with them using a pointer "handle" representing our connection with that resource.

Using heap allocation, our task from the beginning of this subsection could be written as

```cpp
#include <iostream>
auto main() -> int
{
    std::cout << "Please enter a positive integer: ";
    auto num = 0U;
    std::cin >> num;
    double* X = new double[num]; // no need to be a compile-time constant size
    for (unsigned i=0U; i < num; ++i) {
        X[i] = 0.;
    }
    // More work...
    delete [] X; // Extremely important
}
```

Notice one important contrast to our use of pointers before this subsection. Up to this point, we used to initialise pointers to addresses of named variables, e.g., `int* p = &i`. If somehow the pointer `p` runs out of its scope, we can still reach `i`, read its value, assign to it, make another pointer point to it. The object or array that `new` creates "somewhere" in memory is a nameless entity. We store its address in a pointer. As long as the pointer exists, we can reach that block. If the pointer runs out of scope and is gone, there is no way for us to do anything useful with that block of memory. It is lost to our program. It is leaked. In the above listing, `X` is called a "resource owner". It is our handle to a nameless chunk of memory somewhere. We have to make sure that before the resource owner runs out of its scope, it releases its owned resources, or transfers ownership to another entity. For instance, we could let `X` expire without calling `delete` above, if we created another pointer to store that value before `X` expires.

Notice that in the above example, we do not "declare a variable" on the heap. We have a variable called `X`, which is of a pointer type, which lives on the stack. It holds a value, which happens to be the address of some resource we acquired from the system. That nameless object is not a variable declared in our function. Variables like `X` play by the scoping rules of C++ we learned earlier. The nameless entities we create outside of the stack tree by allocating memory with `new` have different rules about their life time. They come into existence when we call `new`, and their life time only ends either when we call `delete` on the same address, or at the end of the program. Every `new` in code should always be paired with a corresponding `delete` somewhere.

In modern C++, manual memory management with `new` and `delete` is considered bad practice.

It is too easy to forget to match every `new` with a delete. Even when we do write a matching `delete` for all calls to `new`, existence of `exception` in C++ means that the line with the `delete` expression may never be reached. Even when an exception is not thrown, and the `delete` instruction runs as intended, it is too easy to forget that despite `delete` freeing a memory region, it does not change the value contained in *the original pointer* in any way, as the following example illustrates.

```
 1  // examples/asan0.cc
 2  auto main() -> int
 3  {
 4      auto* X = new double[400]; // space for 400 doubles reserved on the heap
 5      // X holds the address of where those 400 doubles are on the heap.
 6      // For instance, X could be the numeric address 337932100.
 7      // work with X, e.g.,
 8      if (X != nullptr) {
 9          for (int i = 0; i < 400; ++i) {
10              X[i] = 0;  // Perfectly valid use
11          }
12      }
13
14      // Done with calculations using X
15      delete [] X; // Previously allocated space for 400 doubles is now released
16      // X is still 337932100, but the previously allocated 400 doubles at
17      // that address on the heap is already gone!
18
19      if (X != nullptr) {
20          for (int i = 0; i < 400; ++i) {
21              X[i] = 1;  // Nasty use after free bug!
22          }
23          return X[33] > X[23];
24      }
25  }
```

In line 15 above, we freed the array we had previously allocated on the heap. But, the pointer `X` still held the value of the previously valid location in the heap. Checking the validity of the pointer as we do in line 19 does not help, because `X` is not `nullptr` as it still holds the address of the previously allocated array, since line 4. It was not a `nullptr` in line 8, and nothing changed it since then. `delete` is not an operation *on* the pointer itself, but a request to clear memory at the location specified by the value held by the pointer. When we try to access and assign to elements of the no-longer-present array in line 21, a lot of bad things can happen. If we are lucky, our program crashes, and we debug and find the mistake. We may not always be lucky. One simple thing one can do to mitigate things is to assign nullptr to the pointer `X` immediately after calling `delete`. The memory location it was pointing to would be invalid after the delete, so, we might as well zero it out. This restores correct behaviour in many instances. But, now we have remember not only to pair each `new` with a `delete`, but also to zero out the pointer after calling `delete`. Chances of making this kind of mistakes are not small.

The above code, when compiled and run, does not immediately crash or give any errors, as shown below in the first two lines. This would then hide a possibly serious bug, which can hurt our code base later at some unfortunate time. Tools like address and memory sanitizers are designed to detect bugs like the above. To diagnose possible memory bugs like this, one could compile with address sanitizer.

```
examples> g++ -std=c++20 -g asan0.cc -o nosanitizer
examples> ./nosanitizer
examples> g++ -std=c++20 -g -fsanitize=address asan0.cc -o with_sanitizer
examples> ./with_sanitizer
=================================================================
==15277==ERROR: AddressSanitizer: heap-use-after-free on address 0x61f000000080 at pc 0x0000
WRITE of size 8 at 0x61f000000080 thread T0
    #0 0x4008df in main /home/sandipan/Work/courses/2021/examples/ asan0.cc:21
    #1 0x7fea46bb1349 in __libc_start_main (/lib64/libc.so.6+0x24349)
    #2 0x400769 in _start (/home/sandipan/Work/courses/2021/examples/a.out+0x400769)

0x61f000000080 is located 0 bytes inside of 3200-byte region [0x61f000000080,0x61f000000d00)
freed by thread T0 here:
    #0 0x7fea4795d117 in operator delete[](void*) /home/sandipan/src/gcc/libsanitizer/asan/asa
    #1 0x400897 in main /home/sandipan/Work/courses/2021/examples/ asan0.cc:15
    #2 0x7fea46bb1349 in __libc_start_main (/lib64/libc.so.6+0x24349)

previously allocated by thread T0 here:
    #0 0x7fea4795c757 in operator new[](unsigned long) /home/sandipan/src/gcc/libsanitizer/asa
    #1 0x400827 in main /home/sandipan/Work/courses/2021/examples/asan0.cc:4
    #2 0x7fea46bb1349 in __libc_start_main (/lib64/libc.so.6+0x24349)

SUMMARY: AddressSanitizer: heap-use-after-free /home/sandipan/Work/courses/2021/examples/asan0
<<<<< Lots of more output >>>>>>
==15277==ABORTING
```

Notice how the run with the address sanitizer not only recognised that we were using the heap region after freeing it, it told us exactly `where we allocated that heap region` , `where we freed that heap region` , and `where we tried to illegally access it after releasing it` .

In C++ since C++11, there is really no reason to do any explicit memory management with `new` and `delete` any more. The detailed explanation given above was to give you some insight into what is happening behind the scenes. A vast majority of situations where you might need to work with dynamic memory, you should prefer standard library containers and smart pointers. When implementing a library which has to deal with dynamic memory, the orchestration of allocation and deallocation is usually handled in C++ by writing resource handler classes. Remember the special property of blocks of code in C++? If a variable is declared in a block, it expires at the end of the block. In C++, when an object of a class type expires, there is a special "clean up" function associated with that class, which runs *automatically*. These functions are called "destructors". Imagine that our pointer belonged to one object of such a user defined type. When it is about to expire, the destructor will be compulsorily called. If we write a destructor which releases any heap resources held by the pointer, it would be impossible to "forget to deallocate", even if an exception is thrown between the lines containing the `new` and the `delete` expressions. That's what classes like `std::vector` , `std::unique_ptr` etc. do. They wrap a resource owning pointer in an object of another type. They manage allocation of resources in special "constructor" functions and clean everything up in their destructor functions. Since it is impossible to forget this clean up function, it is necessarily called when a variable expires, we can match all `new` instances with a corresponding `delete` without ever failing. In our chapter on classes, you will learn to do this kind of resource management.

Continue using `std::vector` , `std::string` etc. to manage all the heap allocations for you. Need to store the contents of a file as a character sequence of unknown (at compile time) size? Use a `std::string` . Need to work with a very large array of numbers? Use a `std::vector` . Nowadays, a bare `new` or `delete` call in user code is seen as likely problematic code. Unless you are writing library code, for instance, to provide a type like the `std::vector` there are no cases where you need to write `new` or `delete` in your code.

Notice that the recommendation does not say "avoid pointers". That advice is about as sane as "avoid house numbers". The guideline regarding `new` and `delete` is only about doing heap resource allocation and deallocation directly. It is about *owning* pointers. The pointers holding return values from `new` are sole handles to those resources, i.e., they "own" those resources. It is best to leave these owning pointers out of your code, and use library facilities as mentioned above. They are very well tested solutions following all safe practices, most often (almost always) with negligible overhead.

Let's close this subsection with one valid use case for "non-owning" pointers. The very "C-style" function `axpy` below is easy to understand, easy to implement, and most compilers will generate fast code from it. Its purpose is to calculate the result array `res` from input arrays `x` and `y` and input constant `a` , using the mathematical function $\mathbf{R} = a\mathbf{x} + \mathbf{y}$.

```cpp
void axpy(double* res, const double a, double* x, double* y, unsigned long N)
{
    for (auto i = 0UL; i < N; ++i) res[i] = a * x[i] + y[i];
}

auto main() -> int
{
    std::vector<double> u(100'000'000UL, 0.);
    auto v = u;
    auto w = u;
    for (auto i = 0UL; i < u.size(); ++i) {
```

```
12          u[i] = 0.11 * (i % 7);
13          v[i] = 0.123 * (i % 365);
14      }
15      axpy(w.data(), 3.5, u.data(), v.data(), w.size());
16  }
```

The function `axpy` does not do any memory management. We give it the location in memory of 3 arrays and it performs a simple action on those arrays. When we use the function in line 15, we simply extract and forward the raw pointers holding heap allocated arrays (using the function `data()`) from our `std::vector<double>` objects `u`, `v`, `w`. There are no `new` and `delete` calls in sight. `std::vector` will correctly allocate and deallocate memory for us. Think how we would feed the function `axpy` its inputs without being able to give it the addresses where the data is stored. Copying over one or two double precision numbers to the next white board for processing is one thing, but copying over a billion would just be silly. Instead, we make the function accept pointer inputs. Pointers store addresses. The function can access the data at those addresses. We copy to the next white board nothing more than the addresses on the heap corresponding to where the inputs are to be found.

