PB161 Programming in C++

Petr Ročkai

Part A: Preliminaries

This document is a collection of exercises and commented source code examples. All the sources are also available as separate files that you can edit and compile (we will refer to these files as the source bundle). Additionally, this section contains the rules and general guidelines that apply to the course as a whole.
The latest version of this document along with the source bundle is available both in the study materials in IS and on the student server `aisa`:

- `https://is.muni.cz/auth/el/fi/jaro2022/PB161/um/` – a PDF in `pb161.seminar.pdf` and the source bundle in directories `01` through `12`, `t1` through `t3` and `sol` – use the 'download as ZIP' option in the sidebar to get entire directories in one go,
- log into `aisa` using `ssh` or `putty`, run `pb161 update`, then look under `~/pb161` (this chapter is in subdirectory `00`).

We will update the files as needed, to correct mistakes or include additional material. On `aisa`, running `pb161 update` at any time will update your working copies, taking care not to overwrite your changes. It will also tell you which files have been updated.

Part A.1: Course Overview

Welcome to PB161 Programming in C++. The course consists of lectures, weekly seminars, programming tasks, and a programming test (exam) at the end. Since this is a programming subject, most of the coursework – and grading – will center around actual programming. You will write a few tiny programs (15-20 minutes each) every week, a few bigger programs (though still small, at a couple hundred lines each) during the semester and there will be a simple (but strict) programming test at the end (in the exam period) that you have to pass.
Writing programs is hard and consequently, this course will also be hard – you absolutely need to put in effort to pass the subject. Hopefully, you will have learned something by the end of it.
Further details on the organisation of this course are in this directory or, if you are reading the PDF, in the following sections:

- `2_grading.txt` – what is graded and how; what you need to pass,
- `3_tasks.txt` – general guidelines that govern assignments,
- `4_reviews.txt` – writing and receiving peer reviews,
- `5_quality.txt` – code quality guidelines,
- `6_exam.txt` – about the final programming exam,
- `7_plagiarism.txt` – about cheating.

A.1.1 Topics The semester is organized as three four-week blocks. Each week corresponds to a single chapter in this document, for a total of 12 chapters. The study materials for each week are in directories `01` through `12` (one per week). Start by reading the introduction (`00_intro.txt` in the 'source' version). Each block is followed by a set of bigger tasks, in directories `t1` through `t3`. Again, start by reading the introduction (`00_intro.txt`) in there.

block		topic	lect.	prep	end
1	1.	semantics 1, classes, …	15.2.	19.2.	
	2.	semantics 2, lambdas, …	22.2.	26.2.	
	3.	containers, algorithms	1.3.	5.3.	
	4.	overloading, types, …	8.3.	12.3.	19.3.
2	5.	operators, IO	15.3.	19.3.	
	6.	RAII & exceptions	22.3.	26.3.	
	7.	memory, `unique_ptr`	29.3.	2.4.	
	8.	OOP	5.4.	9.4.	16.4.
3	9.	templates 1	12.4.	16.4.	
	10.	templates 2	19.4.	23.4.	
	11.	iterators	26.4.	30.4.	
	12.	review	3.5.	7.5.	14.5.
–	13.	C++20	10.5.	–	

Part A.2: Grading Overview

There is a number of ways to obtain points:

max	what	notes
36pt	tasks	3pt each, max 12
12pt	code review	2pt each, max 6
12pt	seminars	1pt/week: prep exercises + attendance
6pt	sets	2pt extra for each set with 4+ points
3pt	peer review	0.3pt per review, 10 reviews max
3pt	activity	2pt + 1pt
18pt	exam	3pt + 4pt + 5pt + 6pt
90pt	total	72pt semester + 18pt exam

The semester maximum is 72 point. You need 40 points to pass the semester (failing to do this means grade X). To pass the exam, you need 8 points on the exam. The final grade is then awarded as follows:

- [68, 90] points → A
- [63, 68) points → B
- [58, 63) points → C
- [53, 58) points → D
- [48, 53) points → E
- [40, 72] points → Z (if your ending type is 'z').

A.2.1 Seminars The preparatory exercises are to be worked out in the corresponding week of the semester, with a deadline every Saturday at midnight (see the semester overview in the previous section, column 'prep' for exact dates). Only the enclosed tests are executed upon submission, and the result should appear in the corresponding notepad within 5-10 minutes.
Together with seminar attendance, these exercises are worth a significant fraction of what you need to pass the semester. You are awarded a point for each unit (week) in which you submit at least 3 preparatory exercises and attend[1] the corresponding seminar in the following week.

[1] If you cannot attend the seminar and file the requisite paperwork excusing you from attendance, you can instead show original solutions for 3 of the r-type exercises from the affected unit to your seminar tutor. If they are satisfactory, you will get a point as if you have attended the seminar. Especially in case of an extended illness, you can also ask for an extension of affected prep exercise submission deadlines. Be sure to do this as soon as you reasonably can.

In addition to gaining points, you may also get feedback and a chance to discuss the submitted solutions in the seminar.
The activity points are likewise awarded in the seminar, in this case for demonstrating the solution of one of the r-type exercises for the week. These will be 'live' demos: you should solve the exercise on the spot, without looking at a prepared solution (whether your own or the reference one). You can do them from your own computer (using `pb161 beamer` on `aisa`) or you can go to the front and use the teacher's computer. In any case, in addition to writing down a solution, you will be expected to comment what and why you are doing. On the other hand, the teacher will help you out if you get stuck in a blind alley.
You can do this twice during the semester: the first instance counts as 2 points, the second as 1. If nobody else is interested, you can also volunteer to do an exercise 'for free' (that is, if you already have your 3 activity points).

A.2.2 **Programming Tasks** In each block, there are 4 tasks of increasing difficulty (both within and between blocks). There will be 8 deadlines for each block, spread out over 6 weeks (there's a deadline once a week for the first month, then twice a week for another 2 weeks). Most deadlines are on Saturday (same as weekly exercises), the 2 extras are on Wednesdays. Submissions open 7 days before the first verity deadline (that is 19.2., 26.3. and 23.4. – submissions done before these dates will not be evaluated).
Each deadline gives you one chance to pass the automated test suite. It does not matter when you pass any given task, but the test suite is strictly binary: you either pass or you fail. More details and guidelines are in `3_tasks.txt`.
Verity tests continue to run after the last deadline: you can finish tasks and still get results after they expire, but you will not get any points for doing so. However, it does unlock peer reviews for the given task. The deadline schedule is as follows:

week	set 1		set 2		set 3	
	Wed	Sat	Wed	Sat	Wed	Sat
1		26.2.				
2		5.3.				
3		12.3.				
4		19.3.				
5	23.3.	26.3.		26.3.		
6	30.3.	2.4.		2.4.		
7				9.4.		
8				16.4.		
9			20.4.	23.4.		23.4.
10			27.4.	30.4.		30.4.
11						7.5.
12						14.5.
13					18.5.	21.5.
14					25.5.	28.5.

A.2.3 **Code Quality** We should all strive to always write clean, readable and well-designed code. Of course, this takes more time (often a lot more time) than just going with the first thing that sort of works.
You will be able to submit six of your task solutions for teacher review. Which assignments you choose to submit is up to you. Make sure that you put in adequate effort to make the code as clean and nice as you possibly can. The code must pass verity tests (within the designated deadlines).
The last day on which you can file a request for (teacher) review is 3.4. for T.1, 1.5. for T.2 and 29.5. for T.3 (i.e. the day after the last test deadline). Your teacher will have 10 days to complete the review.

- reviewer deadline: request + 10 days (i.e. 13.4., 25.5. and 8.6.)
- resubmit until 17.4., 15.5. and 12.6.
- earlier request → more time to fix stuff

With each review, you get a grade which corresponds to the following point value:

- A = 2 points,
- B = 1 point,
- C = no points.

The detailed criteria for individual grades (and for code quality in general) are provided in `5_quality.txt`. If your grade is not A, your tutor will point out what you need to improve.
You then get a chance (once) to improve your code and submit the task for a second round of review. If your code has sufficiently improved, you can get the next better grade (i.e. B if your first grade was C and A if your first grade was B).

A.2.4 **Peer Review** Reading code is an important skill – sometimes more so than writing it. While the space to practice reading code in this subject is limited, you will be able to earn a few points doing just that. The rules for peer review are quite different from those for teacher reviews above:

- you can submit any code (even completely broken) for peer review,
- to write a review for any given submission, you must have already passed the respective assignment yourself,
- there are no specific deadlines for requesting or providing peer reviews.

It is okay to point out correctness problems during peer reviews, with the expectation that this might help the recipient pass the assignment. This is the only allowed form of cooperation (more on that below).

A.2.5 **Examples** A lot more work is available that what you need to do, even for an A. We do not expect you to solve all the exercises nor tasks – pick a subset you like, but be sure to spread the work through the entire semester (there are very significant bonuses for doing it this way). To give you an idea, there are some point calculations:

- below tables give achievable grades by passed task count,
- the specific grade in the range depends on the outcome of your exam (8pt – 18pt),
- 4 tasks are the minimum to pass,
  - all 3 sets must be covered,
  - the 2 that are alone in their set must get 1pt or better review,
- 5 tasks are more or less the expected count,
- solving 6 tasks (i.e. half of them) allows considerable leeway in other semester work,
- 6 tasks is also the minimum to get an A.

Maxing out all the optional work gives you this:

tasks	pts	rev	other	total	grade
3	9	6	24	39	X
4	12	8	24	44	E-C
5	15	10	24	49	D-B
6	18	12	24	54	C-A
7	21	12	24	57	B-A
8	24	12	24	60	A

If you forfeit 2 weeks of seminars and the second 'activity' point and get, on average, a B on teacher reviews, these are your prospects (you do need to distribute work across task sets, to get the 2 point bonus on all of them):

tasks	pts	rev	other	total	grade
4	12	4	21	37	X
5	15	5	21	41	E-C
6	18	6	21	45	D-B
7	21	6	21	48	D-B
8	24	6	21	51	C-A
9	27	6	21	54	C-A

Part A.3: Task Sets

The general principles outlined here apply to all tasks. The first and most important rule is, use common sense – the specifications are not exhaustive and sometimes leave room for different interpretations. Do your best to apply the most sensible one. Do not try to find loopholes (all you are likely to get is failed tests). Technically correct is not the best kind of correct.
Think about pre- and postconditions. Aim for weakest preconditions that still allow you to guarantee the postconditions required by the assignment. If your preconditions are too strong (i.e. you disallow inputs that are not ruled out by the spec) you will likely fail the tests.
Do not print anything that you are not specifically directed to. Programs which print garbage (i.e. anything that wasn't specified) will fail tests.
You can use the standard C++ library. External libraries or header files are not allowed, unless specified as part of the assignment. Make sure that your classes and methods use the correct spelling, and that you accept and/or return the correct types. In most cases, either the 'syntax' or the 'sanity' test suite will catch problems of this kind, but we cannot guarantee that it always will – do not rely on it.
If you don't get everything right the first time around, do not despair. The expectation is that most of the time, you will pass in the second or third verity run (especially if you test your program carefully). If you strongly disagree with a test outcome and you believe you adhered to the specification and resolved any ambiguities in a sensible fashion, please raise the issue in the discussion forum.

A.3.1 Submitting Solutions The easiest way to submit, for instance, a solution to the task `t1_cellular` is this:

```
$ ssh aisa.fi.muni.cz
$ cd ~/pb161/t1
… edit files until satisfied …
$ pb161 submit t1_cellular
```

NB. Only the files listed in the assignment will be submitted and evaluated. Please put your entire solution into existing files (or into files you are instructed to create).
You can check the status of your submissions by issuing the following command:

```
$ pb161 status
```

In case you already submitted a solution, but later changed it, you can see the differences between your most recent submitted version and your current version by issuing:

```
$ pb161 diff t1_cellular
```

The lines starting with `-` have been removed since the submission, those with `+` have been added and those with neither are common to both versions.

A.3.2 Compilation To compile and test your solution, use the `make` command: each `tX` directory has a `makefile` in it. Typing `make cellular` in this directory will first compile your solution into an executable binary and then run `clang-tidy`, any tests you may have written, and `valgrind`. If you want to work on your own computer instead of `aisa`, you need to figure out the settings yourself. The `makefile` will tell you which compiler we use and how we invoke it.

A.3.3 Evaluation There are three sets of automated tests which are executed on the solutions you submit. The first set is called 'syntax' and runs immediately after you submit. Only 2 checks are performed: the code compiles and it passes `clang-tidy`.
The next step is 'sanity' and runs every midnight and noon. Its main role is to check that your program meets basic semantic requirements, e.g. that it recognizes correct inputs and produces correctly formatted outputs. The 'sanity' test suite is for your information only and does not guarantee that your solution will be accepted. The 'sanity' test suite is only executed if you passed 'syntax'.
The 'verity' test suite covers most of the specified functionality and runs once or twice a week (the exact schedule is in the previous section). If you pass the verity suite, the assignment is considered complete and you are awarded the points. The verity suite will not run unless the code passes 'sanity' (with the exceptions specified in the task descriptions). Please note that any memory errors (including memory leaks, as reported by `valgrind`) will cause 'verity' to fail.
Only the most recent submission is evaluated, and each submission is evaluated at most once in the 'sanity' and once in the 'verity' mode. You will find your latest evaluation results in the IS in notepads (one notepad per task).

Part A.4: Peer Reviews

You can optionally participate in peer reviews, both as a reviewer and as a review recipient. While reviewers get points for their effort, the recipients do not – instead, they get (hopefully) useful information.

A.4.1 Requesting Reviews If you would like to have your code reviewed, you can issue the following command:

```
$ pb161 review --request t1_cellular
```

Substitute other programming tasks for `t1_cellular` as appropriate. You can request a peer review on a task which you did not pass yet. You may get up to 3 reviews for any given request. The reviewer will work with the submission that was current at the time you have created the request. Make sure you submit the code you want reviewed before requesting the review.
The `pb161 update` command will indicate whether someone reviewed your code, by printing a line of the form `A reviews/t1_cellular.by.xlogin`. To read the review, look at the files in `~/pb161/reviews/t1_cellular.by.xlogin` – you will find a copy of your submitted sources along with comments provided by the reviewer. After you read your review, you should write a few sentences for the reviewer into `note.txt` in the review directory (please wrap lines to 80 columns) and then run:

```
$ pb161 review --accept 100
```

Instead of 100, you can use a smaller number, indicating what percentage of the points the reviewer deserves for their job. Please make sure that you grade the review honestly – the reviews will be screened for abuse and depending on the type of misconduct, one or both parties will be punished.
To request a review from a teacher (as opposed to peer review), add `--teacher` to the command:

```
$ pb161 review --request t1_cellular --teacher
```

The output from `pb161 status` will indicate the task submissions for which you have requested a teacher review.

A.4.2 Writing Reviews To participate as a reviewer, start with the following command:

```
$ pb161 review --list
```

You will get a list of review requests for which you are an eligible reviewer. In particular, only tasks that you have already successfully solved will show up. If you like one of the entries, note its number (e.g. 7) and type:

```
$ pb161 review --checkout 7
$ cd ~/pb161/reviews/
$ ls
```

There will be a directory for each of the reviews you agreed to write.

Each directory contains the source code submitted for review, along with further instructions (the file `readme.txt`).
When inserting your comments, please use double `**` to make the comment stand out, like this:

```
/** A short, one-line remark. **/
```

or for longer comments:

```
/** A longer comment, which should be wrapped to 80 columns or
 ** less, and where each line should start with the ** marker.
 ** It is okay to end the comment on the last line of text like
 ** this. **/
```

Part A.5: Code Quality

As mentioned earlier, when you submit your code for teacher review, it will be graded A–C. The following criteria apply.

A.5.1 Vices This is a list of things your code should not do, and the best grade that is possible if they make an appearance. In all cases, only 'nontrivial' instances matter, but unfortunately, there is no obvious line between trivial and nontrivial. Your reviewer's judgment will apply.

- Code duplication: this is a very serious problem, both when the code is literal copy&paste and when there are minor modifications between the copies. Any significant duplication is an automatic C. Code which is highly redundant (multiple implementations of the same concept or pattern, even if not literally copied) is still a problem, including duplication of the standard library. All but trivial cases warrant a C.
- Spaghetti: another common and very serious problem, often paired with the previous. Long functions, an excessive number of local variables, non-obvious side effects which affect control flow down the line, functions which do too many things at once. A minor instance caps your grade at B, anything more than that means a C.
- Bad naming caps your grade at B (if the problem is pervasive, a C is very likely). This includes:
  - meaningless names – single-letter global variables, names which say nothing about the purpose of the thing (`tmp1` through `tmp7`, `tmp`, `tmptmp`, `pom`, `pomoc`, ...),
  - names which are not English nor established placeholders or abbreviations (these are fine: `a`, `b` for arguments of binary operators, `i`, `j` for loop variables, etc.),
  - overlong, completely redundant names for local objects (`first_plus_operand`, `loop_index_variable_1`).
- Inappropriate data types, data structures or algorithms: using building blocks which do not fit the intended purpose makes programs hard to follow and reason about, and often also leads to poor performance. Abuse of strings is especially common. Caps the grade at B (but may contribute strongly to a C).

If your code is free of the above vices, it will get a B or an A, depending on the virtues described below.

A.5.2 Virtues To earn a grade better than C, your code should be free from vices and also demonstrate some of the following virtues.

- Cohesion and orthogonality: each code unit (class, function, ...) does one well-defined thing and has a clear and fitting name. Required for A.
- Good naming: names should be clear, descriptive, respect word categories (based around verbs for functions and nouns for types and variables), be free from spelling or grammatical errors. Names should not be redundant – context matters. The verbosity of a name should be inversely proportional to its scope. Do not repeat established context (no `list::list_length`). Required for A.
- Comments: each non-obvious code unit should have a comment concisely describing what it does and why. Contributes towards an A, but is not required. Comments which establish correctness of the code are especially valued.
- Preconditions: each function should clearly state its preconditions, preferably in executable form (`assert`). Contributes significantly toward an A.

Part A.6: Exam

The raison d'être of this course is to teach you to write correct C++ programs on your own – and the programming test is designed to ensure that this was indeed the outcome for you personally. Of course, we recognize that there is additional pressure when you are programming for an exam. You will get plenty of time to solve the exercises (in relation to their difficulty).
During the exam, it'll be possible to submit the solutions and get back results of a 'sanity' test. The 'sanity' assertions will also be included with the exam source files, for your convenience. There is no requirement to pass `clang-tidy`.
The exam will take place on 31st of May and 2nd of June (Tuesday and Thursday, respectively) starting at 12:30 and will extend until 17:00. Additional attempts will be possible on 14th and 28th of June. You will get a 'yes/no' verity result (without any indication of what went wrong) at 14:00 and 15:30, and full results at 17:00.
You will also get a chance for a 'rehearsal': there are two practice exams in an appendix of this document (directory `pex`). You can work them out and submit them for evaluation, as if they were a real exam. This is strictly optional and will not be graded in any way. It is up to you to complete them within a reasonable time limit (e.g. 3 hours, 2/3 of the official time limit for the real exam) and on your own.
You can submit multiple times for the practice tests and get 'verity' test results immediately, but please keep in mind that this will not be possible at the actual exam.

A.6.1 Evaluation The programming test will be evaluated using automated tests, just like the 12 'major' tasks. Each exercise is evaluated in a binary fashion: you must pass all tests in order to succeed on the given task, in which case you are awarded its points.
If you fail to obtain 8 points (i.e. pass 2 out of the 4 exercises), you get an F and you can try again according to the standard rules for repeating exams.

A.6.2 Materials The exam will be offline. The exam computers will have a standard selection of text editors and development tools[2] installed, in their default configurations. This document (but without exercise solutions) and lecture slides (again without example source code) will be available for reference.

Part A.7: Plagiarism

tl;dr: Please work alone and do not cheat. Cheating is a colossal waste of everyone's time. We would prefer to spend that time on improving the course for everyone. Thank you.
And now for the long version, because sadly, the above is not enough. The goal of this subject is to teach you to write programs in C++ – from understanding the problem, through designing the solution and writing it down in C++. You must be able to do all of this on your own. Teamwork has its place, but it's not in this subject.
You must work out all graded exercises and tasks entirely on your own. Discussing the solution, even in abstract terms, is not permitted. If you do not understand something, ask your tutor privately. If you are caught cheating, 'we have only shared ideas' or even 'we only discussed

[2] You can expect VS Code, Geany, `micro` and `vim` to be present. Qt Creator might be available. CLion has been excluded due to stability issues (we don't want to deal with any more crashes in the middle of an exam than we absolutely have to).

the problem statement' will not hold as a valid defence. If you want to study together, that is fine, and encouraged – there are plenty of ungraded exercises for this purpose. You can discuss those, solve them together, share and compare your solutions and so on.
Please note that you are also responsible for keeping your solutions private. If you only use the `pb161` command on `aisa`, it will make your `~/pb161` directory inaccessible to anyone else (this also applies to school-provided UNIX workstations). Keep it that way. If you work on your solution using other computers, make sure they are secure. Do not publish your solutions anywhere (on the internet or otherwise) and do not share them for any reason. All parties in a copying incident will be treated equally.

A.7.1 Penalties Any points awarded for work that has been shared with another student are voided (including any points from reviews, and any other bonuses they enabled – that is, if you copy a prep exercise and you had been awarded a point for that week's seminar, that point will be revoked). Additionally, following penalties apply:

items copied	penalty	total
1st task	-3pt	-3pt
2nd task	-5pt	-8pt
3rd task	-7pt	-15pt
1st prep ex.	-1pt	-1pt
2nd prep ex.	-2pt	-3pt
3rd prep ex.	-3pt	-6pt
4th prep ex.	-5pt	-11pt

The 'counter' is shared between tasks and exercises (with tasks coming first), so if you copy (or let someone copy, same thing) a task and two exercises, that'd be -3 (1st task) + -2 (2nd exercises) + -3 (3rd exercise), for a total of -8pt.

Part 1: Strings and Classes

Welcome to PB161. If you haven't read the rules and guidelines in Part A (directory `00` in the source bundle), please do so now, before going on.
The exercises this week will look at some of the basics that you have seen in the lecture: strings, dynamic arrays – `std::vector`, classes with methods, and `const` references. These concepts are further explored in the 'demonstrations' (commented examples; please note that these do not replace the lectures – they are meant to be complementary). The corresponding files in the source bundle are named `d?_*.cpp`, i.e. `d1_fibonacci.cpp` through `d4_lemmings.cpp`.

1. `fibonacci` – using `std::vector` (dynamic array)
2. `hamming` – introduction to `std::string`
3. `hero` – introduction to object composition
4. `lemmings` – collections of custom objects

The second part of the study materials for each week gives you a couple of 'elementary' exercises, which you should be able to quickly solve to make sure you understand the concepts from the lecture and from the commented examples above. Sample solutions are in Part S at the end of the PDF, or in the directory `sol` in the source code bundle. Please note that the sample solution is not always the simplest possible – it's fine to take a more roundabout approach. The source files for this section are named `e?_*.cpp`.
This week, the elementary exercises are:

1. `predicates` – properties of lists of numbers
2. `palindrome` – checking that an `std::string` is a palindrome
3. `pascal` – fill in an `std::vector`

The next section has slightly more difficult exercises. These are labelled preparatory, since they exist to let you prepare for the corresponding seminar. You are strongly encouraged to solve at least 3 of them every week (if you submit them by Saturday, you can then gain a point for attending the seminar). The respective source files in the bundle are called `p?_*.cpp`. Note: Discussing and sharing solutions is strictly forbidden – you must solve the exercises on your own. For details, see Part A (directory `00` in the source bundle).

1. `counting` – count words and lines in a string
2. `fraction` – evaluate a continued fraction
3. `words` – break a string into a vector of one-word strings
4. `account` – encapsulation of state, `const` methods
5. `shapes` – object composition
6. `contacts` – collections of your own objects

The last section has regular exercises – those will be (on average) yet more difficult and let you further practice programming with the concepts that you have learned this week. Like with elementary exercises earlier, solutions to these can be freely discussed and shared, you can work on the exercises with your friends, and you can compare your solution to those included in Part S. Some of these exercises will be solved interactively in the seminar. The files are named `r?_*.cpp`.

1. `wrap` – wrap long lines into paragraphs of a given width
2. `digits` – representing numbers in a positional system
3. `sieve` – find prime numbers
4. `bsearch` – binary search in a sorted `std::vector`
5. `qsort` – the staple of in-place sorting algorithms
6. `radix` – a fast comparison-free sorting algorithm

Part 1.a: Using the source bundle

We recommend that you work on `aisa`, which has all the required tools installed and set up correctly. You can use `micro` as your editor if you are not familiar with `vim`, or you can use a remote editing feature in your code editor of choice. If you prefer to set up your own, local, environment, you are of course free to do that, but please keep in mind that your tools are your responsibility.
If you work on `aisa`, you can check your solutions against the test cases provided with the exercises by running `make` (this tries to compile all of them in order) or `make p1_counting` to compile and test only one of them. Additionally, `make` will run your code through `clang-tidy` and `valgrind`.

Part 1.d: Demonstrations

1.d.1 [`fibonacci`] We will assume some familiarity with C (or at least some C-style braces-and-semicolons language, like Java). First things first: subroutines, statements, types and values. In C++, variables, containers and so on hold values. Assignment updates those values and does not rebind the name to a different object. If you come from Java or Python, this is a bit of a culture shock. See also lectures.
In this demo, we will implement the mother of all programming language examples, the Fibonacci sequence (forget hello world, this is not that kind of a course). First the function (subroutine) signature – in order come:

- return type – in this case `std::vector< int >`, then
- the name of the subroutine – `fibonacci`, and finally
- the argument list: `int n`.

```
std::vector< int > fibonacci( int n )
```

A vector is a sequence container – it holds a sequence of values. In

C++, containers are generic, that is, parametrized by the type of their elements, and these type parameters are specified in angle brackets. In this case, we are declaring that fibonacci returns a vector (sequence) of integers (int is the 'default' integer type in C++). The curly braces after the signature enclose the function body.

```
{
```

The body is a sequence of statements, separated by semicolons (with the exception of compound statements – which are enclosed in braces and are not followed by a semicolon). The first statement in this function is a local variable declaration, which consists of the type (the already familiar std::vector< int >), possible declarators (like pointers and references... again, we will get to those later – there are none in this particular case) and the name of the variable: fib.

```
    std::vector< int > fib;
```

Vectors are generalized arrays: unlike traditional C arrays, they can be resized on demand. To set the size of a vector, we can use its method resize: to call a method of an object (and vector is an object), we use the following syntax:

- the variable holding the object – fib, then
- a dot,
- then the name of the method to call,
- then an argument list, enclosed in parentheses.

Of course, like everything in C++, method calls can get a lot more complicated, and it is a topic that we will likewise revisit.

```
    fib.resize( n, 1 );
```

Now that fib is an appropriately sized vector, with the number 1 stored at each index, we can go on to rewrite the values to the actual Fibonacci sequence. We will use a for loop, which you probably know from C – the for statement has 3 sections enclosed in parentheses and separated by semicolons:

- initialisation, which usually declares or initializes the loop variable,
- the loop condition,
- the iteration expression, which is executed after every iteration, before the loop condition.

The head of the loop is followed by a statement, which is the body: the code that is repeatedly executed. Often, this is a compound statement (enclosed in braces) but it doesn't have to.

```
    for ( int i = 2; i < n; ++ i )
```

In this case, the body consists of a single statement: an assignment, which updates the i-th position in the vector fib with the sum of the values stored at the two preceding indices. Square brackets after a variable name indicate the indexing operator and works analogously to array indexing in C.

```
        fib[ i ] = fib[ i - 1 ] + fib[ i - 2 ];
```

The return statement does two things, like in most imperative languages: it provides the return value, and it immediately stops execution of the function, transferring control back to the caller.

```
    return fib;
}
```

All demonstrations and exercises in this collection contain a short collection of test cases. In the demos, they usually serve to show how the code explained in the main part works, and for you to change and experiment.

```
int main() /* demo */
{
    std::vector fib_7{ 1, 1, 2, 3, 5, 8, 13 };
    std::vector fib_1{ 1 };

    assert( fibonacci( 7 ) == fib_7 );
    assert( fibonacci( 1 ) == fib_1 );
}
```

1.d.2 [hamming] Besides sequences of numbers, another type of sequence frequently appears in computer programs: strings, which are made of letters. In C++, the basic data type for working with strings is std::string, and it is rather similar to a vector, though strings provide additional methods, for operations commonly performed on strings (but not so commonly on other sequences).
In this demo, we will show some basic usage of std::string. The following function, called hamming returns an integer (of type int) and accepts 2 arguments. Notice that there are some new elements in the declarations of those arguments: the const qualifier, meaning that we do not intend to modify the values a and b, and a reference declarator, denoted &.
These two often go together – in this arrangement, they declare a constant reference. In a function argument list, this means that the data will not be copied when the function is called, but the function promises not to change the original. Since a string might contain a lot of data, copying all of it might be expensive: this is why we prefer to use a constant reference to pass it into a function, if the function only needs to examine, but not change, the content of the string. In other words, a and b are not values in their own right; instead, they are aliases (new names) for existing values, albeit such that the original values cannot be modified through these new names. If you try to, the compiler will complain.[3]

```
int hamming( const std::string &a, const std::string &b )
{
```

First, we declare a precondition: the strings must be of equal size. In other words, calling hamming on two strings of different length is a programming error: the caller is responsible for ensuring that the condition holds.

```
    assert( a.size() == b.size() );
```

We declare a local variable to hold the computed distance, of type int (the 'default' integral type in C++).

```
    int distance = 0;
```

Again, a standard C-style for loop. Notice that strings can be indexed, just like arrays and vectors. Also notice that the loop variable is now of type size_t – an unsigned integer type. This is because the size methods of standard containers in C++ return unsigned numbers,[4] and comparing signed and unsigned integers can cause problems.

```
    for ( size_t i = 0; i < a.size(); ++ i )
        if ( std::toupper( a[ i ] ) != std::toupper( b[ i ] ) )
            distance ++;
```

And a return statement.

```
    return distance;
}
```

That is all. If you have never heard of Hamming distance before, it might be a good idea to look it up.

```
int main() /* demo */
```

[3] Of course, this being C++, there is a way around that. It is only needed very rarely, and only in 'plumbing' – low-level code which implements, for instance, new data types.

[4] Arguably, this is a design mistake in C++. There are proposals to fix it, but a change in this regard is going to take a long time, if it ever happens. In the meantime, it makes sense to use unsigned types for straightforward loop variables (i.e. those that count up).

```
{
    assert( hamming("Python", "python") == 0 );
    assert( hamming("AbCd", "aBcD") == 0 );
    assert( hamming("string", "string") == 0 );
    assert( hamming("aabcd", "abbcd") == 1 );
    assert( hamming("abcd", "ghef") == 4 );
    assert( hamming("Abcd", "bbcd") == 1 );
    assert( hamming("gHefgh", "ghefkl") == 2 );
}
```

1.d.3 [hero] In many programs, pre-made data types included in the standard library are more than sufficient. However, it is also often the case that a custom data type could be useful – most often to describe a particular concept from the domain which the program models.

Let us consider a dungeon crawler, or some other role-playing game set in a fantasy world. In such games, the protagonist will be able to pick up items and make use of them, for instance wield a sword to fight the critters in the dungeon.

This would be a rather typical use case for a custom data type: there might be many individual swords in the game, but they all share the same essential set of attributes, like weight, or the amount of damage they deal to the opponent. Of course, we could store these attributes as a tuple, with anonymous fields, and remember that the weight is the first element and the attack strength is the second. While fine in a small program, this approach is not very scalable.[5]

With `struct` (and `class`, in a short while), we can create user-defined data types, with named attributes and methods. The `struct` keyword is inherited from C, where it defines an aggregate (or record) data type. C++ extends this concept with methods, constructors, destructors, inheritance, and so on. However, at their heart, C++ objects are really just fancy record types. We will start by exploring these.

A record type describes a composite (or aggregate) value, made of a fixed number of attributes (fields), possibly of different types. In this sense, it is very much like a tuple. However, in a record type, the fields have names, and their values are accessed by using those names (instead of their positions as in a tuple). To define a record type, we use the keyword `struct`, followed by the name of the type, followed by the definition of the individual fields. Let's start by defining a type which will describe a sword.

```
struct sword
{
```

The most important attribute of a sword is, clearly, a fancy name. Recall that we can use `std::string` to conveniently store strings. Let us then declare the attribute `name` of type `std::string`:

```
    std::string name;
```

Then there are some attributes that deal with game mechanics. Let us just describe them using two integers, `weight` and `attack`. In actual games, things usually get a bit more complicated. It is possible to give default values to attributes – in this case, when a value of type `sword` is created, `weight` and `attack` will be both set zero. How this is achieved or why it is important will be discussed later.

```
    int weight = 0;
    int attack = 0;
};
```

That is all. At this point, `sword` is a type, like `int` or `std::string`, and we can declare variables of type `sword`, return values of type `sword` from functions, or pass values of type `sword` as function arguments. For example, let's write a trivial predicate on values of type `sword`. Notice the syntax for attribute access: it is the same that we have used for calling methods of 'built-in' types like `std::string` earlier. This is not a coincidence.

```
bool sword_is_heavy( const sword &s )
{
    return s.weight > 50;
}
```

Let us define another record type, `shield`, before moving on.

```
struct shield
{
    std::string name;
    int weight = 0;
    int defense = 0;
};
```

Swords and shields are usually rather passive. However, programs often also model more dynamic entities; user-defined record types would seem like a good fit to describe their static side (i.e. their attributes). For instance, a `hero` would have a health bar (how much damage they can take before dying), and some weapons (a sword and a shield, for instance). And obviously a name.

Given a record type which models an entity, it is possible, of course, to write functions which describe the behaviours of this entity. For instance `hero_walk` or `hero_attack` could be functions which take the specific hero to act on as one of their arguments.

You perhaps notice the imbalance though: attributes use a nice and concise syntax, `value.attribute`, but functions use much clumsier `type_method( value, … )`. But we did not have to say `string_size( string )` earlier.

Indeed, in C++, it is possible to also bundle functions into user-defined data types, in addition to attributes. Such data types are no longer called 'record types' – instead, they are known as classes. In other words, a C++ class is a user-defined data type with attributes and methods (associated functions). Let us define one of those – the syntax is analogous to record types:[6]

```
class hero
{
```

In classes, attributes are often private: only methods of the same class are allowed to directly access them. This is the default: unlike `struct`, when we start writing declarations into a class, they will be inaccessible to the outside code. This is okay for our current purposes. It is also common practice to prefix attributes in a class (unless they are public) with an underscore, or some other short string (`m_`, for member, is also sometimes used), to avoid naming conflicts: it is not allowed to have an attribute and a method with the same name.

```
    std::string _name;
    shield _shield;
    sword _sword;
```

To mark further attributes and methods as accessible to the outside world, we use the label `public`, like this:

```
public:
```

Methods are declared just like functions, the only immediate difference being that this is done inside a class. And the odd `const` keyword at the end of the signature. This `const` tells the compiler that the method does not change the object in any way when it is called (again, this is

[5] After all, we could just use void pointers, remember how big the data is and which attribute is stored at which offset. There is a good reason why nobody writes serious programs in this style.

[6] In fact, `struct` and `class` are essentially the same thing, and only differ in minor syntactic details. Nonetheless, we will usually write `struct` for plain record types (without methods) and `class` for actual classes.

enforced by the compiler).

```
    bool wields_heavy_sword() const
    {
        return sword_is_heavy( _sword );
    }
```

An example of non-const method would be the following, which causes the hero to wield a sword given by the argument. The method assigns into one of the attributes, which obviously changes the object, and hence cannot be marked const.

```
    void wield( const sword &s )
    {
        _sword = s;
    }
```

Finally, we will add a constructor: a special kind of method which is called automatically by the compiler whenever a value of type hero is created, e.g. by declaring a local variable. The constructor's name is the name of the class, and it has no return type, bit it can have arguments. Unlike standard functions (and standard methods), constructors have an initialization section, which can initialize attributes, e.g. by passing arguments to their constructors. When the body of the constructor is entered, all the attributes will have been already constructed. The initialization section starts with a colon, and is followed by a list of expressions of the form attribute(argument list).

```
    hero( std::string name )
        : _name( name )
    {}
};
```

That is quite enough for now. Let us look at a few examples of code using the above types.

```
int main() /* demo */
{
    sword katana = { "Katana", 10, 17 };
    hero protagonist( "Hiro Protagonist" );
    protagonist.wield( katana );
    assert( !protagonist.wields_heavy_sword() );
}
```

1.d.4 [lemmings] While we are talking about computer games, you might have heard about a game called Lemmings (but it's not super important if you didn't). In each level of the game, lemmings start spawning at a designated location, and immediately start to wander about, fall off cliffs, drown and generally get hurt. The player is in charge of saving them (or rather as many as possible), by giving them tasks like digging tunnels, or stopping and redirecting other lemmings. Let's try to design a class (reminder: a class is a user-defined data type with attributes and methods) which will capture the state of a single lemming:

```
class lemming
{
```

Each lemming is located somewhere on the map: coordinates would be a good way to describe this. For simplicity, let's say the designated spawning spot is at coordinates (0, 0).

```
    double _x = 0, _y = 0;
```

Unless they hit an obstacle, lemmings simply walk in a given direction – this is another candidate for an attribute; and being rather heedless, it's probably good idea to keep track of whether they are still alive.

```
    bool _facing_right = true;
    bool _alive = true;
```

Finally, they might be assigned a task, which they will immediately start performing. An enumerated type is another kind of a user-defined type and consists of a discrete set of named labels. You have most likely encountered them in C.

```
    enum task { no_task, digger, stopper, /* … */ };
    task _task = no_task;

public:
```

Let us define a couple methods:

```
    void start_digging() { _task = digger; }
    bool busy()  const { return _task != no_task; }
    bool alive() const { return _alive; }

    void step()
    {
        _x += _facing_right ? 1 : -1;
        _y += 0; // TODO gravity, terrain, …
    }
};
```

Earlier, we have mentioned that user-defined types are essentially the same as built-in types – their values can be stored in variables, passed to and from functions and so on. There are more ways in which this is true: for instance, we can construct collections of such values. Earlier, we have seen a sequence of integers, the type of which was std::vector< int >. We can create a vector of lemmings just as easily: as an std::vector< lemming >. Let us try:

```
int count_busy( const std::vector< lemming > &lemmings )
{
```

Note that the vector is marked const (because it is passed into the function as a constant reference). That extends to the items of the vector: the individual lemmings are also const. We are not allowed to call non-const methods, or assign into their attributes here. For instance, calling lemmings[0].start_digging() would be a compile error.

```
    int count = 0;
```

Now is perhaps a good time to introduce a new piece of syntax: the range for loop. Its main purpose is to iterate over all items in a given collection, which is exactly what we want to do. It consists of a declaration of the loop variable, followed by a colon, and an expression which ought to yield an iterable sequence.

```
    for ( const lemming &l : lemmings )
        if ( l.busy() )
            count ++;

    return count;
}

int main() /* demo */
{
```

We first create an (empty) vector, then fill it in with 7 lemmings.

```
    std::vector< lemming > lemmings;
    lemmings.resize( 7 );
```

We can call methods on the lemmings as usual, by indexing the vector:

```
    lemmings[ 0 ].start_digging();
    assert( count_busy( lemmings ) == 1 );
```

We can also modify the lemmings in a range for loop – notice the absence of const; this time, we use a mutable reference (often called just a reference, or an lvalue reference – more on that later):

```
    for ( lemming &l : lemmings )
```

```
    {
        assert( l.alive() );
        l.start_digging();
    }

    assert( count_busy( lemmings ) == 7 );
}
```

Part 1.e: Elementary Exercises

1.e.1 [predicates] Write the following predicates (pure functions which return a boolean value). The first two return true if all (all_odd) or at least one (any_odd) number in the list is odd:

```
bool all_odd( const std::vector< int > & );
bool any_odd( const std::vector< int > & );
```

The third returns true if there are at least n numbers divisible by k:

```
bool count_divisible( const std::vector< int > &, int k, int n );
```

1.e.2 [palindrome] Write a predicate which decides whether a given string is a palindrome, i.e. reads the same in both directions.

```
bool is_palindrome( const std::string &s );
```

1.e.3 [pascal] Write a function which builds the n-th row of Pascal's triangle as a vector of integers and returns it.

```
std::vector< int > pascal( int n );
```

Part 1.p: Preparatory Exercises

1.p.1 [counting] In this exercise, we will work with strings in a read-only way: by counting things in them. Write two functions, word_count and line_count: the former will count words (runs of characters without spaces) and the latter will count the number of non-empty lines. Use range for to look at the content of the string.
Here are the prototypes of the functions – you can simply turn those into definitions. We pass arguments by const references: for now, consider this to be a bit of syntax, the purpose of which is to avoid making a copy of the string. It will be explained in more detail later. Also notice that in a prototype, the arguments do not need to be named (but you will have to give them names to use them).

```
int word_count( const std::string & );
int line_count( const std::string & );
```

1.p.2 [fraction] Write a function which evaluates a continued fraction: given a vector of coefficients of the continued fraction, it computes a numerator and a denominator of a traditional fraction with the same value.
A continued fraction is a representation of a rational number q as a sum of a_0 and the reciprocal of a second number, q_0, which is itself written as a continued fraction: $q_0 = a_0 + 1/q_1$ where $q_1 = a_1 + 1/q_2$, $q_2 = a_2 + 1/q_3$ and so on. The sequence $a_0, a_1, a_2, \ldots$ are the coefficients of the continued fraction. For a rational number, one of the q_n eventually becomes 0 and the sequence ends there.
For more details, see e.g. wikipedia.
Define a traditional fraction as a struct with two integer attributes, p and q (the numerator and the denominator, respectively).

```
struct fraction;

fraction eval_continued( const std::vector< int > &coeff );
```

1.p.3 [words] Write a function that breaks up a string into individual words. We consider a word to be any string without whitespace (spaces, newlines, tabs) in it.

Since we are lazy to type the long-winded type for a vector of strings, we define a type alias. The syntax is different from C, but it should be clearly understandable. We will encounter this construct many times in the future.

```
using string_vec = std::vector< std::string >;
```

The output of words should be a vector of strings, where each of the strings contains a single word from in.

```
string_vec words( const std::string &in );
```

1.p.4 [account] In this exercise, you will create a simple class: it will encapsulate some state (account balance) and provide a simple, safe interface around that state. The class should have the following interface:

- the constructor takes 2 integer arguments: the initial balance and the maximum overdraft
- a withdraw method which returns a boolean: it performs the action and returns true iff there was sufficient balance to do the withdrawal
- a deposit method which adds funds to the account
- a balance method which returns the current balance (may be negative) and that can be called on const instances of account

```
class account;
```

1.p.5 [shapes] Another exercise about objects, this time about their composition. We will write 2 classes: point and rectangle. Points have 2 coordinates (x and y) and rectangles are defined by 2 points (their opposing corners).
Points are constructed from two doubles: the x and y coordinates, and they have x() and y() methods which return doubles.

```
class point;
```

A function to compute euclidean distance between two points. Writing it is a part of the exercise, but it will be also useful when implementing the diagonal method in rectangle.

```
double distance( point a, point b );
```

Rectangles are constructed from a pair of points (bottom left and upper right corner) and provide methods: width, height and diagonal which all return a double, and a method center which returns a point.

```
class rectangle;
```

1.p.6 [contacts] We will look at using collections of objects. We only know one type of collection: a dynamic array, so that's what we will use. The objects we will consider are simple entries in a contact list: they have a name and a phone number (both stored as strings).
We need contact to possess a two-parameter constructor (which initializes both its fields) and two getters (methods), name and phone.

```
class contact;
using contacts = std::vector< contact >; /* type alias */
```

Let's write a helper function which checks whether the string small is a prefix of the string big.

```
bool is_prefix( const std::string &small, const std::string &big );
```

And finally, a function to return all contacts whose names start with the given prefix (use is_prefix in a loop).

```
contacts search( const contacts &list, const std::string &prefix );
```

Part 1.r: Regular Exercises

1.r.1 [wrap] We will look at `std::string` again. Our first task will be to implement a simple word wrapping (paragraph filling) algorithm.
Input: An `std::string` with ASCII text (letters, spaces, newlines and punctuation) and `columns` (a number of columns). Each line of the input text represents a single paragraph.
Output: A string in which there are actual paragraphs with line breaks, not too far after the given column number. That is, at most a single word crosses the `column`-th column. Newlines in the input are replaced by double newlines in the output.

```
std::string fill( const std::string &in, int columns );
```

1.r.2 [digits]

```
std::vector< int > digits( int n, int base );
```

1.r.3 [sieve] Implement the Sieve of Eratosthenes for quickly finding the largest prime smaller than or equal to a given bound.

```
int sieve( int bound );
```

1.r.4 [bsearch] Implement binary search on a vector. In this case, we will use a non-const reference to pass the vector, because we don't know yet how to deal with const iterators properly. We also don't know how to write generic algorithms (we will see that at the end of this course), so we use a vector of integers.
It is customary to return the end iterator if an element is not found. A pair of iterators in C++, by convention, denotes a left-closed / right-open interval, like this: [begin, end).

```
std::vector< int >::iterator bsearch( std::vector< int > &vec, int
val );
```

Part 2: References and Lambdas

In this chapter, we will work with references (both constant and mutable) and also look at the basics of higher-order functions in C++. Demonstrations:

1. `stats` – input and output parameters
2. `primes` – fill in a vector with prime numbers
3. `iterate` – building sequences by iterating a function
4. `newton` – a general routine for numeric approximation

Elementary exercises:

1. `fibonacci` – old sequence, new function signature
2. `normalize` – divide out the gcd from a fraction
3. `accumulate` – sum up $f(x)$ for all x in an `std::vector`

Preparatory exercises:

1. `rewrap` – word wrapping redux, this time in-place
2. `golden` – basic uses of output parameters
3. `divisors` – collections as in/out parameters
4. `midpoints` – in/out parameters of custom types
5. `higher` – higher-order function primer: `map` and `zip`
6. `fixpoint` – find a fixed point of a monotonic function

Regular exercises:

1. `euler` – implement Euler's totient function ϕ
2. `approx` – somewhat easier approximation
3. `solve` – a very simple game solver
4. `sort` – selection sort with a comparator
5. `permute` – compute a vector of digit permutations
6. `bsearch` – binary search with a comparator

Part 2.d: Demonstrations

2.d.1 [stats] In this demo, we will do some basic descriptive statistics. Last week, we have used constant references to pass input arguments into functions. We will now see how to use non-constant (mutable) references to implement output and in/out arguments. The syntax for a mutable reference is simply the type, the reference declarator (`&`) and the name of the argument, i.e. dropping the `const` (compare `data` vs `median` in the following function signature.

```
void stats( const std::vector< double > &data,
            double &median, double &mean, double &stddev )
{
    int n = data.size();
    double sum = 0, square_error_sum = 0;
    for ( double x_i : data )
        sum += x_i;
```

Notice that we do not read the value of `median` before overwriting it with the resulting value: this is a hallmark of an output argument – it is never read before being written by the function.

```
    mean = sum / n;

    if ( n % 2 == 1 )
        median = data[ n / 2 ];
    else
        median = ( data[ n / 2 ] + data[ n / 2 - 1 ] ) / 2;
```

However, after we have assigned a value to `mean`, we can continue to use it like a normal read-write variable. It is important that the read cannot be reached without executing the write first (e.g. it would be a problem if the write above was conditional).

```
    for ( double x_i : data )
        square_error_sum += ( x_i - mean ) * ( x_i - mean );

    double variance = square_error_sum / ( n - 1 );
    stddev = std::sqrt( variance );
```

No return statement: the function was declared with `void` as its return type, meaning that it does not return anything. The values are all passed to the caller via output arguments.

```
}

int main() /* demo */
{
    double median, mean, stddev;
    std::vector< double > sample = { 2, 4, 4, 4, 5, 5, 5, 7, 9 };
    stats( sample, median, mean, stddev );

    assert( mean == 5 );
    assert( median == 5 );
    assert( stddev == 2 );

    sample.push_back( 1100 );
    stats( sample, median, mean, stddev );

    assert( median == 5 );
    assert( mean > 100 );
    assert( stddev > 100 );
}
```

2.d.2 [primes] Besides simple output arguments, like in the previous

demo, we can pass values out of functions by manipulating existing objects, most straightforwardly containers. In this demo, we will write a function primes which appends prime numbers from a given range to an existing std::vector. We will still call out an output argument, though the concept is clearly more nuanced here. Like before, we will use a mutable reference to achieve the desired semantics.

```
void primes( int from, int to, std::vector< int > &out )
{
    for ( int candidate = from; candidate < to; ++ candidate )
    {
        bool prime = true;
        int bound = std::sqrt( candidate ) + 1;
```

Decide whether a given number is prime, naively, by trial division.

```
        for ( int div = 2; div < bound; ++ div )
            if ( div != candidate && candidate % div == 0 )
            {
                prime = false;
                break;
            }
```

Now the interesting part: if the number was found to be prime, we append it to the object referenced by out (i.e. the original object which was declared outside this function and passed into it by reference). Below in main, you can see that the content of the vector p_out changes when we call this function on it.

```
        if ( prime )
            out.push_back( candidate );
    }
}

int main() /* demo */
{
    std::vector< int > p_out;
    std::vector< int > p7 = { 2, 3, 5 },
                       p15 = { 2, 3, 5, 7, 11, 13 };

    primes( 2, 7, p_out );
    assert( p_out == p7 );
    primes( 7, 15, p_out );
    assert( p_out == p15 );
}
```

2.d.3 [iterate] In this short demo, we will introduce new syntax for writing functions. The type of function we will use is called a lambda, from the symbol that is used in lambda calculus to introduce anonymous functions. In C++, lambdas are like regular functions with a few extras.

Notice that iterate is declared as a variable – the function is on the right-hand side, and does not have an intrinsic name (i.e. it is anonymous). The type of iterate is not specified – instead, we have used auto, to instruct the compiler to fill in the type.

Besides the missing name and the empty square brackets, the signature of the lambda is similar to a standard function. However, on closer inspection, another thing is missing: the return type. This might be specified using -> type after the argument list, but if it is not, the compiler will, again, deduce the type for us. The return type is commonly omitted.

```
auto iterate = []( auto f, auto x, int count )
```

An advantage of a lambda is that we do not need to know the types of all the arguments in advance: in particular, we don't know the type of f – this will most likely be a lambda itself (i.e. iterate is a higher-order function). When this is the case, instead of the type, we specify auto, instructing the compiler to deduce a type when the function is used. This is the same principle which we have applied to the variable iterate itself: we do not know the type, so we ask the compiler to fill it in for us (by using auto). Let us continue by writing the body of iterate:

```
{
```

We want to build a vector of values, starting with x, then f(x), f(f(x)), and so on. Immediately, we face a problem: what should be the type of the vector? We need to specify the type parameter to declare the variable, and this time we won't be able to weasel out by just saying auto, since the compiler can't tell the type without an unambiguously typed initializer. We have two options here:

1. in some circumstances, it is possible to omit the type parameter of std::vector and let the compiler deduce only that. This would be written std::vector out{ x } – by putting x into the vector right from the start, the compiler can deduce that the element type should be the same as the type of x, whatever that is; we will deal with this mechanism much later in the course (in the last block); in the meantime,
2. we can use decltype to obtain the type of x and use that to specify the required type parameter for out, i.e.:

```
    std::vector< decltype( x ) > out;
    out.push_back( x );
```

We build the return vector by repeatedly calling f on the previous value, until we hit count items.

```
    for ( int i = 1; i < count; ++ i )
        out.push_back( f( out.back() ) );
```

And we return the value, like in a regular function. Please also note the semicolon after the closing brace: definition of a lambda is an expression, and the variable declaration as a whole needs to be delimited by a semicolon, just like in int x = 7;.

```
    return out;
};

int main() /* demo */
{
    auto f = []( int x ) { return x * x; };
    auto g = []( int x ) { return x + 1; };
```

Of course, we can use auto in declaration of regular variables too, as long as they are initialized.

```
    auto v = iterate( f, 2, 4 );

    std::vector< int > expect{ 2, 4, 16, 256 };
    assert( v == expect );

    std::vector< int >
        iota = iterate( g, 1, 4 ),
        iota_expect{ 1, 2, 3, 4 };

    assert( iota == iota_expect );
}
```

2.d.4 [newton] This demonstration is as far as we'll venture with regards to numeric approximation – the exercises that deal with approximations are all much simpler than this demo. Here, we will implement the general Newton-Raphson method. This can be used for finding all kinds of roots (zeroes of functions) numerically and for solving 'hard' (transcendental) equations.

The input to Newton's method is a function f and its derivative, df. A single improvement step then takes the current estimate x_0 and subtracts $f(x)/df(x)$ from it. It is actually quite simple.

```
auto newton = []( auto f, auto df, double ini, double prec )
{
    double x = ini, y = ini - f( x ) / df( x );
```

```
    while ( std::fabs( y - x ) >= prec )
    {
        x = y;
        y = y - f( x ) / df( x );
    }

    return y;
};
```

We can straightforwardly apply the above generic function to suitable arguments to immediately implement some familiar functions, like square roots or cube roots (we just need to find a function which becomes zero if x is the square root of the argument of the function; that function would be $f(z) = z^2 - x$ and its derivative is $f'(z) = 2z$).

```
double sqrt( double x, double prec ) /* square root */
{
    return newton( [=]( double z ) { return z * z - x; },
                   [=]( double z ) { return 2 * z; }, 1, prec );
}

double cbrt( double n, double prec ) /* cube root */
{
    return newton( [=]( double z ) { return z * z * z - n; },
                   [=]( double z ) { return 3 * z * z; }, 1, prec );
}
```

Compute nth root of x, generalizing sqrt and cbrt above.

```
double root( int n, double x, double prec )
{
    auto  f = [=]( double z ) { return std::pow( z, n ) - x; };
    auto df = [=]( double z ) { return n * std::pow( z, n - 1 ); };
    return newton( f, df, 1, prec );
}
```

Scroll to the end to see the test cases. The following code computes π using only basic arithmetic and the Newton method... It's all a bit fast and loose, but it works. Enjoy.
Approximately evaluate a function using its truncated Taylor expansion.

```
auto taylor = []( auto coeff, double x, double prec )
{
    double r = 0, pow = 1, fact = 1;
    int i = 0;

    while ( pow / fact > prec / 10 )
    {
        r += coeff( i ) * pow / fact;
        fact *= ++i;
        pow *= x;
    }

    return r;
};
```

Shorthand for 4-periodic Taylor coefficients (like those that appear in trigonometric functions).

```
auto trig_coeff( int a, int b, int c, int d )
{
    return [=]( int i ) { return i % 4 == 0 ? a : i % 4 == 1 ? b :
                                 i % 4 == 2 ? c : d; };
}
```

Sine and cosine, to feed into Newton.

```
double sine( double x, double prec )
{
    return taylor( trig_coeff( 0, 1, 0, -1 ), x, prec );
}

double cosine( double x, double prec )
{
    return taylor( trig_coeff( 1, 0, -1, 0 ), x, prec );
}
```

Compute π/2 as the root of cosine.

```
double pi( double prec )
{
    auto  f = [=]( double x ) { return cosine( x, prec ); };
    auto df = [=]( double x ) { return -sine( x, prec ); };
    return 2 * newton( f, df, 1, prec );
}

int main() /* demo */
{
    for ( int decimals = 1; decimals < 10; ++ decimals )
    {
        double p = std::pow( 10, -decimals );

        assert( std::fabs( sqrt( 2, p ) - std::sqrt( 2 ) ) < p );
        assert( std::fabs( cbrt( 2, p ) - std::cbrt( 2 ) ) < p );

        assert( std::fabs( root( 2, 2, p ) - std::sqrt( 2 ) ) < p );
        assert( std::fabs( root( 3, 2, p ) - std::cbrt( 2 ) ) < p );
        assert( std::fabs( root( 4, 16, p ) - 2 ) < p );

        assert( std::fabs( pi( p ) - 4 * std::atan( 1 ) ) < p );
    }
}
```

Part 2.e: Elementary Exercises

2.e.1 [fibonacci] Fill in an existing vector with a Fibonacci sequence (i.e. after calling fibonacci(v, n), the vector v should contain the first n Fibonacci numbers, and nothing else).

```
// void fibonacci( … )
```

2.e.2 [normalize] Write a function to normalize a fraction, that is, find the greatest common divisor of the numerator and denominator and divide it out. Both numbers are given as in/out parameters.

```
// void normalize( … )
```

2.e.3 [accumulate] Write a function accumulate(f, vec) which will sum up $f(x)$ for all x in the given std::vector< int > vec.

```
// auto accumulate = …
```

Part 2.p: Preparatory Exercises

2.p.1 [rewrap] A different take on word-wrapping. The idea is very similar to last week – break lines at the first opportunity after you ran out of space in your current line. The twist: do this by modifying the input string. Additionally, undo existing line breaks if they are in the wrong spot.

```
void rewrap( std::string &str, int cols );
```

2.p.2 [golden] The function next_fib should behave like this:

- given: a == fib(i) and b == fib(i + 1)
- execute: next_fib(a, b)
- to get: a == fib(i + 1) and b == fib(i + 2).

```
void next_fib( int &a, int &b );
```

Optional: Compute the n-th Fibonacci number using next_fib. Make it so that: fib(1) == 1, fib(2) == 1, fib(3) == 2. This is just to practice working with next_fib in case you aren't sure.

```
int fib( int n );
```

Approximate the golden ratio as the ratio of two consecutive Fibonacci numbers. The precision argument gives an upper bound on the approximation error. The number rounds is an output parameter and gives us the number of iterations (calls to next_fib) that were required to satisfy the precision requirement.
Notice that:

- the golden mean $\phi = 1.618...$
- fib(2) / fib(1) = 1 / 1 = 1 is a lower bound
- fib(3) / fib(2) = 2 / 1 = 2 is an upper bound
- fib(4) / fib(3) = 3 / 2 = 1.5 is a lower bound
- fib(5) / fib(4) = 5 / 3 = 1.667 is an upper bound

and so on. Surely the error – distance from ϕ itself – in any given round is smaller than its distance from the previous round.

```
double golden( double precision, int &rounds );
```

2.p.3 [divisors] Take a number, find all its prime divisors and add them into divs, unless they are already there. Be sure to do this in time proportional (linear) to the input number.
Bonus: If you assume that divs is sorted in ascending order when you get it, you can make add_divisors a fair bit more efficient. Can you figure out how?

```
void add_divisors( int num, std::vector< int > &divs );
```

2.p.4 [midpoints] A familiar class: add a 2-parameter constructor and x(), y() accessors (the coordinates should be double-precision floating point numbers).

```
class point;
```

Consider a closed shape made of line segments. Replace each segment A with one that starts at the midpoint of A and ends at the midpoint of B, the segment that comes immediately after A. The input is given as a sequence of points (each point shared by two segments). The last segment goes from the last point to the first point (closing the shape).

```
void midpoints( std::vector< point > &pts );
```

helper functions for floating-point almost-equality

```
bool near( double a, double b )
{
    return std::fabs( a - b ) < 1e-8;
}

bool near( point a, point b )
{
    return near( a.x(), b.x() ) && near( a.y(), b.y() );
}
```

2.p.5 [higher] Write a map function, which takes a function f and a vector v and returns a new vector w such that w[i] = f(v[i]) for any valid index i. We will need to use the 'lambda' syntax for this, since we don't yet know any other way to write functions which accept functions as arguments.

```
// static auto map = []( ... ) { ... };
```

Similar, but f is a binary function, and there are two input vectors of equal length. You do not need to check this.

```
// static auto zip = []( ... ) { ... };
```

You can assume that the output vector is of the same type as the input vector (i.e. f is of type a → a in map, and of type a → b → a for zip.

2.p.6 [fixpoint] A fixed point of a function f is an x such that $f(x) = x$. A function is monotonic if $\forall x, y . x \leq y \rightarrow f(x) \leq f(y)$. Assume that f is a monotonic function and hence, since there are only finitely many int values, that it has at least one fixed point. Find the greatest fixed point of f.

```
// auto fixpoint = …

int f( int x ) { return x / 2; }
int g( int x ) { return x - x / 20; }
int h( int x ) { return std::max( x / 5, 20 ); }
int i( int x ) { return x < INT_MAX ? x + 1 : x; }
```

Part 2.r: Regular Exercises

2.r.1 [euler] This is a straightforward math exercise. Implement Euler's [ϕ], for instance using the product formula $\phi(n) = n\Pi(1 - 1/p)$ where the product is over all distinct prime divisors of n. You may need to take care to compute the result exactly.

```
long phi( long n ); /* ref: 21 lines */
```

2.r.2 [approx] Remember fib.cpp? We can do a bit better. Let's decompose our golden() function differently this time.
The approx function is a higher-order one. What it does is it calls f() repeatedly to improve the current estimate, until the estimates are sufficiently close to each other (closer than the given precision). The init argument is our initial estimate of the result.

```
// auto approx = []( auto f, double init, double prec ) { ... };
```

Use approx to compute the golden mean. Note that you don't need to use the previous estimate in your improvement function. Use by-reference captures to keep state between iterations if you need some.

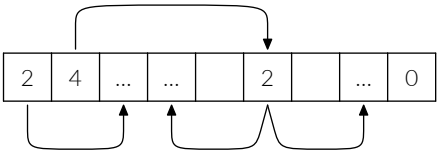
```
double golden( double prec );
```

The Babylonian (Heron) method to compute square roots. Please take note, you may find it helpful later. This is how approx is supposed to be used.

```
double sqrt( double n, double prec )
{
    auto improve = [=]( double last )
    {
        double next = n / last;
        return ( last + next ) / 2;
    };

    return approx( improve, 1, prec );
}
```

2.r.3 [solve] Consider a single-player game that takes place on a 1D playing field like this:



The player starts at the leftmost cell and in each round can decide whether to jump left or right. The playing field is given by the input vector jumps. The size of the field is jumps.size() + 1 (the rightmost cell is always 0). The objective is to visit each cell exactly once.

```
bool solve( std::vector< int > jumps );
```

Part 3: Containers

This week will be about containers (collections).
Demonstrations:

1. `freq` – a word frequency histogram
2. `dfs` – reachability using recursive depth-first search
3. `closure` – closure properties of relations
4. `bfs` – find the nearest matching node

Elementary exercises:

1. `unique` – remove duplicated entries from a vector
2. `reflexive` – compute a reflexive closure of a relation
3. `normalize` – scale an input vector into a 0-1 range

Preparatory exercises:

1. `brackets` – check that brackets in a string balance out
2. `connected` – decompose a graph into connected components
3. `dag` – check whether a graph is acyclic (dfs again)
4. `rel` – a tiny bit of relational algebra
5. `numbers` – a slightly enriched set of numbers
6. `bipartite` – bipartiteness checking using BFS

Regular exercises:

1. `mode` – find the mode (most common value) in a vector
2. `buckets` – sort items into buckets based on an attribute
3. `shortest` – shortest distances in an unweighted graph
4. `flood` – flood fill in a grid
5. `colour` – brute-force a 3-colouring of a graph
6. `life` – the game of life

Part 3.d: Demonstrations

3.d.1 [freq] In this demo, we will build up a histogram of word appearances. Since we will want to process the input incrementally, we will implement the word counter as a class with 2 methods: `process`, which will add each word that appears in its argument to the histogram, and `count`, which takes a single-word string as an argument and returns how many times it has been encountered by `process`.

```
class freq
{
```

All the heavy lifting will be done by the standard associative container, `std::map`. We will use `std::string` as the key type (holding the word of interest) and `int` as the value type: the number of appearances of this word.

```
    std::map< std::string, int > _counter;

public:
```

We will first implement a helper method, for counting a single word. It will be convenient to ignore empty strings here, so we will do just that. Notice that we use the indexing (subscript) operator to access the value which `std::map` associates with the given key. Also notice that the key is automatically added to the map in case it is not yet present.

```
    void add( const std::string &word )
    {
        if ( !word.empty() )
            _counter[ word ] ++;
    }
```

Now the main workhorse: `process` takes an input string, decomposes it into individual words and counts them. Notice the use of `+=` to append a letter to an existing string.

```
    void process( const std::string &str )
    {
        std::string word;

        for ( auto c : str )
            if ( std::isblank( c ) )
            {
                add( word );
                word.clear();
            }
            else
                word += c;
```

Do not forget to add the last word, in case it was not followed by a blank.

```
        add( word );
    }
```

We would clearly like to mark the `count` method, which simply returns information about the observed frequency of a word, as `const`. However, the subscript operator on an `std::map` is not `const` – this is because, as we have mentioned earlier, should the key not be present in the map, it will be added automatically, thus changing the content of the container.
Instead, we can ask `std::map` to check whether the key is present (by using `count`), without adding it. If the key is missing, we simply return 0. Otherwise, we ask the map to find the value associated with the key, again without adding it if it is missing. Note that dereferencing the result of `find` is undefined if the key is not present (in this case, we know for sure that the key is present – we just checked). All `std::map` methods which we used are marked `const` and hence we can mark our `count` method `const` as well, as we desired.

```
    int count( const std::string &s ) const
    {
        return _counter.count( s ) ? _counter.find( s )->second : 0;
    }

};

int main() /* demo */
{
    freq f;
```

We create a `const` alias for `f`, so that we check that it is indeed possible to call `count` on it.

```
    const freq &cf = f;

    assert( cf.count( "hello" ) == 0 );
    assert( cf.count( "" ) == 0 );

    f.process( "hello world" );
    assert( cf.count( "hello" ) == 1 );
    assert( cf.count( "hell" ) == 0 );
    assert( cf.count( "world" ) == 1 );
    assert( cf.count( " world" ) == 0 );

    f.process( "hello hello" );
    assert( cf.count( "hello" ) == 3 );
    assert( cf.count( "world" ) == 1 );

    f.process( "world hello world hello world" );
    assert( cf.count( "hello" ) == 5 );
    assert( cf.count( "world" ) == 4 );
}
```

3.d.2 [dfs] In this demo, we will do some basic exploration of directed graphs. Probably the simplest possible algorithm for that is recursive depth-first search, so that is what we will use. We will be interested

in the question 'is vertex a reachable from vertex b?'.
The input graph is given using adjacency lists: the graph type gives the successors for each vertex present in the graph. Please note that in principle, the set of vertices does not need to be contiguous, or composed of only small numbers (hence the std::map and not an std::vector).

```
using edges = std::vector< int >;
using graph = std::map< int, edges >;
```

Besides the graph itself, we will need to represent the visited set – the set of vertices that have already been visited by the algorithm. In a graph with loops, not keeping track of this information would lead to infinite recursion. In an acyclic graph, it could still lead to exponential running time. Since in pseudocode, this is a literal set, using std::set sounds like a good idea. Indeed, std::set is a container which keeps at most one copy of any element, and provides efficient (logarithmic time) lookup and insertion.

```
using visited = std::set< int >;
```

The main recursive function needs 2 auxiliary arguments: the set of already-visited vertices seen and the boolean moved, which guards against the case when we ask whether a vertex is reachable from itself – this is traditionally only answered in affirmative when there is a path from that vertex to itself, but a naive solution would always answer true. Hence, we need to ensure at least one edge was traversed before returning true. The question this function answers is 'is there a path which starts in vertex from, does not visit any of the vertices in seen and ends in to?'
Notice that seen is passed around by reference: there is only a single instance of this set, shared by all recursive calls. That is, if one branch of the search visits a vertex, it will also be avoided by any subsequent sibling branches (not just by the recursive calls made within the branch).

```
bool is_reachable_rec( const graph &g, int from, int to,
                       visited &seen, bool moved )
{
```

The base case of the recursion is when we reach the target vertex and have already traversed at least one edge. In this case, we return true: we have found a path connecting the two vertices.

```
    if ( from == to && moved )
        return true;
```

The main loop looks at each successor of from and calls is_reachable recursively, asking whether there is a path from the successor to the goal state, avoiding the current state. The result of the g.at call is a (reference to) the edges container (i.e. the std::vector of vertices). Hence next ranges over the successors of the vertex from.

```
    for ( auto next : g.at( from ) )
```

In case next was not yet seen (it is not present in the visited set), mark it as visited and proceed to explore it recursively.

```
        if ( !seen.count( next ) )
        {
            seen.insert( next );
            if ( is_reachable_rec( g, next, to, seen, true ) )
                return true;
        }
```

We have failed to find a satisfactory path, having exhausted all the options. Return false.

```
    return false;
}
```

Finally, we provide a simple wrapper around the recursive function above, providing initial values for the two auxiliary arguments. Check whether to can be reached by following one or more edges if we start at from.

```
bool is_reachable( const graph &g, int from, int to )
{
    visited seen;
    return is_reachable_rec( g, from, to, seen, false );
}

int main() /* demo */
{
    graph g{ { 1, { 2, 3, 4 } },
             { 2, { 1, 2 } },
             { 3, { 3, 4 } },
             { 4, {} },
             { 5, { 3 }} };

    assert(  is_reachable( g, 1, 1 ) );
    assert( !is_reachable( g, 4, 4 ) );
    assert(  is_reachable( g, 1, 2 ) );
    assert(  is_reachable( g, 1, 3 ) );
    assert(  is_reachable( g, 1, 4 ) );
    assert( !is_reachable( g, 4, 1 ) );
    assert(  is_reachable( g, 3, 3 ) );
    assert( !is_reachable( g, 3, 1 ) );
    assert(  is_reachable( g, 5, 4 ) );
    assert( !is_reachable( g, 5, 1 ) );
    assert( !is_reachable( g, 5, 2 ) );
}
```

3.d.3 [closure] In this demo, we will check closure properties of relations: reflexivity, transitivity and symmetry. A relation is a set of pairs, and hence we will represent them as std::set of std::pair instances. We will work with relations on integers. Recall that std::set has an efficient membership test: we will be using that a lot in this program.

```
using relation = std::set< std::pair< int, int > >;
```

The first predicate checks reflexivity: for any x which appears in the relation, the pair (x, x) must be present. Besides membership testing, we will use structured bindings and range for loops. Notice that a braced list of values is implicitly converted to the correct type (std::pair< int, int >).

```
bool is_reflexive( const relation &r )
{
```

Structured bindings are written using auto, followed by square brackets with a list of names to bind to individual components of the right-hand side. In this case, the right-hand side is the loop variable – i.e. each of the elements of r in turn.

```
    for ( auto [ x, y ] : r )
    {
        if ( !r.count( { x, x } ) )
            return false;
        if ( !r.count( { y, y } ) )
            return false;
    }
```

We have checked all the elements of r and did not find any which would violate the required property. Return true.

```
    return true;
}
```

Another, even simpler, check is for symmetry. A relation is symmetric if for any pair (x, y) it also contains the opposite, (y, x).

```
bool is_symmetric( const relation &r )
{
```

```
    for ( auto [ x, y ] : r )
        if ( !r.count( { y, x } ) )
            return false;
    return true;
}
```

Finally, a slightly more involved example: transitivity. A relation is transitive if $\forall x, y, z.(x, y) \in r \wedge (y, z) \in r \rightarrow (x, z) \in r$.

```
bool is_transitive( const relation &r )
{
    for ( auto [ x, y ] : r )
        for ( auto [ y_prime, z ] : r )
            if ( y == y_prime && !r.count( { x, z } ) )
                return false;
    return true;
}

int main() /* demo */
{
    relation r_1{ { 1, 1 }, { 1, 2 } };
    assert( !is_reflexive( r_1 ) );
    assert( !is_symmetric( r_1 ) );
    assert(  is_transitive( r_1 ) );

    relation r_2{ { 1, 1 }, { 1, 2 }, { 2, 2 } };
    assert(  is_reflexive( r_2 ) );
    assert( !is_symmetric( r_2 ) );
    assert(  is_transitive( r_2 ) );

    relation r_3{ { 2, 1 }, { 1, 2 }, { 2, 2 } };
    assert( !is_reflexive( r_3 ) );
    assert(  is_symmetric( r_3 ) );
    assert( !is_transitive( r_3 ) );
}
```

3.d.4 [bfs] The goal of this demonstration will be to find the shortest distance in an unweighted, directed graph:

1. starting from a fixed (given) vertex,
2. to the nearest vertex with an odd value.

The canonical 'shortest path' algorithm in this setting is breadth-first search. The algorithm makes use of two data structures: a queue and a set, which we will represent using the standard C++ containers named, appropriately, `std::queue`[7] and `std::set`.
In the previous demonstration, we have represented the graph explicitly using adjacency list encoded as instances of `std::vector`. Here, we will take a slightly different approach: we well use `std::multimap` – as the name suggests, it is related to `std::map` with one crucial difference: it can associate multiple values to each key. Which is exactly what we need to represent an directed graph – the values associated with each key will be the successors of the vertex given by the key.

```
using graph = std::multimap< int, int >;
```

The algorithm consists of a single function, `distance_to_odd`, which takes the graph `g`, as a constant reference, and the vertex `initial`, as arguments. It then returns the sought distance, or if no matching vertex is found, -1.

```
int distance_to_odd( const graph &g, int initial )
{
```

We start by declaring the visited set, which prevents the algorithm from getting stuck in an infinite loop, should it encounter a cycle in the input graph (and also helps to keep the time complexity under control).

```
    std::set< int > visited;
```

The next piece of the algorithm is the exploration queue: we will put two pieces of information into the queue: first, the vertex to be explored, second, its BFS distance from `initial`.

```
    std::queue< std::pair< int, int > > queue;
```

To kickstart the exploration, we place the initial vertex, along with distance 0, into the queue:

```
    queue.emplace( initial, 0 );
```

Follows the standard BFS loop:

```
    while ( !queue.empty() )
    {
        auto [ vertex, distance ] = queue.front();
        queue.pop();
```

To iterate all the successors of a vertex in an `std::multimap`, we will use its `equal_range` method, which will return a pair of iterators – generalized pointers, which support a kind of 'pointer arithmetic'. The important part is that an iterator can be incremented using the `++` operator to get the next element in a sequence, and dereferenced using the unary `*` operator to get the pointed-to element. The result of `equal_range` is a pair of iterators:

- begin, which points at the first matching key-value pair in the multimap,
- end, which points one past the last matching element; clearly, if `begin == end`, the sequence is empty.

Incrementing `begin` will eventually cause it to become equal to `end`, at which point we have reached the end of the sequence. Let's try:

```
        auto [ begin, end ] = g.equal_range( vertex );

        for ( ; begin != end; ++ begin )
        {
```

In the body loop, `begin` points, in turn, at each matching key-value pair in the graph. To get the corresponding value (which is what we care about), we extract the second element:

```
            auto [ _, next ] = *begin;

            if ( visited.count( next ) )
                continue; /* skip already-visited vertices */
```

First, let us check whether we have found the vertex we were looking for:

```
            if ( next % 2 == 1 )
                return distance + 1;
```

Otherwise we mark the vertex as visited and put it into the queue, continuing the search.

```
            visited.insert( next );
            queue.emplace( next, distance + 1 );
        }
    }
```

We have exhausted the queue, and hence all the vertices reachable from `initial`, without finding an odd-valued one. Indicate failure to the caller.

```
    return -1;
}

int main() /* demo */
{
```

[7] Strictly speaking, `std::queue` is not a container, but rather a container adaptor. Internally, unless specified otherwise, an `std::queue` uses another container, `std::deque` to store the data and implement the operations. It would also be possible, though less convenient, to use `std::deque` directly.

```
    graph g{ { 1, 2 }, { 1, 6 }, { 2, 4 }, { 2, 5 }, { 6, 4 } },
          h{ { 8, 2 }, { 8, 6 }, { 2, 4 }, { 2, 5 }, { 5, 8 } },
          i{ { 2, 4 }, { 4, 2 } };

    assert( distance_to_odd( g, 1 ) ==  2 );
    assert( distance_to_odd( g, 2 ) ==  1 );
    assert( distance_to_odd( g, 6 ) == -1 );

    assert( distance_to_odd( h, 8 ) ==  2 );
    assert( distance_to_odd( h, 5 ) ==  3 );
    assert( distance_to_odd( i, 2 ) == -1 );
}
```

Part 3.e: Elementary Exercises

3.e.1 [unique] Filter out duplicate entries from a vector, maintaining the relative order of entries. Return the result as a new vector.

```
std::vector< int > unique( const std::vector< int > & );
```

3.e.2 [reflexive] Build a reflexive closure of a relation given as a set of pairs, returning the result.

```
using relation = std::set< std::pair< int, int > >;

relation reflexive( const relation &r );
```

3.e.3 [normalize] Given a vector of non-negative floating-point numbers, produce a new vector where all entries fall into the 0-1 range, and they are all related to the original entries by the same factor.

```
using signal_t = std::vector< double >;

signal_t normalize( const signal_t & );
```

Part 3.p: Preparatory Exercises

3.p.1 [brackets] Check that curly and square brackets in a given string balance out correctly.

```
bool balanced( const std::string & );
```

3.p.2 [connected] Decompose an undirected graph into connected components (described by a set of sets of numbers). The graph is given as a symmetric adjacency matrix. Vertices are numbered from 1 to n where n is the dimension of the input matrix.

```
using graph = std::vector< std::vector< bool > >;

using component = std::set< int >;
using components = std::set< component >;

components decompose( const graph &g );
```

3.p.3 [dag] Another exercise for graph exploration, this time we will look for cycles. There are a few algorithms to choose from, those based on DFS are probably the most straightforward.
This time, the input graph is given as a multimap: a map which can contain multiple values for each key. In other words, it behaves as a set of pairs with additional support for efficient retrieval based on the value of the first field of the pair. The `is_dag` function should return `false` iff `g` contains a cycle. The graph does not need to be connected.

```
using graph = std::multimap< int, int >;
bool is_dag( const graph &g );
```

3.p.4 [rel] This exercise demonstrates use of `std::tuple` and structural bindings. Since we cannot write generic code yet (and even if we did, writing the below operators in full generality would be rather tricky), we will only work with a fixed set of types (relations).