Non-owning pointers are safer and easier to manage than resource managing pointers, but they still require some care. We have to ensure that the addresses held by the non-owning pointers correspond to valid objects. Recall the discussion of the dangling pointers above. That problem is independent of memory management. One strategy to reduce the risk of dangling pointers is to not store the non-owning pointers as local variables at all, but instead retrieve them directly from resource handling classes as and when needed. In any project, it is also a good idea to set up the debug builds to run with the address sanitizer. There may be subtle cases where a memory bug is not detected by the sanitizer, but quite often the cases are not that subtle, and will be correctly detected. Once we have ensured that the sanitizers do not find any errors such as "use after free", we can compile with full optimisation and without the sanitizers, for speed.

### 2.6.5 References

References in C++ are a similar idea to pointers. There are two kinds of references. For a variable of type `Type`, there is a corresponding L-value reference type, written as `Type&`, and an R-value reference type, written as `Type&&`. L-values are expressions which can be on the left hand side of an assignment expression. For instance, variable names, expressions like `v[i]` etc. can be on the left or right hand side of an assignment. But, things like `5`, `x * y` or `cos(x)` can't be on the left hand side of an assignment expression. Such expressions are called R-values, and references to R-values are R-value references. Let's first discuss L-value references.

```
double pi{ 3.141 };
double& lvr { pi };
std::cout << pi << ", " << lvr << "\n"; // 3.141, 3.141
lvr = 2.7;
std::cout << pi << ", " << lvr << "\n"; // 2.7, 2.7
pi = 3.14159;
std::cout << pi << ", " << lvr << "\n"; // 3.14159, 3.14159
```

An L-value reference is like an alias for the same object that it refers to. Just like a pointer, we can read and write the variable (unless it is `const`) through the reference. But unlike a pointer, we don't need to dereference a reference to access the object. In fact, we can now look at our variable declarations in a new way. When we say `int var{0};`, we create an object of the `int` type somewhere (4 bytes, `int` rules for interpretation and functions...), set its value to 0, and *create a reference to that object* called `var`. The variable name itself is the first reference we create to an object. The type of `3.14159` above is `double`, but the type of `pi` is `double&`, a reference to a `double`.

A reference is attached to a specific object in memory, like a glued pointer. A reference can never be re-assigned to point to a different object. We can not create a null reference. And we don't use pointer like dereferencing syntax when working with references. A reference is therefore like an automatically dereferenced constant pointer. It must be initialised to an object at creation.

Another important difference between pointers and references is that it is not possible to create an array of references, whereas an array of pointers is a pretty normal thing, e.g.:

`auto main(int argc, char* argv[]) -> int`
References behave like fixed automatically dereferenced pointers. They might be replaced by addresses by the compiler, but they are not real objects in the C++ language.

Being similar to pointers, references inherit some of their problems. Dangling references can cause similar problems as dangling pointers. But constant references are extremely valuable as function argument types, and are the most common way of declaring formal parameters in function headers. Let's understand them a little better.

We saw earlier that large objects can be passed to a function cheaply by using pointers. We just copy the address to the function stack and begin. That is exactly the purpose of references as well. When a function expects its input as a reference, when we call the function with an object, the compiler initialises the function input parameter (the reference) with our object, essentially giving it the address of our object. In the function code, we end up dealing directly with the object used to call the function. We will see plenty of examples of this later.

```
1  std::string obj{"Some concrete object somewhere"};
2  std::string& ref{obj};
3  const std::string& cref1{obj};
4  std::string const& cref2{obj};
5
6  std::cout << ref << "\n"; // same content as if we had printed the obj
7  std::cout << cref1 << "\n";// same content as if we had printed the obj
8  std::cout << cref2 << "\n";// same content as if we had printed the obj
9
10 ref[1] = ref[0]; // Overwrite the second character with the first. OK.
11 cref1[3] = cref1[2]; // Error! can not modify through const-reference!
12 cref2[3] = cref2[2]; // Error! can not modify through const-reference!
13
14 std::string const obj2{"Constant string object"};
15 std::string& ref2{ obj2 }; // Error! Binding reference to obj2 discards qualifiers!
```

In the above, we created a string and a few references to it, differing by how the `const` qualifier is used. The declarations for `cref1` and `cref2` only differ in the placement of `const` (right or left of the object). They mean the same thing. For pointers, it is meaningful to try to distinguish between constant and non-constant pointers to constant or non-constant objects. We saw a lot of different ways of writing "constant pointer to (constant) X". References always point to the same object, so that the only thing which can be non-constant is the object it is pointing to. Therefore, there are no variations above with a `const` appearing on the right of the reference symbol, `&`. The `const` in a constant reference can only refer to the object, and therefore it does not matter whether we put it on the left or the right of the object. The argument of consistency in all cases (normal objects, pointers and references) is still valid, and many programmers therefore prefer writing `int const&` instead of `const int&`. In both cases, we are talking about a reference to a `const int`. Notice that we created constant references to a string object which itself was not a constant. That's because the constant reference is more restrictive in the constantness guarantees than the guarantees in the object itself. It does not seek any previleges to modify the object. It would not be possible to do the opposite, i.e., initialising a non-constant reference from an object that is a `const`, as shown in line 15. The same applies to initialising non-constant references from constant references.

As shown in the above code example, when we try to use an object through the reference, we don't need to dereference it. If we merely want to read information from the object without modifying anything, both the `const` as well as non-`const` reference can be used. This is illustrated in lines 6–8. However, if we want to modify the object, we can use the object directly or, equivalently, use a non-constant reference. We can not use a constant reference in any operation which does not promise to keep the entity unchanged, as shown in lines 11 and 12. Because of this, a function taking a particular argument as a constant reference, can not perform any operation that might modify that object, whereas a function expecting a normal non-constant reference may modify the object used in the input.

```
auto sum1(std::vector<double>& v) -> double
{
    auto sum = 0.;
    for (auto i = 0UL; i < v.size(); ++i) {
        sum += v[i];
```

```cpp
        v[i] = 0;
    }
    return sum;
}
auto sum2(const std::vector<double>& v) -> double
{
    auto sum = 0.;
    for (auto i = 0UL; i < v.size(); ++i) {
        sum += v[i];
        // v[i] = 0.; // this won't compile
    }
    return sum;
}
void elsewhere()
{
    std::vector<double> X{1., 2., 3., 4., 5.};
    std::cout << "Sum 1 = " << sum1(X) << "\n";
    for (auto el : X) std::cout << el << "\n";
}
```

In the above, we call the function `sum1` with an input vector. The answer we receive is correct, but after the call to `sum1`, the vector is all 0. This is because the function `sum1` takes its argument as a non-const L-value reference. Inside the body of the function seen at the top in the snippet above, we modify the input vector by replacing the elements by 0 in each step of the iteration. This is allowed because as far as that function is concerned, the input parameter is a non-constant object, so that it can do any modifications it wants. The second version of the sum function, `sum2` takes its input as a constant reference. Any attempt to modify that input in that function would result in a compiler error. If we had used `sum2` instead of `sum1` later, the vector `X` would remain unchanged after we have evaluated the sum.

Why does this matter? We often need to call functions in libraries written by other people. When calling such a function, it makes a great deal of difference whether the carefully constructed object we pass as an input to that function remains unchanged by that call or not. It affects what we can write after that. To verify whether or not a function tampers with its input, all we have to do is to look at the function header, not its entire code! If the function takes its argument as a constant reference, and the compiler accepts the implementation of the function, it means that at the end of the function, the object we passed as a reference to the function is unchanged. We don't need to write any tests to verify it, we don't need to carefully scan through the source code of that library function. If it takes a constant reference, it has read-only access.

### 2.6.6 Pointers, references and auto

We have seen the C++ variable declaration syntax using the `auto` keyword. To create a new variable, we write something like `auto x {initializer};` to create `x` whose type is the deduced from the type of the initializer. If the initializer is a pointer, e.g., `auto x{&other_var};`, then so is the object of the newly created variable `x`. The value of `x` is a pointer. The name `x`, as we discussed earlier, is the first reference we create to that pointer. It would however be nice to make this a bit more on the face, without spelling out the full typenames. We just want to indicate that the new variable we want to create stores a pointer type, and not an ordinary type like `int` or `double`. This can be done by using the so called qualified `auto` declarations:

```cpp
std::string preexisting{"Exemplar"};
auto ptr{&preexisting};
auto* ptr2{&preexisting};
```

The last two of the above declarations will create `std::string*` objects, because preexisting is declared to be an `std::string` (`preexisting` is a `std::string`, so `&preexisting` is `std::string*`). But if you mistype and forget the `&` when creating `ptr` above, there is no error, because what remains after the typo is a syntactically valid statement, although it means something different than what you intended. You wanted a pointer, and you got a normal `std::string`. And the compiler said nothing. It is the job of the compiler to catch such errors. The second version, with the qualified `auto` tells the compiler that you were trying to create a pointer. If you forget the `&` or use some other initialiser which is not a real pointer, it can't be used to create a pointer, so the compiler will most assuredly flag it as an error.

In general in C++, we are trying to write code with as strict a specification of our intents as we can. The more precisely we can formulate our intent, the more strictly the compiler can "proof read" our code. The compiler is our friend, and we are not in the game of trying to "get something past the gatekeeper". It is a tool to help us translate our vision precisely into C++ code. Therefore, it is regarded as good practice to use qualified `auto` whenever possible.

When creating references, using qualified `auto` is even more important. Because ordinary variable names are themselves L-value references. When we say `auto I{3.141}`, it is clear that we want to create a `double` object with the value 3.141, and refer to it by the name `I`. If we then go on to say `auto J{I}`, do we want to create yet another object of the same type and value or do we want another reference to the same existing object? References are normally initialised from existing variables, the same as full objects:

```cpp
std::string preexisting{"Exemplar"};
std::string another{preexisting};
std::string& ref{preexisting};
```

The creation of the `std::string` object when declaring `another` and the reference `ref` required exactly the same initialiser. If we skip the explicit type description on the left, how is the `auto` going to figure out what we wanted? To reduce the potential for ambiguities, the type deduction rules for `auto` are formulated in such a way that it usually does not create a duplicate reference when given a pre-existing variable or some other L-value reference. It creates a duplicate object instead. In order to create an alias reference using `auto`, we have to specify the reference symbol explicitly:

```cpp
std::string preexisting{"Exemplar"};
auto& ref{preexisting};
const auto& cref{preexisting};
```

Writing `auto var{preexisting}` will make `var` a new `std::string`, where as `auto& var{preexisting}` will make `var` a reference to the existing `std::string`. To create a constant reference, we use `const auto& cref{preexisting}`. Note that even if the initialiser was a constant reference, using plain `auto` will create a new object without the `const` qualifier. To create constant or non-constant references to existing objects, we have to be explicit by using qualified `auto`.

The rules for automatic type deduction with `auto` are formulated in terms of C++ templates. We therefore limit ourselves to these working recipes for now. We will revisit the topic of why `auto` behaves the way it does after we have discussed templates. We will close this section with another "recipe" for declarations of references with `auto`:

```cpp
std::string changeable{"Non-constant string"};
std::string const fixed{"Fixed"};
auto&& x{changeable};
auto&& y{fixed};
```

With the notation `auto&&`, we use the rules of the so called "universal references". The result above is that `x` is deduced as `std::string&` whereas `y` is deduced as `std::string const&`, i.e., we retain the `const` qualifier in the initialiser. The `auto&&` notation also lets us initialise R-value references, but that is a topic we leave out until we discuss C++ classes in detail. Let's examine a popular use case and explain what is happening there:

```cpp
const std::vector<std::string> V(100, "Very long and different strings");
for (auto s : V) std::cout << s << "\n";
for (const auto& s: V) std::cout << s << "\n";
for (auto&& s: V) std::cout << s << "\n";
```

All the above loops print out the strings in the vector `V`. For every iteration round, the range based for loop initialises the loop variable with the next element from the given sequence, and executes the loop body with that variable. In the first version, it will take every element of `V` one by one, and initialise the loop variable with it, for example, `auto s{V[0]}` for the first iteration. As we discussed in this section, this deduces the type of `s` to be `std::string`, and hence a new string is created with the same content as `V[0]`. Therefore, in the first loop we construct a new, potentially long, string for every iteration. In the second case, the loop variable is declared as `const auto& s{V[0]}`. Therefore, it will deduce the type of `s` as `std::string const&`, and initialise the reference to the string object in the array. No new copy will be created. The third version will also create constant references because our initialisers coming from `V`, which is a `const` vector, will be `const` qualified. If `V` were not `const` qualified, the third version of the loop will use non-constant references. Both the second and third version are more efficient because they do not create unnecessary copies of the strings for each loop iteration.

When introducing the range based for loops, I used the plain `auto` like in the first version above, because that is all I could explain without a background on references. Now, I can recommend the second or third version for almost all cases where the elements of the sequence we are iterating over are non-trivial entities (anything other than built-in types). The second one wins out on clarity, but the third one is more general while still being as efficient.

## 2.7 More about functions

Let's now discuss functions in some more detail. We are already familiar with their basic syntax.

```
//declaration
auto function_name(parameter list) -> return_type ;

// definition
auto function_name(parameter list) -> return_type
{body}
```

The specification of the function name and its inputs is called its "signature". It is possible to only "declare" a function without providing explicit code of it, as shown above. A declaration consists of the function header followed by a simple semi-colon instead of the function body. A function declaration is also a promise. We are saying that "there is a function with these properties, just assume that it exists and look at the rest of the code". In the following, the compiler can check the code for correctness and generate "object" code for our program.

```
auto term(int i) -> double;

auto series_sum(int from, int to) -> double
{
    auto sum{0.};
    while (from < to) sum += term(from++);
    return sum;
}
auto main() -> int
{
    std::cout << series_sum(0, 100) << "\n";
}
```

You can copy this code to a file (e.g., `seriessum.cc`), and compile it:

```
g++ -std=c++20 -c seriessum.cc
```

It will compile without errors. But obviously, since we didn't write a function body for `term()`, we can not build an executable program out of it. With the `-c` option, we told the compiler to stop after it has finished compiling, without trying to invoke the "linker", which creates our final executable. The compiler can check that our code is syntactically correct. Every time we refer to `term()`, the compiler looks for a definition or a declaration. Since it only finds a declaration, it just checks that we are we are using the promised function right. We are using it as `term(from++)` which means we are trying to call it with a single integer input. The compiler checks that the function prototype (the declared function) can accept that kind of input. It checks what we are doing with the output. If any of these aspects were inconsistent with the prototype, we would have an error. Just for fun, check that if you replace `term(from++)` by `term(&from)` the compiler does not accept it. With the prototype, you promised a function called `term` taking an integer argument, not a pointer! But after the syntax check, the compiler can only go so far as to generate instructions to call that promised function. It still has no idea what that function actually is. It is the job of the linker to match such calls to functions defined elsewhere, perhaps in a library, or in another file you are using to build the program. If you try to build an executable, i.e., run the compiler without a `-c` option, you will get an error saying something like `undefined reference to 'term(int)'`.

### 2.7.1 Parameter passing

```cpp
// examples/parameter_passing0.cc
#include <iostream>
auto sum_till(unsigned N) -> unsigned
{
    auto sum{0U};
    while (N) sum += N--;
    return sum;
}
auto main() -> int
{
    auto lim = 10U;
    std::cout << sum_till(lim) << "\n";
    std::cout << lim << "\n";
}
```

Run the above program. Why does it print what it does?

We call the function `sum_till` like this: `sum_till(lim)`. In the `sum_till` function, we change the input parameter in the `while` loop. Since non-zero values are regarded as `true`, the loop runs as long as `N` is non-zero. In every loop iteration, we decrement `N` by one. By the time we are done, `N` is 0. Yet when we are done executing the function, `lim`, which was passed as the input parameter, is still 10. This is because we are passing the parameters as a `value` in this instance. When a function is declared with a non-reference type as its parameters, an object is created for each of the parameters from the function call expression. If we had called `sum_till(7)`, instead of a named variable, we couldn't have decreased 7 in that `while` loop until 7 became 0, could we? The function `sum_till` declares in its header that it needs to start with an integer object that it is going to call `N`. When we call that function with an expression `expr`, i.e., like `sum_till(expr)`, the function sees it as

- Initialize parameter `N` with `expr`
- Run the code of the function
- Return to the expression where the function was called, replacing the function call with the return value

In our case, it was like `unsigned N{ lim }` at the start of the function. It is that newly created integer `N` which we decreased to 0 in the loop. Not the original `lim`. We could have written `sum_till(lim + 15)`, in which case the expression `lim+15` would have served to initialise the variable `N` without any problems.

If you now change the header of `sum_till` a little to specify that the input parameter is a reference: `auto sum_till(unsigned& N) -> unsigned`
the behaviour changes, although it is still possible to reason as above. We now get a final output 0 for the value of `lim` after the function has run. In this case, again, the function will create a variable `N` with the type it is declared: `unsigned&`. Something like `unsigned& N{lim}`. Remember what happened when we declared references like that? The reference became kind of an alias for the original variable.

Same happens here. `N` becomes just another name for the object `lim`. Therefore, `lim` undergoes all the changes that `N` does in the function, and ends up with a value of 0. Using reference types in function parameter list is called passing parameters by reference. Without the `&` in the parameter list, the function worked with a copy of object that it created. With the `&`, it created a reference or an alias. Now, try changing the input to `lim + 1` with the `unsigned&` input type. You should get an error, saying something like:

```
error: cannot bind non-const lvalue reference of type 'unsigned int&' to an rvalue
of type 'unsigned int'
```

This is an error because we are trying to give another name to `lim + 1`, but you can only give another name to something that has a name! `lim+1` is an R-value. The reference we created is an L-value reference. The error message says that you can not attach a non-constant L-value reference to an R-value, but only to objects which can be on the left side of the `=` sign. There are two parts to the compiler's grievances: *non-const* and *L-value*. If we make it a `const unsigned&` in the function header, passing `lim + 1` will be OK, but the function code as written wouldn't work, as it tries to change `N`. An R-value reference is made exactly for this kind of nameless objects: somewhere in memory there exists an object which is the result of the calculation `lim + 1`, although that entity does not have any name. An R-value reference can attach itself to that kind of object. To make the input parameter an R-value reference, you would write it as `unsigned&&`. Calling `sum_till(lim + 1)` would then be acceptable to the compiler. It is however not a real use-case for R-value references. For a good use case, you should wait for the discussion of *move semantics* in connection with our discussion of C++ classes. For simple numeric types, it is best to pass the arguments by value (without any `&` or `&&` in the function signature).

One use of non-constant L-value references in function signatures is "in-out" arguments. We want the function to work with the input value we provide, but also change it in a meaningful way. One can also use it to create multiple outputs from a function:

```cpp
void findlims(int &i0, int&i1)
{
// Find a suitable range
   i0 = beginning_of_the_range;
   i1 = end_of_the_range;
}
auto elsewhere() -> int
{
    int x1, x2;
    findlims(x1, x2);
    for (auto i=x1; i < x2; ++i) {
        // work
    }
}
```

This is no longer the recommended approach in C++ if you need multiple outputs from a function. Any number of arguments of any type can be packaged in an `std::tuple` and returned from a function. For example,

```cpp
auto findlims() -> std::tuple<int, int>
{
// Find a suitable range
   return {beginning_of_the_range, end_of_the_range};
}
auto elsewhere() -> int
{
    auto [x1, x2] = findlims();
    for (auto i=x1; i < x2; ++i) {
        // work
    }
}
```

Compared to the implementation using L-value references, the tuple output combined with the "structured binding" we see in line 8, is more expressive about our intents. `x1` and `x2` are the two outputs of the function, not the input. Input is what goes inside the function parentheses `()`, and output is what we write after the `->`. It is useful to know both approaches, so that you can choose the method that seems most understandable on a case by case basis.

Constant references, e.g., `const std::string&` are rather commonly used as function inputs.

```cpp
auto encoder(const std::string& input) -> std::string
{
    // ...
}
auto elsewhere() -> anything
{
    std::string userstr;
    // work
    auto encoded = encoder(userstr);
    // more work
}
```

When we use the function `encoder` above, we know that we are not creating a whole new string out of `userstr` in the function. The function takes a reference input after all! Strings are somewhat expensive to copy, especially if they are more than a few characters long. Since `encoder` was written with a reference argument, we know that the function is going to use an alias for the existing `userstr` instead of creating a brand new string with the same value. That alone would not be great though: we know that a function taking a reference argument can modify its input parameter. Perhaps we don't want `encoder` to be able to change our `userstr`. Can we trust it that it won't? We can, if `encoder` uses a `const` reference in its input. A `const` reference is like a read-only access to the original that we are using as function input. No expensive copy. No danger of the input being modified. This is why, with the exception of when the inputs are simple numbers, or other light weight objects with only a few bytes of data, you will see that C++ programmers like to write functions with constant references to various types as inputs. I have waited to use parameters like `const std::vector<double>&` etc. in the examples, only because the concept had to be explained first. From now on, we will (gradually) introduce more of the `const Type&` syntax whenever appropriate.