First a bunch of type aliases: `item` and its variants each represent a single row, while `rel` and its variants represent an entire relation.

```
using item     = std::tuple< std::string, int, double >;
using item_dbl = std::tuple< std::string, double >;
using item_int = std::tuple< std::string, int >;

using rel     = std::set< item >;
using rel_dbl = std::set< item_dbl >;
using rel_int = std::set< item_int >;
```

Projections: keep a subset of columns, in this case the string and either of the numeric columns.

```
rel_int project_int( const rel & );
rel_dbl project_dbl( const rel & );
```

Selection: keep a subset of rows – those that match on the given column. Throw away all the rest.

```
rel select_str( const rel &, const std::string &n );
rel select_int( const rel &, int n );
```

3.p.5 [numbers] The class represents a set of integers; operations:

- `add` – adds a number, returns true if it was new
- `del` – removes a number, returns true if it was present
- `del_range` – removes numbers within given bounds (inclusive)
- `merge` – adds all numbers from another instance
- `has` – returns true if the given number is in the set

Complexity requirements:

- `del_range` and `merge` must run in O(n) time
- everything else in O(logn) time

```
class numbers;
```

3.p.6 [bipartite]

```
using edges = std::vector< int >;
using graph = std::map< int, edges >;
```

Check whether a given graph is bipartite. The graph is undirected, i.e. its adjacency relation is symmetric.

```
bool is_bipartite( const graph &g );
```

Part 3.r: Regular Exercises

3.r.1 [mode] Find the mode (most common value) in a non-empty vector and return it. If there are more than one, return the smallest.

```
int mode( const std::vector< int > & );
```

3.r.2 [buckets] Sort stones into buckets, where each bucket covers a range of weights; the range of stone weights to put in each bucket is given in an argument – a vector with one element per bucket, each element a pair of min/max values (inclusive). Assume the bucket ranges do not overlap. Stones are given as a vector of weights. Throw away stones which do not fall into any bucket. Return the weights of individual buckets.

```
using bucket = std::pair< int, int >;

std::vector< int > sort( const std::vector< int > &stones,
                         const std::vector< bucket > &buckets );
```

3.r.3 [shortest] Compute single-source shortest path distances for all vertices in an unweighted directed graph. The graph is given using adjacency (successor) lists. The result is a map from a vertex to its shortest distance from `initial`. Vertices which are not reachable from

initial should not appear in the result.

```
using edges = std::vector< int >;

using graph = std::map< int, edges >;

std::map< int, int > shortest( const graph &g, int initial );
```

Part 4: Overloading, Constructors and Lifetime

First, we will look at function and method overloading, including overloading of constructors and overloading on reference kinds. We will also touch the topic of object lifetime (which considers the question of when exactly is an object valid and can be used) and ownership (controlling the lifetime of 'subordinate' objects, e.g. elements in a container).
Demonstrations:

1. art – overloading basics, with books and paintings
2. numbers – a list of numbers which remember their type
3. refs – overloading with references
4. pool – ownership and indirect references

Elementary exercises:

1. diameter – basic function overloading (circle diameter)
2. circle – same story, but with constructors
3. index – access elements of different types using indices

Preparatory exercises:

1. format – method overloading 101
2. least – return a least element without making copies
3. area – geometry with function and ctor overloads
4. zipper – const method overloading on a zipper
5. rpn – postfix arithmetic with more overloading
6. eval – infix evaluation with a node pool

Regular exercises:

1. complex – complex numbers and function overloading
2. XXX – ... references again?
3. search – binary search tree with a pool of nodes
4. bitptr – pointer to a single bit
5. readint – read integers from various string types
6. XXX

Part 4.d: Demonstrations

4.d.1 [art] In this demo, we will look at overloading of standard toplevel functions. We will use 3 record types to represent artistic works: books of fiction, musical compositions and paintings. They have some common attributes, but they are also quite different. We will use function overloading to provide uniform access to the common attributes.
We will use a very simplified view of periodization of art, one that can be more-or-less applied to all 3 types of work which we are interested in. It's perhaps important to note, that the historical periods associated with those styles do not exactly coincide in the 3 disciplines.

```
enum class style_t
{
    antique, medieval, renaissance, baroque, classical, romantic,
modern
};
```

The three record types: a book has an author, a name and a publisher, along with a style. A composition additionally has a key (e.g. 'c minor') and a list of parts. On the other hand, a painting does not have a publisher, but we can associate a technique with it (say, 'oil on canvas'). For simplicity, we store everything as free-form strings .

```
struct book
{
    std::string author, name, publisher;
    style_t style;
};

struct composition
{
    std::string author, name, key, publisher;
    std::vector< std::string > parts;
    style_t style;
};

struct painting
{
    std::string author, name, technique;
    style_t style;
};
```

Now the functions: the first will be the simplest, essentially just forwarding to attribute access. In practice, a function like this is not especially useful, but it is simple.

```
std::string author( book b ) { return b.author; }
std::string author( composition c ) { return c.author; }
std::string author( painting p ) { return p.author; }
```

A slightly more interesting function will be description, which takes some of the attributes and combines them into a single human-readable string describing the work.

```
std::string description( book b )
{
    return b.name + " by " + b.author;
}

std::string description( composition c )
{
    return c.name + " in " + c.key + " by " + c.author;
}

std::string description( painting p )
{
    return p.name + " by " + p.author + " (" + p.technique + ")";
}
```

Another attribute that is shared by books and composition is the name of the publisher. But there is no equivalent concept for paintings. What now? There are a few options: we could leave the overload undefined, which is clearly correct, but not super helpful. Or we can implement an overload which returns some placeholder value. Let's do that here.

```
std::string publisher( book b ) { return b.publisher; }
std::string publisher( composition c ) { return c.publisher; }
std::string publisher( painting ) { return "n/a"; }
```

And finally, for the thorny issue of periods. We sort-of managed to come up with a list of periods which we can sort-of apply to everything, but the years covered differ in each discipline. So the overloads will take care of this.

```
std::pair< int, int > period( book b )
{
    switch ( b.style )
    {
        case style_t::antique: return { -1200, 455 };
```

```
        case style_t::medieval: return { 455, 1485 };
        case style_t::renaissance: return { 1485, 1660 };
        case style_t::baroque: return { 1600, 1680 };
        case style_t::classical: return { 1660, 1790 };
        case style_t::romantic: return { 1770, 1850 };
        case style_t::modern: return { 1850, 2021 };
        default: assert( false );
    }
}

std::pair< int, int > period( composition c )
{
    switch ( c.style )
    {
        case style_t::antique: return { -1300, 500 };
        case style_t::medieval: return { 500, 1400 };
        case style_t::renaissance: return { 1400, 1600 };
        case style_t::baroque: return { 1580, 1750 };
        case style_t::classical: return { 1750, 1820 };
        case style_t::romantic: return { 1800, 1910 };
        case style_t::modern: return { 1890, 2021 };
        default: assert( false );
    }
}

std::pair< int, int > period( painting p )
{
    switch ( p.style )
    {
        case style_t::antique: return { -3000, 500 };
        case style_t::medieval: return { 500, 1400 };
        case style_t::renaissance: return { 1300, 1600 };
        case style_t::baroque: return { 1600, 1730 };
        case style_t::classical: return { 1780, 1850 };
        case style_t::romantic: return { 1800, 1860 };
        case style_t::modern: return { 1860, 2021 };
        default: assert( false );
    }
}
```

Finally, we will check that we can indeed call the functions uniformly on different types input types.

```
int main() /* demo */
{
    book antigone{ "Sophocles", "Antigone", "n/a", style_t::antique
},
         miserables{ "Victor Hugo", "Les Misérables",
                     "A. Lacroix, Verboeckhoven & Cie.",
                     style_t::romantic };

    composition
        bach_mass{ "J. S. Bach", "Mass", "b minor",
                   "Bach Gesellshaft",
                   { "soprano 1", "soprano 2", "alto", "tenor",
                     "bass", "flute 1", "flute 2", "oboe/d'amore 1",
                     "oboe/d'amore 2", "oboe 3", "bassoon 1",
                     "bassoon 2", "horn", "trumpet 1", "trumpet 2",
                     "trumpet 3", "timpani", "violin 1", "violin 2",
                     "viola", "basso continuo" },
                   style_t::baroque },

        fantasia{ "Bohuslav Martinů", "Fantasia H.301", "n/a",
                  "Max Eschig",
                  { "theremin", "oboe",
                    "violin 1", "violin 2", "viola", "violoncello",
                    "piano" },
                  style_t::modern };

    painting babel{ "Pieter Bruegel the Elder",
                    "The Tower of Babel",
                    "oil on wood", style_t::renaissance },
              boon{ "James Brooks", "Boon", "oil on canvas",
                    style_t::modern };
```

Getting a description:

```
    assert( description( bach_mass ) ==
            "Mass in b minor by J. S. Bach" );
    assert( description( babel ) ==
            "The Tower of Babel by Pieter Bruegel the Elder "
            "(oil on wood)" );
    assert( description( antigone ) == "Antigone by Sophocles" );
```

And periods:

```
    assert( period( bach_mass ) == std::pair( 1580, 1750 ) );
    assert( period( fantasia ) == std::pair( 1890, 2021 ) );
    assert( period( boon ) == std::pair( 1860, 2021 ) );
}
```

4.d.2 [numbers] In this demonstration, we will look at overloading: both of regular methods and of constructors. The first class which we will implement is number, which can represent either a real (floating-point) number or an integer. Besides the attributes integer and real which store the respective numbers, the class remembers which type of number it stores, using a boolean attribute called is_real.

```
struct number
{
    bool is_real;
    int integer;
    double real;
```

We provide two constructors for number: one for each type of number that we wish to store. The overload is selected based on the type of argument that is provided.

```
    explicit number( int i ) : is_real( false ), integer( i ) {}
    explicit number( double r ) : is_real( true ), real( r ) {}
};
```

The second class will be a container of numbers which directly allows the user to insert both floating-point and integer numbers, without converting them to a common type. To make insertion convenient, we provide overloads of the add method. Access to the numbers is index-based and is provided by the at method, which is overloaded for entirely different reasons.

```
class numbers
{
```

The sole attribute of the numbers class is the backing store, which is an std::vector of number instances.

```
    std::vector< number > _data;
public:
```

The two add overloads both construct an appropriate instance of number and push it to the backing vector. Nothing surprising there.

```
    void add( double d ) { _data.emplace_back( d ); }
    void add( int i )    { _data.emplace_back( i ); }
```

The overloads for at are much more subtle: notice that the argument types are all identical – there are only 2 differences, first is the return type, which however does not participate in overload resolution. If two functions only differ in return type, this is an error, since there is no way to select which overload should be used.
The other difference is the const qualifier, which indeed does participate in overload resolution. This is because methods have a hidden argument, this, and the trailing const concerns this argument. The const method is selected when the call is performed on a const object (most often because the call is done on a constant reference).

```
    const number &at( int i ) const { return _data.at( i ); }
    number &at( int i ) { return _data.at( i ); }
};

int main() /* demo */
{
    numbers n;
    n.add( 7 );
    n.add( 3.14 );

    assert( !n.at( 0 ).is_real );
    assert(  n.at( 1 ).is_real );

    assert( n.at( 0 ).integer == 7 );
```

Notice that it is possible to assign through the `at` method, if the object itself is mutable. In this case, overload resolution selects the second overload, which returns a mutable reference to the `number` instance stored in the container.

```
    n.at( 0 ) = number( 3 );
    assert( n.at( 0 ).integer == 3 );
```

However, it is still possible to use `at` on a constant object – in this case, the resolution picks the first overload, which returns a constant reference to the relevant `number` instance. Hence, we cannot change the number this way (as we expect, since the entire object is constant, and hence also each of its components).

```
    const numbers &n_const = n;
    assert( n_const.at( 0 ).integer == 3 );

    // n_const.at( 1 ) = number( 1 ); this will not compile
}
```

4.d.3 [refs] In this demonstration, we will look at overloading functions based on different kinds of references. This will allow us to adapt our functions to the kind of value (and its lifetime) that is passed to them, and to deal with arguments efficiently (without making unnecessary copies). But first, let's define a few type aliases:

```
using int_pair   = std::pair< int, int >;
using int_vector = std::vector< int >;
using int_matrix = std::vector< int_vector >;
```

Our goal will be simple enough: write a function which gives access to the first element of any of the above types. In the case of `int_matrix`, the element is an entire row, which has some important implications that we will discuss shortly.
Our main requirements will be that:

1. `first` should work correctly when we call it on a constant,
2. when called on a mutable value, `first( x ) = y` should work and alter the value `x` (i.e. update the first element of `x`).

These requirements are seemingly contradictory: if we return a value (or a constant reference), we can satisfy point 1, but we fail point 2. If we return a mutable reference, point 2 will work, but point 1 will fail. Hence we need the result type to be different depending on the argument. This can be achieved by overloading on the argument type. However, we still have one problem: how do we tell apart, using a type, whether the passed value was constant or not? Think about this: if you write a function which accepts a mutable reference, it cannot be called on an argument which is constant: the compiler will complain about the value losing its `const` qualifier (if you never encountered this behaviour, try it out; it's important that you understand this).
But that means that `first( int_pair &arg )` can only be called on mutable arguments, which is exactly what we need. Fortunately for us, if the compiler decides that this particular `first` cannot be used (because of missing `const`), it will keep looking for some other `first` that might work. You hopefully remember that `first( const int_pair &arg )` can be called on any value of type `int_pair` (without creating a copy). If we provide both, the compiler will use the non-`const` version if it can, but fall back to the `const` one otherwise. And since overloaded functions can differ in their return type, we have our solution:

```
int &first(       int_pair &p ) { return p.first; }
int  first( const int_pair &p ) { return p.first; }
```

The case of `int_vector` is completely analogous:

```
int &first(       int_vector &v ) { return v[ 0 ]; }
int  first( const int_vector &v ) { return v[ 0 ]; }
```

Since in the above cases, the return value was of type `int`, we did not bother with returning `const` references. But when we look at `int_matrix`, the situation has changed: the value which we return is an `std::vector`, which could be very expensive to copy. So we will want to avoid that. The first case (mutable argument), stays the same – we already returned a reference in this case.

```
int_vector &first( int_matrix &v ) { return v[ 0 ]; }
```

At first glance, the second case would seem straightforward enough – just return a `const int_vector &` and be done with it. But there is a catch: what if the argument is a temporary value, which will be destroyed at the end of the current statement? It's not a very good idea to return a reference to a doomed object, since an unwitting caller could get into serious trouble if they store the returned reference – that reference will be invalid on the next line, even though there is no obvious reason for that at the callsite.
You perhaps also remember, that the above function, with a mutable reference, cannot be used with a temporary as its argument: like with a constant, the compiler will complain that it cannot bind a temporary to an argument of type `int_matrix &`. So is there some kind of a reference that can bind a temporary, but not a constant? Yes, that would be an rvalue reference, written `int_matrix &&`. If the above candidate fails, the next one the compiler will look at is one with an rvalue reference as its argument. In this case, we know the value is doomed, so we better return a value, not a reference into the doomed matrix. Moreover, since the input matrix is doomed anyway, we can steal the value we are after using `std::move` and hence still manage to avoid a copy.

```
int_vector first( int_matrix &&v ) { return std::move( v[ 0 ] ); }
```

If both of the above fail, the value must be a constant – in this case, we can safely return a reference into the constant. The argument is not immediately doomed, so it is up to the caller to ensure that if they store the reference, it does not outlive its parent object.

```
const int_vector &first( const int_matrix &v )
{
    return v[ 0 ];
}
```

That concludes our quest for a polymorphic accessor. Let's have a look at how it works when we try to use it:

```
int main() /* demo */
{
    int_vector v{ 3, 5, 7, 1, 4 };
    assert( first( v ) == 3 );
    first( v ) = 5;
    assert( first( v ) == 5 );

    const int_vector &const_v = v;
    assert( first( const_v ) == 5 );

    int_matrix m{ int_vector{ 1, 2, 3 }, v };
    const int_matrix &const_m = m;

    assert( first( first( m ) ) == 1 );
    first( first( m ) )= 2;
```

```
    assert( first( first( const_m ) ) == 2 );
    assert( first( first( int_matrix{ v, v } ) ) == 5 );
```

What follows is the case where the rvalue-reference overload of `first` (the one which handles temporaries) saves us: try to comment the overload out and see what happens on the next 2 lines for yourself.

```
    const int_vector &x = first( int_matrix{ v, v } );
    assert( first( x ) == 5 );
}
```

4.d.4 [pool] This demo will be our first serious foray into dealing with object lifetime. In particular, we will want to implement binary trees – clearly, the lifetime of tree nodes must exactly match the lifetime of the tree itself:

- if the nodes were released too early, the program would perform invalid memory access when traversing the tree,
- if the nodes are not released with the tree, that would be a memory leak – we keep the nodes around, but cannot access them.

This is an ubiquitous problem, and if you think about it, we have encountered it a lot, but did not have to think about it yet: the characters in an `std::string` or the items in an `std::vector` have the same property: their lifetime must match the lifetime of the instance which owns them.

This is one of the most important differences between C and C++: if we do C++ right, most of the time, we do not need to manage object lifetimes manually. This is achieved through two main mechanisms:

1. pervasive use of automatic variables, through value semantics – local variables and arguments are automatically destroyed when they go out of scope,
2. cascading – whenever an object is destroyed, its attributes are also destroyed automatically, and a mechanism is provided for classes which own additional, non-attribute objects (e.g. elements in an `std::vector`) to automatically destroy them too (this is achieved by user-defined destructors which we will explore in part 6, two weeks from now).

In general, destroying objects at an appropriate time is the job of the owner of the object – whether the owner is a function (this is the case with by-value arguments and local variables) or another object (attributes, elements of a container and so on). Additionally, this happens transparently for the user: the compiler takes care of inserting the right calls at the right places to ensure everything is destroyed at the right time.

The end result is modular or composable resource management – well-behaved objects can be composed into well-behaved composites without any additional glue or boilerplate.

To make things easy for now, we will take advantage of existing containers to do resource management for us, which will save us from writing destructors (the proverbial glue, which is boring to write and easy to get wrong). In part 7, we will see how we can use smart pointers for the same purpose.

We will be keeping the nodes of our binary tree in an `std::vector` – this means that each node has an index which we can use to refer to that node. In other words, in this demo (and in some of this week's exercises) indices will play the role of pointers. Since 0 is a valid index, we will use -1 to indicate an invalid ('null') reference. Besides 'pointers' to the left and right child, the node will contain a single integer value.

```
struct node
{
    int left = -1, right = -1;
    int value;
};
```

As mentioned earlier, the nodes will be held in a vector: let's give a name to the particular vector type, for convenience:

```
using node_pool = std::vector< node >;
```

Working with `node` is, however, rather inconvenient: we cannot 'dereference' the `left`/`right` 'pointers' without going through `node_pool`. This makes for verbose code which is unpleasant to both read and to write. But we can do better: let's add a simple wrapper class, which will remember both a reference to the `node_pool` and an index of the `node` we are interested in: this class can then behave like a proper reference to `node`: only a value of the `node_ref` type is needed to access the node and to walk the tree.

```
class node_ref
{
    node_pool &_pool;
    int _idx;
```

To make the subsequent code easier to write (and read), we will define a few helper methods: first, a `get` method which returns an actual reference to the `node` instance that this `node_ref` represents.

```
    node &get() { return _pool[ _idx ]; }
```

And a method to construct a new `node_ref` using the same pool as this one, but with a new index.

```
    node_ref make( int idx ) { return { _pool, idx }; }
```

Normally, we do not want to expose the `_pool` or `node` to users directly, hence we keep them private. But it's convenient for `tree` itself to be able to access them. So we make `tree` a friend.

```
    friend class tree;

public:
    node_ref( node_pool &p, int i ) : _pool( p ), _idx( i ) {}
```

For simplicity, we allow invalid references to be constructed: those will have an index -1, and will naturally arise when we encounter a node with a missing child – that missing node is represented as index -1. The `valid` method allows the user to check whether the reference is valid. The remaining methods (`left`, `right` and `value`) must not be called on an invalid `node_ref`. This is the moral equivalent of a null pointer.

```
    bool valid() const { return _idx >= 0; }
```

What follows is a simple interface for traversing and inspecting the tree. Notice that `left` and `right` again return `node_ref` instances. This makes tree traversal simple and convenient.

```
    node_ref left()  { return make( get().left ); }
    node_ref right() { return make( get().right ); }

    int &value() { return get().value; }
};
```

Finally the class to represent the tree as a whole. It will own the nodes (by keeping a `node_pool` of them as an attribute, will remember a root node (which may be invalid, if the tree is empty) and provide an interface for adding nodes to the tree. Notice that removal of nodes is conspicuously missing: that's because the pool model is not well suited for removals (smart pointers will be better in that regard).

```
class tree
{
    node_pool _pool;
    int _root_idx = -1;
```

A helper method to append a new `node` to the pool and return its index.

```
    int make( int value )
    {
        _pool.emplace_back();
```

```
            _pool.back().value = value;
            return _pool.size() - 1;
        }

    public:
        node_ref root() { return { _pool, _root_idx }; }
        bool empty() const { return _root_idx == -1; }
```

We will use a vector to specify a location in the tree for adding a node, with values -1 (left) and 1 (right). An empty vector represents at the root node.

```
    using path_t = std::vector< int >;
```

Find the location for adding a node recursively and create the node when the location is found. Assumes that the path is correct.

```
    void add( node_ref parent, path_t path, int value,
              unsigned path_idx = 0 )
    {
        assert( path_idx < path.size() );
        int dir = path[ path_idx ];

        if ( path_idx < path.size() - 1 )
        {
            auto next = dir < 0 ? parent.left() : parent.right();
            return add( next, path, value, path_idx + 1 );
        }

        if ( dir < 0 )
            parent.get().left = make( value );
        else
            parent.get().right = make( value );
    }
```

Main entry point for adding nodes.

```
    void add( path_t path, int value )
    {
        if ( root().valid() )
            add( root(), path, value );
        else
        {
            assert( path.empty() );
            _root_idx = make( value );
        }
    }
};

int main() /* demo */
{
    tree t;
    t.add( {}, 1 );

    assert( t.root().value() == 1 );
    assert( t.root().valid() );
    assert( !t.root().left().valid() );

    t.add( { -1 }, 7 );
    assert( t.root().value() == 1 );
    assert( t.root().left().valid() );
    assert( t.root().left().value() == 7 );

    t.add( { -1, 1 }, 3 );
    assert( t.root().left().right().value() == 3 );
}
```

Part 4.e: Elementary Exercises

4.e.1 [diameter] Standard point in a plane, with x and y coordinates, stored as double-precision floating point numbers, with the obvious constructor.

```
struct point;
```

Define a structure which describes a circle with a given center and a given radius (a point and a non-negative number). Include a straightforward constructor.

```
struct circle_radius;
```

And a structure, which describes a circle using two points: the center and a point on the circle. Again, add a constructor.

```
struct circle_point;
```

Finally, define function diameter which given either of the above representations of a circle, returns its diameter (i.e. twice the radius).

```
// double diameter( ??? );
```

4.e.2 [circle] Standard 2D point.

```
struct point;
```

Implement a structure circle with 2 constructors, one of which accepts a point and a number (center and radius) and another which accepts 2 points (center and a point on the circle itself). Store the circle using its center and radius, in attributes center and radius respectively.

```
struct circle;
```

4.e.3 [index] In this exercise, you will provide index-based access to pairs and vectors of integers, using function overloading. The element function should take an std::vector or an std::pair as its first argument and an index as its second argument. A companion size function should return the number of valid indices for either of the two types of objects.

```
// ??? element( ???, int idx );
// ??? size( ??? );
```

Part 4.p: Preparatory Exercises

4.p.1 [format] In this exercise, we will implement a very simple 'string builder': a class that will help us create strings from smaller pieces. It will have a single overloaded method called add, in 3 variants: it will accept either a string, an integer or a floating-point number (use std::to_string for conversions).
To make it easier to use, add should return a reference to the instance it was called on. See below for examples. The method get should return the constructed string.

```
class string_builder;
```

4.p.2 [least] The class element represents a value which, for whatever reason, cannot be duplicated. Our goal will be to write a function which takes a vector of these, finds the smallest and returns it. Do not change the definition of element in any way.

```
class element
{
    int value;
public:
    element( int v ) : value( v ) {}
    element( element &&v ) : value( v.value ) {}
    element &operator=( element &&v ) = default;
    bool less_than( const element &o ) const { return value <
o.value; }
    bool equal( const element &o ) const { return value == o.value;
}
};

using data = std::vector< element >;
```

Write function `least` (or a couple of function overloads) so that calling `least( d )` where `d` is of type `data` returns the least element in the input vector.

```
// ??? least( ??? )
```

4.p.3 [area] Implement 2 classes which represent 2D shapes: (regular) `polygon` and `circle`. Each of the shapes has 2 constructors:

- `circle` takes either 2 points (center and a point on the circle) or a point and a number (radius),
- `polygon` takes an integer (the number of sides ≥ 3) and either two points (center and a vertex) or a single point and a number (the major radius).

Add a toplevel function `area` which can compute the area of either.

```
struct point;
struct polygon;
struct circle;
```

4.p.4 [zipper] In this exercise, we will implement a simple data structure called a zipper – a sequence of items with a single focused item. Since we can't write class templates yet, we will just make a zipper of integers. Our zipper will have these operations:

- (constructor) constructs a singleton zipper from an integer
- `shift_left` and `shift_right` move the point of focus, in O(1), to the nearest left (right) element; they return true if this was possible, otherwise they return false and do nothing
- `insert_left` and `insert_right` add a new element just left (just right) of the current focus, again in O(1)
- `focus` access the current item (read and write)
- bonus: add `erase_left` and `erase_right` to remove elements around the focus (return `true` if this was possible), in O(1)

```
class zipper;
```

4.p.5 [rpn] Write a simple stack-based evaluator for numeric expressions in an RPN form. The operations:

- `push` takes a number and pushes it onto the working stack,
- `apply` accepts an instance of one of the three operator classes defined below; like with the string builder earlier, both those methods should return a reference to the evaluator,
- again like with the zipper, a `top` method should give access to the current top of the stack, including the possibility of changing the value,
- `pop` which also returns the popped value, and
- `empty` which returns a `bool`.

All three operators are binary (take 2 arguments).

```
struct add {};  /* addition */
struct mul {};  /* multiplication */
struct dist {}; /* absolute value of difference */

class eval;
```

4.p.6 [eval] We will do an infix version of the evaluator from the previous exercise. Additionally, we will want to store common subexpressions only once. For this reason, we will store the nodes in a pool and only take out references to them.

```
struct node
{
```

The type of the node. Only `mul` and `add` nodes have children.

```
    enum op_t { mul, add, constant } op;
```

The attributes `left` and `right` are indices, with -1 indicating an invalid (null) reference. The `is_root` boolean indicates whether this node is a root – that is, it does not appear as a child of any other node.

```
    int left = -1, right = -1;
    bool is_root = true;
```

The value stored in a `constant`-type node.

```
    int value = 0;
};

using node_pool = std::vector< node >;
```

An 'ephemeral' reference to a node – one that can be used to traverse an expression tree, but which is only valid as long as the `eval` instance which created it is alive. Add `const` methods `left()`, `right()` which return another `node_ref` instance, a `const` method `compute()` which evaluates the subtree, and a non-`const` method `update( int )` which only works on nodes of type `constant`.

```
class node_ref;
```

The `eval` class represents an entire expression which can be evaluated, traversed (starting from root nodes – those which have no parent) and, most importantly, extended by creating new nodes.

```
class eval
{
    node_pool _pool;
public:
    std::vector< node_ref > roots();

    node_ref add( node_ref, node_ref );
    node_ref mul( node_ref, node_ref );
    node_ref number( int );
};
```

Part 4.r: Regular Exercises

4.r.1 [complex] Structure `angle` simply wraps a single double-precision number, so that we can use constructor overloads to allow use of both polar and cartesian forms to create instances of a single type (`complex`).

```
struct angle;
struct complex;
```

Now implement the following two functions, so that they work both for real and complex numbers.

```
// double magnitude( … )
// … reciprocal( … )
```

The following two functions only make sense for complex numbers, where `arg` is the argument, normalized into the range $(-\pi, \pi\rangle$:

```
double real( complex );
double imag( complex );
double arg( complex );
```

4.r.2 [search] Implement a binary search tree, i.e. a binary tree which maintains the search property. That is, a value of each node is:

- ≥ than all values in its left subtree,
- ≤ than all values in its right subtree.

Store the nodes in a pool (a vector or a list, your choice). The interface is as follows:

- `node_ref root() const` returns the root node,
- `bool empty() const` checks whether the tree is empty,
- `void insert( int v )` inserts a new value into the tree (without rebalancing).

The `node_ref` class then ought to provide:

- `node_ref left() const` and `node_ref right() const`,
- `bool valid() const`,
- `value() const` which returns the value stored in the node.

Calling `root` on an empty tree is undefined.

```
struct node; /* ref:  6 lines */
```

```
using node_pool = std::vector< node >;

class node_ref; /* ref: 12 lines */
class tree;     /* ref: 28 lines */

std::tuple< bool, int, int > verify( node_ref n, int bound );
bool has( node_ref n, int v );
```

Part T.1: Introductory Tasks

The programming tasks for this block are as follows:

1. `cellular.*` – a simple cellular automaton simulator,
2. `magic.*` – a backtracking magic square solver,
3. `reversi.*` – a 3D version of the game reversi,
4. `chess.*` – a simple simulator of standard chess.

In this set, the tasks only require basic programming skills and C++ constructs that you have encountered in the first two chapters. In other words, no advanced language constructs or library features are necessary.

Part T.1.1: [cellular]

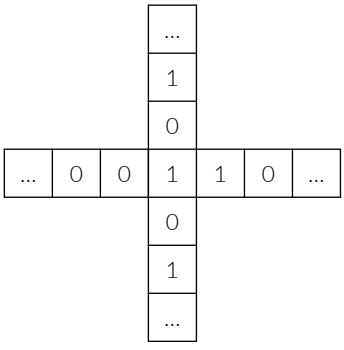
The goal of this task is to implement a simulator for one-dimensional cellular automata. You will implement this simulator as a class, the interface of which is described below You are free to add additional methods and data members to the class, and additional classes and functions to the file, as you see fit. You must, however, keep the entire interface in this single file. The implementation can be in either `cellular.hpp` or in `cellular.cpp`. Only these two files will be submitted. The class `automaton_state` represents the state of a 1D, infinite binary cellular automaton. The `set` and `get` methods can be passed an arbitrary index.

```
class automaton_state
{
```

Attributes are up to you.

```
public:
    automaton_state(); /* create a blank state (all cells are 0) */
    void set( int index, bool value ); /* change the given cell */
    bool get( int index ) const;
};
```

The `automaton` class represents the automaton itself. The automaton is arranged as a cross, with a horizontal and a vertical automaton, which are almost entirely independent (each has its own state and its own rule), with one twist: the center cell (index 0 in both automata) is shared. The new state of the shared center cell (after a computation step is performed in both automata independently) is obtained by combining the two values (that either automaton would assign to that cell) using a specified boolean binary operator. The new center is obtained as `horizontal_center OP vertical_center`. The state can look, for example, like this:



The automaton keeps its state internally and allows the user to perform simulation on this internal state. Initially, the state of the automaton is 0 (false) everywhere. The rules for both the vertical and the horizontal component are given to the constructor by their Wolfram code.
The center-combining operator is given by the same type of code, but instead of 3 cells, only 2 need to be combined: there are only 16 such operators (compared to 256 rules for each of the automata). The input vectors to the binary operator are numbered by their binary code as:

left	right	index
0	0	0
0	1	1
1	0	2
1	1	3

The operator code is then a 4-digit binary number, e.g 0110 gives the code for `xor` ($0\,0 \to 0, 0\,1 \to 1, 1\,0 \to 1, 1\,1 \to 0$) while 1000 gives code for `and` (everything is zero except if both inputs are 1). And so on and so forth. The same process but with 3 input cells is used to construct the Wolfram code for the automata.

```
class automaton
{
```

Attributes are up to you.

```
public:

    enum direction { vertical, horizontal };
```

Constructs an automaton based on a rule given by its Wolfram code for the horizontal component, another for the vertical component, and a 4-bit code for the center operator. Assume that neither of the rules contains the transition $000 \to 1$.

```
    automaton( int h_rule, int v_rule, int center );
```

The `read` method returns the current value of the (shared) center cell. The set method sets the specified cell to the value given.

```
    bool read() const;
    void set( direction dir, int index, bool value );
```

Finally, the following methods run the simulation – either perform a single step (update each cell exactly once) or a given number of steps (assume a non-negative number of steps).

```
    void step();
    void run( int steps );
};
```

The `compute_cell` function takes two rule numbers, two initial states, a center operator and a number of steps. It then computes the value of the central cell after n steps of the automaton such described. Like above, the number of steps is a non-negative number. Assume that the center cell in both input states has the same value.

```
bool compute_cell( int vertical_rule, int horizontal_rule,
                   int center_op,
                   const automaton_state &vertical_state,
```

```
                const automaton_state &horizontal_state,
                int steps );
```

Part T.1.2: [chess]

The goal of this task is to implement the standard rules of chess.

```
struct position
{
    int file; /* column 'letter', a = 1, b = 2, ... */
    int rank; /* row number, starting at 1 */
};

enum class piece_type
{
    pawn, rook, knight, bishop, queen, king
};

enum class player { white, black };
```

The following are the possible outcomes of `play`. The outcomes are shown in the order of precedence, i.e. the first applicable is returned.

`capture`	the move was legal and resulted in a capture
`ok`	the move was legal and was performed
`no_piece`	there is no piece on the `from` square
`bad_piece`	the piece on `from` is not ours
`bad_move`	this move is not available for this piece
`blocked`	another piece is in the way
`lapsed`	en passant capture is no longer allowed
`has_moved`	one of the castling pieces has already moved
`in_check`	the player is currently in check and the move does not get them out of it
`would_check`	the move would place the player in check
`bad_promote`	promotion to a pawn or king was attempted

Attempting an en passant when the pieces are in the wrong place is a `bad_move`. In addition to `has_moved`, (otherwise legal) castling may give:

- `blocked` – some pieces are in the way,
- `in_check` – the king is currently in check,
- `would_check` – would pass through or end up in check.

```
enum class result
{
    capture, ok, no_piece, bad_piece, bad_move, blocked, lapsed,
    in_check, would_check, has_moved, bad_promote
};

struct occupant
{
    bool is_empty;
    player owner;
    piece_type piece;
};

class chess
{
public:
```

Construct a game of chess in its default starting position. The first call to `play` after construction moves a piece of the white player.

```
    chess();
```

Move a piece currently at `from` to square `to`:

- in case the move places a pawn at its 8th rank (rank 8 for white, rank 1 for black), it is promoted to the piece given in `promote` (otherwise, the last argument is ignored),
- castling is described as a king move of more than one square,
- if the result is an error (not `capture` nor `ok`), calling `play` again will attempt another move by the same player.

```
    result play( position from, position to,
                 piece_type promote = piece_type::pawn );
```

Which piece is at the given position?

```
    occupant at( position ) const;
};
```

Part T.1.3: [magic]

A magic square is an $n \times n$ grid of natural numbers $1–n^2$, such that all rows and columns and both diagonals add up to a fixed 'magic constant' and each number appears exactly once. Solving the square means filling in all empty cells in a manner that gives the full square the magic property. The goal of this task is to implement a simple backtracking solver for completing partially filled magic squares.

```
class magic
{
public:
```

Construct an empty $n \times n$ square.

```
    magic( int n );
```

Get the value at the given position. A return value of 0 indicates an empty square.

```
    int get( int x, int y ) const;
```

Set a cell at the given position to a given value. The behaviour is undefined if `v` is already present in the square. If `v` is negative, the cell is empty, but must not take `std::abs( v )` as its value in the solved square.

```
    void set( int x, int y, int v );
```

Solve the square: fill in all empty cells so that the square has the magic property and return `true`. If the square cannot be solved, do not change its content and return `false`.

```
    bool solve();
};
```

Part T.1.4: [reversi]

The subject of this task is the game of reversi (also known as othello), played by two players on a 3D board (a box) of a given shape (given as 3 even, non-negative numbers). Size 0 in a given direction means the board is infinite in that direction.
The cells are cubes (a cube has 8 vertices, 12 edges and 6 faces). The coordinates start at the center (which is a vertex) and extend in two directions (positive and negative) along the 3 axes. The 8 cells which share the center vertex have coordinates [1, 1, 1], [1, 1, -1], [1, -1, 1], [1, -1, -1], ...
The rules are a straightforward extension of the standard 2D rules into three dimensions:

- each player starts with 4 stones placed around the center such that no two (of the same colour) share a face, with white taking the [1, 1, 1] cell,
- players take turns in placing a new stone, which must be placed adjacent (share an edge, vertex or a face) to an opposing player's stone, and enemy stones must form a straight, uninterrupted line

to one of current players' own stones (along straight lines – sharing a face, along diagonals which share an edge, or along diagonals which share a vertex),
- the colour of all opposing stones on all such lines connecting the new stone to existing stones of the current player is flipped.

The white player starts. The game ends when no new stones can be placed and the player with more stones wins. It must be possible to make a copy of an in-progress game.

```
class reversi
{
public:
    reversi( int x_size, int y_size, int z_size );
```

Place a stone at the given coordinates. If the placement was legal, returns `true` and the next call places a stone of the opposing player; otherwise, no change is made, the function returns `false` and the same player must try a different move.
As a special case, if the current player has no legal move left, but the game is not finished, `play` must be called with $x = y = z = 0$ to continue. Doing this is illegal in any other circumstances.
It is undefined behaviour to call `play` when the game is already over.

```
    bool play( int x, int y, int z );
```

Return true if the game is finished (no further moves are possible).

```
    bool finished() const;
```

Only defined if the game is already over (i.e. `finished` would return `true`). Returns the difference in the number of stones of each player: positive for white's victory, negative for black's victory, 0 for a draw.

```
    int result() const;
};
```

Part 5: Operators and IO

The main topics for week 5 are operator overloading (which will build on what we learned about function and method overloading in week 4). The second topic for this week will be IO: we will look at formatted input and output and at reading and writing files.
Demonstrations:

1. `arithmetic` – introduction to operator overloading,
2. `relational` – implementing equality and ordering,
3. `access` – dereference, indexing and other access ops,
4. `convert` – conversion and assignment,
5. `files` – opening files, reading and writing strings
6. `streams` – from values to strings and back
7. `format` – overloading formatting operators

Elementary exercises:

1. `cartesian` – complex numbers in algebraic form,
2. `force` – composing and scaling forces,
3. `forcefmt` – vectors redux, this time with IO

Preparatory exercises:

1. `polar` – complex numbers in polar form,
2. `rational` – rational numbers with ordering,
3. `tmpfile` – an auto-erasing temporary file
4. `nibble` – a pointer-like class for sub-byte access,
5. `grep` – print matching lines
6. `fixnum` – more numbers, this time with a parser.

Regular exercises:

1. `poly` – polynomials with addition and multiplication
2. `csv` – parse comma-separated numeric data
3. `set` – a set of integers with set operators,
4. `email` – a simplified RFC 822 parser
5. `json` – format a string → string map as JSON
6. `cpp` † – a very simple C preprocessor

Part 5.d: Demonstrations

5.d.1 **[arithmetic]** Operator overloading allows instances of classes to behave more like built-in types: it makes it possible for values of custom types to appear in expressions, as operands. Before we look at examples of how this looks, we need to define a class with some overloaded operators. For binary operators, it is customary to define them using a 'friends trick', which allows us to define a top-level function inside a class.

As a very simple example, we will implement a class which represents integral values modulo 7 (this happens to be a finite field, with addition and multiplication).

```
class gf7
{
    int value;
public:
```

The constructor is trivial, it simply constructs a `gf7` instance from an integer. We mark it `explicit` to avoid surprising automatic conversions of integers into `gf7` instances.

```
    explicit gf7( int v ) : value( v % 7 ) {}
```

This is the 'friend trick' syntax for writing operators, and for binary operators, it is often the preferred one (because of its symmetry). The function is not really a part of the class in this case – the trick is that we can write it here anyway.

```
    friend gf7 operator+( gf7 a, gf7 b )
    {
        return gf7( a.value + b.value ); // [a]₇ + [b]₇ = [a + b]₇
    }
```

For multiplication, we will use the more 'orthodox' syntax, where the operator is a `const` method: the left operand is passed into the operator as `this`, the right operand is the argument. In general, operators-as-methods have one explicit argument less (unary operators take 0 arguments, binary take 1 argument).

```
    gf7 operator*( gf7 b ) const
    {
        return gf7( value * b.value ); // [a]₇ * [b]₇ = [a * b]₇
    }
```

Values of type `gf7` cannot be directly compared (we did not define the required operators) – instead, we provide this method to convert instances of `gf7` back into `int`'s.

```
    int to_int() const { return value; }
};
```

Operators can be also overloaded using 'normal' top-level functions, like this unary minus (which finds the additive inverse of the given element). Notice that we cannot access private fields (attributes) of the class here:

```
gf7 operator-( gf7 x ) { return gf7( 7 - x.to_int() ); }
```

Now that we have defined the class and the operators, we can look at how is the result used.

```
int main() /* demo */
{
    gf7 a( 3 ), b( 4 ), c( 0 ), d( 5 );
```

Values a, b and so forth can be now directly used in arithmetic expressions, just as we wanted.

```
    gf7 x = a + b;
    gf7 y = a * b;
```

Let us check that the operations work as expected:

```
    assert( x.to_int() == c.to_int() ); /* [3]7 + [4]7 = [0]7 */
    assert( y.to_int() == d.to_int() ); /* [3]7 * [4]7 = [5]7 */
    assert( (-a + a).to_int() == c.to_int() ); /* unary minus */
}
```

That was arithmetic operator overloading. Let's now look at relational (ordering) operators, in relational.cpp.