### 2.7.2 Overloading

Suppose, as an example, that we are writing a function to calculate $x^y$, for real number $x$. If the power $y$ is a positive integer, we can just multiply $x$ with itself the required number of times. If it is a negative integer, the answer would be $\frac{1}{x^{-y}}$. If $y$ is it self not known to be an integer, power calculation becomes more complicated, but for the purpose of this example, let's just say we calculate it as $exp(ylog(x))$. This does not work for negative $x$, and in reality we have to handle the whole problem much more carefully for special cases. But, let's just recognise that there are operations which could be (would have to be, or would preferably be) done differently depending on the type of the input.

We could write two different functions for the two ways to calculate the power, something like `integral_power` and `real_power`. That would work, but it is needlessly verbose. In this approach, `integral_power(x, y)` is intended to be called with integer $y$, and `real_power(x,y)` is intended for use with `double` or `float` $y$, every single time. We are therefore providing the integer/real information twice: once in the name of the function, and once in the type of the argument `y`. If after using these functions hundreds of times in a program, we decide to change the type of a parameter from integer to a floating point number, we have to change each instance of the use of the specialised power function. Wouldn't it be nice if the function to call could be automatically inferred based on the type of the parameter $y$? Something like:

```cpp
// examples/overload0.cc
#include <iostream>
#include <cmath>
#include <limits>

```

```cpp
auto power(double x, int n) -> double
{
    if (n == 0) return 1.;
    else if (n < 0) return 1.0/power(x, -n);
    auto ans{1.};
    while (n--) ans *= x;
    std::cout << "Using a simple loop to calculate power\n";
    return ans;
}

auto power(double x, double y) -> double
{
    std::cout << "Using exp of log method to calculate power\n";
    if (x < 0) {
        std::cerr << "Invalid input for non-integral power function: " << x << "\n";
        exit(1);
    } else if (x == 0) {
        if (y > 0) return 0.;
        else if (y < 0) return std::numeric_limits<double>::infinity();
        else return std::numeric_limits<double>::quiet_NaN();
    } else return std::exp(y * std::log(x));
}

auto main() -> int
{
    std::cout << power(4.0, 3) << "\n\n"; // int power
    std::cout << power(4.0, 3.) << "\n\n"; // double power
}
```

Here, in the `main` function, we have called `power(x,y)`, twice, once with an integer power and another time with a **`double`** power. We also have two functions just called `power`, differing only in the type of the second input. The compiler has all the information to know in each case which variant we want to call. You are probably thinking this would be very reasonable. Does C++ really allow that? Try it!

As you have probably found out, it is allowed without any fuss. The C++ compiler does indeed distinguish functions based on the types of its input arguments. If, to achieve one goal, different methods need to be used depending on the type of the input arguments, we can choose a single intuitive name for the function, and write multiple versions of it for the different input parameter types, all sharing the same name. This is called function “overloading”. The collection of functions with the same human readable name but different combinations of types in the input parameter list is called an “overload set”. When using the functions, the compiler chooses the variant from the overload set that best matches the inputs. The “best match” is found during compilation, using the types of the input at the point of usage. If we were calculating $x^y$ for real numbers $x$ and $y$, where $y$ was varying from 1 to 5 in steps of 0.1, there will be points where the value of $y$ will be a whole number. The call to `power(x,y)` will not automatically change to the integer variants at those points. The overload resolution is done using the `type` of the input, not its value. When the compiler translates C++ code to machine code, it always knows the type of the inputs to a function. It may not know the values.

One could think of this as an example of “polymorphic” behaviour for the function `power`. This kind of polymorphic behaviour has no cost at program execution time. When the program is running, we are not spending CPU cycles deciding which variant of the function to call. The compiler already made a choice based on the types of the inputs when the code was compiling. This is one of the many forms of the so called “static polymorphism” available in C++.

## Exercise 20:

Read the function signatures in `examples/binform.cc` and note where function overloading is used.

Overloading works because the C++ compiler does not use the human-readable name of a function as its true identifier. That name is merged with the types of all the inputs of the function to create a composite name. When we try to use a function with some parameters, the compiler looks for a function with the same composite name. Other functions in the overload set will have different composite names, so that as far as the C++ compiler is concerned, there is no ambiguity about which function we are calling at any point.

## 2.7.3 Function templates

Sometimes, we have the exact opposite requirement compared to what we saw in the last section. The `power` function needed to follow different procedures depending on the specific types of the input parameters. Sometime we want to do the exact same operations in code, irrespective of the input types. Let's take an example. Let's say we are writing a function which takes two numbers, and returns the smaller of the two. For input types `int` and `double`, we would need something like this...

```cpp
auto min_int(int x, int y) -> int
{
    if (x <= y) return x;
    return y;
}
auto min_double(double x, double y) -> double
{
    if (x <= y) return x;
    return y;
}
```

Here, the code in the function body is the same for both of them. Perhaps, we could just write it without reference to any specific type, and let the type be inferred from the invocation of the function.

This can be done using what is known as a `template` in C++. We first declare a placeholder name and then write code with a placeholder name for the type of the function parameters. The placeholder name(s) to be used as types in function signatures are declared as follows: `template <class T>` or `template <typename Koala>`. As of C++20, `typename` and `class` in a template declaration are equivalent.

```cpp
template <class T>
auto minval(T x, T y) -> T
{
    if (x <= y) return x;
    return y;
}
```

When we use it as `minval(1,2)`, the compiler does the following,

- It looks for a function called `minval` with two integer arguments. If it finds such a function, it uses it. Let's say such a function does not exist, and only our template above exists.
- It tries to make a substitution for `T` so that the the function `minval` defined as above becomes a match for the expression. If we substitute `T` with `int`, we have an exact match. It then creates a function with the signature `minval<int>(int x, int y) -> int` and translates the template code above with a concrete type for `x` and `y`. After all, how the comparison operation can really be done by the processor depends on what we are comparing (remember the bit layouts of integers and floating point numbers?).

From then on, anytime we call `min` with two integer arguments in our program, the compiler uses that generated `minval<int>` function. If it then encounters, for instance, `minval(2.1,3.1)`, it follows the same process as above, and generates code for us with the substitution $T \longmapsto double$. In the rest of the code, we can use the above single function code as `minval(1,2)`, `minval(1.0, 2.0)`, and even `minval("one"s,"two"s)`. As long as it is possible to make a template substitution to match our function call, the compiler will generate code using the template. If we tried to use it like this though, `minval(1, "two"s)`, the compiler can not make a successful substitution. The first type matches if we substitute $T \longmapsto int$, but `"two"s` is a string. The type of the second argument matches if we substitute $T \longmapsto std :: string$, but then the first one fails. The compiler will then complain that it got "conflicting types" when it tried to substitute for `T`.

**Exercise 21:**

Use the program `examples/temp0.cc` to familiarize yourself with function templates. It contains the `minval` function above with a few uses of the function. First compile and run the function. See how a single definition of a function template can be useful for various types. Now, uncomment the line containing an invocation of that function with two different types of arguments. Try to compile, and read the error message.

Template error messages in a big program can be daunting to the uninitiated. Here, we have a tiny program, and a relatively small error message. Usually, in a big program the compiler will find lots of possible substitutions which will all be listed, and a reason will be given why the compiler rejected each option. This makes the error messages long. But the content of the messages is usually helpful in finding what's wrong. Consider yourself initiated.

Some of you might be wondering why we don't simply let the compiler infer the types of `x` and `y` in `minval` by declaring them as `auto`. Since, the variables in the input parameter list of a function are initialised from the argument list when the function is called, and we have already seen the initialiser being used to deduce the type of a variable (e.g., `auto x = 1.0`), perhaps we can declare the function parameters as `auto`, and infer the output type as `decltype(x)`?

```
auto minval(auto x, auto y) -> decltype(x)
{
    if (x <= y) return x;
    return y;
}
```

We can. This is legal code in C++20, but this was not allowed till C++17. It will work for `minval(12, 45)`, `minval(0.99, -3.2)`, and `minval("one"s, "two"s)`. But in this way of writing the function, the variable type for `x` and `y` will be inferred independently of each other. If we wrote `min("one"s, 2)` it will infer `x` to be a `std::string` and `y` to be an `int`, and run into all kinds of difficulties, trying to generate concrete code from the above. The second return statement will now try to construct a `std::string` from an integer. The boolean expression `x <= y` is now ill-formed. Instead of complaining about the difficulty of inferring the correct substitution for `T` as above, the compiler will now find errors throughout the *body* of the function instead. Depending on the particular function at hand, and where that might be called, it may be better to use the `auto` function parameter approach, or the explicit `template` approach. The `auto` in the function parameters also makes the function a function template, only we don't explicitly write `template <class T>` etc.

Many properties we have learned about functions also hold for lambda functions. Their definition:

```
[](parameters)->return_type { body }
```

looks quite similar to ordinary functions, except that we don't have a function name, and we have those weird `[]` capture brackets. Lambda functions could be declared generically with `auto` in their parameter list since C++14. Regular functions only got that ability with C++20. But regular functions could be written as function templates since time immemorial since the first official standard. Lambda functions only got that ability with C++20, to make these two very similar. The syntax of using an explicit templates in a lambda function is like this:

```
[]<class T>(parameters) -> return_type {body}
```

### 2.7.4 Attributes

When the job of a function is to acquire resources for you, the worst disrespect a user of the function can show is to ignore the valuable resource returned from the function.

```
// examples/ignored.cc
template <class T>
auto getmem(unsigned long N, T init) -> T*
{
    T* ans = new T[N];
    for (auto i = 0UL; i < N; ++i) ans[i] = init;
    return ans;
}
auto main() -> int
{
    double* x = getmem(10'000'000, 0.);
    double* Y = getmem(10'000'000, 0.);
    getmem(10'000'000, 0.); // return value not stored!!
    delete [] X;
    delete [] Y;
}
```

Remember how when you acquire memory with `new` you have to call a matching `delete`? How can you do that if you ignore the return value of such a function? There can be many other reasons why you as the developer might want to make sure that the user of a function does not unceremoniously drop the return value, as shown above. In such cases, you can place `[[nodiscard]]` before the `auto` in the function header. The compiler will then detect and notify when the function is used and the return value not saved. Try compiling the above program as it is and after inserting a `[[nodiscard]]` at the right place.

`[[nodiscard]]` is called an "attribute". Attributes are additional hints to the compiler. The code is syntactically complete without them, but if a compiler knows about an attribute, it may be able to produce better error messages or compiled code with better performance. Other examples of attributes are `[[deprecated(your deprecation message)]]`, to emit a warning when a function is used which you want to replace with a newer version, `[[likely]]` or `[[unlikely]]` to mark the `if` or `else` parts if you, as the programmer, know that one of them is far far more likely than the other.

## 2.8 Exercises

### Exercise 22:

If `line` is a variable of type `std::string`, the function `getline()` can be used to retrieve entire lines at a time from an input stream. E.g., `getline(std::cin, line)` will read from the standard input till the end of the line (until you press ENTER), and put the entire line of text into the variable `line`. Pressing ENTER without any input reads an empty line. Use this, to make a program which asks for new names to be entered. Names could have multiple parts. Append the name to the end of a name list. This should repeat until an empty name is entered. After this, you should print the names sorted in alphabetical order.

Use a `std::vector<std::string>` as the type for the name list. Appending an element to the end of a vector `vec` is done like this: `vec.push_back(newelement)`. Sorting the vector by the natural comparison criteria of the element type (alphabetical, for strings) is done simply as `std::ranges::sort(vec)`. To change the sorting criterion, you can give `sort` another argument, which is a function one can use to decide what element goes before what. Usually it is a simple $a < b$ comparison. But, you can give sort, for example, `[](auto a, auto b)->bool { return a.size() < b.size(); }` as the second argument to sort the names by the length of the names. Try this sorting criterion as well.

### Exercise 23:

Take the partially written program `examples/cleanup.cc` as a starting point. This exercise does not work on the online platforms as it needs standard input. The program asks the user to type some text. When the user presses ENTER or RETURN, the entire line of content is read in as a string. The program cleans the typed string using a function `cleanup()`, and prints the result. It stops when you press ENTER without any content. The `cleanup` function as given to you does not do anything at all. It returns the input string unmodified.

Your task is to modify the clean up function so that it removes any leading and trailing spaces or punctuation marks.

# Chapter 3

# One illustrative exercise

## 3.1 Many ways to skin a cat, and what we can learn from them

Note: A lot of exercises in this chapter use the C++20 `<ranges>` header and the facilities it provides. As of this writing, they can be compiled with `g++` version 10 or newer or the Microsoft Visual C++ compiler version 19.29. Although LLVM clang++ now supports some aspects of the C++20 ranges feature, other important bits are missing (as of clang 14.0), so that it may not be possible to experiment with these using Clang. Most concepts you learn in the following are valuable for older C++ standards, and some even for other programming languages. Let's consolidate the concepts learned so far, and add a few by exploring an easy to understand problem from school mathematics lessons. We know that given any real number $x$,

- $sin^2(x) + cos^2(x) = 1$
- $sin(2x) = 2sin(x)cos(x)$
- $sin(3x) = 3sin(x) - 4sin^3(x)$

You know these things when you were 10–12 years old, and you know these now. We are not trying to prove these basic trigonometric relations. But we will use these as our vehicle for learning some more C++ syntax, and getting some practice along the way. To that end, we will be verifying that the relations hold for a lot of values in a given range. The 3 relations listed above are there for you to avoid getting bored doing the exact same calculation in 9 different ways. Switch them around as you wish.

So, a rough description of what we want to do: we generate a sequence of values in a given range, and for each generated value, we evaluate whether the relation holds. If for any generated value of $x$ the left and right sides of the equations are not the same, we would have proven that the relation does not hold.

There are many components to this problem, spanning from very fundamental concepts to some very elegant high level facilities in C++. Since each of the above relations can be written as some function $f(x) = 0$, e.g., $sin^2(x) + cos^2(x) - 1 = 0$, at the heart of our program, we will be evaluating a function returning real numbers, and comparing its result to 0.0. I trust you can now write such a function easily. But the first theme I would like to discuss is what is meant by comparing the result to 0.0 or any other number for that matter.

With this knowledge of how real numbers are stored (see section 2.1.1) in types like `float` and `double`, it should come as no surprise that a boolean expression like `x == 0.` will only be true if all the bits of `x` are 0. If your calculation returns a number with every single non-sign bit set to 0, the expression will be true. Otherwise, it could be a very small number, like $10^{-100}$, but that's not the same as 0. It has lots of non-zero bits as you might have seen when running the example program `binform.cc`. Remember, the density of representable numbers near 0 is high, so there is no a priori reason for the computer to judge something like $10^{-100}$ as small enough to be considered almost equal to 0. To make the concept of "almost zero" concrete, we need a scale. If all the numbers we are working with in a certain problem are between 0 and $10^{-100}$, then the latter most certainly does not seem close to 0. If on the other hand a quantity takes any value between 0 and 100, a value of $10^{-100}$ can probably be safely treated as "equal to" zero. Therefore, for floating point numbers, it is usually a bad idea to use equality comparison, as in, `x == 0.`. What we mean is $abs(x) < \epsilon$ where $\epsilon$ is a scale that defines what is sufficiently close to 0. That scale depends on the calculation we are trying to perform.

Let's go back to our fixed width decimal number representation in section 2.1.1 for a moment. Imagine we have two numbers, $x$ and $y$ both of which have the initial value of 1 million. If we now add 3 to $y$. As discussed before, $y$ will stay put at 1 million, and the fact that we did the addition will not be visible in any way in the stored value. Then we subtract the $x$ from $y$, we suddenly have a much smaller number, i.e., 0 in our representation. The true answer here, 3, will be lost, because numbers around a million can not be resolved in such fine detail in our representation. Before we do the subtraction, we lost the information about the addition of 3 to $y$. Just because we now want to subtract a million from $y$ does not mean that we can miraculously recover that lost information. It's forgotten. So, even when the final answer we calculated was a number around 0, it carried the same resolution as the numbers we subtracted to get there. For our original problem of checking if $sin^2(x) + cos^2(x) - 1 = 0$, we calculate one number which will probably be very close to 1 and then subtract 1 from it. It will be a number very close to 0, but with a resolution of numbers around 1. So, what is this resolution of representable real numbers around 1?

How close can a larger number get to 1 while having a different bit representation? If we change the exponent, we have gone too far, as it is possible to have smaller increments by changing the mantissa. How little can we change the mantissa? How about changing the smallest digit of mantissa by 1? For our decimal numbers, it gives $1 + 10^{-5}$, as there are 5 mantissa digits. For the `double` type, it becomes $1 + 2^{-52}$, which is about $2.22 \times 10^{-16}$ more than 1. Using a `double` type, we can not represent any number that is closer to 1 than $1 + 2^{-52}$. This resolution around the value of 1 is called `epsilon` for a given numeric data type. Properties of different numeric types in C++ are defined in a header called `<limits>`. In that header, `1+epsilon` is the smallest number bigger than 1 which has its own bit representation. The epsilon for the type `double` is obtained by invoking `std::numeric_limits<double>::epsilon()`.

Above, we considered an extreme example of taking a large number and adding very small numbers to it, for dramatic effect. But the core issue is present whenever we add or subtract two numbers of unequal exponents. Consider a calculation $x + y - z$ where $x = 1.11111 \times 10^2$ and $y = \frac{1}{30} = 3.33333 \times 10^{-2}$, and $z = 1.11111 \times 10^2$. First, observe that the number $\frac{1}{30}$ can not be represented exactly in this representation, and therefore we save only 5 digits after the decimal point in a recurring decimal number. That much is easy. To calculate $x + y$, you would align the exponents, so that the second number becomes something like $0.000333333 \times 10^2$, and then add the mantissae, to obtain $1.11144 \times 10^2$, because in our example representation, the mantissa only holds 5 digits. Every time we add two quantities of different exponents, we lose a few digits at the end of the smaller number, because there is no place to put them! If we then subtract the $z = 1.11111 \times 10^2$, we are left with $3.3 \times 10^{-2}$. Some how we started with 6 significant digits in three numbers, performed two simple arithmetic operations, and are left with something with only 2 significant digits! If on the other hand we had rearranged the calculation so that we do the subtraction first, i.e., $x + y - z = x - z + y$, we would be adding 0 to $y$, and the result would contain 6 valid digits. We learn an important property of floating point numbers from this: floating point addition is not associative, i.e., $(x + y) + z \neq x + (y + z)$. The same mathematical operations can often be rearranged in ways to reduce the loss of precision due to truncation.

## Exercise 24:

In the above, we used our toy fixed width decimal representation to illustrate that the floating point arithmetic is non-associative. The same claim holds true for actual `float` and `double` types. The program `truncation0.cc` illustrates this using `float` types, and printing bits of the numbers. We use numbers $\frac{1000}{9}$ and $\frac{1}{30}$ as before, but now our `float` type has 23 bits for the mantissa, which corresponds to about 7 decimal digits. The process plays out analogously in binary digits. When we convert back to decimal numbers for printing, we see some artefacts in the last two decimal digits. The fractional parts of a binary number add up to a value as a sum of a series $m_0 2^{-1} + m_1 2^{-2} + \ldots$. If we need a lot of terms in the series to approximate a number like $\frac{1}{30}$. If we truncate the series too early, it looks odd when converted to decimal numbers. Study the output of this program and see how floating point arithmetic actually plays out.

The effects of truncation are very interesting, and scientists and engineers should learn numerical techniques and choose the best available method to arrange their calculations. But, that would be a full course in its own right, and will take us too far off our current objective. So, let's be aware of the pitfalls, but move on with some more C++. For now, let's use `double` whenever we can, for our floating point numbers (more mantissa bits), and not be overzealous in setting the precision for our floating point comparisons. Epsilon for `double` is around $10^{-16}$. Allowing for some rounding errors in our arithmetic, let's use without much further consideration, $100\epsilon$ as our tolerance when verifying our trigonometric relations.