5.d.2 [relational] In this example, we will show relational operators, which are very similar to the arithmetic operators from previous example, except for their return types, which are bool values.
The example which we will use in this case are sets of small natural numbers (1-64) with inclusion as the order. We will implement the full set of comparison operators, which is still required in C++17 but will no longer be needed in C++20 (with the spaceship operator).
NB. Standard ordered containers like std::set and std::map require the operator less-than to define a linear order. The comparison operators in this example do not define a linear order.

```
class set
{
```

Each bit of the below number indicates the presence of the corresponding integer (the index of that bit) in the set.

```
    uint64_t bits;
public:
```

Like before, we add an explicit constructor that takes an initial value. We use a default argument to say that the constructor can be used as a default constructor (without arguments), in which case it will create an empty set.

```
    explicit set( uint64_t to_set = 0 ) : bits( to_set ) {}
```

We also define a few methods to add and remove numbers from the set, to test for presence of a number and an emptiness check.

```
    void add( int i ) { bits |=    1ul << i; }
    void del( int i ) { bits &= ~( 1ul << i ); }
    bool has( int i ) const { return bits & ( 1ul << i ); }
    bool empty() const { return !bits; }
```

We will use the method syntax here, because it is slightly shorter. We start with (in)equality, which is very simple (the sets are equal when they have the same members):

```
    bool operator==( set b ) const { return bits == b.bits; }
    bool operator!=( set b ) const { return bits != b.bits; }
```

It will be quite useful to have set difference to implement the comparisons below, so let us also define that:

```
    set operator-( set b ) const { return set( bits & ~b.bits ); }
```

Since the non-strict comparison (ordering) operators are easier to implement, we will do that first. Set b is a superset of set a if all elements of a are also present in b, which is the same as the difference a - b being empty.

```
    bool operator<=( set b ) const { return ( *this - b ).empty(); }
    bool operator>=( set b ) const { return ( b - *this ).empty(); }
};
```

And finally the strict comparison operators, which are more conveniently written using top-level function syntax:

```
bool operator<( set a, set b ) { return a <= b && a != b; }
bool operator>( set a, set b ) { return a >= b && a != b; }

int main() /* demo */
{
    set a; a.add( 1 ); a.add( 7 ); a.add( 13 );
    set b; b.add( 1 ); b.add( 6 ); b.add( 13 );
```

In each pair of assertions below, the two expressions are not quite equivalent. Do you understand why?

```
    assert( a != b ); assert( !( a == b ) );
    assert( a == a ); assert( !( a != a ) );
```

The two sets are incomparable, i.e. neither is less than the other, but as shown above they are not equal either.

```
    assert( !( a < b ) ); assert( !( b < a ) );

    a.add( 6 ); // let's make <a> a superset of <b>
```

And check that the ordering operators work on ordered sets.

```
    assert( a > b ); assert( a >= b ); assert( a != b );
    assert( b < a ); assert( b <= a ); assert( b != a );

    b.add( 7 ); /* let's make the sets equal */
    assert( a == b ); assert( a <= b ); assert( a >= b );
}
```

That's all regarding relational operators, you will have a chance to implement your own in one of the exercises later. In the meantime, let us move on to 'access' operators: dereference, indirect access and indexing, in access.cpp.

5.d.3 [access] This set of operators will be slightly more difficult. Surely, you remember the unary * operator from C, where it is used to dereference pointers. We haven't seen much of that in C++, except perhaps with iterators. We will now see how to implement a class which can be dereferenced like a pointer. We will also add indexing to the mix (like with plain C arrays, or std::vector or even std::map).
Let us revisit the zipper class from last week. We will add indexing (relative to the focus), use a dereference operator to access the focus and we will not store integers, but points in a plane. Cue our favourite class, a point:

```
struct point
{
    double x, y;
    point( double x, double y ) : x( x ), y( y ) {}
```

We know equality comparison from previous examples. We will need it later on for writing test cases for zipper.

```
    bool operator==( point o ) const { return x == o.x && y == o.y;
}
};
```

Now for the zipper. We will need to use std::vector to be able to index elements, but we will still use left and right like stacks.

```
class zipper
{
    using stack = std::vector< point >;
    stack left, right;
```

```
    point focus;
public:

    zipper( double x, double y ) : focus( x, y ) {}
```

Inserting points into the zipper.

```
    zipper &emplace_left( double x, double y )
    {
        left.emplace_back( x, y );
        return *this;
    }

    zipper &emplace_right( double x, double y )
    {
        right.emplace_back( x, y );
        return *this;
    }
```

A helper method, so we don't repeat ourselves in the increment/decrement operators below. The trick is to pass the `left`/`right` stacks by reference, since moving left and right is symmetric with regards to those (i.e. the code to move left is the same as to move right, with all occurrences of `left` and `right` swapped).

```
    void shift( stack &a, stack &b )
    {
        b.push_back( focus );
        focus = a.back();
        a.pop_back();
    }
```

First the pre-increment operators, i.e. `++x` and `--x`. Here, we use those operators in the manner of C pointer arithmetic (you may want to review that topic).

```
    zipper &operator++() { shift( right, left ); return *this; }
    zipper &operator--() { shift( left, right ); return *this; }
```

Now the post-increment: `x++` and `x--`. In this particular data structure, they are expensive and should not be used. They are here just to demonstrate the syntax and a common implementation technique. The difference is that post-increment needs to make a copy, since the value of the expression is the object before the increment/decrement was applied to it.

```
    zipper operator++( int ) { auto r = *this; ++*this; return r; }
    zipper operator--( int ) { auto r = *this; --*this; return r; }
```

The dereference (unary `*`) and indirect member access operators (mutable, i.e. non-`const` overloads first, then the `const` overloads). Those operators allow us to treat `zipper` as if it was a pointer to a `point` instance (the one that is in focus). See `main` below to see how this works when used.

```
    point &operator*()  { return  focus; }
    point *operator->() { return &focus; }

    const point &operator*()  const { return focus; }
    const point *operator->() const { return &focus; }
```

And finally an indexing operator. We will not bother with the `const` version at this time: it would be certainly possible, but ugly and/or repetitive.

```
    point &operator[]( int i )
    {
        if ( i == 0 ) return focus;
        if ( i < 0 ) return left[ left.size() + i ];
        if ( i > 0 ) return right[ right.size() - i ];
        assert( false );
    }
};

int main() /* demo */
{

    zipper z( 0, 0 ); // [0,0]
```

Notice the correspondence between `*x` and `x[ 0 ]` that we carried over from C pointers.

```
    assert( z[ 0 ] == point( 0, 0 ) );
    assert( *z == point( 0, 0 ) );
```

We will add a few items to the zipper, so that we can demonstrate the other operators.

```
    z.emplace_left( 1, 1 );  // (1,1) [0,0]
    z.emplace_right( 2, 1 ); // (1,1) [0,0] (2,1)
```

Check that the indexing operators behave as expected: negative indices give us items on the left and positive indices give us items on the right.

```
    assert( z[ -1 ] == point( 1, 1 ) );
    assert( z[ 1 ]  == point( 2, 1 ) );
```

Let us check that indexing also works further out.

```
    z.emplace_left( 2, 2 ); // (1,1) (2,2) [0,0] (2,1)
    assert( z[ -2 ] == point( 1, 1 ) );
    assert( z[ -1 ] == point( 2, 2 ) );
```

The pre-decrement operator moves the focus of the zipper tho the left. Let's check that (and demonstrate the correspondence between `z[ 0 ]` and `*z` again, for a good measure).

```
    -- z; // (1,1) [2,2] (0,0) (2,1)
    assert( z[ -1 ] == point( 1, 1 ) );
    assert( z[ 0 ] == point( 2, 2 ) );
    assert( *z == point( 2, 2 ) );
```

Finally the indirect access operators let us look at `x` and `y` of the focused point in a nice, succinct way. The syntax is the same that you used to access `struct` members via a pointer to the `struct` in C.

```
    assert( z->x == 2 );
    assert( z->y == 2 );
```

Move the zipper twice to the right and do a final check.

```
    ++ z; ++ z; // (1,1) (2,2) (0,0) [2,1]
    assert( z->x == 2 );
    assert( z->y == 1 );
}
```

Next: quick introduction to exceptions, in `exceptions.cpp`.

5.d.4 [convert] In this example, we will implement a class which behaves like a nullable reference to an integer. Taking a hint from Java, we will throw an exception when the user attempts to use a null reference.
We first define the type which we will use to indicate an attempt to use an invalid (null) reference.

```
class null_pointer_exception {};
```

Now for the reference-like class itself. We need two basic ingredients to provide simple reference-like behaviours: we need to be able to (implicitly) convert a value of type `maybe_ref` to a value of type `int`. The other part is the ability to assign new values of type `int` to the referred-to variable, via instances of the class `maybe_ref`.

```
class maybe_ref
{
```

We hold a pointer internally, since real references in C++ cannot be null.

```
    int *_ptr = nullptr;
```

We will also define a helper (internal, private) method which checks whether the reference is valid. If the reference is null, it throws the above exception.

```
    void _check() const
    {
        if ( !_ptr )
            throw null_pointer_exception();
    }

public:
```

Constructors: the default-constructed `maybe_ref` instances are nulls, they have nowhere to point. Like real references in C++, we will allow `maybe_ref` to be initialized to point to an existing value. We take the argument by reference and convert that reference into a pointer by using the unary `&` operator, in order to store it in `_ptr`.

```
    maybe_ref() = default;
    maybe_ref( int &i ) : _ptr( &i ) {}
```

As mentioned earlier, we need to be able to (implicitly) convert `maybe_ref` instances into integers. The syntax to do that is `operator type`, without mentioning the return type (in this case, the return type is given by the name of the operator, i.e. `int` here). It is also possible to have reference conversion operators, by writing e.g. `operator const int &()`. However, we don't need one of those here because `int` is small, and we can't have both since that would cause a lot of ambiguity.

```
    operator int() const
    {
        _check();
        return *_ptr;
    }
```

The final part is assignment: as you have learned in the lecture, `operator=` should return a reference to the assigned-to instance. It usually takes a `const` reference as an argument, but again we do not need to do that here. Below in the demo, we will point out where the assignment operator comes into play.

```
    maybe_ref &operator=( int v )
    {
        _check();
        *_ptr = v;
        return *this;
    }
};

int main() /* demo */
{
    int i = 7;
```

When initializing built-in references, we use `int &i_ref = i`. We can do the same with `maybe_ref`, but we need to keep in mind that this syntax calls the `maybe_ref( int )` constructor, not the assignment operator.

```
    maybe_ref i_ref = i;
```

Let us check that the reference behaves as expected.

```
    assert( i_ref == 7 ); /* uses conversion to <int> */
    i_ref = 3;            /* uses the assignment operator */
    assert( i_ref == 3 ); /* conversion to <int> again */
```

Check that the original variable has changed too.

```
    assert( i == 3 );
```

Let's also check that null references behave as expected.

```
    bool caught = false;
    maybe_ref null;
```

Comparison will try to convert the reference to `int`, but that will fail in `_check` with an exception.

```
    try { assert( null == 7 ); }
    catch ( const null_pointer_exception & ) { caught = true; }
```

Make sure that the exception was thrown and caught.

```
    assert( caught );
    caught = false;
```

Same but with assignment into the null referenc.

```
    try { null = 2; }
    catch ( const null_pointer_exception & ) { caught = true; }

    assert( caught );
}
```

5.d.5 [files] This example will be brief: we will show how to open a file for reading and fetch a line of text. We will then write that line of text into a new file and read it back again to check that things worked. We will split up the example into functions for 2 reasons: first, to make it easier to follow, and second, to take advantage of RAII: the file streams will close the underlying resource when they are destroyed. In this case, that will be at the end of each function.

```
std::string read( const char *file )
{
```

The default method of doing IO in C++ is through streams. Reading files is done through a stream of type `std::ifstream`, which is short for input file stream. The constructor of `ifstream` takes the name of the file to open. We will use a file given to us by the caller.

```
    std::ifstream f( file );
```

The simplest method to read text from a file is using `std::getline`, which will fetch a single line at a time, into an `std::string`. We need to prepare the string in advance, since it is passed into `std::getline` as an output argument.

```
    std::string line;
```

The `std::getline` function returns a reference to the stream that was passed to it. Additionally, the stream can be converted to `bool` to find out whether everything is okay with it. If the reading fails for any reason, it will evaluate to `false`. The newline character is discarded.

```
    if ( !std::getline( f, line ) )
```

In real code, we would of course want to handle errors, because opening files is something that can fail for a number of reasons. Here, we simply assume that everything worked.

```
        assert( false );

    return line;
}
```

Next comes a function which demonstrates writing into files.

```
void write( const char *file, std::string line )
{
```

To write data into a file, we can use `std::ofstream`, which is short for output file stream. The output file is created if it does not exist.

```
    std::ofstream f( file );
```

Writing into a file is typically done using operators for formatted out-

put. We will look at those in more detail in the next section. For now, all we need to know that writing an object into a stream is done like this:

```
f << line;
```

We will also want to add the newline character that getline above chomped. We have two options: either use the "\n" string literal, or std::endl – a so-called stream manipulator which sends a newline character and asks the stream to send the bytes to the operating system. Let's try the more idiomatic approach, with the manipulator:

```
f << std::endl;
```

At this point, the file is automatically closed and any outstanding data is sent to the operating system.

```
}

int main() /* demo */
{
```

We first use read to get the first line of this file.

```
std::string line = read( "d5_files.cpp" );
```

And we check that the line we got is what we expect. Remember the stripped newline.

```
assert( line ==  "/* This example will be brief:"
                 " we will show how to open a file for" );
```

Now we write the line into another file. After you run this example, you can inspect files.out with an editor. It should contain a copy of the first line of this file.

```
write( "d5_files.out", line );
```

Finally, we use read again to read "file.out" back, and check that the same thing came back.

```
std::string check = read( "d5_files.out" );
assert( check == line );
}
```

5.d.6 [streams] File streams are not the only kind of IO streams that are available in the standard library. There are 3 'special' streams, called std::cout, std::cerr and std::cin. Those are not types, but rather global variables, and represent the standard output, the standard error output and the standard input of the program. However, the first two are instances of std::ostream and the third is an instance of std::istream.

We don't know about class inheritance yet, but it is probably not a huge stretch to understand that instances of std::ofstream (output file stream) are also at the same time instances of std::ostream (general output stream). The same story holds for std::ifstream (input file stream) and std::istream (general input stream).

There is another pair of classes: std::ostringstream and std::istringstream. Those streams are not attached to OS resources, but to instances of std::string: in other words, when you write to an ostringstream, the resulting bytes are not sent to the operating system, but are instead appended to the given string. Likewise, when you read from an istringstream, the data is not pulled from the operating system, but instead come from an std::string. Hopefully, you can see the correspondence between files (the content of which are byte sequences stored on disk) and strings (the content of which are byte sequences stored in RAM).

In any case, string streams are ideal for playing around, because we can use the same tools as we always do: create some simple instances, apply operations and use assert to check that the results are what we expect. String-based streams are defined in the header sstream.

Everything that we will do with string streams applies to other types of streams too (i.e. the 3 special streams mentioned earlier, and all file streams).

Like in the previous example, we will split up the demonstration into a few sections, mainly to avoid confusion over variable names. We will first demonstrate reading from streams. We have already seen std::getline, so let's start with that. It is probably noteworthy that it works on any input stream, not just std::ifstream.

```
void getline_1()
{
    std::istringstream istr( "a string\nwith 2 lines\n" );
    std::string s;

    assert( std::getline( istr, s ) );
    assert( s == "a string" );
    assert( std::getline( istr, s ) );
    assert( s == "with 2 lines" );
    assert( !std::getline( istr, s ) );
    assert( s.empty() );
}
```

We can also override the delimiter character for std::getline, to extract delimited fields from input streams.

```
void getline_2()
{
    std::istringstream istr( "colon:separated fields" );
    std::string s;

    assert( std::getline( istr, s, ':' ) );
    assert( s == "colon" );
    assert( std::getline( istr, s, ':' ) );
    assert( s == "separated fields" );
    assert( !std::getline( istr, s, ':' ) );
}
```

So far so good. Our other option is so-called formatted input. The standard library doesn't offer much in terms of ready-made overloads for such inputs: there is one for strings, which extracts individual words (like the scanf specifier %s, if you remember that from C, but the C++ version is actually safe and it is okay to use it). Then there is an instance for char, which extracts a single character (regardless of whether it is a whitespace character or not) and a bunch of overloads for various numeric types.

```
void formatted_input()
{
    std::istringstream istr( "integer 123 float 3.1415 s t" );
    std::string s, t;
    int i; float f;

    istr >> s; assert( s == "integer" );
    istr >> i; assert( i == 123 );
    istr >> s; assert( s == "float" );
```

Notice that float numbers are not very exact. They are usually just 32 bits, which means 24 bits of precision, which is a bit less than 8 decimal digits.

```
istr >> f; assert( std::fabs( f - 3.1415 ) < 1e-7 );
```

The last thing we want to demonstrate with regards to the formatted input operators is that we can chain them. The values are taken from left to right (behind the scenes, this is achieved by the formatted input operator returning a reference to its left operand.

```
istr >> s >> t;
assert( s == "s" && t == "t" );
```

When we reach the end of the stream (i.e. the end of the buffer, or of

the file), the stream will indicate an error. A stream in error condition converts to `false` in a `bool` context.

```
    assert( !( istr >> s ) );
}
```

Output is actually quite a bit simpler than input. It is almost always reasonable to use formatted output, since strings are simply copied to the output without alterations.

```
void formatted_output()
{
    std::ostringstream a, b, c;
    a << "hello world";
```

To read the buffer associated with an output string stream, we use its method `str`. Of course, this method is not available on other stream types: in those cases, the characters are written to files or to the terminal and we cannot access them through the stream anymore.

```
    assert( a.str() == "hello world" );
```

Like with formatted input, output can be chained.

```
    b << 123 << " " << 3.1415;
    assert( b.str() == "123 3.1415" );
```

When writing delimited values to an output stream, it is often desirable to only put the delimiter between items and not after each item: this is an endless source of headaches. Here is a trick to do it without too much typing:

```
    int i = 0;
    for ( int v : { 1, 2, 3 } )
        c << ( i++ ? ", " : "" ) << v;

    assert( c.str() == "1, 2, 3" );
}
```

5.d.7 [format] We have seen the basics of input and output, and that formatted input and output is realized using operators. Like many other operators in C++, those operators can be overloaded. We will show how that works in this example.
We will revisit the `cartesian` class from last week, to represent complex numbers in algebraic form, i.e. as a sum of a real and an imaginary number. We do not care about arithmetic this time: we will only implement a constructor and the formatted input and output operators. We will, however, need equality so that we can write test cases.

```
class cartesian
{
    double real, imag;
public:
```

We have seen default arguments before: those are used when no value is supplied by the caller. This also allows instances to be default-constructed.

```
    cartesian( double r = 0, double i = 0 ) : real( r ), imag( i )
{}
```

The comparison is fuzzy, due to the limited precision available in `double`.

```
    friend bool operator==( cartesian a, cartesian b )
    {
        return std::fabs( a.real - b.real ) < 1e-10 &&
               std::fabs( a.imag - b.imag ) < 1e-10;
    }
```

Now the formatted output, which is a little easier than the input. Since the first operand of this operator is not an instance of `cartesian`, the operator cannot be implemented as a method. It must either be a function outside the class, or use the 'friend trick'. Since we will need to access private attributes in the operator, we will use the `friend` syntax here. The return type and the type of the first argument are pretty much given and are always the same. You could consider them part of the syntax. The second argument is an instance of our class (this would often be passed as a `const` reference).

```
    friend std::ostream &operator<<( std::ostream &o, cartesian c )
    {
```

We will use `27.3±7.1*i` as the output format. We can use 'simpler' overloads of the `<<` operator to build up ours: this is a fairly common practice. We write to the `ostream` instance given to us in the argument. We must not forget to return that instance to our caller.

```
        o << c.real;
        if ( c.imag >= 0 )
            o << "+";
        return o << c.imag << "*i";
    }
```

The input operator is similar. It gets a reference to an `std::istream` as an argument (and has to pass it along in the return value). The main difference is that the object into which we read the data must be passed as a non-constant (i.e. mutable) reference, since we need to change it.

```
    friend std::istream &operator>>( std::istream &i, cartesian &c )
    {
```

Like above, we will build up our implementation from simpler overloads of the same operator (which all come from the standard library). The formatted input operators for numbers do not require that the number is followed by whitespace, but will stop at a character which can no longer be part of the number. A `+` or `-` character in the middle of the number qualifies.

```
        i >> c.real;
```

We will slightly abuse the flexibility of the formatted input operator for `double` values: it accepts numbers starting with an explicit `+` sign, hence we do not need to check the sign ourselves. Just read the imaginary part.

```
        i >> c.imag;
```

We do need to deal with the trailing `*i` though.

```
        char ch;
```

When formatted input fails, it should set a `failbit` in the input stream. This is how the `if ( stream >> value )` construct works.

```
        if ( !( i >> ch ) || ch != '*' ||
             !( i >> ch ) || ch != 'i' )
            i.setstate( i.failbit );
```

And as mentioned above, we need to return a reference to the input stream.

```
        return i;
    }

};

int main() /* demo */
{
    std::ostringstream ostr;
    ostr << cartesian( 1, 1 );
```

We first check that the output behaves as we expected.

```
    assert( ostr.str() == "1+1*i" );
```

We write a few more complex numbers into the stream, using operator chaining.

```
    ostr << " " << cartesian( 3, 0 ) << " " << cartesian( 1, -1 )
         << " " << cartesian( 0, 0 );

    assert( ostr.str() == "1+1*i 3+0*i 1-1*i 0+0*i" );
```

We now construct an input stream from the string which we created above, and check that the values can be read back.

```
    std::istringstream istr( ostr.str() );
    cartesian a, b, c;
```

Let's read back the first number and check that the result makes sense.

```
    assert( istr >> a );
    assert( a == cartesian( 1, 1 ) );
```

We can also check that chaining works as expected, using the remaining numbers in the string.

```
    assert( istr >> a >> b >> c );

    assert( a == cartesian( 3, 0 ) );
    assert( b == cartesian( 1, -1 ) );
    assert( c == cartesian( 0, 0 ) );
```

We can reset an `istringstream` by calling its `str` method with a new buffer. We want to demonstrate that trying to read an ill-formatted complex number will fail.

```
    std::istringstream bad1( "7+3*j" );
    assert( !( bad1 >> a ) );

    std::istringstream bad2( "7" );
    assert( !( bad2 >> a ) );
}
```

Part 5.e: Elementary Exercises

5.e.1 [cartesian] In this exercise, we will implement complex numbers with addition, subtraction, unary minus and equality.
The class should be called `complex` (do not mind the syntax highlight). The constructor should take 2 real numbers (the real and imaginary parts).

```
class complex;
```

5.e.2 [force] In this example, we will define a class that represents a (physical) force in 3D. Forces are vectors (in the mathematical sense): they can be added and multiplied by scalars (scalars are, in this case, real numbers). Forces can also be compared for equality (we will use fuzzy comparison because floating point computations are inexact).
Hint: It may be useful to know that when overloading binary operators, the operands do not need to be of the same type.

```
class force;
```

5.e.3 [forcefmt] This week in the physics department, we will deal with formatting and parsing vectors (forces, just to avoid confusion with `std::vector`... for now).
The class will be called `force`, and it should have a constructor which takes 3 values of type `double` and a default constructor which constructs a 0 vector. In addition to that, it should have a (fuzzy) comparison operator and formatting operators, both for input and for output. Use the following format: `[F_x F_y F_z]`, that is, a left square bracket, then the three components of the force separated by spaces, and a closing square bracket. Do not forget to set `failbit` in the input stream if the format does not match expectations.

```
class force;
```

Part 5.p: Preparatory Exercises

5.p.1 [polar] The first thing we will do is implement a simple class which represents complex numbers using their polar form. This form makes multiplication and division easier, so that is what we will do here (see also `cartesian.cpp` for definition of addition).

- the constructor takes the modulus and the argument (angle)
- add `abs` and `arg` methods to access the attributes
- implement multiplication and division on `polar`
- implement equality for `polar`; keep in mind that if the modulus is zero, the argument (angle) is irrelevant

NB. The argument is periodic: either normalize it to fall within [0, 2π), or otherwise make sure that `polar( 1, x ) == polar( 1, x + 2π )`. The equality operator you implement should be tolerant of imprecision: use `std::fabs( x - y ) < 1e-10` instead of `x == y` when dealing with real numbers.

```
class polar; /* reference implementation: 29 lines */
```

5.p.2 [rational] In this exercise, we will represent rational numbers (fractions) with addition and ordering. The constructor of `rat` should take the numerator and the denominator (in this order), which are both integers. It should be possible to compare `rat` instances for equality and inequality (in this exercise, it is enough to implement the less-than operator , i.e. `a < b`).
NB. Recall how fractions with different denominators are compared (and added). Your implementation does not need to be very efficient, or work for very large numbers.

```
class rat; /* reference implementation: 9 lines */
```

5.p.3 [tmpfile] We will implement a simple wrapper around `std::fstream` that will act as a temporary file. When the object is destroyed, use `std::remove` to unlink the file. Make sure the stream is closed before you unlink the file.
The `tmpfile` class should have the following interface:

- a constructor which takes the name of the file
- method `write` which takes a string and replaces the content of the file with that string; this method should flush the data to the operating system (e.g. by closing the stream)
- method `read` which returns the current content of the file
- method `stream` which returns a reference to an instance of `std::fstream` (i.e. suitable for both reading and writing)

Calling both `stream` and `write` on the same object is undefined behaviour. The `read` method should return all data sent to the file, including data written to `stream()` that was not yet flushed by the user.

```
class tmpfile;
```

5.p.4 [nibble] In this exercise, we will implement a class that represents an array of nibbles (half-bytes) stored compactly, using a byte vector as backing storage. We will need 3 classes: one to represent reference-like objects: `nibble_ref`, another for pointer-like objects: `nibble_ptr` and finally the container to hold the nibbles: `nibble_vec`. NB. In this exercise, we will not consider `const`-ness of values stored in the vector.[8]
The `nibble_ref` class needs to remember a reference or a pointer to the byte which contains the nibble that we refer to, and whether it is the upper or the lower nibble. With that information (which should be

[8] In particular, obtaining a pointer (e.g. by using `begin`) will allow you to change the value that it points to, even if the vector itself was marked `const`. We will deal with this issue properly toward the end of the semester.

passed to it via a constructor), it needs to provide:

- an assignment operator which takes an `uint8_t` as an argument, but only uses the lower half of that argument to overwrite the pointed-to nibble,
- a conversion operator which allows implicit conversion of a `nibble_ref` to an `uint8_t`.

```
class nibble_ref; /* reference implementation: 17 lines */
```

The `nibble_ptr` class works as a pointer. Dereferencing a `nibble_ptr` should result in a `nibble_ref`. There is no indirect access, because the target (pointed-to) type does not have any fields. To make `nibble_ptr` more useful, it should also have:

- a pre-increment operator, which shifts the pointer to the next nibble in memory. That is, if it points at a lower nibble, after `++x`, it should point to an upper half of the same byte, and after another `++x`, it should point to the lower half of the next byte,
- an equality comparison operator, which checks whether two `nibble_ptr` instances point to the same place in memory.

```
class nibble_ptr; /* reference implementation: 18 lines */
```

And finally the `nibble_vec`: this class should provide 4 methods:

- `push_back`, which adds a nibble at the end,
- `begin`, which returns a `nibble_ptr` to the first stored nibble (lower half of the first byte),
- `end`, which returns a `nibble_ptr` past the last stored nibble (i.e. the first nibble that is not in the container), and finally
- indexing operator, which returns a `nibble_ref`.

```
class nibble_vec; /* reference implementation: 16 lines */
```

5.p.5 [grep] To practice working with IO streams a little, we will write a two simple functions which reads lines from an input stream, process them a little and possibly print them out or their part into an output stream.
The `grep` function checks, for every line on the input, whether it matches a given `pattern` (i.e. the pattern is a substring of the line) and if it does (and only if it does) copies the line to the output stream.

```
void grep( std::string pattern, std::istream &, std::ostream & );
```

The other function to add is called `cut` and it will process the lines differently: it splits each line into fields separated by the character `delim` and only prints the column given by `col`. Unlike the `cut` program, index columns starting at 0. If there are not enough columns on a given line, print an empty line.

```
void cut( char delim, int col, std::istream &, std::ostream & );
```

5.p.6 [fixnum] In this exercise, we will implement fixed-precision numbers, with 2 fractional digits and up to 6 integral digits (both decimal), i.e. numbers of the form '123456.78'.
This is the class which we will use for indicating that parsing of the `fixnum` has failed (i.e. this class will be thrown as an exception in that case).

```
class bad_format;
```

The `fixnum` class should provide following operations: addition, subtraction and multiplication. It should have explicit constructors which construct the number from an integer or from a string. The latter constructor should throw an exception if the string is ill-formed (it is okay to only handle positive numbers in string form). Finally, it should be possible to compare `fixnum` instances for equality. All operations should round toward zero, to the nearest representable number.

```
class fixnum; /* reference implementation: 32 lines */
```

Part 5.r: Regular Exercises

5.r.1 [poly] Goal: implement polynomials with addition (easy) and multiplication (less easy). A polynomial is a term of the form $7x^4 + 0x^3 + 0x^2 + 3x + x^0$ – i.e. a sum of non-negative integral powers of x, with each power carrying a fixed (constant) coefficient. Adding two polynomials will simply give us a polynomial where coefficients are sums of the coefficients of the two addends. The case of multiplication is more complicated, because:

- each term of the first polynomial has to be multiplied by each term of the second polynomial
- some of those products give equal powers of x and hence their coefficients need to be summed

For each polynomial, there is some n, such that all powers higher than n have a zero coefficient. This is important when you want to store the polynomials in a computer.
The default constructor of the class `poly` should generate a polynomial which has all coefficients set to 0. Besides addition and multiplication (which are implemented as operators), also implement equality and a method `set`, which takes an exponent (power of x) and a coefficient, both integers.

```
class poly; /* reference implementation is 45 lines */
```

5.r.2 [csv] In this exercise, we will deal with CSV files: we will implement a class called `csv` which will read data from an input stream and allow the user to access it using the indexing operator.
The exception to throw in case of format error.

```
class bad_format;
```

The constructor should accept a reference to `std::istream` and the expected number of columns. In the input, each line contains integers separated by value. The constructor should throw an instance of `bad_format` if the number of columns does not match.
Additionally, if `x` is an instance of `csv`, then `x[ 3 ][ 1 ]` should return the value in the third row and first column.

```
class csv;
```

5.r.3 [set] In this exercise, we will implement a set of arbitrary integers, with the following operations: union using `|`, intersection using `&`, difference using `-` and inclusion using `<=`. Use efficient algorithms for the operations (check out what's available in the standard header `algorithm`). Provide methods `add` and `has` to add elements and test their presence.

```
class set; /* reference implementation: 36 lines */
```

Part 6: Exceptions and RAII

Demonstrations:

1. `exceptions` – throwing and catching exceptions
2. `stdexcept` – the standard exception hierarchy
3. `semaphore` – automatic management of finite resources
4. `swarm` – keeping the swarm under control

Elementary exercises:

1. `default` – read a number or return a default value

2. counter – count the number of instances of a class
3. coffee – a simple model of a coffee machine

Preparatory exercises:

1. fd – POSIX file descriptors
2. loan – database-style transactions with resources
3. library – borrowing books
4. parse – a simple parser which throws exceptions
5. invest – we further stretch the banking story
6. linear – linear equations, with some exceptions

Regular exercises:

1. printing – printing with a monthly budget
2. bsearch – a key-value vector which throws on failure
3. enzyme – cellular chemistry with RAII
4. tinyvec – a vector in a fixed memory buffer
5. lock – a movable mutual exclusion token
6. bounded – a bounded queue that throws when full

Part 6.d: Demonstrations

6.d.1 [exceptions] Exceptions are, as their name suggests, a mechanism for handling unexpected or otherwise exceptional circumstances, typically error conditions. A canonic example would be trying to open a file which does not exist, trying to allocate memory when there is no free memory left and the like. Another common circumstance would be errors during processing user input: bad format, unexpected switches and so on.
NB. Do not use exceptions for 'normal' control flow, e.g. for terminating loops. That is a really bad idea (even though try blocks are cheap, throwing exceptions is very expensive).
This example will be somewhat banal. We start by creating a class which has a global counter of instances attached to it: i.e. the value of counter tells us how many instances of counted exist at any given time. Fair warning, do not do this at home.

```
int counter = 0;

struct counted
{
    counted()  { ++ counter; }
    ~counted() { -- counter; }
};
```

A few functions which throw exceptions and/or create instances of the counted class above. Notice that a throw statement immediately stops the execution and propagates up the call stack until it hits a try block (shown in the main function below). The same applies to a function call which hits an exception: the calling function is interrupted immediately.

```
int f() { counted x; return 7; }
int g() { counted x; throw std::bad_alloc(); assert( 0 ); }
int h() { throw std::runtime_error( "h" ); }
int i() { counted x; g(); assert( 0 ); }

int main() /* demo */
{
    bool caught = false;
```

A try block allows us to detect that an exception was thrown and react, based on the type and attributes of the exception. Otherwise, it is a regular block with associated scope, and behaves normally.

```
    try
    {
        counted x;
        assert( counter == 1 );
        f();
        assert( counter == 1 );
    }
```

One or more catch blocks can be attached to a try block: those describe what to do in case an exception of a matching type is thrown in one of the statements of the try block. The catch clause behaves like a prototype of a single-argument function – if it could be 'called' with the thrown exception as an argument, it is executed to handle the exception.
This particular catch block is never executed, because nothing in the associated try block above throws a matching exception (or rather, any exception at all):

```
    catch ( const std::bad_alloc & ) { assert( false ); }
```

The counted instance x above went out of scope:

```
    assert( counter == 0 );
```

Let's write another try block. This time, the i call in the try block throws, indirectly (via g) an exception of type std::bad_alloc.

```
    try { i(); }
```

To demonstrate how catch blocks are selected, we will first add one for std::runtime_error, which will not trigger (the 'prototype' does not match the exception type that was thrown):

```
    catch ( const std::runtime_error & ) { assert( false ); }
```

As mentioned above, each try block can have multiple catch blocks, so let's add another one, this time for the bad_alloc that is actually thrown. If the catch matches the exception type, it is executed and propagation of the exception is stopped: it is now handled and execution continues normally after the end of the catch sequence.

```
    catch ( const std::bad_alloc & ) { caught = true; }
```

Execution continues here. We check that the catch block was actually executed:

```
    assert( caught );
    assert( counter == 0 ); // no <counted> instances were leaked
}
```

6.d.2 [stdexcept] It is possible to sub-class standard exception classes. For most uses, std::runtime_error is the most appropriate base class.

```
class custom_exception : public std::runtime_error
{
public:
    custom_exception() : std::runtime_error( "custom" ) {}
};
```

This demo simply demonstrates some of the standard exception types (i.e. those that are part of the standard library, and which are thrown by standard functions or methods; as long as those methods or functions are not too arcane).

```
int main() /* demo */
{
    try
    {
        throw custom_exception();
        assert( false );
    }
```

As per standard rules, it's possible to catch exceptions of derived classes (of course including user-defined types) via a catch clause which accepts a reference to a superclass.

```
    catch ( const std::exception & ) {}
```

```
try
{
    std::vector x{ 1, 2 };
```

Attempting out-of-bounds access through `at` gives `std::out_of_range`

```
    x.at( 7 );
    assert( false );
}
catch ( const std::out_of_range & ) {}

try
{
```

If the string passed to `stoi` is not a number, we get back an exception of type `std::invalid_argument`.

```
    std::stoi( "foo" );
    assert( false );
}
catch ( const std::invalid_argument & ) {}

try
{
```

If an integer is too big to fit the result type, `stoi` throws `std::out_of_range`.

```
    std::stoi( "123456123456123456" );
    assert( false );
}
catch ( const std::out_of_range & ) {}

try
{
```

System-interfacing functions may throw `std::system_error`. Here, for instance, trying to detach a thread which was not started.

```
    std::thread().detach();
    assert( false );
}
catch ( const std::system_error & ) {}

try
{
```

Throwing a `system_error` is the appropriate reaction when dealing with a failure of a POSIX function which sets `errno`.

```
    int fd = ::open( "/does/not/exist", O_RDONLY );
    if ( fd < 0 )
        throw std::system_error( errno, std::system_category(),
                                 "opening /does/not/exist" );
    assert( false );
}
catch ( const std::system_error & ) {}

try
{
```

Passing a size that is more than `max_size()` when constructing or resizing an `std::string` or an `std::vector` gives us back an `std::length_error`. Note that the -1 turns into a really big number in this context.

```
    std::string x( -1, 'x' );
    assert( false );
}
catch ( const std::length_error & ) {}

try
{
    std::bitset< 128 > x;
    x[ 100 ] = true;
```

Trying to convert an `std::bitset` to an integer type may throw `std::overflow_error`, if there are bits set that do not fit into the target integer type.

```
        x.to_ulong();
        assert( false );
    }
    catch ( const std::overflow_error & ) {}
}
```

6.d.3 **[semaphore]** In this demo, we will implement a simple semaphore. A semaphore is a device which guards a resource of which there are multiple instances, but the number of instances is limited. It is a slight generalization of a mutex (which guards a singleton resource). Internally, semaphore simply counts the number of clients who hold the resource and refuses further requests if the maximum is reached. In a multi-threaded program, semaphores would typically block (wait for a slot to become available) instead of refusing. In a single-threaded program (which is what we are going to use for a demonstration), this would not work. Hence our `get` method returns a `bool`, indicating whether acquisition of the lock succeeded.

```
class semaphore
{
    int _available;
public:
```

When a semaphore is constructed, we need to know how many instances of the resource are available.

```
    explicit semaphore( int max ) : _available( max ) {}
```

Classes which represent resource managers (in this case 'things that can be locked' as opposed to 'locks held') have some tough choices to make. If they are impossible to copy/move/assign, users will find that they must not appear as attributes in their classes, lest those too become un-copyable (and un-movable) by default. However, this is how the standard library deals with the problem, see `std::mutex` or `std::condition_variable`. While it is the safest option, it is also the most annoying. Nonetheless, we will do the same.

```
    semaphore( const semaphore & ) = delete;
    semaphore &operator=( const semaphore & ) = delete;
```

We allow would-be lock holders to query the number of resource instances currently available. Perhaps if none are left, they can make do without one, or they can perform some other activity in the hopes that the resource becomes available later.

```
    int available() const
    {
        return _available;
    }
```

Finally, what follows is the 'low-level' interface to the semaphore. It is completely unsafe, and it is inadvisable to use it directly, other than perhaps in special circumstances. This being C++, such interfaces are commonly made available. Again see `std::mutex` for an example. However, it would also be an option to be strict about it, make the following 2 methods private, and declare the RAII class defined below, `semaphore_lock`, to be a friend of this one.

```
    bool get()
    {
        if ( _available > 0 )
            return _available --;
        else
            return false;
    }
```

```
    void put()
    {
        ++ _available;
    }
};
```

We will want to write a RAII 'lock holder' class. However, since `get` above might fail, we need a way to indicate the failure in the RAII class as well. But constructors don't return values: it is therefore a reasonable choice to throw an exception. It is reasonable as long as we don't expect the failure to be a common scenario.

```
class resource_exhausted : public std::runtime_error
{
public:
    resource_exhausted()
        : std::runtime_error( "semaphore full" )
    {}
};
```

Now the RAII class itself. It will need to hold a reference to the semaphore for which it holds a lock (good thing the semaphore is not movable, so we don't have to think about its address changing). Of course, it must not be possible to make a copy of the resource class: we cannot duplicate the resource, which is a lock being held. However, it does make sense to move the lock to a new owner, if the client so wishes. Hence, both a move constructor and move assignment are appropriate.

```
class semaphore_lock
{
    semaphore *_sem = nullptr;
public:
```

To construct a semaphore lock, we understandably need a reference to the semaphore which we wish to lock. You might be wondering why the attribute is a pointer and the argument is a reference. The main difference between references and pointers (except the syntactic sugar) is that references cannot be null. In a correct program, all references always refer to valid objects. It does not make sense to construct a semaphore_lock which does not lock anything. Hence the reference. Why the pointer in the attributes? That will become clear shortly.
Before we move on, notice that, as promised, we throw an exception if the locking fails. Hence, no `noexcept` on this constructor.

```
    semaphore_lock( semaphore &s ) : _sem( &s )
    {
        if ( !_sem->get() )
            throw resource_exhausted();
    }
```

As outlined above, semaphore locks cannot be copied or assigned. Let's make that explicit.

```
    semaphore_lock( const semaphore_lock & ) = delete;
    semaphore_lock &operator=( const semaphore_lock & ) = delete;
```

The new object (the one initialized by the move constructor) is quite unremarkable. The interesting part is what happens to the 'old' (source) instance: we need to make sure that when it is destroyed, it does not release the resource (i.e. the lock held) – the ownership of that has been transferred to the new instance. This is where the pointer comes in handy: we can assign `nullptr` to the pointer held by the source instance. Then we just need to be careful when we release the resource (in the destructor, but also in the move assignment operator) – we must first check whether the pointer is valid.
Also notice the `noexcept` qualifier: even though the 'normal' constructor throws, we are not trying to obtain a new resource here, and there is nothing in the constructor that might fail. This is good, because move constructors, as a general rule, should not throw.

```
    semaphore_lock( semaphore_lock &&src ) noexcept
        : _sem( src._sem )
    {
        src._sem = nullptr;
    }
```

We now define a helper method, `release`, which frees up (releases) the resource held by this instance. It will do this by calling `put` on the semaphore. However, if the semaphore is null, we do nothing: the instance has been moved from, and no longer owns any resources.
Why the helper method? Two reasons:

1. it will be useful in both the move assignment operator and in the destructor,
2. the client might need to release the resource before the instance goes out of scope or is otherwise destroyed 'naturally' (compare `std::fstream::close()`).

```
    void release() noexcept
    {
        if ( _sem )
            _sem->put();
    }
```

Armed with `release`, writing both the move assignment and the destructor is easy. The move assignment is also `noexcept`, which is

```
    semaphore_lock &operator=( semaphore_lock &&src ) noexcept
    {
```

First release the resource held by the current instance. We cannot hold both the old and the new resource at the same time.

```
        release();
```

Now we reset our `_sem` pointer and update the `src` instance – the resource is now in our ownership.

```
        _sem = src._sem;
        src._sem = nullptr;
        return *this;
    }

    ~semaphore_lock() noexcept
    {
        release();
    }
};

int main() /* demo */
{
    semaphore sem( 3 );
    sem.get();
    semaphore_lock l1( sem );
    bool l4_made = false;

    try
    {
        semaphore_lock l2( sem );
        assert( sem.available() == 0 );
        semaphore_lock l3 = std::move( l2 );
        assert( sem.available() == 0 );
        semaphore_lock l4 = std::move( l1 );
        assert( sem.available() == 0 );
        l4_made = true;
        semaphore_lock l5( sem );
        assert( false );
    }
    catch ( const resource_exhausted & ) {}

    assert( l4_made );
    assert( sem.available() == 2 );
```

```
    // clang-tidy: -clang-analyzer-deadcode.DeadStores
}
```

6.d.4 [swarm] TBD. Create overlords which create a resource and non-overlords which consume it. Enforce the balance by throwing an exception on exhaustion.

```
class swarm;

class unit
{
    swarm &owner;
};

class overlord : unit
{
};

class zergling : unit
{
};

class swarm
{
    int _control  = 3;
    int _resource = 200;
public:
    overlord spawn_overlord();
    zergling spawn_zergling();
};

int main() /* demo */
{
    swarm s;

    std::vector< zergling > zerglings;
    std::vector< overlord > overlords;

    zerglings.emplace_back( s );
}
```

Part 6.e: Elementary Exercises

6.e.1 [default] Write a function stoi_or which takes a string and an int. If the string can be parsed using std::stoi, return the result of stoi, otherwise return the 'default' value from the second argument.