The first version of the program to do this will use a for loop.

## Exercise 25:

```
 1  // examples/trig_forloop.cc
 2  #include <iostream>
 3  #include <cmath>
 4  #include <limits>
 5  auto f(double x) -> double
 6  {
 7      return sin(x) * sin(x) + cos(x) * cos(x) - 1.0;
 8  }
 9  auto main() -> int
10  {
11      auto npoints { 1000'000UL };
12      auto pi { std::acos(-1) };
13      auto eps { 100 * std::numeric_limits<double>::epsilon() };
14
15      bool invalidated = false;
16      for (auto i = 0UL; i < npoints; ++i) {
17          double x{ 2 * pi * i / npoints };
18          if (std::fabs(f(x)) > eps) {
19              invalidated = true;
20              break;
21          }
22      }
23      if (invalidated) {
24          std::cout << "There relation was found to be invalid.\n";
25      } else {
26          std::cout << "The relation was found to be valid for all points.\n";
27      }
28  }
```

Notice how we use a single quote mark in line 11 to group the digits in a large numeric literal. This is valid C++, and you should never let your head get fuzzy by counting the number of 0s in a number. Notice how when we want to check if the result deviates from 0, we compare its absolute value with `eps` rather than using `==` for equality comparison. The code uses concepts you are already familiar with, so, just satisfy yourself that you understand every line. Change the trigonometric relation we are verifying to one of the other two. Could you use a smaller `eps` ?

Cat skinned. The code works and does what it is supposed to do. But this way of writing it leaves us open to lots of errors, some of which can be reduced by using a bit more of C++ syntax. What if we mistyped and wrote `++pi` instead of `++i` in the `for` loop? We will be incrementing `pi` after each execution of the loop body. And since the continuation condition is written in terms of `i`, which would remain unchanged, the loop will keep running for ever! And what sense does it make to allow something intended to represent the mathematical constant $\pi$ to change during the program execution? If you read section 2.5, you know what to do!

**Exercise 26:**

Find the variable declarations in `examples/trig_forloop.cc`, and make as many variables as you can `const` qualified.

### 3.1.1 using valarrays ...

Some of you would be familiar with programming in Python, and when you read the problem statement regarding our trigonometry task, you would immediately think of something like the following:

```python
# Python code as an example
import numpy as np

npoints = 1000000
eps = 1.0e-14

X = np.linspace(0., 2*np.pi, npoints)
res = np.sin(X) * np.sin(X) + np.cos(X) * np.cos(X) - 1

if any(res > abs(eps)) :
    print("The relation does not hold")
else :
    print("The relation holds for all points")
```

Sometimes, it would be nice to be able to write compact expressions like the above calculation of the `res` variable. `X` is a `numpy` array, containing linearly spaced values in the range 0. and $2\pi$ with a million points. When we say

```python
res = np.sin(X) * np.sin(X) + np.cos(X) * np.cos(X) - 1
```

it is understood that we are generating another `numpy` array through element wise application of a function `X`. The task for the above line is an element wise operation across an array like object. Let's try to do such a thing in C++.

There is something in the C++ standard library, which is a bit like the `numpy` arrays. It's called `std::valarray`. If `v1` and `v2` are of the type `std::valarray<double>`, operations like `v1+v2`, `v1*v2`, `sin(v1)` etc. are understood as element wise operations. The technique used to implement this functionality is called "expression templates". If you can make your own expression templates, you can achieve a similar (or more elegant) syntax as the `valarray` using any array like container. Many C++ mathematical libraries (e.g., Eigen, Blaze, Armadillo ...) achieve this. Let's see what is possible using only the C++ standard library. What we don't have is an equivalent of `linspace`, so let's write a very simple version.

```cpp
auto linspace(double min, double max, unsigned long howmany)
    -> std::valarray<double>
{
    std::valarray answer(0., howmany);
    for (auto i = 0UL; i < howmany; ++i) {
        answer[i] = min + i * (max - min) / howmany;
    }
    return answer;
}
```

The central line, where we are assigning a value to `answer[i]` shows how to access an element in an array in C++. The same syntax is used to access elements of many different "containers" in C++, such as `std::vector`, `std::valarray`, `std::array`, and plain C-style arrays. The right square brackets on the right of a variable name are interpreted as the "indexing operator". The right hand side of that assignment just interpolates a value between `min` and `max`, using a number between 0 and 1 to place intermediate values, i.e., `i * 1.0/howmany`. There is a function with this exact job in the standard library: `std::lerp`. So, we can replace the right hand side with `std::lerp(min, max, i* 1.0/howmany)`.

The main function can now look somewhat similar to the python version:

```cpp
// examples/trig_valarray.cc
#include <iostream>
#include <cmath>
#include <ranges>
#include <valarray>

auto linspace(double min, double max, unsigned long howmany)
    -> std::valarray<double>
{
    std::valarray answer(0., howmany);
    for (auto i = 0UL; i < howmany; ++i) {
        answer[i] = std::lerp(min, max, static_cast<double>(i) / howmany);
    }
    return answer;
}
// The static_cast above is another way to ensure floating point arithmetic
auto main() -> int
{
    using namespace std;
    using namespace std::ranges;
    constexpr auto npoints = 1000'000UL;
    const auto pi = acos(-1) ;
    constexpr auto eps = 1.0e-14;

    const auto X = linspace(0., 2 * pi, npoints);
    const decltype(X) res = sin(X) * sin(X) + cos(X) * cos(X) - 1;
    auto is_bad = [=](double x){ return fabs(x) > eps; };

    if (any_of(res, is_bad )) {
        cout << "There relation was found to be invalid.\n";
    } else {
        cout << "The relation was found to be valid for all points.\n";
    }
}
```

The code now starts to resemble the python code quite a bit. The brevity and elegance of python syntax is a virtue, as (when?) it makes code easy to read. This should not be pursued at the cost of the extra guarantees provided by the C++ code above. For instance, there is nothing in the above python code that prevents accidental statements like `np.pi = 0`. In a large project with hundreds or even thousands of files, implementing thousands of functions to accomplish complex tasks, ensuring that none of the components introduce errors like this or the less drastic and more insiduous `np.pi = 3.1`, is very hard. If a programming language enforces stricter ownership regulations and constant-ness checks, it carries a lot of that burden for us and prevents a lot of errors. So, despite slight verbosity of our `constexpr` and its friends, as C++ programmers, we should keep them. They add value to our programs.

Let's now discuss this curious line in the code.

```cpp
    auto is_bad = [=](double x){ return fabs(x) > eps; };
```

It made the validity checking `if` statement very readable. You recognize the right hand side as a lambda expression, representing the mapping $x \mapsto (abs(x) > eps)$. Previously we had used such lambda expressions directly when we needed locally defined functions. Lambda functions don't need to have names in order to be used, but they are not "Voldemort"-functions. Nothing prevents us from giving them descriptive names (as we have done here) when that makes code clearer. Once we create a variable `is_bad` storing our lambda, we can use `is_bad` as a function, e.g., `is_bad(0.44)`. If a `double` value `val` from our results array is mapped to `true` by this mapping, it proves that our trigonometric relation does not hold.

The code in the body of the above lambda, `return std::fabs(x) > eps;`, should be understandable. But, the lambda function, like any other function, defines its own scope for the variables.

Visually, the braces enclosing the body of this lambda are inside the block scope defined by another function. The scope defined by the body of the lambda is nevertheless sequestered from the surrounding scope. Lambda functions are as close as we can get to nested function definitions in C++, because normal functions can not be defined in block scope.

Our lambda needs `x` and `eps` to be able to calculate its answer. `x` is the argument it receives. But what is `eps` in the lambda function body `return std::fabs(x) > eps;` ? We want to refer to a variable called `eps`, defined outside the lambda function, in the body of the lambda. To do this, we have to "capture" that value. This is a crucial way in which lambdas differ from ordinary functions. Lambdas can be defined in block scope, e.g., inside another function, another lambda, within the block in the `if` or `else` parts of a conditional statement and so on. We can't do these things with ordinary functions. But since they can be defined within another function, perhaps inside a loop or a block covered by a branch, it becomes interesting to ask whether it should have any awareness of temporary variables defined in the block surrounding their declaration. By default, it doesn't. The lambda ignores all block scope variables surrounding its definition, and simply works with its own local variables and input arguments. The square brackets `[]` which indicate the beginning of a lambda function, are called "capture" brackets. One could explicitly write the names of all variables from the external context one wishes to use in the lambda in those brackets. In our case, something like `[eps](double x){ return std::fabs(x) > eps; }`. With that, we would request read-only access to a variable named `eps` inside the lambda function, which should be visible in the scope where the lambda itself is defined. The notation we have used instead, i.e., `[=]`, requests read-only access to any variables from the surrounding scope inside the lambda.

`std::ranges::any_of` is one of the many algorithms implemented in the C++ standard library. It needs as its argument a range of values and a predicate (predicate is a function, or something which behaves like a function, mapping a certain type of input values to a boolean value). `std::ranges::any_of` tests each element in the input range with the predicate, until one of them returns true. If nothing in the range satisfies the predicate, `any_of` returns false. What is a range though?

A `range` is anything which has a `std::begin` and a `std::end`. The ranges library is one of the 4 most important new features of C++20. `std::valarray` is a range. So are all other standard library containers, `std::string`s etc. When you write your own classes, we will see how easily you can make them usable with these algorithms from the ranges library. But C++ has had an `std::any_of` function all the way since C++11. That function needed three input parameters, a starting point, an ending point and a predicate. We could have written our validity check like using that function like this: `if (std::any_of(std::begin(res), std::end(res), is_bad)) ...`
The version using `std::any_of` will continue to work, of course, but it is likely that the ranges version, `std::ranges::any_of` will find increasing use in the coming years.

### 3.1.2 Using more ranges and compositions

Recall that at the beginning of this chapter, we formulated the problem as follows: "we generate a sequence of values in a given range, and for each generated value, we evaluate whether the relation holds. If for any generated value of $x$ the left and right sides of the equations are not the same, we would have proven that the relation does not hold". That way of formulating the problem guided our thinking when we wrote the `for` and `while` loop versions of our solutions.

Another way to think about it would be:

1. Let there be a sequence $S = \{0, 1, 2, ..., N-1\}$
2. Transform $S$ into a sequence of equally spaced real values between $x_{min}$ and $x_{max}$, by using the mapping $n \mapsto (x_{min} + \frac{(x_{max}-x_{min})n}{N})$
3. Transform the resulting sequence with the mapping $x \mapsto sin^2(x) + cos^2(x) - 1$
4. Check if any element of the resulting range satisfy the predicate $x \mapsto (abs(x) > \epsilon)$

If we formulate our solution in this way, we see that the problem consists of smaller sub-problems which can be individually solved using small pieces of code which are individually easy to check. We have already seen the last item in the form `std::ranges::any_of(res, is_bad);`. Now let's look at the rest of the items in that list.

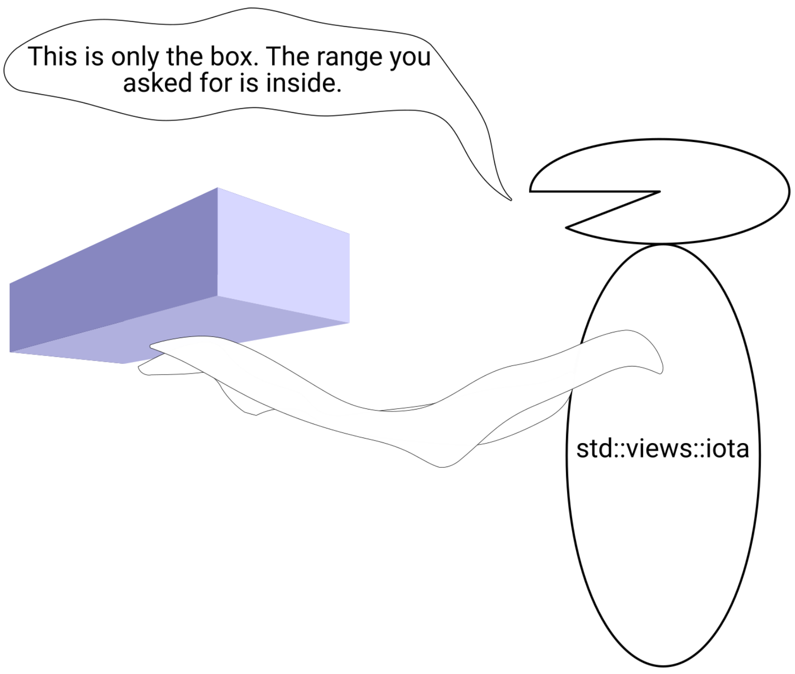

Figure 3.1: `std::view::iota` "effectively" gives us a sequence of integers in a specified finite or infinite range. It does not try to fill any container with the values of the integers. It just gives us "something" that works as a proxy for that sequence we want, like the box containing the sheep in "The Little Prince" (Antoine de Saint-Exupéry).

### 3.1.3 Creating sequences

The first problem, generating a sequence of integers within some bounds is extremely general, and in C++ standard library, the function that provides this is `std::views::iota`. `std::views::iota(4)` gives us the infinite sequence $4, 5, 6, 7, ...$ with no upper bound. `std::views::iota(4,10)` gives us precisely $4, 5, 6, 7, 8, 9$. What does it really mean that it "gives" an infinite sequence of integers? Where are they stored?

The idea behind constructs like `std::views::iota` is that when we want to work with sequential values like that, they don't need to be present in the computer's memory. That sequence is an idea:

- There is a start to the sequence that we can identify.
- We can focus our attention on one "current" element, and retrieve its value
- We can move our focus to the next element along the sequence
- There is a way to tell if we have reached the end of the sequence

If we calculated all the elements of a finite sequence and stored it in a long list, and returned that list from such a sequence generator, we would satisfy all those above requirements. But, we could also return just a "box" (see Figure. 3.1), not actually containing pre-calculated values, but imbued with some properties which help it satisfy the above requirements. If we store the return value of `std::views::iota` somewhere, like `auto box = std::views::iota(1,10);`, the box is a "stand-in" for the concept of a sequence of integers in the half open range $[1, 10)$. Among other properties, the box has something called an "iterator" associated with it. The iterator is like our finger pointing to one element along the sequence. It can be moved to the next element of the sequence by incrementing it with the `++` operator like an integer. To obtain the value of the actual element of the sequence, we need to "dereference" the iterator. So, to find the start of the range the box is supposed to represent, we use something like `auto pos = std::begin(box)`, where `pos` will be an iterator. Dereferencing iterators has a fairly uniform syntax across lots of C++ types. It looks like `*pos`. So, if `pos` is an iterator (your finger) along a sequence, `*pos` is the element of the sequence it is pointing at. Two iterators are equal if they point to the same element. `std::end()` usually returns an iterator which points to a fictitious element just past the end of the sequence, so that we can stop when the iterator compares equal to this end iterator. The loop over the sequence would then look like this:

```
for (auto it = std::begin(seq); it != std::end(seq); ++it)
    do_something();
```

When we discuss C++ classes, you will learn that when you create a new data type as a C++ class, you have complete control over the meaning of operators acting on that data type. It's called operator overloading. If you create a new data type called `FuzzyNumber`, you can make your own rules about what happens when you use operators like `+`, `-`, `&` ... with objects of that type.

```
FuzzyNumber a = somefunc(1);
FuzzyNumber b = somefunc(2);
auto c = 2 * a * a + 3 * a * b + b * b;
```

The author of the `FuzzyNumber` type can decide what should happen when an expression like `a * a` or `2 * a` is evaluated. Here the multiplication by the scalar 2 can have a completely different effect than the multiplication with another object of the same type. It's entirely up to the person writing that class.

The iterators are objects of specially designed classes with very specific properties. The iterators we get from our `box` above could, for instance, have the property that they keep track of the starting value of the sequence and their current position. When you dereference them, e.g., using the prefix unary `*` operator, they actually calculate and return the correct value for their position. The comparison operators, `==`, `!=` etc. are also designed to perform comparisons not only with other iterators, but sometimes also some "sentinel" types. Comparison with the sentinels is usually written in a way so that some logical criterion is satisfied. For instance, if we were generating an infinite sequence, there is no place to point with our fingers as the end of the sequence. So, `std::end(box)` in such a case will return a sentinel object, such that for any given iterator `it` the comparison `it != std::end(box)` returns true. This way, a `for` loop over such a range can continue indefinitely, we can generate as many numbers from the sequence as we want and still not reach the end.

### 3.1.4 Element wise operations on a sequence

The core of the strategy formulated in Section 3.1.2 was transformations on a given sequence. If we have one sequence, $S_1 = \{s_0, s_1, s_2...\}$, we want to transform it to another sequence $S_2 = \{f(s_0), f(s_1), f(s_2)...\}$ by using a mapping $x \mapsto f(x)$. $S_2$ can be written as application of a transformation on $S_1$, i.e.,

$$\begin{aligned} S_2 &= T_{12}(S_1) \\ &= T(x \mapsto f(x))(S_1) \end{aligned}$$

where $T(x \mapsto f(x))$ in the second line is an element wise transformation which applies $x \mapsto f(x)$ to its inputs. $T$ itself can be considered as a higher order function (a function with at least one input which is another function) representing a general element wise transformation over an input sequence. The higher order function needs as its arguments, a range of elements it will process, and a specific mapping it will apply to each element. If we apply another transformation to the resulting sequence,

$$\begin{aligned} S_3 &= T_{23}(S_2) \\ &= T_{23} \otimes T_{12}(S_1) \\ &= T(x \mapsto g(x)) \otimes T(x \mapsto f(x))(S_1) \\ &= T(x \mapsto g(f(x)))S_1 \end{aligned} \tag{3.1}$$

For a moment, let's focus on the second last line in the above equation. The resultant sequence $S_3$ is obtained by applying a series of transformations on the input $S_1$, which are applied in the right to left order. Notationally, we could make it more intuitive by writing it something like this:

$$\begin{aligned} S_3 &= T(x \mapsto g(x)) \otimes T(x \mapsto f(x))(S_1) \\ &= S_1 \mid T(x \mapsto f(x)) \mid T(x \mapsto g(x)) \end{aligned}$$

This means, we take the input sequence and "pipe" it through a transformation $x \mapsto f(x)$ and then pipe it through another transformation $x \mapsto g(x)$. For people used to reading from left to right, this is more intuitive. It is also familiar to people who have used the command line or shell in UNIX like operating systems. Something like:

```
find . -name "*.o" | xargs rm -f
```

i.e., find all files ending with a `.o` suffix and send the resulting sequence of filenames as arguments to the command `rm -f`. The shell consists of a large number of simple utilities, which do one thing and do it well (e.g., `find`, `rm`). Crucially, we have the ability to "pipe" the output generated in one command to another, i.e., to connect the output of one command to the input of another. The result of the second command can be piped again to another another and so on. Such composability exponentially amplifies the power of the small utilities of the UNIX like shells, for instance in Linux and Mac OS. If we could break a programming task into small components, which could be implemented robustly and reliably with no room for error, and if we could compose them in a UNIX pipe like syntax, we would have a very powerful tool in our hands. Well, we can.

```
1  // examples/trig_views.cc
2  #include <iostream>
3  #include <ranges>
4  #include <algorithm>
5  #include <cmath>
6  #include <limits>
7
8  int main()
9  {
10     namespace sr = std::ranges;
11     namespace sv = std::views;
12     const auto pi = std::acos(-1);
13     constexpr auto npoints = 10'000'000UL;
14     constexpr auto eps = 100 * std::numeric_limits<double>::epsilon();
15     auto T12 = [=](auto idx){ return std::lerp(0., 2*pi, idx * 1.0 / npoints); };
16     auto T23 = [ ](auto x)  { return sin(x) * sin(x) + cos(x) * cos(x) - 1.0;  };
17     auto is_bad = [=](double x){ return std::fabs(x) > eps; };
18
19     auto res = sv::iota(0UL, npoints) | sv::transform(T12) | sv::transform(T23);
20     if (sr::any_of(res, is_bad) ) {
21         std::cerr << "There is at least one input for which the relation does not hold.\n"
22                   << "They are...\n";
23         for (auto bad_res : res | sv::filter(is_bad)) {
24             std::cerr << bad_res << "\n";
25         }
26     } else {
27         std::cout << "The relation holds for all inputs\n";
28     }
29 }
```

The syntax of the so called view adaptors in C++20, which we have used above, is modelled after the pipe like composition mechanism we just discussed. `std::views::transform(lambdafunction)` creates a "transformation" like the ones we have discussed above. `std::views::filter` *filters* the input sequence based on a predicate and produces an output sequence containing only the inputs for which the predicate is true. In line 23, we show how to use `filter` and iterate over a resulting range.