6.e.2 [counter]

```
static int counter = 0;
```

Add constructors and a destructor to counted in such a way that counter above always corresponds to the number of instances of counted that exist at any given time.

```
struct counted;
```

6.e.3 [coffee] Implement a coffee machine which gives out a token when the order is placed and takes the token back when it is done... at most one order can be in progress.
Throw this when the machine is already busy making coffee.

```
class busy {};
```

And this when trying to use a default-constructed or already-used token.

```
class invalid {};
```

Fill in the two classes. Besides constructors and assignment operators, add methods make and fetch to machine, to create and redeem tokens respectively.

```
class machine;
class token;
```

6.e.4 [lock] TBD lock a resource, with ownership transfer but no copy

Part 6.p: Preparatory Exercises

6.p.1 [fd] In POSIX systems, opening a file or a file-like resource gives us a file descriptor, a small number that can be passed to system calls such as read or write. The descriptor must be closed when it is no longer needed, by calling close on it exactly once (it is important not to close the same descriptor twice). Write a class which safely wraps a file descriptor so that we can't accidentally lose it or close it twice.
It should be possible to move-construct and move-assign file descriptors. A new valid descriptor can be created in 2 ways: by calling fd::open("file", flags) or fd::dup(raw_fd) where flags and raw_fd are both int. Use POSIX functions open and dup to implement this. Run man 2 open and man 2 dup on aisa for details about these POSIX functions.
Add methods read and write to the fd class, the first will simply take an integer, read the given number of bytes and return an std::string. The latter will take an std::string and write it into the descriptor. Again see man 2 read and man 2 write on aisa for advice.
If open, read or write fails, throw std::system_error. Attempting to call read or write on an invalid descriptor (one that was default-constructed or already closed) should throw std::invalid_argument.

6.p.2 [loan] Let us revisit the bank account story from first week. We will have 2 classes this time: an account, which has the usual methods: deposit, withdraw, balance; to simplify things, we will only add a default constructor, which sets the initial balance to 0.
The other class will be called loan, and its constructor will take a reference to an account and the amount loaned (an int). Constructing a loan object will deposit the loaned amount to the referenced account. It will also have a method called repay which takes an integer, which withdraws the given amount from the associated account and reduces the amount owed by the same sum. Attempting to repay more than is owed should throw std::out_of_range.
Make sure that we can't accidentally destroy a loan without repaying it first. Does it make sense to make a copy of a loan? How about move? And assignment?

```
class account;
class loan;
```

6.p.3 [library] A very simple library model: patrons can borrow books and borrowed books can be moved around, and must be eventually returned. The library should have the following methods:

- add_book, which creates a book record based on 2 arguments – the title (a string) and the number of copies (an integer) of the book – and returns a suitable object (a handle) to represent that book,
- add_patron which creates a patron, given a name (a string), and again returns a suitable object to represent the patron.

It should be possible to call borrow on objects which represent patrons, passing either a reference to a library or another patron as the first argument, and the book handle as a second argument. It returns true if the borrowing was a success, or false otherwise (no copies were available). If a patron is destroyed, all books in their possession return to the library. Destroying a book handle does nothing.

```
class library;
```

Finally, the class loan holds information about a loan. Both library and the patron object get a method give which returns a loan object associated with the book passed to it, and take, which accepts a loan

object and takes ownership of the associated book. If `give` is called on an object which does not have a copy of the requested book, return an invalid (empty) `loan` object.
While a book is held in a `loan` instance, it is not in the possession of any of the objects, but it is checked out from the library. If the `loan` object is destroyed without being `taken` by anyone, the book returns to the library.

```
class loan;
```

6.p.4 [parse] Write a simple parser for an assembly-like language with one instruction per line (each taking 2 operands, separated by spaces, where the first is always a register and the second is either a register or an 'immediate' number).
The opcodes (instructions) are: `add`, `mul`, `jnz`, the registers are `rax`, `rbx` and `rcx`. The result is a vector of `instruction` instances (see below). Set `r_2` to `reg::immediate` if the second operand is a number.
If the input does not conform to the expected format, throw `no_parse`, which includes a line number with the first erroneous instruction and the kind of error (see `enum error`), as public attributes `line` and `type`, respectively. If multiple errors appear on the same line, give the one that comes first in the definition of `error`. You can add attributes or methods to the structures below, but do not change the enumerations.

```
enum class opcode { add, mul, jnz };
enum class reg { rax, rbx, rcx, immediate };
enum class error { bad_opcode, bad_register, bad_immediate,
                   bad_structure };

struct instruction
{
    opcode op;
    reg r_1, r_2;
    int32_t immediate;
};

struct no_parse
{
    int line;
    error type;
};

std::vector< instruction > parse( const std::string & );

#include <iostream>
```

6.p.5 [invest] We will revisit (again) our familiar example of a bank account. This time, we add exceptions to the story: withdrawals that would exceed the overdraft limit will throw. We will also add a class dual to `loan` from the last time: an `investment`, which will deduct money from an account upon construction, accrue interest, and upon destruction, deposit the money into the original account.
We will use this class as the exception type. It is okay to keep it empty.

```
class insufficient_funds;
```

First the `account` class, which has the usual methods: `balance`, `deposit` and `withdraw`. The starting balance is 0. The balance must be non-negative at all times: an attempt to withdraw more money than available should throw an exception of type `insufficient_funds`.

```
class account; /* reference implementation: 13 lines */
```

And finally the class `investment`, which has a three-parameter constructor: it takes a reference to an `account`, the sum to invest and a yearly interest rate (in percent, as an integer). Upon construction, it must withdraw the sum from the account, and upon destruction, deposit the original sum plus the interest. The method `next_year` should update the accrued interest.

```
class investment; /* reference implementation: 15 lines */
```

6.p.6 [linear] Write a solver for linear equations in 2 variables. The interface will be a little unconventional: overload operators `+`, `*` and `==` and define global constants `x` and `y` of suitable types, so that it is possible to write the equations as shown in `main` below.
Note that the return type of `==` does not have to be bool. It can be any type you like, including of course custom types. For `solve`, I would suggest looking up Cramer's rule.
ref: class `eqn` 25 lines, `solve` 8 lines, `x` and `y` 2 lines
If the system has no solution, throw an exception of type `no_solution`. Derive it from `std::exception`.

Part 6.r: Regular Exercises

6.r.1 [printing] TBD. Jobs need resources (printing credits) which must be reserved when the job is queued, but are only consumed at actual printing time; jobs can be moved between queues (printers) by the system, for load balancing.
Define a class `job` which ...

6.r.2 [car] TBD. Define classes `car`, `key` and `person`. A key is required to drive the car, but cannot be cloned. People can borrow the keys from each other.

6.r.3 [enzyme] TBD. Reactions tie up enzymes, which return to the pool after the reaction is done. Different reactions need different sets of enzymes present, and a given enzyme cannot be used by more than one reaction at a time.

Part 7: Memory and Smart Pointers

Before you dig into the demonstrations and exercises, do not forget to read the extended introduction below. That said, the units for this week are, starting with demonstrations:

1. `queue` – a queue with stable references
2. `finexp` – like regexps but finite
3. `expr` – expressions with operators and shared pointers
4. `family` – genealogy with weak pointers

Elementary exercises:

1. `dynarray` – a simple array with a dynamic size
2. `list` – a simple linked list with minimal interface

Preparatory exercises:

1. `unrolled` – a linked list of arrays
2. `bittrie` – bitwise tries (radix trees)
3. `solid` – efficient storage of optional data
4. `chartrie` – binary tree for holding string keys
5. `bdd` – binary decision diagrams
6. `rope` – a string-like structure with cheap concatenation

Regular exercises:

1. `circular` – a singly-linked circular list
2. `zipper` – implementing zipper as a linked list
3. `segment` – a binary tree of disjoint intervals
4. `diff` – automatic differentiation
5. `critbit` – more efficient version of binary tries
6. `refcnt` † – implement a simple reference-counted heap

Part 7.A: Exclusive Ownership

So far, we have managed to almost entirely avoid thinking about memory management: standard containers manage memory behind the scenes. We sometimes had to think about copies (or rather avoiding them), because containers could carry a lot of memory around and copying all that memory without a good reason is rather wasteful (this is why we often pass arguments as `const` references and not as values). This week, we will look more closely at how memory management works and what we can do when standard containers are inadequate to deal with a given problem. In particular, we will look at building our own pointer-based data structures and how we can retain automatic memory management in those cases using `std::unique_ptr`.
XXX

Part 7.B: Shared Ownership

While `unique_ptr` is very useful and efficient, it only works in cases where the ownership structure is clear, and a given object has a single owner. When ownership of a single object is shared by multiple entities (objects, running functions or otherwise), we cannot use `unique_ptr`.
To be slightly more explicit: shared ownership only arises when the lifetime of the objects sharing ownership is not tied to each other. If A owns B and A and B both need references to C, we can assign the ownership of C to object A: since it also owns B, it must live at least as long as B and hence there ownership is not actually shared.
However, if A needs to be able to transfer ownership of B to some other, unrelated object while still retaining a reference to C, then C will indeed be in shared ownership: either A or B may expire first, and hence neither can safely destroy the shared instance of C to which they both keep references. In many modern languages, this problem is solved by a garbage collector, but alas, C++ does not have one.
Of course, it is usually better to design data structures in a way that allows for clear, 1:1 ownership structure. Unfortunately, this is not always easy, and sometimes it is not the most efficient solution either. Specifically, when dealing with large immutable (or persistent, in the functional programming sense) data structures, shared ownership can save considerable amount of memory, without introducing any ill side-effects, by only storing common sub-structures once, instead of cloning them. Of course, there are also cases where shared mutable state is the most efficient solution to a problem.

Part 7.d: Demonstrations

7.d.1 [queue] In this example, we will demonstrate the use of `std::unique_ptr`, which is an RAII class for holding (owning) values dynamically allocated from the heap. We will implement a simple one-way, non-indexable queue. We will require that it is possible to erase elements from the middle in O(1), without invalidating any other iterators. The standard containers which could fit:

- `std::deque` fails the erase in the middle requirement,
- `std::forward_list` does not directly support queue-like operation, hence using it as a queue is possible but awkward; wrapping `std::forward_list` would be, however, a viable approach to this task, too,
- `std::list` works well as a queue out of the box, but has twice the memory overhead of `std::forward_list`.

As usual, since we do not yet understand templates, we will only implement a queue of integers, but it is not hard to imagine we could generalize to any type of element.
Since we are going for a custom, node-based structure, we will need to first define the class to represent the nodes. For sake of simplicity, we will not encapsulate the attributes.

```
struct queue_node
{
```

We do not want to handle all the memory management ourselves. To rule out the possibility of accidentally introducing memory leaks, we will use `std::unique_ptr` to manage allocated memory for us. Whenever a `unique_ptr` is destroyed, it will free up any associated memory. An important limitation of `unique_ptr` is that each piece of memory managed by a `unique_ptr` must have exactly one instance of `unique_ptr` pointing to it. When this instance is destroyed, the memory is deallocated.

```
    std::unique_ptr< queue_node > next;
```

Besides the structure itself, we of course also need to store the actual data. We will store a single integer per node.

```
    int value;
};
```

We will also need to be able to iterate over the queue. For that, we define an iterator, which is really just a slightly generalized pointer (you may remember `nibble_ptr` from last week). We need 3 things: pre-increment, dereference and inequality.

```
struct queue_iterator
{
    queue_node *node;
```

The `queue` will need to create instances of a `queue_iterator`. Let's make that convenient.

```
    queue_iterator( queue_node *n ) : node( n ) {}
```

The pre-increment operator simply shifts the pointer to the `next` pointer of the currently active node.

```
    queue_iterator &operator++()
    {
        node = node->next.get();
        return *this;
    }
```

Inequality is very simple (we need this because the condition of iteration loops is `it != c.end()`, including range `for` loops):

```
    bool operator!=( const queue_iterator &o ) const
    {
        return o.node != node;
    }
```

And finally the dereference operator. This should be familiar by now (perhaps notice the `const` overload). Depending on element type, the `const` overload would in many cases return a `const` reference instead of a value.

```
    int &operator*()       { return node->value; }
    int  operator*() const { return node->value; }
};
```

This class represents the queue itself. We will have `push` and `pop` to add and remove items, `empty` to check for emptiness and `begin` and `end` to implement iteration.

```
class queue
{
```

We will keep the head of the list in another `unique_ptr`. An empty queue will be represented by a null head. Also worth noting is that when using a list as a queue, the head is where we remove items. The end of the queue (where we add new items) is represented by a plain

pointer because it does not own the node (the node is owned by its predecessor).

```
    std::unique_ptr< queue_node > first;
    queue_node *last = nullptr;
public:
```

As mentioned above, adding new items is done at the 'tail' end of the list. This is quite straightforward: we simply create the node, chain it into the list (using the last pointer as a shortcut) and point the last pointer at the newly appended node. We need to handle empty and non-empty lists separately because we chose to represent an empty list using null head, instead of using a dummy node.

```
    void push( int v )
    {
        if ( last ) /* non-empty list */
        {
            last->next = std::make_unique< queue_node >();
            last = last->next.get();
        }
        else /* empty list */
        {
            first = std::make_unique< queue_node >();
            last = first.get();
        }

        last->value = v;
    }
```

Reading off the value from the head is easy enough. However, to remove the corresponding node, we need to be able to point first at the next item in the queue.
Unfortunately, we cannot use normal assignment (because copying unique_ptr is not allowed). We will have to use an operation that is called move assignment and which is written using a helper function in from the standard library, called std::move.
Operations which move their operands invalidate the moved-from instance. In this case, first->next is the moved-from object and the move will turn it into a null pointer. In any case, the next pointer which was invalidated was stored in the old head node and by rewriting first, we lost all pointers to that node. This means two things:

1. the old head's next pointer, now null, is no longer accessible
2. memory allocated to hold the old head node is freed

```
    int pop()
    {
        int v = first->value;
        first = std::move( first->next );
```

Do not forget to update the last pointer in case we popped the last item.

```
        if ( !first ) last = nullptr;
        return v;
    }
```

The emptiness check is simple enough.

```
    bool empty() const { return !last; }
```

Now the begin and end methods. We start iterating from the head (since we have no choice but to iterate in the direction of the next pointers). The end method should return a so-called past-the-end iterator, i.e. one that comes right after the last real element in the queue. For an empty queue, both begin and end should be the same. Conveniently, the next pointer in the last real node is nullptr, so we can use that as our end-of-queue sentinel quite naturally. You may want to go back to the pre-increment operator of queue_iterator just in case.

```
    queue_iterator begin() { return { first.get() }; }
    queue_iterator end()   { return { nullptr }; }
```

And finally, erasing elements. Since this is a singly-linked list, to erase an element, we need an iterator to the element before the one we are about to erase. This is not really a problem, because erasing at the head is done by pop. We use the same move assignment construct that we have seen in pop earlier.

```
    void erase_after( queue_iterator i )
    {
        assert( i.node->next );
        i.node->next = std::move( i.node->next->next );
    }
};

int main() /* demo */
{
```

We start by constructing an (empty) queue and doing some basic operations on it. For now, we only try to insert and remove a single element.

```
    queue q;
    assert( q.empty() );
    q.push( 7 );
    assert( !q.empty() );
    assert( q.pop() == 7 );
    assert( q.empty() );
```

Now that we have emptied the queue again, we add a few more items and try erasing one and iterating over the rest.

```
    q.push( 1 );
    q.push( 2 );
    q.push( 7 );
    q.push( 3 );
```

We check that erase works as expected. We get an iterator that points to the value 2 from above and use it to erase the value 7.

```
    queue_iterator i = q.begin();
    ++ i;
    assert( *i == 2 );
    q.erase_after( i );
```

We can use instances of queue in range for loops, because they have begin and end, and the types those methods return (i.e. iterators) have dereference, inequality and pre-increment.

```
    int x = 1;
    for ( int v : q )
        assert( v == x++ );
```

That went rather well, let's just check that the order of removal is the same as the order of insertion (first in, first out). This is how queues should behave.

```
    assert( q.pop() == 1 );
    assert( q.pop() == 2 );
    assert( q.pop() == 3 );
    assert( q.empty() );
}
```

7.d.2 [finexp] We will do a simpler version of regular expressions that can only capture finite languages, but somewhat more compactly than just listing all the words that belong to the language. There will be two operations: concatenation and alternative.
In this and the next demo, we will make use of late dispatch, which will be properly explained in the next chapter. All you need to know for now is, that, given:

- a class base and its derived class derived,

- a pointer, `base *ptr`, that in fact points to an instance of class `derived`, and
- a method `late` which is marked `virtual` in `base`, and `override` in `derived`,

a call `ptr->late()` will execute the implementation of the method from `derived` (and not from `base`, as would be the case with a non-`virtual` method).
Our goal will be to implement class `finexp`, with the following interface:

- an instance of `finexp` can be constructed from a string; the resulting `finexp` will match that exact string and nothing else
- two instances of `finexp` can be combined using `*`: the resulting `finexp` matches if the input string can be split in such a way that the first part matches the left `finexp` and the second part matches the right `finexp`
- two instances of `finexp` can be comined using `+`: the result matches a string if either of the operands does

Hint: it might be a worthwhile exercise to compare the below implementation with one based on `std::shared_ptr`.

```
struct node;
using node_ptr = std::unique_ptr< node >;
```

TBD explain things!

```
struct node
{
    std::string x;
    node_ptr l, r;

    virtual std::set< int > match( const std::string &s ) const
    {
        assert( !l && !r );
        if ( s.substr( 0, x.size() ) == x )
            return { int( x.size() ) };
        else
            return {};
    }

    node_ptr copy_into( node_ptr &&n ) const
    {
        n->l = l ? l->clone() : nullptr;
        n->r = r ? r->clone() : nullptr;
        return std::move( n );
    }

    virtual node_ptr clone() const
    {
        return copy_into( std::make_unique< node >( x ) );
    }

    node( std::string x ) : x( x ) {}
    node( const node_ptr &l_, const node_ptr &r_ )
        : l( l_->clone() ), r( r_->clone() )
    {}
    virtual ~node() = default;
};

struct alt : node
{
    using node::node;

    node_ptr clone() const override
    {
        return copy_into( std::make_unique< alt >( x ) );
    }

    std::set< int > match( const std::string &s ) const override
    {
        std::set< int > lout = l->match( s ), rout = r->match( s );
        rout.insert( lout.begin(), lout.end() );
        return rout;
    }
};

struct seq : node
{
    using node::node;

    node_ptr clone() const override
    {
        return copy_into( std::make_unique< seq >( x ) );
    }

    std::set< int > match( const std::string &s ) const override
    {
        std::set< int > out;

        for ( int i : l->match( s ) )
            for ( int j : r->match( s.substr( i ) ) )
                out.insert( i + j );

        return out;
    }
};

class finexp
{
    node_ptr n;
public:
    finexp( std::string s ) : n( new node( s ) ) {}
    finexp( node_ptr &&p ) : n( std::move( p ) ) {}
    finexp( const finexp &o ) : n( o.n->clone() ) {}

    finexp operator+( finexp b ) const
    {
        return { std::make_unique< alt >( n, b.n ) };
    }

    finexp operator*( finexp b ) const
    {
        return { std::make_unique< seq >( n, b.n ) };
    }

    friend bool match( const finexp &f, const std::string &s )
    {
        return f.n->match( s ).count( s.size() );
    }
};

int main() /* demo */
{
    finexp a( "a" ), b( "b" ), ab( "ab" ), ba( "ba" ),
           abba( "abba" );

    assert(  match( a, "a" ) );
    assert(  match( b, "b" ) );
    assert( !match( a, "b" ) );
    assert( !match( b, "a" ) );

    assert(  match( abba, "abba" ) );
    assert( !match( abba, "a" ) );
    assert( !match( abba, "abb" ) );
    assert( !match( a, "ab" ) );

    assert(  match( a + b, "a" ) );
    assert(  match( a + b, "b" ) );
    assert( !match( a + b, "ab" ) );
    assert( !match( a + b, "c" ) );

    assert(  match( a + abba, "a" ) );
    assert( !match( a + abba, "b" ) );
    assert(  match( a + abba, "abba" ) );

    assert(  match( ( ab + a ) * a, "aba" ) );
    assert(  match( ( a + ab ) * a, "aba" ) );
    assert( !match( ( ba + ab ) * a, "ba" ) );
```

```
    assert(  match( a * ( ba + ab ), "aba" ) );
    assert( !match( a * ( b + a ), "aba" ) );
}
```

7.d.3 [expr] In this example program, we will look at using shared pointers and operator overloading to get a nicer version of our expression examples, this time with sub-structure sharing: that is, doing something like `a + a` will not duplicate the sub-expression `a`.
Like in week 7, we will define an abstract base class to represent the nodes of the expression tree.

```
struct expr_base
{
    virtual int eval() const = 0;
    virtual ~expr_base() = default;
};
```

Since we will use (shared) pointers to `expr_base` quite often, we can save ourselves some typing by defining a convenient type alias: `expr_ptr` sounds like a reasonable name.

```
using expr_ptr = std::shared_ptr< expr_base >;
```

We will have two implementations of `expr_base`: one for constant values (nothing much to see here),

```
struct expr_const : expr_base
{
    const int value;
    expr_const( int v ) : value( v ) {}
    int eval() const override { return value; }
};
```

and another for operator nodes. Those are more interesting, because they need to hold references to the sub-expressions, which are represented as shared pointers.

```
struct expr_op : expr_base
{
    enum op_t { add, mul } op;
    expr_ptr left, right;
    expr_op( op_t op, expr_ptr l, expr_ptr r )
        : op( op ), left( l ), right( r )
    {}

    int eval() const override
    {
        if ( op == add ) return left->eval() + right->eval();
        if ( op == mul ) return left->eval() * right->eval();
        assert( false );
    }
};
```

In principle, we could directly overload operators on `expr_ptr`, but we would like to maintain the illusion that expressions are values. For that reason, we will implement a thin wrapper that provides a more natural interface (and also takes care of operator overloading). Again, the `expr` class essentially provides Java-like object semantics – which is quite reasonable for immutable objects like our expression trees here.

```
struct expr
{
    expr_ptr ptr;
    expr( int v ) : ptr( std::make_shared< expr_const >( v ) ) {}
    expr( expr_ptr e ) : ptr( e ) {}
    int eval() const { return ptr->eval(); }
};
```

The overloaded operators simply construct a new node (of type `expr_op` and wrap it up in an `expr` instance.

```
expr operator+( expr a, expr b )
{
    return { std::make_shared< expr_op >( expr_op::add,
                                          a.ptr, b.ptr ) };
}

expr operator*( expr a, expr b )
{
    return { std::make_shared< expr_op >( expr_op::mul,
                                          a.ptr, b.ptr ) };
}

int main() /* demo */
{
    expr a( 3 ), b( 7 ), c( 2 );
    expr ab = a + b;
    expr bc = b * c;
    expr abc = a + b * c;

    assert( a.eval() == 3 );
    assert( b.eval() == 7 );
    assert( ab.eval() == 10 );
    assert( bc.eval() == 14 );
    assert( abc.eval() == 17 );
}
```

7.d.4 [family] For many tasks, shared pointers (reference counting) are quite adequate (see also Python). However, they do have a weak spot: reference cycles. If you manage to create a loop of shared pointers, the pointers on this cycle (and anything outside the cycle they point to) will never be freed. That is unfortunate, since it reintroduces memory leaks into the rather leak-free subset of C++ that we have been using until now.
However, if we are a little careful, C++ allows us to have cyclic data structures with reference counting without introducing memory leaks: the `std::weak_ptr` class template.
We will implement a bit of genealogy – that is, family trees. This will simply consist of a graph of `person` instances (we will not delve into too much detail). Each `person` will have two parents, a father and a mother, and a list of children. We will want to maintain an invariant: the list of children contains exactly those `person` instances that have this `person` set as one of their parents. Since a fixed number of pointers (parents) are easier to manage than the arbitrary number of children, we will treat parents as the primary information and children as derived. Like before, we will split the class into a shared (data) part and into thin interface part.

```
class person_data
{
    std::shared_ptr< person_data > mother, father;
    std::vector< std::weak_ptr< person_data > > children;
    std::string name;
    friend class person;
};
```

The interface: the data is stored behind a shared pointer, but like in earlier examples, we pretend the `person` instances are values with sharing semantics. The family graph is, on the outside, still quite immutable (we can only add and remove nodes), so the abstraction is still reasonably solid.

```
class person
{
    using data_ptr = std::shared_ptr< person_data >;
    data_ptr _d;
public:
```

Construct a `person` instance from an existing data pointer. We would actually like to make this `private`, but that would give us problems because we actually delegate the constructor call to `std::vector`: we would have to make that a friend class (but that would punch holes

into the model... let's not bother for now).

```
explicit person( data_ptr p ) : _d( p ) {}
```

We need to be able to construct parent-less instances, since the data ends somewhere and we can no longer provide the data about parents.

```
explicit person( std::string name )
    : _d( std::make_shared< person_data >() )
{
    _d->name = name;
}
```

The standard constructor for person, with two parents. We take person by value since it's really just a pointer anyway (we could perhaps save an refcount increment/decrement pair by passing via const references). We also use constructor delegation: in the initialization section, we invoke the above 'parent-less' constructor. This constructor is also in charge of maintaining (half of) the above-mentioned invariant by inserting our data pointer into the children list of both the parents.

```
person( std::string name, person mother, person father )
    : person( name )
{
    _d->mother = mother._d;
    _d->father = father._d;

    _d->mother->children.emplace_back( _d );
    _d->father->children.emplace_back( _d );
}
```

The other half of the invariant is maintained here, with the help of shared_ptr destructors: if a person is completely destroyed (i.e. no copies remain, i.e. the reference count on the corresponding person_data drops to zero), all weak_ptr instances pointing to it will automatically turn into null pointers. We then simply filter those out to obtain the correct list of children.

```
std::vector< person > children() const
{
    std::vector< person > out;
    for ( const auto &c_weak : _d->children )
        if ( auto c = c_weak.lock() )
            out.emplace_back( c );
    return out;
}
```

A few simple accessors.

```
bool valid() const { return !!_d; }
std::string name() const { return _d->name; }
person mother() const { return person{ _d->mother }; }
person father() const { return person{ _d->father }; }
```

Equality: we base equality on object identity: copies of the same person (even those that arise in a roundabout way, without calling the copy constructor, e.g. those that arise from the above mother and father accessors which construct new person instances) will compare as equal.

```
    bool operator==( person o ) const { return o._d == _d; }
};

int main() /* demo */
{
    person unknown( "unknown" );
    person a( "a", unknown, unknown );
    person b( "b", unknown, unknown );

    assert( a.mother().valid() );
    assert( a.father().valid() );
    assert( a.mother() == unknown );
    assert( a.father() == unknown );
    assert( !unknown.mother().valid() );
    assert( a.mother().name() == "unknown" );

    {
        person c( "c", a, b );
        person d( "d", a, b );

        person x( "x", unknown, unknown );
        person e( "e", c, x );
```

Check that the children containers are correctly filled in by the constructors.

```
        assert( c.mother() == a );
        assert( a.children().size() == 2 );
        assert( b.children().size() == 2 );
        assert( c.children().size() == 1 );
        assert( x.children().size() == 1 );

        for ( const auto &ch : x.children() )
            assert( ch == e );

        for ( const auto &ch : c.children() )
            assert( ch == e );
```

The instances c, d, x and e are destroyed at this point (with no surviving copies).

```
    }
```

Check that the invariant is maintained.

```
    assert( a.children().empty() );
    assert( b.children().empty() );
}
```

Part 7.e: Elementary Exercises

7.e.1 **[dynarray]** Implement a dynamic array of integers with 2 operations: element access (using operator[]) and resize. The constructor takes the initial size as its only parameter.

```
class dynarray;
```

7.e.2 **[list]** Implement a linked list of integers, with head, tail (returns a reference) and empty. Asking for a head or tail of an empty list has undefined results. A default-constructed list is empty. The other constructor takes an int (the value of head) and a reference to an existing list. It will should make a copy of the latter.

```
class list;
```

Part 7.p: Preparatory Exercises

7.p.1 **[unrolled]** Another exercise, another data structure. This time we will look at so-called unrolled linked lists. We will need the data structure itself, with begin, end, empty and push_back methods. As usual, we will store integers. The difference between a 'normal' singly-linked list and an unrolled list is that in the latter, each node stores more than one item. In this case, we will use 4 items per node. Of course, the last node might only be filled partially. The iterator that begin and end return should at least implement dereference, pre-increment and inequality, as usual. We will not provide an interface for erasing elements, because that is somewhat tricky.

```
struct unrolled_node;     /* ref: 6 lines */
struct unrolled_iterator; /* ref: 22 lines */
class unrolled;           /* ref: 36 lines */
```

7.p.2 **[bittrie]** More data structures. A bit trie (or a bitwise trie, or a bitwise radix tree) is a binary tree for encoding a set of binary values,

with quick insertion and lookup. Each edge in the tree encodes a single bit (i.e. it carries a zero or a one). To make our life easier, we will represent the keys using a vector of booleans.
The key is a sequence of bits: iteration order (left to right) corresponds to a path through the trie starting from the root. I.e. the leftmost bit decides whether to go left or right from the root, and so on. A key is present in the trie iff it describes a path to a leaf node.

```
using key = std::vector< bool >;

struct trie_node; /* ref: 5 lines */
```

For simplicity, we will not have a normal `insert` method. Instead, the trie will expose its root node via `root` and allow explicit creation of new nodes via `make`, which accepts the parent node and a boolean as arguments (the latter indicating whether the newly created edge represents a 0 or a 1). Both `root` and `make` should return node references. Finally, add a `has` method which will check whether a given key is present in the `trie`.

```
class trie; /* ref: 21 lines */
```

7.p.3 [solid] In this exercise, we will focus on building objects that have optional data attached to them. The idea is that if the optional data is sufficiently big and there are enough instances which do not use this data, it makes sense to split the object into two. Of course, logically (in the interface), the object should still act like a single unit.
To make testing easier, we declare a global counter of matrices. It will be adjusted by the constructor and destructor of `transform_matrix` below. This is not a design pattern that you should normally use (but it is okay in a small demo).

```
int matrix_counter = 0;
```

The two pieces will be, in this case, a general description of a 3D object (a solid) and a 3D transformation matrix with 9 entries (3 rows and 3 columns). The matrix is represented by the class declared below. Make the class default-constructible and do not forget to implement the book-keeping for `matrix_counter`. The class should store the matrix entries inline (i.e. they should be part of the object, not managed in a separate heap allocation).

```
struct transform_matrix;
```

We don't know about inheritance yet, but the below class could be considered a base class in a simple inheritance hierarchy: it will only have properties common to different object types, but will not describe a complete solid in itself. It should have the following methods:

- `pos_x`, `pos_y` and `pos_z` to give the position of the solid
- `transform_entry( int r, int c )` gives the entry in the transformation matrix at row `r` and column `c`
- `transform_set( int r, int c, double v )` sets the corresponding entry in the transformation matrix
- a constructor which takes 3 arguments of type `double` (the x, y and z position coordinates)

The default transformation matrix is the identity matrix (1's on the main diagonal, 0's everywhere else). Memory should only be allocated for the transformation matrix if it changes from the default.

```
class solid;
```

7.p.4 [chartrie] An exercise similar to the `bittrie` earlier (same data structure but with bigger keys). To make it more interesting, the node management will happen within the class itself and will not be part of the interface. The encoding you should use is this:

- the left child of a node adds a single character to the key, like in the bit trie from before (the character is part of the left edge)
- the right child is actually a sibling of the current node and the edge is not labelled
- the chain to the right is sorted in ascending order

In other words, you can imagine the trie to be a 256-ary tree, which is obviously impractical to implement directly (this would need 256 pointers per node). Hence, we encode each 'virtual' node in this 256-ary trie using a singly-linked list made of the right children of each real, binary node.

```
struct trie_node; /* ref: 9 lines */
```

The interface of `trie` is very simple: it has an `add` method, which inserts a key into the data structure, and a `has` method which decides whether a given key is present. Both accept a single `std::string`. Like with the bit trie before, we do not consider prefixes of included keys to be present.

```
class trie; /* ref: 53 lines; has() = 10, add() = 36 */
```

7.p.5 [bdd] Binary decision diagrams are a compact way to write boolean functions in multiple arguments. You could think of the data structure as a DAG with additional semantics: each vertex is either a variable and has two successors which tell us where to go next depending on the value of that variable, or is a 0 or 1, represented by two sink nodes in the DAG (there are no outgoing edges).
The interface should be as follows;

- the constructor takes a `char`: the variable to use for the root node
- `one` returns the true node
- `zero` returns the false node
- `root` returns the initial node
- `add_var` takes a char and creates a new variable node: there may be multiple nodes for the same variable
- `add_edge` takes the parent node, a boolean, and the child
- `eval` takes a map from `char` to `bool` and returns `true` or `false` by traversing the BDD from the root and at each variable node, taking the path dictated by the input map (variable assignment)

Note: It is UB if a variable node does not have both successors set.

```
class bdd_node; /* ref:  6 lines */
class bdd;      /* ref: 19 lines */
```

7.p.6 [rope] A rope is a string-like data structure, represented as a binary tree with traditional strings in leaves and weights in internal nodes. Subtree sharing is allowed and expected.
A weight of a given node is the total length of the string represented by its left subtree. Provides an O(1) concatenation and O(d) indexing, where d is the depth of the tree.
In addition to the indexing operator, provide 2 constructors: one which constructs a singleton rope from a string, and another that joins 2 existing ropes.
You do not need to implement any rebalancing.

```
class rope;
```

Part 7.r: Regular Exercises

7.r.1 [circular] In this exercise, we will implement a slightly unusual data structure: a circular linked list, but instead of the usual access operators and iteration, it will have a `rotate` method, which rotates the entire list. We require that rotation does not invalidate any references to elements in the list.
If you think of the list as a stack, you can think of the `rotate` operation as taking an element off the top and putting it at the bottom of the stack. It is undefined on an empty list.
To add and remove elements, we will implement `push` and `pop` which work in a stack-like manner. Only the top element is accessible, via

the top method. This method should allow both read and write access. Finally, we also want to be able to check whether the list is empty. As always, we will store integers in the data structure.

```
class circular;
```

7.r.2 [zipper] Implement our favourite data structure – a zipper of integers – this time using a unique_ptr-linked list extending both ways from the focus. Methods:

- (constructor) constructs a singleton zipper from an integer,
- shift_left and shift_right move the point of focus, in O(1), to the nearest left (right) element; they return true if this was possible, otherwise they return false and do nothing,
- push_left and push_right add a new element just left (just right) of the current focus, again in O(1),
- focus access the current item (read and write).

7.r.3 [segment] In this exercise, we will go back to building data structures, in this particular case a simple binary tree. The structure should represent a partitioning of an interval with integer bounds into a set of smaller, non-overlapping intervals.
Implement class segment_map with the following interface:

- the constructor takes two integers, which represent the limits of the interval to be segmented,
- a split operation takes a single integer, which becomes the start of a new segment, splitting the existing segment in two,
- query, given an integer n, returns the bounds of the segment that contains n, as an std::pair of integers.

The tree does not need to be self-balancing: the order of splits will determine the shape of the tree.

7.r.4 [diff] In this exercise, we will implement automatic differentiation of simple expressions. You will need the following rules:

- linearity: $(a \cdot f(x) + b \cdot g(x))' = a \cdot f'(x) + b \cdot g'(x)$
- the Leibniz rule: $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
- chain rule: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
- derivative of exponential: $exp'(x) = exp(x)$

Define a type, expr (from expression), such that values of this type can be constructed from integers, added and multiplied, and exponentiated using function expnat (to avoid conflicts with the exp in the standard library).

```
class expr; /* ref: 29 + 7 lines */
expr expnat( expr );
```

Implement function diff that accepts a single expr and returns the derivative (again in the form of expr). Define a constant x of type expr such that diff(x) is 1.

```
expr diff( expr ); /* ref: 11 lines */
// const expr x;
```

Finally, implement function eval which takes an expr and a double and it substitutes for x and computes the value of the expression.

```
double eval( expr, double ); /* ref: 11 lines */
```

Part 8: Inheritance and Polymorphism

This week will be about objects in the OOP (object-oriented programming) sense and about inheritance-based polymorphism. In OOP, classes are rarely designed in isolation: instead, new classes are derived from an existing base class (the derived class inherits from the base class). The derived class retains all the attributes (data) and methods (behaviours) of the base (parent) class, and usually adds something on top, or at least modifies some of the behaviours.
So far, we have worked with composition (though we rarely called it that). We say objects (or classes) are composed when attributes of classes are other classes (e.g. standard containers). The relationship between the outer class and its attributes is known as 'has-a': a circle has a center, a polynomial has a sequence of coefficients, etc.
Inheritance gives rise to a different type of relationship, known as 'is-a': a few stereotypical examples:

- a circle is a shape,
- a ball is a solid, a cube is a solid too,
- a force is a vector (and so is velocity).