It is also interesting that the line 19, where the variable `res` is defined does not actually perform any calculations whatsoever on the sequence! It is just a definition of a composite sequence. It is only when we search through the sequence in the `any_of` function that the actual mappings we have used to create our composite sequence are executed. Because of the meaning of "any of", the evaluation need not go through the entire sequence. As soon as one example is found for which the trigonometric relation does not hold, we can stop scanning the rest of the sequence. That is exactly what `std::views::any_of` does.

## Exercise 27:

Our trigonometric relation is after all, true, so that we don't get to see that `sr::any_of` above would indeed stop processing the input sequence upon the first `true` outcome from the predicate. Therefore, let's take a bogus trigonometric relation that is not always true: $sin^2(x) < 0.99$. It's mostly true, but there are some values of $x$ for which this is false.

The program `examples/trig_views2.cc` checks this incorrect relation using the same procedure as the previous example. Here we have also rigged the lambda functions, so that they first print their input and output values before returning their results. Study the program, compile and run it, and compare the output with what you expect. Notice that the first time the trigonometry lambda function is executed is after we enter the `std::views::any_of` part. Also, notice that it does not continue with the entire sequence of numbers, but stops at the first point for which the relation does not hold. Notice also that the output from the two lambda functions, one implementing the map from integers to a value in the range 0 to $2\pi$ and the other a mapping from $x$ to $sin^2(x) - 0.99$ are staggered. This reflects that the two transformations we have composed are not executed in two passes, but like a single pass over the sequence with a composite transformation as in the last line of Equation 3.1.

Let's play with the views a little more. The above pipe like syntax of composite transformations is not only useful for generated sequences from `std::views::iota`. Any container of objects of any kind can be used as the input, because they all have associated iterator types with the right properties. In the following, we take a list of strings, and sort it in different ways before printing the top 3 elements:

```cpp
// examples/sort_various.cc
#include <iostream>
#include <string>
#include <ranges>
#include <algorithm>
#include <vector>

auto main() -> int
{
    std::vector<std::string> L{ "Magpies", "are", "birds", "of", "the", "Corvidae", "family" };
    namespace sr = std::ranges;
    namespace sv = std::views;
    std::cout << "Top 3 after alphabetical sorting...\n";
    sr::sort(L);
    for (auto el : L | sv::take(3)) std::cout << el << "\n";
    std::cout << "Top 3 after alphabetical sorting in reverse order ...\n";
    // No sorting required, because we already did that,
    // and we can just look at it in the reverse order
    for (auto el : L | sv::reverse | sv::take(3)) std::cout << el << "\n";
    std::cout << "Top 3 after sorting by string length ...\n";
    sr::sort(L, [](auto a, auto b) { return a.size() < b.size();} );
    for (auto el : L | sv::take(3)) std::cout << el << "\n";
}
```

And now, let's try to do something similar, but by using the command line arguments as our input strings. Recall that to receive arguments from the command line, `main` is declared as `auto main(int argc, char *argv[]) -> int`. This has a long history, and in particular it is shared with C (In the older form `int main(int argc, char *argv[])`). The arguments we receive in our C++ program are therefore not C++ standard library strings, nor is the list of arguments a proper C++ container of any kind. There are many ways to convert the input arguments into an `std::vector<std::string>` for convenient processing with C++ algorithms, e.g.,

```cpp
std::vector<std::string> inp;
std::copy_n(argv, argc, std::back_inserter(inp));
```

But, we don't really need to create real C++ strings here. The command line is not going anywhere while the program is executing, so we can use what is known as a `std::string_view`. A `string_view` is like a string, except that it does not own its contents. If there is a string somewhere, we can create a `string_view`, which is a lighter weight object, and work with it. This is safe as long as the original `string` which we are viewing with the `string_view` is alive when we use the `string_view`. If we only have a C-style string, we can still convert it to a `string_view`, as long as the C-style string is alive when we use the `string_view`. The `string_view` converts the C-style string into a nice package for convenient processing using C++ algorithms, without doing any dynamic memory allocation. To make a vector of `string_views`, you would replace the type of the `inp` variable above, and use `string_view` instead. But even the `vector` is not really necessary. A vector is a full fledged container in C++, which owns its contents. Like `string_view` is to `string`, there is something called a `std::span`, which has the same relation to a `vector`.

The usage of `span` with the standard algorithms is demonstrated in the following program, where we print the alphabetically first and last word among all words entered on the command line.

```
 1  // examples/views_and_span.cc
 2  #include <iostream>
 3  #include <span>
 4  #include <ranges>
 5  #include <algorithm>
 6  #include <string>
 7  #include <iomanip>
 8
 9  auto main(int argc, char * argv[]) -> int
10  {
11      std::span args(argv, argc);
12      // reverse order, because span expects a "count" as the second argument
13
14      auto str = [](auto inp) -> std::string_view { return inp; };
15      if (argc < 2) {
16          std::cout << "Usage:\n"
17                    << argv[0] << " some strings in the command line\n";
18          return 1;
19      }
20      namespace sr = std::ranges;
21      namespace sv = std::views;
22      auto [first, last] = sr::minmax( args | sv::drop(1) | sv::transform(str) );
23
24      std::cout << "Alphabetically first and last strings in your input are "
25                << std::quoted(first) << " and " << std::quoted(last) << "\n";
26  }
27
```

In line 14, we create a lambda function to convert the C-style strings in `argv` into `string_view`s. It seems like we are not doing any work, and just returning the input. But the trick is, we explicitly specify the output type of that lambda with the `-> std::string_view` syntax. If the variable `inp` which comes in as the input is a `string` or a C-style string, it can be automatically converted to a `string_view`. When a function must return a value of type `ReturnType`, but we write a return statement `return something` where `something` is of type `AnotherType`, the compiler will try to insert code to make an automatic conversion from `AnotherType` to `ReturnType`. It is as if we had written something like

```
auto func(InputType inp) -> ReturnType
{
    AnotherType something = somevalue();
    // Work
    ReturnType rightkindofsomething{ something };
    return rightkindofsomething;
}
```

If that makes sense, all is good. If not, it would generate a compiler error. For instance, an `int` value can be automatically converted into (used to create ) a `double` , but a `string` can not be. In line 22, we are calling the `std::ranges::minmax` algorithm, which returns a pair of values. We give two names to the two parts of pair. The input to the `minmax` function takes a range. We give it a range constructed out of the command line arguments as a `span` , after being piped through a "drop view" to skip the first element ( `argv[0]` is the name of the program, and not a real argument to it), and again through a transformation which converts the elements of the span from C-style strings to `string_views` .

## 3.2 Exercises

### Exercise 28:

Modify the program `examples/views_and_span.cc` so that it prints the words you type in the command line, but only up to the first word which ends with "ing". You will need a view adaptor called `take_while(Pred)` which takes a predicate as its argument. Hint: Write the predicate as a lambda. Another hint: to check if a string view ends with a certain substring "abc", you can use the function `ends_with` as in, `word.ends_with("abc")` .

### Exercise 29:

Modify the trigonometry exercise example to numerically verify $sin(x + y) = sin(x)cos(y) + cos(x)sin(y)$. Suggestion:

- Let's say, we want to have $nx = 10000$ points along $x$ and $ny = 10000$ along $y$. We can start with a simple sequence of integers from 0 till $nx * ny$.
- Map each integer `i` in the range to a pair of integers `std::pair<int, int>` by performing integer division operations:

```cpp
auto ix = i / ny; // quotient of division of i by ny
auto iy = i % ny; // remainder after division of i by ny
```

or more compactly,

```cpp
auto [ix, iy] = std::div(i, ny);
// ix and iy will be quotient and remainder
```

- Map the pair of integers to a pair of `double` in a manner similar to how we mapped a single integer before. The goal is to create from $ix, iy$ real numbers $x, y$ which are in the range $(0, 2\pi]$.
- Verify the relation for $x$ and $y$.
- If the relation does not hold for any of the integer indexes, it is not valid.

### Exercise 30:

**Sieve of Eratosthenes** Sieve of Eratosthenes is a very old, but rather fast method to find all prime numbers up to a given number. Check out the Wikipedia page in the link to see how it works. Implement the sieve in C++. Suggestion:

1. Create a `std::vector<bool>` to represent a true/false value for each integer from 0 to `N=1000UL`. Let's call it `isprime`. Every position starts out with a value `true`
2. Set the values for the integers 0 and 1 as false.
3. Create a "current" position and initialise it to 2
4. In a `while` loop, repeat the following as long as `current * current < N`.
   (a) Loop from `2*current` till `N`, in steps of `current`, and mark those positions in `isprime` as `false`
   (b) Increment `current` at least once, until `isprime[current]` is true or `current` becomes too large
5. At this point, all positions on `isprime` which still hold a `true` are primes. Make a lambda function which captures `isprime`, takes an `unsigned long index` as an argument, and returns `isprime[index]`. This can be used to test if an integer in the range 0 to $N$ is a prime.
6. Create a sequence of primes:
   `auto primes= sv::iota(0, N) | sv::filter(yourlambda);`
   Loop over `primes` in a range based for loop, or count how many there are with `sr::count(primes)`

Is it really enough to run the outer loop only until $\sqrt{N}$? Why are we using `current * current < N` instead of `current < sqrt(N)`?

## Exercise 31:

**Caesar shift** Caesar shift or Caesar cipher is an extremely simple (and extremely weak) encryption techniques, which was used once upon a time to send secret messages. It has no value as an encryption technique today, but it does make a simple and fun programming exercise. Your task is to make a function which takes as its input a text as a `std::string` and an integer shift value. The function must transform each character in the input text by shifting it along the alphabet by the given shift value. For instance, if shift is 1, each character should be replaced by the next character in the alphabet. When we run out of characters, we start again from 'A'. Assume that we are using the English language. Leave characters other than 'A'–'Z' and 'a'–'z' unchanged. You can write this entirely on your own. If you want, you could also start from a partial implementation in `examples/caesar.cc`.