This is where polymorphism comes into play: a function which doesn't care about the particulars of a shape or a solid or a vector can accept an instance of the base class. However, each instance of a derived class is an instance of the base class too, and hence can be used in its place. This is known as the Liskov substitution principle.
An important caveat: this does not work when passing objects by value, because in general, the base class and the derived class do not have the same size. Languages like Python or Java side-step this issue by always passing objects by reference. In C++, we have to do that explicitly if we want to use inheritance-based polymorphism. Of course, this also works with pointers (including smart ones, like std::unique_ptr).
With this bit of theory out of the way, let's look at some practical examples: the rest of theory (late binding in particular) will be explained in demonstrations:

1. account – a simple inheritance example
2. shapes – polymorphism and late dispatch
3. expr – dynamic and static types, more polymorphism
4. destroy – virtual destructors
5. factory – polymorphic return values

Elementary exercises:

1. resistance – compute resistance of a simple circuit
2. perimeter – shapes and their perimeter length
3. fight – rock, paper and scissors

Preparatory exercises:

1. prisoner – the famous dilemma
2. bexpr – boolean expressions with variables
3. sexpr – a tree made of lists (lisp style)
4. network – a network of counters
5. filter – filter items from a data source
6. geometry – shapes and visitors

Regular exercises:

1. bom – polymorphism and collections
2. circuit – calling virtual methods within the class
3. loops – circuits with loops
4. pretty – turn arithmetic expressions into strings
5. json – a more general JSON pretty-printer
6. while – interpreting while programs using an AST

Part 8.d: Demonstrations

8.d.1 [account] In this example, we will demonstrate the syntax and most basic use of inheritance. Polymorphism will not enter the picture yet (but we will get to that very soon: in the next example). We will consider bank accounts (a favourite subject, surely).
We will start with a simple, vanilla account that has a balance, can

withdraw and deposit money. We have seen this before.

```
class account
{
```

The first new piece of syntax is the `protected` keyword. This is related to inheritance: unlike `private`, it lets subclasses (or rather subclass methods) access the members declared in a `protected` section. We also notice that the balance is signed, even though in this class, that is not strictly necessary: we will need that in one of the subclasses (yes, the system is already breaking down a little).

```
protected:
    int _balance;

public:
```

We allow an account to be constructed with an initial balance. We also allow it to be default-constructed, initializing the balance to 0.

```
    account( int initial = 0 )
        : _balance( initial )
    {}
```

Standard stuff.

```
    bool withdraw( int sum )
    {
        if ( _balance > sum )
        {
            _balance -= sum;
            return true;
        }

        return false;
    }

    void deposit( int sum ) { _balance += sum; }
    int balance() const { return _balance; }
};
```

With the base class in place, we can define a derived class. The syntax for inheritance adds a colon, `:`, after the class name and a list of classes to inherit from, with access type qualifiers. We will always use `public` inheritance. Also, did you know that naming things is hard?

```
class account_with_overdraft : public account
{
```

The derived class has, ostensibly, a single attribute. However, all the attributes of all base classes are also present automatically. That is, there already is an `int _balance` attribute in this class, inherited from `account`. We will use it below.

```
protected:
    int _overdraft;

public:
```

This is another new piece of syntax that we will need: a constructor of a derived class must first call the constructors of all base classes. Since this happens before any attributes of the derived class are constructed, this call comes first in the initialization section. The derived-class constructor is free to choose which (overloaded) constructor of the base class to call. If the call is omitted, the default constructor of the base class will be called.

```
    account_with_overdraft( int initial = 0, int overdraft = 0 )
        : account( initial ), _overdraft( overdraft )
    {}
```

The methods defined in a base class are automatically available in the derived class as well (same as attributes). However, unlike attributes, we can replace inherited methods with versions more suitable for the derived class. In this case, we need to adjust the behaviour of `withdraw`.

```
    bool withdraw( int sum )
    {
        if ( _balance + _overdraft > sum )
        {
            _balance -= sum;
            return true;
        }

        return false;
    }
};
```

Here is another example based on the same language features.

```
class account_with_interest : public account
{
protected:
    int _rate; /* percent per annum */

public:

    account_with_interest( int initial = 0, int rate = 0 )
        : account( initial ), _rate( rate )
    {}
```

In this case, all the inherited methods can be used directly. However, we need to add a new method, to compute and deposit the interest. Since naming things is hard, we will call it `next_year`. The formula is also pretty lame.

```
    void next_year()
    {
        _balance += ( _balance * _rate ) / 100;
    }
};
```

The way objects are used in this exercise is not super useful: the goal was to demonstrate the syntax and basic properties of inheritance. In modern practice, code re-use through inheritance is frowned upon (except perhaps for mixins, which are however out of scope for this subject). The main use-case for inheritance is subtype polymorphism, which we will explore in the next unit, `shapes.cpp`.

```
int main() /* demo */
{
```

We first make a normal account and check that it behaves as expected. Nothing much to see here.

```
    account a( 100 );
    assert( a.balance() == 100 );
    assert( a.withdraw( 50 ) );
    assert( !a.withdraw( 100 ) );
    a.deposit( 10 );
    assert( a.balance() == 60 );
```

Let's try the first derived variant, an account with overdraft. We notice that it's possible to have a negative balance now.

```
    account_with_overdraft awo( 100, 100 );
    assert( awo.balance() == 100 );
    assert( awo.withdraw( 50 ) );
    assert( awo.withdraw( 100 ) );
    awo.deposit( 10 );
    assert( awo.balance() == -40 );
```

And finally, let's try the other account variant, with interest.

```
    account_with_interest awi( 100, 20 );
    assert( awi.balance() == 100 );
    assert( awi.withdraw( 50 ) );
```

```
    assert( !awi.withdraw( 100 ) );
    awi.deposit( 10 );
    assert( awi.balance() == 60 );
    awi.next_year();
    assert( awi.balance() == 72 );
}
```

8.d.2 [shapes] The inheritance model in C++ is an instance of a more general notion, known as subtyping. The defining characteristic of subtyping is the Liskov substitution principle: a value which belongs to a subtype (a derived class) can be used whenever a variable stores, or a formal argument expects, a value that belongs to a supertype (the base class). As mentioned earlier, in C++ this only extends to values passed by reference or through pointers.
We will first define a couple useful type aliases to represent points and bounding boxes.

```
using point = std::pair< double, double >;
using bounding_box = std::pair< point, point >;
```

Subtype polymorphism is, in C++, implemented via late binding: the decision which method should be called is postponed to runtime (with normal functions and methods, this happens during compile time). The decision whether to use early binding (static dispatch) or late binding (dynamic dispatch) is made by the programmer on a method-by-method basis. In other words, some methods of a class can use static dispatch, while others use dynamic dispatch.

```
class shape
{
public:
```

To instruct the compiler to use dynamic dispatch for a given method, put the keyword `virtual` in front of that method's return type. Unlike normal methods, a `virtual` method may be left unimplemented: this is denoted by the `= 0` at the end of the declaration. If a class has a method like this, it is marked as abstract and it becomes impossible to create instances of this class: the only way to use it is as a base class, through inheritance. This is commonly done to define interfaces. In our case, we will declare two such methods.

```
    virtual double area() const = 0;
    virtual bounding_box box() const = 0;
```

A class which introduces `virtual` methods also needs to have a destructor marked as `virtual`. We will discuss this in more detail in a later unit. For now, simply consider this to be an arbitrary rule.

```
    virtual ~shape() = default;
};
```

As soon as the interface is defined, we can start working with arbitrary classes which implement this interface, even those that have not been defined yet. We will start by writing a simple polymorphic function which accepts arbitrary shapes and computes the ratio of their area to the area of their bounding box.

```
double box_coverage( const shape &s )
{
```

Hopefully, you remember structured bindings (if not, revisit e.g. 03/rel.cpp).

```
    auto [ ll, ur ] = s.box();
    auto [ left, bottom ] = ll;
    auto [ right, top ] = ur;

    return s.area() / ( ( right - left ) * ( top - bottom ) );
}
```

Another function: this time, it accepts two instances of shape. The values it actually receives may be, however, of any type derived from shape. In fact, a and b may be each an instances of a different derived class.

```
bool box_collide( const shape &sh_a, const shape &sh_b )
{
```

A helper function (lambda) to decide whether a point is inside (or on the boundary) of a bounding box.

```
    auto in_box = []( const bounding_box &box, const point &pt )
    {
        auto [ x, y ] = pt;
        auto [ ll, ur ] = box;
        auto [ left, bottom ] = ll;
        auto [ right, top ] = ur;

        return x >= left && x <= right && y >= bottom && y <= top;
    };

    auto [ a, b ] = sh_a.box();
    auto box = sh_b.box();
```

The two boxes collide if either of the corners of one is in the other box.

```
    return in_box( box, a ) || in_box( box, b );
}
```

We now have the interface and two functions that are defined in terms of that interface. To make some use of the functions, however, we need to be able to make instances of shape, and as we have seen earlier, that is only possible by deriving classes which provide implementations of the virtual methods declared in the base class. Let's start by defining a circle.

```
class circle : public shape
{
    point _center;
    double _radius;
public:
```

The base class has a default constructor, so we do not need to explicitly call it here.

```
    circle( point c, double r ) : _center( c ), _radius( r ) {}
```

Now we need to implement the `virtual` methods defined in the base class. In this case, we can omit the `virtual` keyword, but we should specify that this method overrides one from a base class. This informs the compiler of our intention to provide an implementation to an inherited method and allows it (the compiler) to emit a warning in case we accidentally hide the method instead, by mistyping the signature. The most common mistake is forgetting the trailing `const`. Please always specify `override` where it is applicable.

```
    double area() const override
    {
        return 4 * std::atan( 1 ) * std::pow( _radius, 2 );
    }
```

Now the other `virtual` method.

```
    bounding_box box() const override
    {
        auto [ x, y ] = _center;
        double r = _radius;
        return { { x - r, y - r }, { x + r, y + r } };
    }
};
```

And a second shape type, so we can actually make some use of polymorphism. Everything is the same as above.

```
class rectangle : public shape
{
    point _ll, _ur; /* lower left, upper right */
public:

    rectangle( point ll, point ur ) : _ll( ll ), _ur( ur ) {}

    double area() const override
    {
        auto [ left, bottom ] = _ll;
        auto [ right, top ] = _ur;
        return ( right - left ) * ( top - bottom );
    }

    bounding_box box() const override
    {
        return { _ll, _ur };
    }
};

int main() /* demo */
{
```

We cannot directly construct a shape, since it is abstract, i.e. it has unimplemented pure virtual methods. However, both `circle` and `rectangle` provide implementations of those methods which we can use.

```
    rectangle square( { 0, 0 }, { 1, 1 } );
    assert( square.area() == 1 );
    assert( square.box() == bounding_box( { 0, 0 }, { 1, 1 } ) );
    assert( box_coverage( square ) == 1 );

    circle circ( { 0, 0 }, 1 );
```

Check that the area of a unit circle is π, and the ratio of its area to its bounding box is π / 4.

```
    double pi = 4 * std::atan( 1 );
    assert( std::fabs( circ.area() - pi ) < 1e-10 );
    assert( std::fabs( box_coverage( circ ) - pi / 4 ) < 1e-10 );
```

The two shapes quite clearly collide, and if they collide, their bounding boxes must also collide. A shape should always collide with itself, and collisions are symmetric, so let's check that too.

```
    assert( box_collide( square, circ ) );
    assert( box_collide( circ, square ) );
    assert( box_collide( square, square ) );
    assert( box_collide( circ, circ ) );
```

Let's make a shape a bit further out and check the collision detection with that.

```
    circle c1( { 2, 3 }, 1 ), c2( { -1, -1 }, 1 );
    assert( !box_collide( circ, c1 ) );
    assert( !box_collide( c1, c2 ) );
    assert( !box_collide( c1, square ) );
    assert(  box_collide( c2, square ) );
}
```

8.d.3 [expr] To better understand polymorphism, we will need to set up some terminology, particularly:

- the notion of a static type, which is, essentially, the type written down in the source code, and of a
- dynamic type (also known as a runtime type), which is the actual type of the value that is stored behind a given reference (or pointer).

The relationship between the static and dynamic type may be:

- the static and dynamic type are the same (this was always the case until this week), or
- the dynamic type may be a subtype of the static type (we will see that in a short while).

Anything else is a bug.

We will use a very simple representation of arithmetic expressions as our example here. An expression is a tree, where each node carries either a value or an operation. We will want to explicitly track the type of each node, and for that, we will use an enumerated type. Those work the same as in C, but if we declare them using `enum class`, the enumerated names will be scoped: we use them as `type::sum`, instead of just `sum` as would be the case in C.

```
enum class type { sum, product, constant };
```

Now for the class hierarchy. The base class will be `node`.

```
class node
{
public:
```

The first thing we will implement is a `static_type` method, which tells us the static type of this class. The base class, however, does not have any sensible value to return here, so we will just throw an exception.

```
    type static_type() const
    {
        throw std::logic_error( "bad static_type() call" );
    }
```

The 'real' (dynamic) type must be a `virtual` method, since the actual implementation must be selected based on the dynamic type: this is exactly what late binding does. Since the method is `virtual`, we do not need to supply an implementation if we can't give a sensible one.

```
    virtual type dynamic_type() const = 0;
```

The interesting thing that is associated with each node is its value. For operation nodes, it can be computed, while for leaf nodes (type `constant`), it is simply stored in the node.

```
    virtual int value() const = 0;
```

We also observe the virtual destructor rule.

```
    virtual ~node() = default;
};
```

We first define the (simpler) leaf nodes, i.e. constants.

```
class constant : public node
{
    int _value;
public:
```

The leaf node constructor simply takes an integer value and stores it in an attribute.

```
    constant( int v ) : _value( v ) {}
```

Now the interface common to all `node` instances:

```
    type static_type() const { return type::constant; }
```

In methods of class `constant`, the static type of `this` is always[9] either `constant *` or `const constant *`. Hence we can simply call the `static_type` method, since it uses static dispatch (it was not declared `virtual` in the base class) and hence the call will always resolve to the method just above.

```
    type dynamic_type() const override { return static_type(); }
```

Finally, the 'business' method:

[9] As long as we pretend that the `volatile` keyword does not exist, which is an entirely reasonable thing to do.

```
    int value() const override { return _value; }
};
```

The inner nodes of the tree are operations. We will create an intermediate (but still abstract) class, to serve as a base for the two operation classes which we will define later.

```
class operation : public node
{
    const node &_left, &_right;

public:
    operation( const node &l, const node &r )
        : _left( l ), _right( r )
    {}
```

We will leave static_type untouched: the version from the base class works okay for us, since there is nothing better that we could do here. The dynamic_type and value stay unimplemented.
We are facing a dilemma here, though. We would like to add accessors for the children, but it is not clear whether to make them virtual or not. Considering that we keep the references in attributes of this class, it seems unlikely that the implementation of the accessors would change in a subclass and we can use cheaper static dispatch.

```
    const node &left() const { return _left; }
    const node &right() const { return _right; }
};
```

Now for the two operation classes.

```
class sum : public operation
{
public:
```

The base class does not have a default constructor, which means we need to call the one that's available manually.

```
    sum( const node &l, const node &r )
        : operation( l, r )
    {}
```

We want to replace the static_type implementation that was inherited from node (through operation):

```
    type static_type() const { return type::sum; }
```

And now the (dynamic-dispatch) interface mandated by the (indirect) base class node. We can use the same approach that we used in constant for dynamic_type:

```
    type dynamic_type() const override { return static_type(); }
```

And finally the logic. The static return type of left and right is const node &, but the method we call on each, value, uses dynamic dispatch (it is marked virtual in class node). Therefore, the actual method which will be called depends on the dynamic type of the respective child node.

```
    int value() const override
    {
        return left().value() + right().value();
    }
};
```

Basically a re-run of sum.

```
class product : public operation
{
public:
```

We will use a trick which will allow us to not type out the (boring and redundant) constructor. If all we want to do is just forward arguments to the parent class, we can use the following syntax. You do not have to remember it, but it can save some typing if you do.

```
    using operation::operation;
```

Now the interface methods.

```
    type static_type() const { return type::product; }
    type dynamic_type() const override { return static_type(); }

    int value() const override
    {
        return left().value() * right().value();
    }
};

int main() /* demo */
{
```

Instances of class constant are quite straightforward. Let's declare some.

```
    constant const_1( 1 ),
             const_2( 2 ),
             const_m1( -1 ),
             const_10( 10 );
```

The constructor of sum accepts two instances of node, passed by reference. Since constant is a subclass of node, it is okay to use those, too.

```
    sum sum_0( const_1, const_m1 ),
        sum_3( const_1, const_2 );
```

The product constructor is the same. But now we will also try using instances of sum, since sum is also derived (even if indirectly) from node and therefore sum is a subtype of node, too.

```
    product prod_4( const_2, const_2 ),
            prod_6( const_2, sum_3 ),
            prod_40( prod_4, const_10 );
```

Let's also make a sum instance which has children of different types.

```
    sum sum_9( sum_3, prod_6 );
```

For all variables which hold values (i.e. not references), static type = dynamic type. To make the following code easier to follow, the static type of each of the above variables is explicitly mentioned in its name. Clearly, we can call the value method on the variables directly and it will call the right method.

```
    assert( const_1.value() == 1 );
    assert( const_2.value() == 2 );
    assert( sum_0.value() == 0 );
    assert( sum_3.value() == 3 );
    assert( prod_4.value() == 4 );
    assert( prod_6.value() == 6 );
    assert( prod_40.value() == 40 );
    assert( sum_9.value() == 9 );
```

However, the above results should already convince us that dynamic dispatch works as expected: the results depend on the ability of sum::value and product::value to call correct versions of the value method on their children, even though the static types of the references stored in operation are const node. We can however explore the behaviour in a bit more detail.

```
    const node &sum_0_ref = sum_0, &prod_6_ref = prod_6;
```

Now the static type of sum_0_ref is const node &, but the dynamic type of the value to which it refers is sum, and for prod_6_ref the static type is const node & and dynamic is product.

```
    assert( sum_0_ref.value() == 0 );
    assert( prod_6_ref.value() == 6 );
```

Let us also check the behaviour of left and right.

```
    assert( sum_0.left().value()  ==  1 );
    assert( sum_0.right().value() == -1 );
```

The static type through which we call left and right does not matter, because neither product nor sum provide a different implementation of the method.

```
    const operation &op = sum_0;
    assert( op.left().value() == 1 );
    assert( op.right().value() == -1 );
```

The final thing to check is the static_type and dynamic_type methods. By now, we should have a decent understanding of what to expect. Please note that sum_0 and sum_0_ref refer to the same instance and hence they have the same dynamic type, even though their static types differ.

```
    assert( sum_0.dynamic_type() == type::sum );
    assert( sum_0_ref.dynamic_type() == type::sum );

    assert( sum_0.static_type() == type::sum );

    try { sum_0_ref.static_type(); assert( false ); }
    catch ( const std::logic_error & ) {}
```

And the same is true about prod_6 and prod_6_ref.

```
    assert( prod_6.dynamic_type() == type::product );
    assert( prod_6_ref.dynamic_type() == type::product );
    assert( prod_6.static_type() == type::product );

    try { prod_6_ref.static_type(); assert( false ); }
    catch ( const std::logic_error & ) {}
}
```

8.d.4 [destroy] In this (entirely synthetic, sorry) example, we will look at object destruction, especially in the context of polymorphism. We first set up a few counters to track constructor and destructor calls.

```
static int bad_base_counter = 0, bad_derived_counter = 0,
           good_base_counter = 0, good_derived_counter = 0;

class bad_base
{
public:
    virtual int bad_dummy() { return 0; }

    bad_base() { bad_base_counter ++; }
```

We will knowingly break the virtual destructor rule here, to see why the rule exists.

```
    ~bad_base() { bad_base_counter --; }
};

class good_base
{
public:
    virtual int good_dummy() { return 0; }

    good_base() { good_base_counter ++; }
```

Notice the virtual.

```
    virtual ~good_base() { good_base_counter --; }
};
```

Let's add some innocent derived classes.

```
class bad_derived : public bad_base
{
public:
    bad_derived() { bad_derived_counter ++; }
    ~bad_derived() { bad_derived_counter --; }
};

class good_derived : public good_base
{
public:
    good_derived() { good_derived_counter ++; }
```

It is good practice to also add override to destructors of derived classes. This will tell the compiler we expect the base class to have a virtual destructor which we are extending. The compiler will emit an error if the base class destructor is (through some unfortunate accident) not marked as virtual.

```
    ~good_derived() override { good_derived_counter --; }
};

int main() /* demo */
{
```

For regular variables, everything works as expected: constructors and destructors of all classes in the hierarchy are called.

```
    {
        bad_base bb;
        assert( bad_base_counter == 1 );
        bad_derived bd;
        assert( bad_base_counter == 2 );
        assert( bad_derived_counter == 1 );
    }

    assert( bad_base_counter == 0 );
    assert( bad_derived_counter == 0 );
```

Same thing with virtual destructors.

```
    {
        good_base gb;
        assert( good_base_counter == 1 );
        good_derived gd;
        assert( good_base_counter == 2 );
        assert( good_derived_counter == 1 );
    }

    assert( good_base_counter == 0 );
    assert( good_derived_counter == 0 );
```

However, problems start if an instance is destroyed through a pointer whose static type disagrees with the dynamic type. This cannot happen with references (unless the destructor is called explicitly), but it is entirely plausible with pointers, including smart pointers. Let's first demonstrate the case that works: good_derived.

```
    using good_ptr = std::unique_ptr< good_base >;
```

Please make good note of the fact, that the static type of the pointer refers to good_base, but the actual value stored in it has dynamic type good_derived.

```
    {
        good_ptr gp = std::make_unique< good_derived >();
        assert( good_base_counter == 1 );
        assert( good_derived_counter == 1 );
    }
```

Since the unique_ptr went out of scope, the instance stored behind it was destroyed. The counters should be both zero again.

```
    assert( good_base_counter == 0 );
    assert( good_derived_counter == 0 );
```

Let's observe what happens with the bad_base and bad_derived combination.

```
using bad_ptr = std::unique_ptr< bad_base >;

{
    bad_ptr bp = std::make_unique< bad_derived >();
    assert( bad_base_counter == 1 );
    assert( bad_derived_counter == 1 );
}
```

The pointer went out of scope. Since the destructor was called using static dispatch, only the base class destructor was called. This is of course very problematic, since resources were leaked and invariants broken.

```
assert( bad_base_counter == 0 );
assert( bad_derived_counter == 1 );
```

Please note that some compilers (recent clang versions) will emit a warning if this happens. Unfortunately, this is not the case with gcc 9.2 which we are using (and which is a rather recent compiler). It is therefore unadvisable to rely on the compiler to catch this type of problem. Stay vigilant.

```
}
```

8.d.5 [factory] As we have seen, subtype polymorphism allows us to define an interface in terms of virtual methods (that is, based on late dispatch) and then create various implementations of this interface.
It is sometimes useful to create instances of multiple different derived classes based on runtime inputs, but once they are created, to treat them uniformly. The uniform treatment is made possible by subtype polymorphism: if the entire interaction with these objects is done through the shared interface, the instances are all, at the type level, interchangeable with each other. The behaviour of those instances will of course differ, depending on their dynamic type.
When a system is designed this way, the entire program uses a single static type to work with all instances from the given inheritance hierarchy – the type of the base class. Let's define such a base class.

```
class part
{
public:
    virtual std::string description() const = 0;
    virtual ~part() = default;
};
```

Let's add a simple function which operates on generic parts. Working with instances is easy, since they can be passed through a reference to the base type. For instance the following function which formats a single line for a bill of materials (bom).

```
std::string bom_line( const part &p, int count )
{
    return std::to_string( count ) + "x " + p.description();
}
```

However, creation of these instances poses a somewhat unique challenge in C++: memory management. In languages like Java or C#, we can create the instance and return a reference to the caller, and the garbage collector will ensure that the instance is correctly destroyed when it is no longer used. We do not have this luxury in C++.
Of course, we could always do memory management by hand, like it's 1990. Fortunately, modern C++ provides smart pointers in the standard library, making memory management much easier and safer. Recall that a unique_ptr is an owning pointer: it holds onto an object instance while it is in scope and destroys it afterwards. Unlike objects stored in local variables, though, the ownership of the instance held in a unique_ptr can be transferred out of the function (i.e. an instance of unique_ptr can be legally returned, unlike a reference to a local variable).
This will make it possible to define a factory: a function which constructs instances (parts) and returns them to the caller. Of course, to actually define the function, we will need to define the derived classes which it is supposed to create.

```
using part_ptr = std::unique_ptr< part >;
part_ptr factory( std::string );
```

In the program design outlined earlier, the derived classes change some of the behaviours, or perhaps add data members (attributes) to the base class, but apart from construction, they are entirely operated through the interface defined by the base class.

```
class cog : public part
{
    int teeth;
public:
    cog( int teeth ) : teeth( teeth ) {}

    std::string description() const override
    {
        return std::string( "cog with " ) +
               std::to_string( teeth ) + " teeth";
    }
};

class axle : public part
{
public:
    std::string description() const override
    {
        return "axle";
    }
};

class screw : public part
{
    int _thread, _length;
public:

    screw( int t, int l ) : _thread( t ), _length( l ) {}

    std::string description() const override
    {
        return std::to_string( _length ) + "mm M" +
               std::to_string( _thread ) + " screw";
    }
};
```

Now that we have defined the derived classes, we can finally define the factory function.

```
part_ptr factory( std::string desc )
{
```

We will use std::istringstream (first described in 06/streams.cpp) to extract a description of the instance that we want to create from a string. The format will be simple: the type of the part, followed by its parameters separated by spaces.

```
    std::istringstream s( desc );
    std::string type;
    s >> type; /* extract the first word */

    if ( type == "cog" )
    {
        int teeth;
        s >> teeth;
        return std::make_unique< cog >( teeth );
    }

    if ( type == "axle" )
        return std::make_unique< axle >();

    if ( type == "screw" )
```

```
    {
        int thread, length;
        s >> thread >> length;
        return std::make_unique< screw >( thread, length );
    }

    throw std::runtime_error( "unexpected part description" );
}

int main() /* demo */
{
```

Let's first use the factory to make some instances. They will be held by part_ptr (i.e. unique_ptr with the static type part.

```
    part_ptr ax = factory( "axle" ),
             m7 = factory( "screw 7 50" ),
             m3 = factory( "screw 3 10" ),
             c8 = factory( "cog 8" ),
             c9 = factory( "cog 9" );
```

From the point of view of the static type system, all the parts created above are now the same. We can call the methods which were defined in the interface, or we can pass them into functions which work with parts.

```
    assert( ax->description() == "axle" );
    assert( m7->description() == "50mm M7 screw" );
    assert( m3->description() == "10mm M3 screw" );
    assert( c8->description() == "cog with 8 teeth" );
    assert( c9->description() == "cog with 9 teeth" );
```

Let's try using the bom_line function which we have defined earlier.

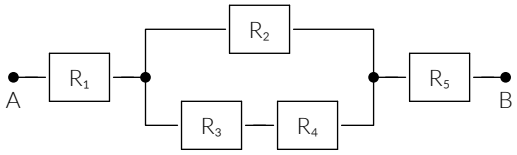
```
    assert( bom_line( *ax, 3 ) == "3x axle" );
    assert( bom_line( *m7, 20 ) == "20x 50mm M7 screw" );
```

At the end of the scope, the objects are destroyed and all memory is automatically freed.

```
}
```

Part 8.e: Elementary Exercises

8.e.1 [resistance] We are given a simple electrical circuit made of resistors and wires, and we want to compute the total resistance between two points. The circuit is simple in the sense that in any given section, all its immediate sub-sections are either connected in series or in parallel. Here is an example:



The resistance that we are interested in is between the points A and B. Given R_1 and R_2 connected in series, the total resistance is $R = R_1 + R_2$. For the same resistors connected in parallel, the resistance is given by $1/R = 1/R_1 + 1/R_2$.
You will implement 2 classes: series and parallel, each of which represents a single segment of the circuit. Both classes shall provide a method add, that will accept either a number (double) which will add a single resistor to that segment, or a const reference to the opposite class (i.e. an instance of series should accept a reference to parallel and vice versa).

```
class series;
class parallel;
```

Then add a top-level function resistance, which accepts either a series or a parallel instance and computes the total resistance of the circuit described by that instance. The exact prototype is up to you.

8.e.2 [perimeter] Implement a simple inheritance hierarchy – the base class will be shape, with a pure virtual method perimeter, the 2 derived classes will be circle and rectangle. The circle is constructed from a radius, while the rectangle from a width and height, all of them floating-point numbers.

```
class shape;
class circle;
class rectangle;

bool check_shape( const shape &s, double p )
{
    return std::fabs( s.perimeter() - p ) < 1e-8;
}
```

8.e.3 [fight] There should be 4 classes: the base class gesture and 3 derived: rock, paper and scissors. Class gesture has a (pure virtual) method fight which takes another gesture (via a const reference) and returns true if the current gesture wins.
To do this, add another method, visit, which has 3 overloads, one each for rock, paper and scissors. Then override fight in each derived class, to simply call visit(*this) on the opposing gesture. The visit method knows the type of both this and the opponent (via the overload) – simply indicate the winner by returning an appropriate constant.

```
class rock;
class paper;
class scissors;
```

Keep the forward declarations, you will need them to define the overloads for visit.

```
class gesture;
```

Now define the 3 derived classes.

Part 8.p: Preparatory Exercises

8.p.1 [prisoner] Another exercise, another class hierarchy. The abstract base class will be called prisoner, and the implementations will be different strategies in the well-known game of (iterated) prisoner's dilemma.
The prisoner class should provide method betray which takes a boolean (the decision of the other player in the last round) and returns the decision of the player for this round. In general, the betray method should not be const, because strategies may want to remember past decisions (though we will not implement a strategy like that in this exercise).

```
class prisoner;
```

Implement an always-betray strategy in class traitor, the tit-for-tat strategy in vengeful and an always-cooperate in benign.

```
class traitor;
class vengeful;
class benign;
```

Implement a simple strategy evaluator in function play. It takes two prisoners and the number of rounds and returns a negative number if the first one wins, 0 if the game is a tie and a positive number if the second wins. The scoring matrix:

- neither player betrays 2 / 2
- a betrays, b does not: 3 / 0
- a does not betray, b does: 0 / 3
- both betray 1 / 1

```
int play( prisoner &a, prisoner &b, int rounds );
```

8.p.2 [bexpr] Boolean expressions with variables, represented as binary trees. Internal nodes carry a logical operation on the values obtained from children while leaf nodes carry variable references.
To evaluate an expression, we will need to supply values for each of the variables that appears in the expression. We will identify variables using integers, and the assignment of values will be done through the type `input` defined below. It is undefined behaviour if a variable appears in an expression but is not present in the provided `input` value.

```
using input = std::map< int, bool >;
```

Like earlier in `expr.cpp`, the base class will be called `node`, but this time will only define a single method: `eval`, which accepts a single `input` argument (as a `const` reference).

```
class node; /* ref: 6 lines */
```

Internal nodes are all of the same type, and their constructor takes an unsigned integer, `table`, and two `node` references. Assuming bit zero is the lowest-order bit, the node operates as follows:

- `false false` → bit 0 of `table`
- `false true` → bit 1 of `table`
- `true false` → bit 2 of `table`
- `true true` → bit 3 of `table`

```
class operation; /* ref: 16 lines */
```

The leaf nodes carry a single integer (passed in through the constructor) – the identifier of the variable they represent.

```
class variable; /* ref: 7 lines */
```

8.p.3 [sexpr] An s-expression is a tree in which each node has an arbitrary number of children. To make things a little more interesting, our s-expression nodes will own their children.
The base class will be called `node` (again) and it will have single (virtual) method: `value`, with no arguments and an `int` return value.

```
class node;
using node_ptr = std::unique_ptr< node >;
```

There will be two types of internal nodes: `sum` and `product`, and in this case, they will compute the sum or the product of all their children, regardless of their number. A sum with no children should evaluate to 0 and a product with no children should evaluate to 1.
Both will have an additional method: `add_child`, which accepts (by value) a single `node_ptr` and both should have default constructors. It is okay to add an intermediate class to the hierarchy.

```
class sum;
class product;
```

Leaf nodes carry an integer constant, given to them via a constructor.

```
class constant;
```

8.p.4 [network] In this exercise, we will define a network of counters, where each node has its own counter which starts at zero, and events which affect the counters propagate in the network. Different node types react differently to the events.
There are three basic events which can propagate through the network: `reset` will set the counter to 0, `increment` and `decrement` add and subtract 1, respectively.

```
enum class event { reset, increment, decrement };
```

The abstract base class, `node`, will define the polymorphic interface. Methods:

- `react` with a single argument of type `event`,
- `connect` which will take a reference to another `node` instance: the connection thus created starts in `this` and extends to the `node` given in the argument,
- `read`, a `const` method that returns the current value of the counter.

Think carefully about which methods need to be `virtual` and which don't. The counter is signed and starts at 0. Each node can have an arbitrary number of both outgoing and incoming connections.

```
class node;
```

Now for the node types. Each node type first applies the event to its own counter, then propagates (or not) some event along all outgoing connections. Implement the following node types:

- `forward` sends the same event it has received
- `invert` sends the opposite event
- `gate` resends the event if the new counter value is positive

```
class forward;
class invert;
class gate;
```

8.p.5 [filter] This exercise will be yet another take on a set of numbers. This time, we will add a capability to filter the numbers on output. It will be possible to change the filter applied to a given set at runtime.
The base class for representing filters will contain a single pure `virtual` method, `accept`. The method should be marked `const`.

```
class filter;
```

The `set` (which we will implement below) will own the filter instance and hence will use a `unique_ptr` to hold it.

```
using filter_ptr = std::unique_ptr< filter >;
```

The `set` should have standard methods: `add` and `has`, the latter of which will respect the configured filter (i.e. items rejected by the filter will always test negative on `has`). The method `set_filter` should set the filter. If no filter is set, all numbers should be accepted. Calling `set_filter` with a `nullptr` argument should clear the filter.
Additionally, `set` should have `begin` and `end` methods (both `const`) which return very simple iterators that only provide dereference to an `int` (value), pre-increment and inequality. It is a good idea to keep two instances of `std::set< int >::iterator` in attributes (in addition to a pointer to the output filter): you will need to know, in the pre-increment operator, that you ran out of items when skipping numbers which the filter rejected.

```
class set_iterator;
class set;
```

Finally, implement a filter that only accepts odd numbers.

```
class odd;
```

8.p.6 [geometry] We will go back to a bit of geometry, this time with circles and lines: in this exercise, we will be interested in planar intersections. We will consider two objects to intersect when they have at least one common point. On the C++ side, we will use a bit of a trick with `virtual` method overloading (in a slightly more general setting, the trick is known as the visitor pattern).
First some definitions: the familiar `point`.

```
using point = std::pair< double, double >;
```

Check whether two floating-point numbers are 'essentially the same' (i.e. fuzzy equality).

```
bool close( double a, double b )
{
    return std::fabs( a - b ) < 1e-10;
}
```

We will need to use forward declarations in this exercise, since methods of the base class will refer to the derived types.

```
struct circle;
struct line;
```

These two helper functions are already defined in this file and may come in useful (like the `slope` class above).

```
double dist( point, point );
double dist( const line &, point );
```

A helper class which is constructed from two points. Two instances of `slope` compare equal if the slopes of the two lines passing through the respective point pairs are the same.

```
struct slope : std::pair< double, double >
{
    slope( point p, point q )
        : point( ( q.first  - p.first  ) / dist( p, q ),
                 ( q.second - p.second ) / dist( p, q ) )
    {}

    bool operator==( const slope &o ) const
    {
        auto [ px, py ] = *this;
        auto [ qx, qy ] = o;

        return ( close( px, qx ) && close( py, qy ) ) ||
               ( close( px, -qx ) && close( py, -qy ) );
    }

    bool operator!=( const slope &o ) const
    {
        return !( *this == o );
    }
};
```

Now we can define the class `object`, which will have a `virtual` method `intersects` with 3 overloads: one that accepts a `const` reference to a `circle`, another that accepts a `const` reference to a `line` and finally one that accepts any `object`.

```
class object;
```

Put definitions of the classes `circle` and `line` here. A `circle` is given by a `point` and a radius (`double`), while a `line` is given by two points. NB. Make the `line` attributes `public` and name them `p` and `q` to make the `dist` helper function work.

```
struct circle; /* ref: 18 lines */
struct line;   /* ref: 18 lines */
```

Definitions of the helper functions.

```
double dist( point p, point q )
{
    auto [ px, py ] = p;
    auto [ qx, qy ] = q;
    return std::sqrt( std::pow( px - qx, 2 ) +
                      std::pow( py - qy, 2 ) );
}

double dist( const line &l, point p )
{
    auto [ x2, y2 ] = l.q;
    auto [ x1, y1 ] = l.p;
    auto [ x0, y0 ] = p;
    return std::fabs( ( y2 - y1 ) * x0 - ( x2 - x1 ) * y0 +
                      x2 * y1 - y2 * x1 ) /
           dist( l.p, l.q );
}
```

Part 8.r: Regular Exercises

8.r.1 **[bom]** Let's revisit the idea of a bill of materials that made a brief appearance in `factory.cpp`, but in a slightly more useful incarnation. Define the following class hierarchy: the base class, `part`, should have a (pure) virtual method `description` that returns an `std::string`. It should also keep an attribute of type `std::string` and provide a getter for this attribute called `part_no()` (part number). Then add 2 derived classes:

- `resistor` which takes the part number and an integral resistance as its constructor argument and provides a description of the form `"resistor ?Ω"` where `?` is the provided resistance,
- `capacitor` which also takes a part number and an integral capacitance and provides a description of the form `"capacitor ?μF"` where `?` is again the provided value.

```
class part;
class resistor;
class capacitor;
```

We will also use owning pointers, so let us define a convenient type alias for that:

```
using part_ptr = std::unique_ptr< part >;
```

That was the mechanical part. Now we will need to think a bit: we want a class `bom` which will remember a list of parts, along with their quantities and will own the `part` instances it holds. The interface:

- a method `add`, which accepts a `part_ptr` by value (it will take ownership of the instance) and the quantity (integer)
- a method `find` which accepts an `std::string` and returns a `const` reference to the part instance with the given part number,
- a method `qty` which returns the associated quantity, given a part number.

```
class bom;
```

8.r.2 **[circuit]** In this exercise, we will look at calling `virtual` methods from within the class, in an 'inverted' approach to inheritance. Most of the implementation will be part of the base class, in terms of a few (or in this case one) `protected virtual` methods.
We will implement a simple class hierarchy to represent a logical circuit: a bunch of components connected with wires. Each component will have at most 2 inputs and a single output (all of which are boolean values). Implement the following (non-virtual) methods:

- `connect` which takes an integer (0 or 1, the index of the input to connect) and a reference to another `component` and connects the output of the given component to the input of this component
- `read` with no arguments, which returns the current output of the component (this will of course depend on the state of the input components).

Both inputs start out unconnected. Unconnected inputs always read out `false`. Behaviour is undefined if there is a loop in the circuit (but see also `loops.cpp`).

```
class component;
```

The derived classes should be as follows:

- nand for which the output is the NAND logical function of the two inputs,
- source which ignores both inputs and reads out true,
- delay which behaves as follows: first time read is called, it always returns zero; subsequent read calls return a value that the input 0 had at the time of the previous call to read.

```
class nand;
class source;
class delay;
```

8.r.3 [loops] Same basic idea as circuit.cpp: we model a circuit made of components. Things get a bit more complicated in this version:

- loops are allowed
- parts have 2 inputs and 2 outputs each

The base class, with the following interface:

- read takes an integer (decides which output) and returns a boolean,
- connect takes two integers and a reference to a component (the first integer defines the input of this and the second integer defines the output of the third argument to connect).

There is more than one way to resolve loops, some of which require read to be virtual (that's okay). Please note that each loop must have at least one delay in it (otherwise, behaviour is still undefined). NB. Each component should first read input 0 and then input 1: the ordering will affect the result.

```
class component; /* ref: 30 lines */
```

A delay is a component that reads out, on both outputs, the value it has obtained on the corresponding input on the previous call to read.

```
class delay; /* ref: 20 lines */
```

A latch remembers one bit of information (starting at false):

- if both inputs read false, the remembered bit remains unchanged,
- if input 0 is false while input 1 is true the remembered bit is set to true,
- in all other cases, the remembered bit is set to false.

The value on output 0 is the new value of the remembered bit: there is no delay. The value on output 1 is the negation of output 0.

```
class latch; /* 15 lines */
```

Finally, the cnot gate, or a controlled not gate has the following behaviour:

- output 0 always matches input 0, while
- output 1 is set to:
  - input 1 if input 0 is true
  - negation of input 1 if input 0 is false

```
class cnot; /* ref: 11 lines */
```

Part T.2: Tasks with Operators, Exceptions and OOP

The programming tasks for this block are as follows:

1. machine.* – a simple register machine simulator,
2. natural.* – arbitrary-size natural numbers,
3. parser.* – parsing simplified JSON,
4. complex.* – arbitrary-precision complex numbers.

The first task only relies on knowledge from the first block and you should be able to start working on it immediately. Tasks 2 and 4 additionally require operator overloading (chapter 5) and basic understanding of exceptions (chapter 6). Finally, task 3 needs unique_ptr (chapter 7) and virtual dispatch (chapter 8), but the parser itself (and hence xml_validate) can be implemented with knowledge from block 1 alone, so you can still start early.

Part T.2.1: [complex]

In this exercise, you will implement exact (arbitrary-precision) real and complex numbers. You can use the natural task as a base, if you wish. Both real and complex numbers should provide the standard array of arithmetic operators: addition, subtraction, unary minus, multiplication and division. Real numbers should have all relational operators and complex numbers should have equality (== and !=).
Note: keep your representation normalised – complexity of operations should only depend on the represented number, not on the way it was obtained.

```
// extra files: natural.hpp natural.cpp

class real
{
public:
    explicit real( int v = 0 );
    real abs() const;
    real reciprocal() const;
    real power( int n ) const;
};
```

```
class complex
{
    static inline const real epsilon = real( 1 ) / real( 1000000 );
public:
    explicit complex( real real_part = real( 0 ),
                      real imaginary_part = real( 0 ) );

    real real_part();
    real imaginary_part();

    complex reciprocal() const;
    complex power( int n ) const;
```

Compute the:

- exponential function $exp(z)$,
- the natural logarithm $ln(1+z)$,

where z is this. Use the respective Taylor expansions at 0 (i.e. the Maclaurin series). The number of terms to use is given by terms.

```
    complex exp( int terms ) const;
    complex log1p( int terms ) const;
```

Compute the absolute value of the given complex number to the given precision (the argument prec gives the upper bound on the admissible approximation error). You may find the newton demo from week 2 helpful to compute abs. Don't forget to find a suitable starting point for the approximation, otherwise convergence will be very slow.

```
    real abs( real prec = epsilon ) const;
```

To compute the argument, you will need the inverse tangent (atan), which can be approximated using its Maclaurin series in the closed interval $\langle -1, 1\rangle$. There is only one problem: the convergence near ± 1 is very slow. Hence, you want to use a different series here (discovered by Euler):

$$\sum 2^{2n}(n!)^2 x^{2n+1}/(2n+1)!(1+x^2)^{n+1}$$

Though this one will eventually converge everywhere, it is particularly good in the same $\langle -1, 1 \rangle$ interval. In this interval, it can be truncated at the first term less than half the required precision.
Now note that for any given x, either x or $1/x$ falls into $\langle -1, 1 \rangle$: hence, you can use the reciprocal formula (`atan(1/x)` is `2*atan(1) - atan(x)`) to find an expression for the argument which always falls into the interval of (fast) convergence.
Don't forget that adding two numbers each with error $\leq \varepsilon$ only guarantees that the sum has an error $\leq 2\varepsilon$. Likewise, multiplication by an exact constant also multiplies the error.

```
    real arg( real prec = epsilon ) const;
```

Compute the exponential and `log1p` to a given precision. Assume that z (`this`) is in the area of convergence for the required power series (the open unit disc for `log1p`, the entire complex plane for `exp`).
Tip: to judge the precision, use the norm (square of the modulus), not the modulus itself. For `exp`, depending on the argument, the terms of the power series may grow before they start to shrink. Once they start to shrink and their norm falls below `prec` squared, you have achieved the required precision. How things work out with `log1p` is left as an exercise (it's much simpler).

```
    complex exp( real prec = epsilon ) const;
    complex log1p( real prec = epsilon ) const;
};
```

Part T.2.2: [machine]

In this task, you will implement a simple register machine (i.e. a simple model of a computer). The machine has an arbitrary number of integer registers and byte-addressed memory. Registers are indexed from 1 to `INT_MAX`. Each instruction takes 2 register numbers (indices) and an 'immediate' value (an integral constant). Each register can hold a single value of type `int32_t` (i.e. the size of the machine word is 4 bytes). In memory, words are stored LSB-first. The entire memory and all registers start out as 0.
The machine has the following instructions (whenever `reg_x` is used in the description, it means the register itself (its value or storage location), not its index; the opposite holds in the column `reg_2` which always refers to the register index).

opcode	reg_2	description
mov	≥ 1	copy a value from reg_2 to reg_1
	= 0	set reg_1 to immediate
add	≥ 1	store reg_1 + reg_2 in reg_1
	= 0	add immediate to reg_1
mul	≥ 1	store reg_1 * reg_2 in reg_1
jmp	= 0	jump to the address stored in reg_1
	≥ 1	jump to reg_1 if reg_2 is nonzero
load	≥ 1	copy value from addr. reg_2 into reg_1
stor	≥ 1	copy reg_1 to memory address reg_2
halt	= 0	halt the machine with result reg_1
	≥ 1	same, but only if reg_2 is nonzero

Each instruction is stored in memory as 4 words (addresses increase from left to right). Executing a non-jump instruction increments the program counter by 4 words.

opcode	immediate	reg_1	reg_2

```
enum class opcode { mov, add, mul, jmp, load, stor, hlt };

class machine
{
public:
```

Read and write memory, one word at a time.

```
    int32_t get( int32_t addr ) const;
    void    set( int32_t addr, int32_t v );
```

Start executing the program, starting from address zero. Return the value of `reg_1` given to the `hlt` instruction which halted the computation.

```
    int32_t run();
};
```

Part T.2.3: [natural]

In this task, you will implement a class which represents arbitrary-size natural numbers (including 0). In addition to the methods prescribed below, the class must support the following:

- arithmetic operators `+`, `-`, `*`, `/` and `%` (the last two implementing division and reminder),
- all relational operators,
- bitwise operators `^` (xor), `&` (and) and `|` (or).

The usual preconditions apply (divisors are not 0, the second operand of subtraction is not greater than the first).

```
class natural
{
public:
```

Construct a natural number, optionally from an integer. Throw `std::out_of_range` if `v` is negative.

```
    explicit natural( int v = 0 );

    natural power( natural exponent ) const;
    natural digit_count( natural base ) const;
    natural digit_sum( natural base ) const;
};
```

Part T.2.4: [parser]

Write a parser for simplified JSON: in our version, object keys are barewords (i.e. there is no escaping to deal with) and values are integers, arrays or objects (no strings or floats). The EBNF:

```
(* toplevel elements *)
value   = blank, ( integer | array | object ), blank ;
integer = [ '-' ], digits  | '0' ;
array   = '[', valist, ']' | '[]' ;
object  = '{', kvlist, '}' | '{}' ;

(* compound data *)
valist  = value,  { ',', value } ;
kvlist  = kvpair, { ',', kvpair } ;
kvpair  = blank, key, blank, ':', value ;

(* lexemes *)
digits  = nonzero, { digit } ;
nonzero = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
digit   = '0' | nonzero ;
key     = keychar, { keychar } ;
keychar = ? ASCII upper- or lower-case alphabetical character ? ;
blank   = { ? ASCII space, tab or newline character ? } ;
```

It is okay to use `std::isspace` to implement the `blank` nonterminal. The interface should be as follows:

```
class json_value;
using json_ptr = std::unique_ptr< json_value >;
```

```
using json_ref = const json_value &;

enum class json_type { integer, array, object };

class json_error
{
public:
    const char *what() const;
};

class json_value
{
public:
    virtual json_type type() const = 0;
    virtual int int_value() const = 0;
    virtual json_ref item_at( int ) const = 0;
    virtual json_ref item_at( const std::string & ) const = 0;
    virtual int length() const = 0;
    virtual ~json_value();
};
```

Semantic requirements:

- when a string that is passed to `json_parse` does not conform to the above grammar, the function should throw an exception of type `json_error`,
- a duplicated key within an object should throw `json_error` too,
- the `item_at` overloads should throw an instance of `std::out_of_range` when the specified element does not exist,
- the `item_at` overload which accepts an integer, when called on a value of type `json_type::object`, should return elements in key-alphabetical order,
- `length` should never throw (integers should have a length of 0).

Check that the input document is well-formed (i.e. it conforms to the grammar). Return `true` or `false` depending on the outcome. Do not throw any exceptions.

```
bool json_validate( const std::string & );
```

Read a simplified-JSON document. Throw `json_error` if the document is ill-formed.

```
json_ptr json_parse( const std::string & );
```

Part 9: Templates

You have hopefully already noticed that certain classes in the C++ standard library are parametric: they can be instantiated with different type parameters: that is, we can create an `std::vector` of integers, but we can also create an `std::vector` of floating-point numbers. Even more interestingly, we can create an `std::vector` of instances of our own classes. It is probably quite clear that there is a single entity called `std::vector`: this entity is called a class template. Unfortunately, the terminology gets slightly confusing here: the instances of class templates are classes. This is different from objects which are also known as class instances.
Demonstrations:

1. `zipper` – our favourite data structure, now generic
2. `expr` – a different take on expressions
3. `fold` – function templates
4. `rel` – non-type template arguments

Elementary exercises:

1. `iota` – generate an integer sequence
2. `quot` – quotient fields (aka fields of fractions)
3. `split` – slice a string view into two on a delimiter

Preparatory exercises:

1. `circular` – a circular list, but generic
2. `buffer` – a fixed-size queue-like data structure
3. `stats` – median, quartiles, mode over any container
4. `sparse` – sparse copy-on-write arrays
5. `bbox` – a bounding box of a point collection
6. `visit` – call a function on each node of a graph

Regular exercises:

1. `tfold` – fold a tree using an arbitrary function
2. `tmap` – apply a function to each node
3. `monoid` – free monoids and homomorphisms
4. `treap` – a combination of a heap and a binary search tree
5. `critbit` – binary tries with generic values
6. `finally` – a generic RAII wrapper

Part 9.d: Demonstrations

9.d.1 [zipper] The canonic use for C++ templates is designing container classes (collections) which can hold different types of values. In C, the solution was to either use `void` pointers (e.g. linked lists in PB071), implement data structures using macros (e.g. linked lists in the Linux kernel), or remember store sizes explicitly and copy in data using `void` pointers and the element size (e.g. `glib` data structures). Let's now look at the C++ way, using the familiar `zipper` as an example. You may find it helpful to compare this example with `05/access.cpp`, which implemented a non-generic version of the same class.
Definition of a class template looks the same as a definition of a normal class, all we need to do is specify that we want a template instead, with a single type parameter, which is, by convention, called `T`:

```
template< typename T >
```

We can now proceed to define a `class` and it will become a class template (i.e. it will be parametrized by `T` above). Anywhere in the body of the class where we can specify a type, we can now also use `T`. When we instantiate the template later, the compiler will find all occurrences of `T` in the class definition and replace them with the supplied type parameter. This process is known as substitution.

```
class zipper
{
```

If you remember from a few weeks ago, a zipper can be represented using 2 stacks. Like before, we will use a pair of `std::vector` instances. However, the zipper now stores values of type `T`, so we will supply that as the type parameter of `vector`.

```
    using stack = std::vector< T >;
    stack left, right;
    T focus;

public:
```

Forwarding arguments of arbitrary types is a little too advanced for us, so we will settle for making a copy of the initial value. This means that we require `T` to be a type with a copy constructor. In other words, we won't be able to create zippers that hold values of type `unique_ptr`.

```
    zipper( const T &f ) : focus( f ) {}
```

Like above, we will settle for a copy.

```
    zipper &push_left( const T &x )
```

```
    {
        left.push_back( x );
        return *this;
    }

    zipper &push_right( const T &x )
    {
        right.push_back( x );
        return *this;
    }
```

The shift helper remains unchanged from our previous implementation.

```
    void shift( stack &a, stack &b )
    {
        b.push_back( focus );
        focus = a.back();
        a.pop_back();
    }
```

This time, we will only have pre-increment and pre-decrement, since those are the only practical variants for this class.

```
    zipper &operator++() { shift( right, left ); return *this; }
    zipper &operator--() { shift( left, right ); return *this; }
```

The dereference and indirect access operators are more interesting, since they need to mention the element type, which is now T.

```
    T &operator*()  { return  focus; }
    T *operator->() { return &focus; }
```

And the const overloads of the same:

```
    const T &operator*()  const { return  focus; }
    const T *operator->() const { return &focus; }
```

Indexing operator (non-const overload only).

```
    T &operator[]( int i )
    {
        if ( i == 0 ) return focus;
        if ( i < 0 ) return left[ left.size() + i ];
        if ( i > 0 ) return right[ right.size() - i ];
        assert( false );
    }
};

int main() /* demo */
{
```

Let's first create a zipper which holds integers.

```
    zipper< int > zi( 0 ); // [0]
    zi.push_left( 2 );     // 2 [0]
    zi.push_left( 1 );     // 2 1 [0]
    zi.push_right( 1 );    // 2 1 [0] 1
```

check

```
    assert( zi[ -2 ] == 2 );
    assert( zi[ -1 ] == 1 );
    assert( zi[ 0 ] == 0 );
    assert( zi[ 1 ] == 1 );

    assert( *zi == 0 );
```

And now a different instance, with pairs.

```
    using p = std::pair< int, int >;
    zipper< p > zp( p( 0, 1 ) );

    assert( *zp == p( 0, 1 ) );
    assert( zp->first == 0 );
    assert( zp->second == 1 );

    zp.push_left( p( 7, 7 ) );
    assert( zp[ -1 ] == p( 7, 7 ) );

    -- zp;
    assert( zp->first == 7 );
    assert( zp->second == 7 );
}
```

9.d.2 [expr] We will do another take at expressions, this time with templates, which will allow us to use values instead of references. While the first two examples were directly comparable to earlier versions, this one will deviate quite far from 07/expr.cpp, though you may still find it useful to quickly go over that one first. The semantics will be the same: we will have sums, products and constants. However, there will be no distinction between static and dynamic types and no virtual methods.

We will start with constants, since those are the simplest. Since we are not using virtual methods, we also don't need a common base class or inheritance. We do, however, need to provide a common interface (i.e. method names and signatures) between the different classes.

```
class constant
{
    int _value;
public:
    constant( int v ) : _value( v ) {}
    int value() const { return _value; }
};
```

So far, we have only seen templates with a single type parameter. However, we can have as many as we want (in fact, we can even have a variable number, though that is outside of the scope of this subject). For sum, we will need 2: the type of the left and of the right sub-expression.

```
template< typename left_t, typename right_t >
class sum
{
```

Unlike before, the static type of the left and the right sub-expressions may be different, and instead of references or pointers, we will simply store them by value in attributes.

```
    left_t _left;
    right_t _right;

public:
```

We need to define a constructor. Like in our earlier take on expressions, the constructor will take the two sub-expressions as arguments. This time, their types are given by the template parameters though. Other than that, the constructor is pretty normal.

```
    sum( const left_t &l, const right_t &r )
        : _left( l ), _right( r )
    {}
```

And the interface to get values out of the expression:

```
    int value() const { return _left.value() + _right.value(); }
};
```

The product class looks pretty much the same:

```
template< typename left_t, typename right_t >
class product
{
    left_t _left;
    right_t _right;

public:
```

```
    product( const left_t &l, const right_t &r )
        : _left( l ), _right( r )
    {}

    int value() const { return _left.value() * _right.value(); }
};
```

The duplication is somewhat unsatisfactory. Maybe we could do a little better by using inheritance, so let's try defining another class. First the base class:

```
template< typename left_t, typename right_t >
class operation
{
protected:
    left_t _left;
    right_t _right;

public:
    operation( const left_t &l, const right_t &r )
        : _left( l ), _right( r )
    {}
};
```

Now a derived class – let's do subtraction. Remember that inheritance works with classes, but operation is a class template: we need to instantiate it to obtain a class before we can inherit from it!

```
template< typename left_t, typename right_t >
class difference : public operation< left_t, right_t >
{
public:
    difference( const left_t &l, const right_t &r )
        : operation< left_t, right_t >( l, r ) {}
```

Plot twist: if the type of the base class depends on template parameters, we cannot directly access inherited attributes. Instead, we have to explicitly tell the compiler that those are attributes of this class using this.

```
    int value() const
    {
        return this->_left.value() - this->_right.value();
    }
};
```

That wasn't much better. Templates are, unfortunately, somewhat verbose. On the upside, notice that we have implemented the first two operations (+, *) somewhat differently from the last (-), but they can still interoperate smoothly. Templates use duck typing: if it looks and quacks like a duck (i.e. it has the right attributes and methods) it probably is a duck, and the compiler will let us use the type with the template.

```
int main() /* demo */
{
```

We first define some constants, those are simple.

```
    constant c_0( 0 ), c_1( 1 ), c_2( 2 );
```

When we create instances of class templates using constructors which take arguments of types that match the type parameters of the template, we do not need to explicitly type them out. This is the same principle that lets us write std::pair(0, 1). The feature is called template argument deduction and we will see more of it with function templates in the next unit. Of course, we can specify the template arguments ourselves if we want to, but it gets tedious rather quickly. We will show both styles, first the explicit one:

```
    sum< constant, constant > s_1( c_0, c_1 );
    sum< sum< constant, constant >, constant > s_2( s_1, c_1 );
```

This is clearly impractical. Let's try the implicit style.

```
    sum s_3( s_2, c_1 );
    product p_0( c_0, c_1 );
    product p_1( c_1, s_1 );
```

That is much better. Let's make some differences and then we will check all the values.

```
    difference d_2( s_3, s_1 );
    difference d_0( d_2, c_2 );

    assert( c_0.value() == 0 );
    assert( s_1.value() == 1 );
    assert( s_2.value() == 2 );
    assert( s_3.value() == 3 );
    assert( p_0.value() == 0 );
    assert( p_1.value() == 1 );
    assert( d_2.value() == 2 );
    assert( d_0.value() == 0 );
}
```

9.d.3 [fold] In this unit, we will look at function templates, which are similar to class templates we have seen in previous units. Function templates rely even more heavily on template parameter deduction than class templates: most of the time, calling function templates looks just like calling standard function: the compiler will deduce the type parameters from the actual argument types. We will see that later down.

One further thing to note is that we have actually met function templates quite early on, we just did not mention they were templates: the call operator of a lambda with an auto parameter is, in fact, a function template, the only difference is that the syntax is (usually) less verbose.

We will start by defining some commonly useful folds on containers. Let's start with sum. The container type will be a type parameter: we want our sum to work on different container types (for instance sets and vectors) and also with different element types: the types of items stored in those containers. The syntax for function templates is pretty much the same as it was for class templates:

```
template< typename container_t >
```

followed by a standard function signature. In this case, we have a small problem, since we don't have a name for the type of the return value. For now, we can use auto.

```
auto sum( const container_t &xs )
{
```

There are two ways to go about writing the summing loop, with different trade-offs in terms of types. Probably the most reasonable thing to do is to declare an accumulator of the correct type and initialize it to 0. For that, however, we need to know the type of the values stored in xs. Fortunately, standard library provides us with a way to do just that: each standard container has a nested type name, which we can access using ::. If the outer type is a template argument, or depends on a template argument, we additionally need to tell the compiler that we intend to refer to a type (since templates are duck typed, the nested name could also turn out to be an attribute or a method). The type of a single element stored in a container is known as its value_type.

```
    using value_t = typename container_t::value_type;
```

Now that we have named the type of values in the container, we can declare an accumulator with the correct type. Again, by the virtue of duck typing, we do not know for certain whether values of this type can be constructed from integers, but we assume that they can. When we attempt to use the template, the compiler will check and emit errors if this is, in fact, not possible.

```
    value_t accum = 0;
```

The loop itself is then quite trivial.

```
    for ( const auto &x : xs )
        accum = accum + x;

    return accum;
}
```

Let's also try to do a product, in a slightly different style, just to see some more options. In this case, since we do not make any use of container_t, it would be easier to simply use a lambda. We will do that in a bit.

```
template< typename container_t >
auto product( const container_t &xs )
{
    auto accum = xs.empty() ? 1 : *xs.begin();
    bool first = true;

    for ( const auto &x : xs )
        if ( first )
            first = false;
        else
            accum = accum * x;

    return accum;
}
```

Let us do mean in a lambda style, so we have a comparison at hand. We can re-use sum from above. We will take the average of an empty sequence to be 0.

```
static auto mean = []( const auto &xs )
{
    return xs.empty() ? 0 : sum( xs ) / xs.size();
};
```

Finally, we will generalize the two folds (sum and product), and add a zip_with for a good measure, in the template style. We can accept functions as arguments the same way we accept any other values. This will work with anything that can be called (remember duck typing?), most importantly lambdas.
The initial value of the accumulator passed in by the user gives us the type of the accumulator 'for free'. In practice, this is a little dangerous in the sense that it could give us some unexpected results if enough implicit conversions are allowed (like accumulating rational numbers into an integer). I will show you another version of fold as a bonus after we do zip_with.

```
template< typename xs_t, typename fun_t, typename init_t >
auto fold( const xs_t &xs, const fun_t &f, const init_t &init )
{
```

The fold itself is pretty trivial, once we have figured out the types.

```
    init_t accum = init;
    for ( const auto &x : xs )
        accum = f( accum, x );
    return accum;
}
```

Now for the zip_with. It will accept two sequences and a function. The result will be a vector, since we do not have a good way to tell the function what type of a container we would like.

```
template< typename xs_t, typename ys_t, typename fun_t >
auto zip_with( const xs_t &xs, const ys_t &ys, const fun_t &f )
{
```

We need a new trick, because there is nothing at hand that would give us the element type of the result (even if we settled for a vector as the container). The way to find out is decltype, an operator that takes an expression and produces its type. Whenever we can write out a name of a type, we can instead use decltype with an expression. The expression must only refer to names that are in scope at the point of the decltype though.

```
    using value_t = decltype( f( *xs.begin(), *ys.begin() ) );
```

Note: there is a bit of a danger in the above: this function will not work with an f that returns a reference. Repairing this deficiency is beyond the scope of this course. Ask if you are interested though.

```
    std::vector< value_t > out;
```

We want our zip_with to terminate when it hits the end of the shorter sequence. This means we cannot use std::transform, unfortunately, so we will type out the loop by hand.

```
    auto x = xs.begin();
    auto y = ys.begin();

    while ( x != xs.end() && y != ys.end() )
        out.push_back( f( *x++, *y++ ) );

    return out;
}
```

And finally, the promised 'bonus' fold, which prefers the return type of f as its accumulator type. We have the basic recipe for that in zip_with.

```
template< typename xs_t, typename fun_t, typename init_t >
auto fold_( const xs_t &xs, const fun_t &f, const init_t &init )
{
    using accum_t = decltype( f( init, *xs.begin() ) );
```

Note that accum_t and init_t may be different types.

```
    accum_t accum = init;

    for ( const auto &x : xs )
        accum = f( accum, x );

    return accum;
}
```

For a good measure, we will define a custom class of numbers. You might remember rat from an earlier exercise. The minimum viable definition follows.

```
struct rat
{
    int p, q;
    rat( int p, int q = 1 ) : p( p ), q( q ) {}

    friend rat operator+( rat r, rat s )
    {
        return { r.p * s.q + s.p * r.q, r.q * s.q };
    }

    rat operator*( rat r ) const { return { p * r.p, q * r.q }; }
    rat operator/( rat r ) const { return { p * r.q, q * r.p }; }

    bool operator<( rat r )  const { return p * r.q <  r.p * q; }
    bool operator==( rat r ) const { return p * r.q == r.p * q; }
};

int main() /* demo */
{
    std::set< int > xs{ 1, 2, 3 };
    std::vector< double > ys{ 1.5, 2 };
    std::set< rat > zs{ 1, { 1, 2 }, { 1, 4 } };
```

The only interesting thing in the below test cases is that the functions are used like standard functions: no angle brackets to be seen anywhere. This is because the compiler deduces the type parameters from

the types of the actual arguments which we pass into the function. Since all template arguments can be deduced this way, we can omit angle brackets entirely.

```
    assert( sum( xs ) == 6 );
    assert( sum( ys ) == 3.5 );
    assert( sum( zs ) == rat( 7, 4 ) );

    assert( product( xs ) == 6 );
    assert( product( ys ) == 3 );
    assert( product( zs ) == rat( 1, 8 ) );

    assert( mean( xs ) == 2 );
    assert( mean( ys ) == 1.75 );
    assert( mean( zs ) == rat( 7, 12 ) );
```

When calling our original `fold`, we have to be careful with the type of the initial value, otherwise we will run into problems. This is somewhat inconvenient.

```
    assert( fold( zs, std::plus<>(), rat( 0 ) ) == rat( 7, 4 ) );
```

On the other hand, our improved version (here named `fold_`) works just fine if we write it in a 'natural' style.

```
    assert( fold_( zs, std::plus<>(), 0 ) == rat( 7, 4 ) );
```

Finally, let's look at `zip_with`.

```
    std::vector xs_ys{ 2.5, 4.0 };
```

The sets are sorted in ascending order, so the pairings will be 1/4 + 1, 1/2 + 2 and 1 + 3.

```
    std::vector xs_zs{ rat( 5, 4 ), rat( 5, 2 ), rat( 4 ) };

    assert( zip_with( xs, ys, std::plus<>() ) == xs_ys );
    assert( zip_with( xs, zs, std::plus<>() ) == xs_zs );
}
```

9.d.4 [rel] We will take a second look at function templates, but this time we will also add non-type template arguments to the mix. We haven't used it much, but this is how `std::get` works. It might be useful to review `03/p4_rel.cpp` before diving into this example.
General projections are still too hard for us, so we will only do a single-column projection. However, selection is easier so let's look at those first. We will need 3 template arguments: one to specify which column to use as the selection criterion, another to specify the type of a single row and the last one to specify the type of the value which we will compare with the entries.

```
template< int index, typename rel_t, typename key_t >
rel_t select( const rel_t &rel, const key_t &key )
{
```

Since the type of the relation does not change under selection, it is simple enough to create an empty relation and add matching rows from `rel` to it.

```
    rel_t out;
```

We assume that it is possible to iterate a `rel_t`, and that it is possible to `insert` things into a `rel_t`. Since templates are duck-typed, this will be checked when the template is instantiated.

```
    for ( const auto &row : rel )
```

We now need to decide whether the row matches the criterion: the `index`-th column should be equal to `key` for that, so let's check that.

```
        if ( std::get< index >( row ) == key )
```

Just insert the row.

```
            out.insert( row );
```

And return the result.

```
    return out;
}
```

Now for a single-column projection. Again, we will pass in `index` as a template parameter. However, we will run into some problems with the return type. Fortunately, in the signature, we can just use `auto` as the return type and worry about it later.

```
template< int index, typename rel_t >
auto project( const rel_t &rel )
{
```

Actually, we can't put that off for very long. We need to declare the variable to hold the resulting relation. First of all, we need to find out the type of a single row. For that, we can use standardized nested types that all `std` containers provide, which we have learned about in the previous unit. The row is the `value_type` of the relation, like this. Remember that the `typename` specifier is compulsory whenever we want to refer to a type nested within a template parameter, or within something that depends on a template parameter.

```
    using row_t = typename rel_t::value_type;
```

Now that we have the row type, we need to extract the type of `index`-th column. For that, the standard library provides the `tuple_element_t` helper template, like this:

```
    using col_t = std::tuple_element_t< index, row_t >;
```

Now we have the type that we need to construct the output relation, which we will construct as a set of `col_t`.

```
    std::set< col_t > out;
```

At this point, the code for `project` is just a variation on what we already saw in `select`.

```
    for ( const auto &row : rel )
        out.insert( std::get< index >( row ) );

    return out;
}

int main() /* demo */
{
    using element = std::tuple< std::string, int, double >;
    using elem_rel = std::set< element >;
```

We first define a testing data set.

```
    elem_rel r{ { "hydrogen",  1, 0.78 },
                { "hydrogen",  2, 1.50 },
                { "hydrogen",  3, 3.09 },
                { "helium",    3, 3.09 },
                { "iron",     56, 9.15 },
                { "iron",     58, 9.14 },
                { "nickel",   60, 9.15 },
                { "nickel",   62, 9.15 } };
```

Using `select` is straightforward: we specify, as a template parameter (i.e. using angle brackets) the column index, and pass in the relation and the key as standard arguments. The types of the relation and the key are then deduced automatically. You may find it helpful to compare the calls with the definition of `select` above.

```
    elem_rel nickel = select< 0 >( r, "nickel" ),
             iron   = select< 0 >( r, "iron" ),
             helium = select< 0 >( r, "helium" );
```

```
    elem_rel helium_expect{ { "helium", 3, 3.09 } },
             iron_expect  { { "iron", 56, 9.15 },
                            { "iron", 58, 9.14 } },
             nickel_expect{ { "nickel", 60, 9.15 },
                            { "nickel", 62, 9.15 } };

    assert( helium == helium_expect );
    assert( iron   == iron_expect );
    assert( nickel == nickel_expect );
```

Now for projection: again, we explicitly specify the index of the column to extract, as a template parameter. The type of the relation is deduced and we therefore do not need to mention it.

```
    auto names = project< 0 >( r );
    std::set< std::string > names_expect{ "hydrogen", "helium",
                                          "iron", "nickel" };

    assert( names == names_expect );

    std::set< std::pair< int, int > > p{ { 1, 1 }, { 1, 2 },
                                         { 2, 2 }, { 2, 4 } };

    std::set< int > left{ 1, 2 }, right{ 1, 2, 4 };

    assert( project< 0 >( p ) == left );
    assert( project< 1 >( p ) == right );
    assert( project< 1 >( select< 0 >( p, 1 ) ) == left );
}
```

Part 9.e: Elementary Exercises

9.e.1 [iota] Implement a generic function iota, which, given a function f, calls f(start), f(start + 1), ... f(end - 1), in this order.

```
// void iota( … f, int start, int end );
```

9.e.2 [quot] A quotient field is a generalization of rational numbers: one can be constructed from any integral domain. When the integral domain is taken to be Z (the integers), the result is Q (the rational numbers). However, the construction is much more general and can be applied to polynomials, Gaussian integers, p-adic numbers and so on. Here, we will construct standard rationals and Gaussian rationals (which are like normal rationals, but with an imaginary part).
Define a class template rat. The type parameter will provide the integral domain: int for integers, gauss for Gaussian integers. The constructor should take the numerator and the denominator as arguments. Define addition, multiplication and division on rat's, as well as equality. When done, implement gauss, which is simply a complex number where both the real and imaginary parts are integers. Store them in algebraic form for simplicity. Define addition, multiplication and equality.

9.e.3 [split] Implement a function split, which given a string view s and a delimiter delim, produces a pair of string_views a and b such that:

- delim is not in a,
- and either
  - s == a + delim + b if delim was present in s,
  - s == a and b is empty otherwise

```
using split_view = std::pair< std::string_view, std::string_view >;
split_view split( std::string_view s, char delim );
```

Part 9.p: Preparatory Exercises

9.p.1 [circular] In this exercise, we will implement a circular list again, but this time generically, i.e. using templates. Like before, instead of the usual access operators and iteration, it will have a rotate method, which rotates the entire list. We require that rotation does not invalidate any references to elements in the list.
If you think of the list as a stack, you can think of the rotate operation as taking an element off the top and putting it at the bottom of the stack. It is undefined on an empty list.
To add and remove elements, we will implement push and pop which work in a stack-like manner. Only the top element is accessible, via the top method. This method should allow both read and write access. Finally, we also want to be able to check whether the list is empty. It is okay to make copies in push, but make sure you return references in top.

```
template< typename T >
struct circular_node; /* ref: 8 lines */

template< typename >
class circular; /* ref: 34 lines */
```

9.p.2 [buffer] We will implement another data structure. We have not demonstrated the use of non-type template parameters with class template, but the principle is the same as it was in function templates in d4_rel.cpp. An additional hint: the size of an std::array is a non-type template argument (of type size_t).
Implement a bounded circular buffer with a fixed size, as a class template with a single type argument T (which comes first) and a single non-type argument size of type size_t (which comes second). The class should be default-constructible and it can assume that T is also default-constructible and that it can be copied. The circular buffer should provide the following methods:

- push inserts a value at one 'end'
- pop removes and returns a value from the other 'end'
- empty which returns true if there are no items
- full if there are already size items

Calling push on a full and pop on an empty buffer is undefined. Pushing new items should wrap around the end of the storage and start re-using storage from the start, as long as pop has been called in the meantime (i.e. the buffer is not full). In other words, buffer with push and pop behave like a FIFO queue which can hold at most size elements.

```
template< typename T, size_t size >
class buffer; /* ref: 23 lines */
```

9.p.3 [stats] In this exercise, we will do some basic statistics: median, quartiles and mode.
Implement the functions mode, median and quartiles, in such a way that it accepts any sequential std container, with element type that allows less-than and equality comparison. Additional notes:

- mode returns an std::set of numbers, since there might be arbitrarily many: include any input number for which the number of occurrences is maximal
- median return a single number; pick the smaller of the two elements if the median lies in between two different numbers
- quartiles returns numbers at indices (size / 4) and ((3 * size) / 4) of the sorted input sequence, in an std::pair [this is slightly incorrect but simpler].

```
// mode:      ref. 15 lines
// median:    ref.  7 lines
// quartiles: ref.  8 lines
```

9.p.4 [sparse] Imagine there is a large array of data (e.g. numbers), but we sometimes need to change a few of those. However, we also need to keep the original data intact, and we don't want to copy all the data around. In this exercise, we will design a simple solution to this problem.
Implement class template sparse, with a type argument T and a size_t argument N, with the following interface:

- construct from an `std::array` of `T` with size `N`,
- construct a copy (the `array` given to the constructor is shared by all copies),
- `set( i, v )` replaces the value stored at index `i` with `v` (without affecting the backing array),
- `get( i )` returns the value at index `i` (that is the `v` passed to latest call to `set( i, v )` with the same `i`, or the value from the backing array if `set` was never called for `i`,
- `merge()` that propagates the changes made in this instance into all other instances sharing the same backing array.

Note: the memory complexity should be O(N) of shared data, and O(M) per instance of `sparse` where M is the number of altered indices. A `merge` on one copy should not affect altered indices in other copies.

```
template< typename T, size_t N >
class sparse;
```

9.p.5 [bbox] We will dust off geometry a little bit: we will look at constructing a bounding box around a sequence of points (this time in 3D).
Points can be constructed from three floating-point numbers (of type `double`.

```
struct point;
```

There is a `dist` function which gives the Euclidean distance of two points.

```
double dist( point a, point b );
```

A helper function to check approximate point equality.

```
bool close( point a, point b ) { return dist( a, b ) < 1e-10; }
```

Now for the bounding box: we want an axis-aligned box (i.e. not the smallest one), and will represent it using 2 points – those in the opposite corners. Some of the resulting dimensions might be 0 (in case the points all lie on a line or in a plane). Return the points in such a way that the coordinates of the first one are smaller along all axes. It should be possible to pass the points using a `const` reference to any container which can be iterated.

```
using box_t = std::pair< point, point >;
// ... box_t box( ... )
```

9.p.6 [visit] The input graph is given using adjacency lists: the `graph` type gives the successors for each vertex present in the graph.

```
template< typename vertex_t >
using graph = std::map< vertex_t, std::vector< vertex_t > >;
```

Visit each vertex of graph `g` reachable from `initial` once and call `f` on its value. The order of calls is not important.

```
// void visit( … g, … f, … initial );
```

Part 9.r: Regular Exercises

9.r.1 [tfold] Fold a proper binary tree using an associative and commutative binary function (proper meaning that each node either has both children, or none).

```
template< typename value_t >
struct tree
{
    std::unique_ptr< tree > left, right;
    value_t value;

    static auto make_tree( const tree &t )
    {
        return std::make_unique< tree >( t );
    }

    tree( const tree &t )
        : left(  t.left  ? make_tree( *t.left )  : nullptr ),
          right( t.right ? make_tree( *t.right ) : nullptr ),
          value( t.value )
    {}

    tree( value_t value, const tree &l, const tree &r )
        : left( make_tree( l ) ),
          right( make_tree( r ) ),
          value( std::move( value ) )
    {}

    tree( value_t value ) : value( std::move( value ) ) {}
};
```

Given a binary function `f` and a tree instance `t`, compute a single value that is the result of folding the entire tree. Since `f` is both associative and commutative, it does not matter in which order you combine the individual values.

```
// … tfold( … f, … t )
```

9.r.2 [tmap] Goal: build a tree by preserving the structure of an existing tree, but obtain new values by applying a given function to the originals. The type of the value may change. Hint: assuming

- `function_t fun` is a unary function,
- `value_t val` is a value such that `fun( val )` makes sense,

you can use:

- `std::invoke_result_t< function_t, value_t >` to obtain the type of `fun( val )` without having either `fun` or `val` (only their types).

```
template< typename value_t >
struct tree
{
    value_t value;
    std::vector< tree > children;

    tree( value_t v, std::vector< tree > ch = {} )
        : value( std::move( v ) ), children( std::move( ch ) )
    {}
};
```

Build a tree of a suitable type given a function `f` which maps values to values and some tree `t`, compatible with `f`.

```
// … tmap( … f, … t )
```

9.r.3 [monoid] Monoids are algebraic structures with a single operation, usually written as multiplication. A free monoid M on a set A is defined as the set of all strings of elements from A with concatenation as the operation. A monoid homomorphism is a map f from M to M' with the property $f(a \cdot b) = f(a) \cdot f(b)$. All monoids arise as homomorphic images of a free monoid on some set.
Define a class template `monoid`, which takes a single type argument, `hom_t`, with the following behaviour:

- the constructor will then accept `hom`, a value (typically a lambda) of type `hom_t`, as an argument and store it in an attribute (by value),
- method `elem`, which takes an `std::string` and returns a value of a suitable type with a multiplication and an equality operator.

The class should work with a fixed underlying set: the minuscule Latin letters (i.e. 'a' through 'z') and use the mechanics of a free monoid to implement multiplication. The provided `hom` will take an `std::string` as an argument, and return a value of some arbitrary type. Assume applying `hom` to a string yields values which can be compared, but

not multiplied (at least not in a way compatible with the provided homomorphism).

```
template< typename hom_t >
class monoid_element; /* ref: 11 lines */

template< typename hom_t >
class monoid; /* ref: 8 lines */
```

Part 10: Templates (cont'd)

This week, we will practice templates some more, and introduce a few more useful template constructs. Among other things, we'll look at more complicated cases of template argument deduction and how function overloading interacts with function templates. We will also look at templated operator overloads.
Demonstrations:

1. `apply` – call functions on scalars in composite data
2. `method` – method templates
3. `expr` – expressions, this time with operator overloading
4. `set` – operators on sets (with arbitrary element types)
5. `call` – overloading the call operator

Elementary exercises:

1. `format` – format collections
2. `concat` – splice two sequences into one
3. `icons` – integer lists with compile-time recursion

Preparatory exercises:

1. `post` – post-order on a generic graph
2. `cons` – heterogeneous lists
3. `map` – more template argument deduction
4. `collect` – extract values from containers
5. `list` – linked lists with sub-list sharing
6. `vecset` – implement a set using a sorted vector

Regular exercises:

1. `select` – create a vector of variants
2. `sorted` – a stateful sequence observer
3. `fsm` – generic finite state machines
4. `tree` – trees with sub-tree sharing
5. `bimap` – map 2 types of keys to each other
6. `tinyvec` – a generic vector in fixed memory

Part 10.d: Demonstrations

10.d.1 **[apply]** In this example, we will show how to use recursion together with templates to walk through composite, templated data types. In particular, we will look at finding and summing up numeric data types in a binary tree made of `std::tuple` instances.
We will need a data type to stop the recursive data definitions: an empty tree, if you will. We will call it `null` (not to be confused with the C macro `NULL` nor with C++ `nullptr`. We want this to be a unique data type, but it does not need to carry any actual data, hence we can use an empty `struct`.

```
struct null {};
```

The summation will be defined recursively, so let's first define the overload for the base type: `null`. The neutral element of addition is 0, so let's use 0 as the value of an empty subtree.

```
int sum( null ) { return 0; }
```

Now for the non-empty subtrees: we will use 3-tuples: the value in the node (integer) and the left and right subtrees. We will use template argument deduction to obtain the type of the composite tuple. Recall that we used to write function templates somewhat like this:

```
template< typename T >
int sum( const T &tup )
{
    int v = std::get< 0 >( tup );
    // ...
}
```

This is not optimal, because there is a conflict with the `null` overload above: the template can be instantiated with `T = null`. The compiler will prefer the non-template version (or rather the most specific version), but the rules are complex and error-prone. It is better to not rely on those rules if they can be easily avoided.
In this case, we can use a more specific (non-overlapping) overload, which will only accept 3-tuples. There is no chance that a `null` is confused for a 3-tuple. Nonetheless, we still need to figure out how to do template argument deduction in this case. Easier shown than described. We will use 2 template type parameters, for the left and right subtree.

```
template< typename L, typename R >
```

However, we cannot directly use L and R as function arguments: we want to accept 3-tuples. Fortunately, the compiler can also deduce parts of an argument type:

```
int sum( const std::tuple< int, L, R > &tup )
{
```

We can also use structured bindings to decompose the tuple, making the code easier to read:

```
    const auto &[ v, left, right ] = tup;
```

The rest of the function now looks like the most straightforward recursive definition from IB015.

```
    return v + sum( left ) + sum( right );
}

int main() /* demo */
{
    std::tuple a{ 3, null(), null() };
    std::tuple b{ 7, null(), null() };
    std::tuple c{ 1, null(), null() };
    std::tuple d{ 10, a, null() };
    std::tuple e{ 2, b, c };
    std::tuple f{ 0, d, e };

    assert( sum( null() ) == 0 );
    assert( sum( a ) == 3 );
    assert( sum( b ) == 7 );
    assert( sum( c ) == 1 );
    assert( sum( d ) == 13 );
    assert( sum( e ) == 10 );
    assert( sum( f ) == 23 );
}
```

10.d.2 **[method]** We already know that we can write class templates and function templates. It is only logical that we can also create method templates in C++ classes and in class templates.
This example will be somewhat synthetic: we will have a class which

does not permit direct access to its elements, but allows them to be folded using a function object.

```
template< typename T >
class foldable
{
    std::vector< T > data;
public:
```

A method to add elements to the internal container.

```
    void push( const T &t ) { data.push_back( t ); }
```

And the method template to accumulate the content using a function object.

```
    template< typename fun_t, typename init_t >
    init_t fold( const fun_t &fun, init_t init )
    {
        for ( const auto &e : data )
            init = fun( init, e );
        return init;
    }
};

int main() /* demo */
{
    foldable< int > f;
    f.push( 7 );
    f.push( 3 );
    assert( f.fold( std::plus<>(), 0 ) == 10 );
    f.push( 10 );
    assert( f.fold( std::multiplies<>(), 1 ) == 210 );
}
```

10.d.3 [expr] This example will demonstrate operator overloading in conjunction with class templates. Again, we will use argument deduction with partial types in function signatures, to match the desired types closely enough to to avoid ambiguities. You can probably imagine than an operator + that accept arbitrary types as arguments would not mesh very well with the rest of the program.
First, we will define an enum to tag expressions with the operator they represent, and a constant class to use as leaf nodes.

```
enum class expr_op { add, mul };
```

Constants will be a straightforward class with an eval method, common with the expr class template below:

```
struct constant
{
    int v;
    int eval() const { return v; }
    constant( int v ) : v( v ) {}
};
```

We will start by defining an expr class template.

```
template< typename left_t, typename right_t >
struct expr
{
    expr_op op;
    left_t left;
    right_t right;

    expr( expr_op op, const left_t &l, const right_t &r )
        : op( op ), left( l ), right( r )
    {}
```

Compute the value of this node.

```
    int eval() const
    {
        int l = left.eval(),
            r = right.eval();

        switch ( op )
        {
            case expr_op::add: return l + r;
            case expr_op::mul: return l * r;
        }

        assert( false );
    }
```

Like with normal operator overloading, there are multiple ways to overload operators for class templates. Let's start by defining a method. However, we immediately run into a problem: the right operand does not have to be of the same type as the left one, even though we want it to be an instance of the same class template. For that reason, we need to define the operator as a template method.

```
    template< typename l2_t, typename r2_t >
```

The return type is mildly infuriating, because it needs to spell out the composite instance. Next time, we will just use auto.

```
    expr< expr< left_t, right_t >, expr< l2_t, r2_t > >
```

Now for the signature of the operator itself:

```
    operator+( const expr< l2_t, r2_t > &o ) const
    {
```

Now that we have spelled out the monster types, the implementation is trivial.

```
        return { expr_op::add, *this, o };
    }
```

Now let's try a friend definition. The gist is the same, and you may remember that we can still use expr without arguments to mean the instance with left_t = left_t and right_t = right_t. Then:

```
    template< typename l2_t, typename r2_t >
    friend auto operator*( const expr &a,
                           const expr< l2_t, r2_t > & b )
    {
```

But now we have a problem again. We are in the definition of a class template, and hence using the name of the class template without arguments means this specific instance. But we want to construct a different instance, but using template argument deduction. We need to tell the compiler that is what we mean by using a qualified name for the class template: if qualified, the name no longer refers to this instance.

```
        return ::expr( expr_op::mul, a, b );
    }
```

That covers the expression + expression cases. But we also need to be able to work with constant instances here. More operators!

```
    friend auto operator+( constant c, const expr &a )
    {
        return ::expr( expr_op::add, c, a );
    }

    friend auto operator+( const expr &a, constant c )
    {
        return ::expr( expr_op::add, a, c );
    }

    friend auto operator*( constant c, const expr &a )
    {
        return ::expr{ expr_op::mul, c, a };
    }
```

```
    friend auto operator*( const expr &a, constant c )
    {
        return ::expr( expr_op::mul, a, c );
    }
};
```

That's not the end yet. We also need to be able to multiply and add two constants, to get a complete set. Since the result is an `expr` instance, it does not make much sense to put those into the `constant` class itself. Let's use toplevel definitions for those. Fortunately, in this case, the operators are not templates at least.

```
auto operator+( constant a, constant b )
{
    return expr( expr_op::add, a, b );
}

auto operator*( constant a, constant b )
{
    return expr( expr_op::mul, a, b );
}

int main() /* demo */
{
    constant a( 1 );
    constant b( 2 );
    auto c = a + b;
    assert( c.eval() == 3 );
    assert( ( a * c ).eval() == 3 );
    assert( ( a + c ).eval() == 4 );
    assert( ( c + c ).eval() == 6 );
}
```

10.d.4 [set] In which we will combine operator templates and template argument deduction to spice up the standard `std::set` container.
We have already seen in `apply.cpp` that the compiler can deduce template arguments based on (self-contained) fragments of function argument types. We will use that along with operator templates to provide overloads for all instances of `std::set`, without affecting any other standard container, or any other type at all.
We will overload operator `&` for intersection, operator `|` for union, `-` for standard difference and finally `^` for symmetric difference of two sets. Please keep in mind that the priorities of bitwise operators in C++ are unintuitive at best: overloaded operators inherit both their priority and associativity from the built-in ones.

```
template< typename T >
std::set< T > operator&( const std::set< T > &a,
                         const std::set< T > &b )
{
    std::set< T > out;
```

Remember standard algorithms?

```
    std::set_intersection( a.begin(), a.end(),
                           b.begin(), b.end(),
                           std::inserter( out, out.begin() ) );
    return out;
}
```

Now the union.

```
template< typename T >
std::set< T > operator|( const std::set< T > &a,
                         const std::set< T > &b )
{
    std::set< T > out;

    std::set_union( a.begin(), a.end(),
                    b.begin(), b.end(),
                    std::inserter( out, out.begin() ) );
    return out;
}
```

And difference. This is getting a little boring.

```
template< typename T >
std::set< T > operator-( const std::set< T > &a,
                         const std::set< T > &b )
{
    std::set< T > out;

    std::set_difference( a.begin(), a.end(),
                         b.begin(), b.end(),
                         std::inserter( out, out.begin() ) );
    return out;
}
```

And finally the symmetric difference. Surprise!

```
template< typename T >
std::set< T > operator^( const std::set< T > &a,
                         const std::set< T > &b )
{
    return ( a | b ) - ( a & b );
}

int main() /* demo */
{
    std::set a{ 1, 2, 3 };
    std::set b{ 1, 3, 5 };

    std::set aib{ 1, 3 };
    std::set aub{ 1, 2, 3, 5 };
    std::set amb{ 2 };
    std::set axb{ 2, 5 };

    assert( ( a & b ) == aib );
    assert( ( a | b ) == aub );
    assert( ( a - b ) == amb );
    assert( ( a ^ b ) == axb );

    assert( ( a & b ) == ( b & a ) );
    assert( ( a | b ) == ( b | a ) );
    assert( ( a - b ) != ( b - a ) );
    assert( ( a ^ b ) == ( b ^ a ) );
}
```

10.d.5 [call] The final example will deal with the function call operator, also known as `operator()`. This will allow us to construct objects which can be called, just like lambdas. However, there is one thing that lambdas can't do very well in C++, and that is provide multiple overloads. Likewise, standard top-level functions cannot be easily passed as arguments to other functions, since overload sets are not first-class in C++: instead, we have to wrap up the overload set in a callable object. We will see that in a minute.
We will re-use the same construction that we have seen in `apply.cpp`, but we will allow different value types to appear in the tree, instead of just integers. We will again define `null` as the empty tree:

```
struct null {};
```

Even though the `struct` above is empty, we need to define equality if we want to use it. Since it will be rather useful in writing tests later, we will define the (trivial) equality operator here:

```
bool operator==( null, null ) { return true; }
```

Like other functions defined on recursive data types, we first need to define the base case for `map`, i.e. the case when the subtree is empty:

```
template< typename fun_t >
auto map( null, const fun_t & )
{
    return null();
```

```
}
```

And instead of `sum`, we will have a generic mapping operator:

```
template< typename val_t, typename left_t, typename right_t,
          typename fun_t >
auto map( const std::tuple< val_t, left_t, right_t > &tuple,
          const fun_t &fun )
{
    const auto &[ val, left, right ] = tuple;
    return std::tuple{ fun( val ),
                       map( left, fun ), map( right, fun ) };
}
```

The call operator uses syntax very similar to the indexing operator which we have seen before, just with parentheses instead of square brackets. Since `()` is the name of the operator, the arguments need to come in another pair of parens. Please keep that in mind!

```
struct to_string
{
    std::string operator()( int i ) const
    {
        return std::to_string( i );
    }

    std::string operator()( const std::string &s ) const
    {
        return s;
    }
};
```

A small thing that we have not seen yet. Normally, if we want to construct an object, we need to call its constructor, e.g. `std::string( "hello" )`. Sometimes, this is quite tedious, like in this example. C++ offers a feature called user-defined literals, where we are allowed to overload certain operators which then make it possible to create object instances using literal syntax. We will not get into the details of creating such user-defined literals, but we will use the one that the standard library provides for constructing `std::string` instances. To use them, we need to use the following declaration first, to get the literal operators into scope:

```
using namespace std::literals;

int main() /* demo */
{
```

After the `using namespace` above, we can say "hello"s to mean `std::string( "hello" )`. Saves us a bit of typing.

```
    std::tuple a{ "hello"s, null(), null() };
    std::tuple b{ 7, null(), null() };
    std::tuple c{ "x"s, a, b };

    std::tuple b_str{ "7"s, null(), null() };
    std::tuple c_str{ "x"s, a, b_str };

    assert( map( a, to_string() ) == a );
    assert( map( b, to_string() ) == b_str );
    assert( map( c, to_string() ) == c_str );
}
```

Part 10.e: Elementary Exercises

10.e.1 [format] In this exercise, we will practice writing functions with more complex argument deduction. The functions in question will use `std::ostringstream` to produce string representation of sets and vectors.
Use a comma-separated format for `std::vector` instances, with arbitrary element type, then do the same for `std::set`. Vectors should use square brackets `[]` and sets should use curly braces `{}` as delimiters. Assume the `value_type` stored in the vector has appropriate `std::ostream` operators. The functions should be called `format`.

10.e.2 [concat] Write a function which takes two sequences, `a` and `b`, and produces a single vector with values from the two sequences (first all values from `a`, then all values from `b`, preserving their order). Assume each sequence has a nested typedef `value_type`. The sequences do not need to be of the same type, but their values must be compatible.

```
// … concat( … a, … b )
```

10.e.3 [select] Write a function which returns a vector of variants, such that the i-th position is taken from input `a` iff `which[ i ]` is true and from input `b` otherwise. Both `a` and `b` must have at least `which.size()` elements. Elements beyond this size are ignored. Both `a` and `b` are sequences with a `value_type` nested typedef.

```
// … select( … a, … b, const std::vector< bool > &which );
```

Part 10.p: Preparatory Exercises

10.p.1 [post] The goal of this exercise is simple: take an oriented graph as the input and produce a list (vector) of vertices in the 'leftmost' DFS post-order. That is, visit the successors of a vertex in order, starting from leftmost (different exploration order will result in different post-orders). The graph is encoded as a neighbourhood list.

```
template< typename value_t >
using graph = std::map< value_t, std::vector< value_t > >;
```

Construct the post-order of `g` starting from vertex `i`.

```
// … dfs_post( … g , … i );
```

10.p.2 [cons] We will elaborate a little on the topic from `icons.cpp`, by making the type of `car` into a template argument. That way, we will be able to make a list that has items of different types in it.
Generalize `cons` from the previous exercise and write a `reduce` function that takes an arbitrary `cons` instance, a function object (e.g. a lambda) and an initial accumulator value. The function object must be able to accept the accumulator as its first argument, and an arbitrary value that appears in the `cons` list as its second argument.
null, cons, reduce
callable object with overloads, for testing

```
struct collect
{
    using pair = std::pair< int, double >;

    pair operator()( pair p, int i ) const
    {
        p.first += i;
        return p;
    }

    pair operator()( pair p, double d ) const
    {
        p.second += d;
        return p;
    }
};
```

10.p.3 [map] Write `keyset`, a function which takes an instance of `std::map` and returns an `std::set` with the keys that were present in the input `map`.
And `intersect`, which takes an `std::set` of keys and an `std::map` (with the same key type and arbitrary value type) and produces another `std::map` which only retains the key/value pairs for which keys were

present in the input key set.

10.p.4 [collect] The goal of this exercise is to write a set of overloads that will, together, allow the user to extract values from standard containers, into a vector. For `std::map` and `std::unordered_map`, this means the value without the key. For other container types, the functions simply copy the contents into an output vector.

10.p.5 [list] In this exercise, we will define a `list` class that behaves like the lists in functional programming: the values and structure will be immutable, but it'll be possible to fairly cheaply create new lists by prepending values to existing lists.
Define a class template `list` with a singe type parameter `T`, with the following interface (all methods are `const`):

- `head` returns the value of the current list
- `tail` returns the remainder of the list as another `list` instance, without copying any values
- `empty` which returns `true` if the list is empty
- a default constructor which creates an empty list (i.e. the `[]` constructor you know from Haskell)
- a 2-parameter constructor which takes the value and the tail of the list (i.e. the `(:)` constructor).

Hint: It is preferable to store the values inline in the nodes. You should also use 2 data types, one for the list itself and another for nodes: this will make it easier to implement empty lists and in general make the implementation nicer.

```
template< typename T >
class list; /* ref: 24 lines */
```

10.p.6 [tree] In this exercise, we will implement an immutable binary tree, similar to the list we saw earlier.
Implement class template `tree` with a single type argument `T` and the following interface (all method are `const`):

- the default constructor creates an empty tree,
- a 3-parameter constructor creates a tree with the value given in the first argument as root and the next two arguments specify the left and right subtrees,
- a 1-parameter constructor creates a leaf node which stores the value given in the argument,
- `empty` returns true if the tree is empty,
- `root` returns a reference to the value in the root,
- `left` returns the left subtree,
- `right` returns the right subtree.

Hint: two of the constructors can be merged using default arguments.

```
template< typename T >
class tree;
```

Part 10.r: Regular Exercises

10.r.1 [icons] In this exercise, your goal will be to define a list-like structure using templates, and then recursively sum numbers stored in it. You will need to use both templates and function overloading.
In LISP-like languages, lists are built out of so-called cons cells (cons being short for constructor). Each cell contains a value and a pointer to the next cell. The value is traditionally called `car` and the pointer to the next cell is called `cdr`. The `cdr` may also be `null`, i.e. an empty list. In our case, the `car` will always be of type `int`. The `cdr` will be given by a template parameter, in the expectation that it is another `cons` instance or `null`.

```
struct null {}; /* empty */

template< typename cdr_t >
struct cons;
```

Overloads and/or templates of `sum` go here.

10.r.2 [sorted] Write a function object that will decide, when passed to `foreach`, whether the iterated sequence was sorted in ascending order. The result is obtained by calling `was_sorted` on the object after the iteration ends.
It might be useful to know that `std::any` can hold a value of any type. Use normal assignment to store a value in an `any` instance and `std::any_cast` to read the value back.

```
struct check_sorted;
```

Unlike `std::foreach`, we take the function by reference, which makes it possible to inspect its state after iteration ends.

```
template< typename iter_t, typename fun_t >
void foreach( iter_t b, iter_t e, fun_t &&f )
{
    while ( b != e )
        f( *b++ );
}
```

10.r.3 [fsm] Everyone's favourite: deterministic finite state machines. We will write a class template that will let us decide whether a generalized word (a sequence of values of a type equipped with equality) belongs to a regular language described by a given finite automaton (finite state machine) or not. The type which represents individual letters is given to `fsm` as a type parameter.
The constructor of `fsm` should accept a single boolean (mark the constructor `explicit`), which determines if the state represented by the instance is accepting or not. A default-constructed `fsm` should be non-accepting. Make the following methods available:

- `next` which accepts a letter (of type `letter_t`) and a `const` reference to another `fsm` instance; the method adds a transition to the automaton,
- `accept` which takes a sequence of `letter_t` values (the type of the sequence is not fixed, but it can be iterated using a range `for`) and returns `true` if the automaton accepts the word stored in the sequence; this method should be marked `const`.

```
// … class fsm;
```

Part 11: Iterators

XXX
Demonstrations:

1. `queue` – an iterable queue
2. `split` – chop up a `string_view` into pieces
3. `glob` – iterate over matches of a pattern in a string

Elementary exercises:

1. `iota` – an iterable sequence of integers
2. `view` – iterate a slice of an existing collection
3. `skip` – iterate every n-th item (a stride) of a collection

Preparatory exercises:

1. `seq` – generic sequences
2. `filter` – filtered sequences
3. `zip` – iterate two sequences in lockstep
4. `nibble` – a fixed-size nibble array

5. `tree` – in-order iteration of a tree
6. `scan` – generalized prefix sum

Regular exercises:

1. `map` – applying a function to a sequence
2. `range` – views with a shared backing store
3. `permute` – iterate all permutations of a sequence
4. `critbit` – iterate a 'critbit' binary trie in order
5. `matrix` – iterate a compact rectangular array
6. `bits` – iterate bits in a word

Part 11.d: Demonstrations

11.d.1 [queue] We will now implement a data structure 'from scratch' (i.e. without using `std` containers) using templates. Again, you may find it useful to go back to `06/queue.cpp` and compare the two implementations.
Like before, since we are going for a custom, node-based structure, we will need to first define the class to represent the nodes. Unlike the previous implementation, however, the node itself needs to be parametrized by the type of the value it should hold.

```
template< typename T >
struct queue_node
{
```

You may have noticed this with the `zipper` earlier: we do not need to mention the type parameter when we want to refer to the instance of `queue_node` where `T` is `T` (though we can if we want to). In other words, within `queue_node`, saying `queue_node` when referring to a type means the same thing as `queue_node< T >`.

```
    std::unique_ptr< queue_node > next;
```

The data attribute will be of type `T`.

```
    T value;
};
```

Like the node, the iterator also needs to be parametric.

```
template< typename T >
struct queue_iterator
{
```

Here, we have no choice but to explicitly spell out the type parameter of `queue_node`, since we are no longer within that class.

```
    queue_node< T > *node;
```

Constructor names are unaffected by templates.

```
    queue_iterator( queue_node< T > *n ) : node( n ) {}
```

The pre-increment operator simply shifts the pointer to the `next` pointer of the currently active node. This method is unchanged from the non-generic version.

```
    queue_iterator &operator++()
    {
        node = node->next.get();
        return *this;
    }
```

The implicit 'current instance of the template' shortcut works in arguments too, including in arguments of `friend` functions, so let's demonstrate that:

```
    friend bool operator!=( const queue_iterator &a,
                            const queue_iterator &b )
    {
        return a.node != b.node;
    }
```

Finally the dereference operator. Unlike before, we don't know much about `T`, hence we prefer to always return a reference, even in the `const` overload.

```
          T &operator*()       { return node->value; }
    const T &operator*() const { return node->value; }
};
```

The `queue` itself will be a template too, of course.

```
template< typename T >
class queue
{
```

Like in the iterator, we need to instantiate any template classes that we use that were defined earlier. That is the only difference compared to our earlier `queue` implementation.

```
    std::unique_ptr< queue_node< T > > first;
    queue_node< T > *last = nullptr;
public:
```

In the integer-only version, we passed the argument by value, but like in the dereference operator above, we will now instead use a `const` reference: `T` might be a big class with an expensive copy operation. We do not want to do that twice.

```
    void push( const T &v )
    {
        if ( last ) /* non-empty list */
        {
```

Notice the `T` in the `make_unique` call.

```
            last->next = std::make_unique< queue_node< T > >();
            last = last->next.get();
        }
        else /* empty list */
        {
            first = std::make_unique< queue_node< T > >();
            last = first.get();
        }

        last->value = v;
    }
```

Now we run into a bit of a problem. Since making copies of `T` is possibly expensive, we would like to return a reference: but we cannot, since `pop` will destroy the node which stores the value. Incidentally, this is the reason why `std::queue::pop` is a void function and you need to use a separate `front` call to get the value. We will simply return by value instead, which can be less efficient, but not terribly so. We can reduce the cost by using `std::move` on the value, since the node is going to be destroyed anyway.

```
    T pop()
    {
        T v = std::move( first->value );
        first = std::move( first->next );
```

Do not forget to update the `last` pointer in case we popped the last item.

```
        if ( !first ) last = nullptr;
        return v;
    }
```

The emptiness check is simple enough.

```
    bool empty() const { return !last; }
```

Same as before, but we need to instantiate the queue_iterator template.

```
    queue_iterator< T > begin() { return { first.get() }; }
    queue_iterator< T > end()   { return { nullptr }; }
```

Same.

```
    void erase_after( queue_iterator< T > i )
    {
        assert( i.node->next );
        i.node->next = std::move( i.node->next->next );
    }
};

int main() /* demo */
{
```

We start by constructing an (empty) queue and doing some basic operations on it. We start by inserting and removing a single element.

```
    queue< std::pair< long, long > > q;

    assert( q.empty() );
    q.push( { 7, 0 } );
    assert( !q.empty() );
    assert( q.pop() == std::pair( 7l, 0l ) );
    assert( q.empty() );
```

Now that we have emptied the queue again, we add a few more items and try erasing one and iterating over the rest.

```
    q.push( { 1, 0 } );
    q.push( { 2, 0 } );
    q.push( { 7, 0 } );
    q.push( { 3, 0 } );
```

We check that erase works as expected.

```
    queue_iterator i = q.begin();
    ++ i;
    assert( *i == std::pair( 2l, 0l ) );
    q.erase_after( i );
```

We can still use instances of queue in range for loops, because they have begin and end, and the types those methods return (i.e. iterators) have dereference, inequality and pre-increment. Since our current instance of queue contains pairs, we can also use structured bindings in the for loop.

```
    int x = 1;

    for ( auto [ v, w ] : q )
    {
        assert( v == x++ );
        assert( w == 0 );
    }
```

That went rather well, let's just check that the order of removal is the same as the order of insertion (first in, first out). This is how queues should behave.

```
    assert( q.pop() == std::pair( 1l, 0l ) );
    assert( q.pop() == std::pair( 2l, 0l ) );
    assert( q.pop() == std::pair( 3l, 0l ) );
    assert( q.empty() );
}
```

11.d.2 [split] TBD

11.d.3 [glob] TBD

Part 11.e: Elementary Exercises

11.e.1 [iota] Write a class iota, which can be iterated using a range for to yield a sequence of numbers in the range start, end - 1 passed to the constructor.

```
class iota;
```

11.e.2 [view] Write a class template view which will allow us to bundle a pair of iterators of an existing sequence and use them as a virtual sequence in its own right. It should be possible to change the underlying sequence through the view. The constructor should accept two iterators, start and end, and iterating the resulting sequence should include everything in this range (end is excluded, as is customary).

```
// … class view;
```

11.e.3 [skip] Write a class template skip which will be like a view, but allow us to iterate every n-th item (a stride) of a given iterator range, instead of every item. Make sure that if the stride does not evenly divide the sequence length, iteration still works correctly.
Hint: please note that this class is significantly more complicated than view was. You might find decltype(auto) useful, particularly as the return type of a function.

Part 11.p: Preparatory Exercises

11.p.1 [seq] In this exercise, the goal will be to implement a class template which will allow us to iterate over a sequence of number-like objects.
The seq class should be constructible from 2 number-like objects: the initial value (included) and the final value (excluded). Use pre-increment (operator ++) and equality (operator ==) to generate the values. The dereference operator should return the generated objects by value.

```
template< typename T >
class seq_iterator; /* ref: 10 lines */

template< typename T >
class seq; /* ref: 9 lines */
```

A nat implementation for testing purposes.

```
struct nat
{
    int v;
    nat( int v ) : v( v ) {}
    bool operator==( nat o ) const { return v == o.v; }
    nat &operator++() { ++v; return *this; }
};
```

11.p.2 [filter] Lazy sequences, part two.
Define a class template, filter, which holds two items:

- a reference to an arbitrary container,
- a lambda f (of type a → bool).

The constructor of filter should accept both, in this order. It should be possible to use instances of filter in range for loops: each element from the underlying container is first passed to f and if the result is true, is returned to the user (via the dereference operator), otherwise it is discarded. You may want to review map.cpp in this unit and filter.cpp from week

7.

```
template< typename, typename >
```

```
struct filter_iterator; /* ref: 25 lines */

template< typename, typename >
struct filter; /* ref: 11 lines */
```

11.p.3 [zip] Lazy sequences, part three.
Define a class template, zip, which holds 2 references to arbitrary containers, possibly of different types.
The constructor of zip should accept both (via const references). It should be possible to use instances of zip in range for loops: in each iteration, the zip iterator fetches a single elemnt from each of the two containers and returns them as a 2-tuple of const references. The iteration ends when the shorter of the two sequences runs out of elements.
Hint: to create a tuple of references, use std::tie.

```
template< typename, typename >
struct zip_iterator; /* ref: 31 lines */

template< typename, typename >
struct zip; /* ref: 11 lines */
```

11.p.4 [nibble] In this exercise, we will create a fixed-size array of nibbles (half-bytes), with an indexing operator and with a basic iterator. You may want to refer back to 05/nibble.cpp for details about operators. The class template nibble_array should take a single size_t-typed non-type template argument. It should be possible to index the array and to iterate it using a range for loop. The storage size should be the least required number of bytes. Default-constructed nibble_array should have zeroes in all its entries.

```
template< size_t N >
class nibble_array;
```

11.p.5 [tree] Write an iterator class and 2 functions, tree_begin and tree_end, which given a proper binary tree (by const reference) construct an iterable range which visits each node of the tree once. The iteration should proceed in-order, that is, the entire left subtree is visited before the current node, and the right subtree afterwards.

```
template< typename value_t >
struct tree
{
    std::unique_ptr< tree > left, right;
    tree *parent = nullptr;
    value_t value;

    static auto make_tree( const tree &t, tree *parent )
    {
        return std::make_unique< tree >( t, parent );
    }

    tree( const tree &t, tree *parent )
        : left(  t.left  ? make_tree( *t.left,  this ) : nullptr ),
          right( t.right ? make_tree( *t.right, this ) : nullptr ),
          parent( parent ),
          value( t.value )
    {}

    tree( value_t value, const tree &l, const tree &r )
        : left( make_tree( l, this ) ),
          right( make_tree( r, this ) ),
          value( std::move( value ) )
    {}

    tree( value_t value )
        : value( std::move( value ) )
    {}
};
```

Given a tree t, construct the two end-points of the iterator range:

```
// … tree_begin( … t );
// … tree_end( … t );
```

11.p.6 [scan] Implement a class template scan, which computes a generalized prefix sum. This is essentially a fold, but instead of simply returning the final value, it also returns all the intermediate results. It should be possible to iterate instances of scan using a range for loop.
Example: a scan of a sequence with elements 1, 2, 3 and 4, using std::plus, and initial value 0 should yield the sequence 1, 3, 6, 10.
The constructor, which should enable class template argument deduction, takes 3 arguments:

- a const reference to a container with value_type = T,
- a lambda with the signature S(T, S), and
- an initial value of type S, by value.

NB. Do not assume that values of either type S or T can be copied. Values of type S can be default-constructed though.

Part 11.r: Regular Exercises

11.r.1 [map] Lazy sequences, part one.
Define a class template, map, which holds two items:

- a reference to an arbitrary container,
- a lambda f (of type a → b).

The constructor of map should accept both (via const references), in this order. It should be possible to use instances of map in range for loops: each element from the underlying container is first passed to f and the result of that is returned to the user (via the dereference operator).
Hint: the type of the iterator that const versions of begin and end return is available in standard containers as a nested type, like this: std::vector< int >::const_iterator.

11.r.2 [range] We have mostly ignored the question of ownership when we worked with on-the-fly `map` and `filter` implementations in week 9: it was up to the user to ensure that the underlying container outlives the range object. However, we may sometimes want to be able to return such mapped ranges from functions which construct the underlying data, and this does not work in our previous model. Let's try a different approach then.
Define a class template range which takes a container as a template argument. The class should offer the following interface (all methods are const):

- construction from a container, with argument template deduction (we will make a copy of the container: in real world, we would be able to avoid doing that),
- iteration interface: begin and end which return suitable iterators (usable in range for at minimum),
- take and drop which construct a new range object that shares the backing data with the current one, but offer a reduced view of it (take reduces the view to first N elements, while drop removes the first N elements from the view),
- an element-wise equality comparison operator (we want this to work with ranges backed by different containers, so you will need to implement this operator as a template).

```
template< typename container_t >
class range;
```

11.r.3 [permute] Implement class permutations, with a constructor which accepts an std::vector of int and which represents a sequence of all distinct permutations of the input vector. Iterating an instance of permutations should yield values which can be both iterated and indexed, yielding, in turn, integers. The permutations object itself does not need to be indexable. For example:

```
std::vector vec{ 1, 3, 2 };

for ( auto p : permutations( vec ) )
    for ( int v : p )
        std::cerr << v << " ";
```

The first permutation should be sorted in ascending order. The sequence of permutations as a whole should be sorted in lexicographic order (as a consequence of this, the last permutation should be sorted in descending order). The output of the above program, therefore, should be: 1 2 3 1 3 2 2 1 3 2 3 1 3 1 2 3 2 1.

Part 12: Review

Since this is a review chapter with no new material in it, there are no demonstrations. You can refer to previous chapters and lectures if you encounter concepts that you do not know, or which you need to refresh.
Elementary exercises:

1. digraph – count digraph frequency
2. spelling – a very simple spell checker
3. ternary – ternary logic

Preparatory exercises:

1. chords – naming minor and major 5 & dominant 7 chords
2. grammar – generate words from regular grammars
3. linear – simple linear equations with a parser
4. poly – searching for roots of polynomials
5. queens – checking a solution of the 8 queens puzzle
6. map – more mapping of sequences

Regular exercises:

1. trie – binary tries with wildcards
2. cooking – storing recipes
3. cards – parsing and comparing playing cards
4. minilisp – a small LISP-like language parser
5. language – detect language of a text fragment
6. union – the union-set data structure

Part 12.e: Elementary Exercises

12.e.1 [digraph] We will write a simple function, digraph_freq, which accepts a string and computes the frequency of all (alphabetic) digraphs. The exact signature is up to you, in particular the return type. The only requirement is that the returned value can be indexed using strings and this returns the count (or 0 if the input string is not a correct digraph). This must also work on const instances of the return value. For examples see main.
Define digraph_freq here, along with any helper functions or classes.

12.e.2 [spelling] The file /usr/share/dict/words contains one English word per line. Write a class, spell, the constructor of which takes the path to a file of this type (one word per line) and with a single method, check, which takes an std::string which contains a single word and returns true if the word is in the provided list.

```
class spell;
```

12.e.3 [ternary] Ternary (or 3-valued) logic uses 3 different truth values: true, false and unknown (maybe).
Define a suitable type tristate and 3 constants yes, no and maybe (to avoid conflicts with built-in boolean constants), along with the standard logical operators and equality.

Part 12.p: Preparatory Exercises

12.p.1 [chords] We will write a simple program to compute and format chords (as in music). Partition the code as you see fit.
The entire western musical scale has 12 notes in it, one semitone (100 cents) apart. Chords are built up from minor (300 cents) and major (400 cents) thirds. We will only deal with chords in the root position, i.e. where the root note is in the bass and we'll use the German names:

- c, d, e, f, g, a and h are the 'base' notes
- cis, dis, eis = f, fis, gis, ais, his = c are sharps,
- ces = h, des, es, fes = e, ges, as and b are flats.

Base notes are 200 cents apart, except the e/f and h/c pairs, which are 100 cents apart. A flat subtracts, and a sharp adds, 100 cents to the base note. The simplified rules for using note names in chords are as follows:

- key is E, G, A, H, D or any note with a sharp → use sharps,
- key is F or a note with a flat → use flats,
- flats and sharps are not mixed in basic chords,
- ignore double flats and double sharps
- instead eis, his, ces and fes, use f, c, h and e.

A (pure) fifth is 700 cents and a minor 7th is 1000 cents. Intervals (in cents) are composed using addition, mod 1200. By convention, 'c' is 0. For instance, if the root is 'g', that is 700 cents, adding a pure fifth yields 1400 mod 1200 = 200 = 'd'. Notes 'g' and 'd' are a fifth apart.
The major fifth chord starts at the key note (tonic) + a major third + minor third, e.g. 'c' → 'c e g' or 'e' → 'e gis h'.

```
std::string major_5( std::string key );
```

The root of a minor fifth chord is a sixth above the key note and adds a minor third + a major third, e.g. 'c' → 'a c e' or 'e' → 'cis e gis'. Alternatively, you could think of it as a minor third below the key note, the key note itself, and a major third above.

```
std::string minor_5( std::string key );
```

The root of a dominant 7th chord is a fifth above the key note (tonic): for key 'c', the root of the dominant is 'g'. On top of the root, add a major third, a minor third, and another minor third. E.g. 'f' → 'c e g b'

```
std::string dominant_7( std::string key );
```

ref: 42 lines in 4 helper functions, major_5, minor_5 & dominant_7 are each 1 line

12.p.2 [grammar] A regular grammar has rules of the form $A \to xB$ or $A \to x$ where A and B are non-terminals and x is a terminal.
Implement a class grammar, which is default-constructible and has 2 methods:

- add_rule, which takes 2 or 3 arguments: a single char for the left-hand non-terminal (a capital alphabetic letter), a terminal and optionally another non-terminal,
- generate, a const method which takes 2 arguments: the starting non-terminal and an integer which gives the maximum length of a word, and returns an std::vector of std::string with all the words the grammar can generate (within the given size bound), sorted lexicographically.

```
class grammar;
```

12.p.3 [linear] Remember the linear equation solver from 08/linear.cpp? Let's do that again, but this time with a simple parser instead of operator overloading.
Write a function solve which takes a string as its input, and returns

an `std::pair` of floating point numbers. The input contains 2 linear equations, one per line, with 2 single-letter alphabetic variables and integer coefficients. The result should be ordered alphabetically (e.g. x, y).

```
std::pair< double, double > solve( const std::string &eq );
```

ref: solve 26 lines, helper class 19 lines

12.p.4 [poly] In this exercise, we will find at least one (real) root of an odd-degree polynomial (this is guaranteed to exist, and is comparatively easy to find using the intermediate value theorem and binary search). Write function `find_root` which takes 2 arguments:

- a vector of coefficients (each represented as a `double`), sorted from the highest-degree to the lowest, e.g. $x^3 - 3x + 2$ is represented as the vector `{ 1, 0, -3, 2 }`,
- an `std::pair` with a lower and an upper bound on the value of the root.

The function should return a root x, that is, a number for which the polynomial evaluates to zero, e.g. $x = -2$ for the above example.
The pre-condition is that the bounds evaluate to numbers with opposite signs (and hence the interval must contain at least one root). There might be multiple roots in the interval, though: it does not matter which one the function finds. The returned `double` should be within `1e-5` of the actual value of the root.

```
using bounds = std::pair< double, double >;
using poly   = std::vector< double >;

double find_root( const poly &, bounds ); /* ref: 28 lines */
```

12.p.5 [queens] Write a function that checks whether a given configuration of 8 queens on a chessboard is such that no two queens endanger each other.
The first number is the column numbered from left, a = 1, b = 2, ...; second is the row (likewise indexed from 1, starting at bottom): { 1, 1 } is the bottom left corner.

```
using position = std::pair< int, int >;
using queens = std::vector< position >;
```

Return true if all queens are safe.

```
bool check( const queens &q ); /* ref: 43 lines */
```

12.p.6 [map] We will revisit one of the sequence-related constructs from earlier, that of on-the-fly (on demand) transformation (mapping) of elements using a given function. In particular, we will look at composing maps.
In this exercise, you should implement a `map` view like the one we did in `09/map.cpp`, with one improvement: it should also work with functions which return references. The easiest way to do that is by creating a type alias using `decltype` and use that as the return type of the dereference operator. (Alternatively, look up `decltype( auto )` in cppreference, though that wouldn't work if you wanted to use the type as a component of another type.)

Part 12.r: Regular Exercises

12.r.1 [trie] We will implement a binary trie (see `06/bintrie.cpp` for more details about the data structure) with a twist.

The user will manage the nodes explicitly, for two reasons: doing it automatically is a fair amount of work, and we want to be able to share subtrees. In particular, our present trie implementation will be able to encode 0 and 1 bits in a key, but also a `?`: a bit which will not affect the outcome. The easiest way to achieve this is by pointing both the left and the right pointer of a tree to the same child node. To make things even more interesting, each leaf node should be able to reconstruct its own key, with question marks always taken to be ones. The interface:

- `root` returns a suitable pointer to the root node,
- `add` takes a suitable node pointer (the parent node), and the bit to append (a `bool`),
- `add_amb` takes a node and appends a question mark to it,
- `find` takes an `std::vector` of `bool` and returns a shared pointer to the corresponding node (or a `nullptr` if not found).

The default-constructed `trie` should be empty. Both `add` and `add_amb` should return a (shared) pointer to the new node. The nodes should provide the method `key` which returns an `std::vector` of `bool`. The nodes must not store the entire key.

```
class trie_node; /* ref: 26 lines */
class trie;      /* ref: 30 lines */
```

12.r.2 [cooking] In this exercise, we will implement a simple model of cooking, with recipes and a pantry. Try to think about code duplication and whether you can reduce it and what is the cost of reduction in duplication.
The class `pantry` will keep a list of available ingredients and their quantity. It should be default-constructible and offer a method `add`, which takes a string (the name of the ingredient) and an integer (the quantity). A `const` method `count` should take a string (name of the ingredient) and return the quantity available (possibly 0).

```
class pantry;
```

We will use another class to represent recipes (in our simplified world, a list of ingredients and quantities required to cook a meal). Like `pantry`, it should be default-constructible and offer a method `add`, which accepts 2 or 3 arguments: name, the required quantity and, if supplied, an optional quantity of the ingredient that will be used if available (in addition to the required amount) but is not required to cook the meal.

```
class recipe;
```

Finally, implement function `cook` with 3 arguments: a mutable reference to the `pantry` which will be used to obtain the ingredients, a `const` reference to the `recipe` to cook and an `int`, the number of portions to prepare. The function then returns `true` if everything went okay (and of course deducts the ingredients used up from the `pantry`) or `false` if some ingredient was missing or there wasn't enough of it, in which case the `pantry` content remains unchanged.

```
bool cook( pantry &, const recipe &, int qty );
```

12.r.3 [cards] In this exercise, we will look back at input/output streams and formatting operators.
Implement class `card` which represents one card from the standard 52-card deck, along with operators for input and output. The format is two letters, first the rank and then the suit. The rank 10 is represented as `T`. Use S, H, C and D to represent suits. Do not forget to handle errors.

```
class card;
```

Part T.3: Tasks with Templates and Iterators

The programming tasks for this block are as follows:

1. `tree.*` – iterating trees in post-order,

2. `bplus.*` – a class template implementing B+ trees,
3. `linalg.*` – complex and real linear algebra,
4. `lisp.*` – a simple interpreter for a LISP-like language.

All tasks in this block make some use of templates, which are covered in chapters 9 and 10, and of course they also rely on knowledge from previous blocks. Since the first task is about writing iterators, you will also need to understand the material from chapter 11 to complete it (though you should be able to make considerable progress with what you know from chapter 5).

Part T.3.1: [bplus]

The goal of this task is to implement B+ search tree, with insertion and lookup of keys. Assume that both keys and values can be copied and that keys can be compared using `<` and `==`.
The `max_fanout` specifies the 'branching factor' b of the tree: the maximum number of children a node can have. Each node then stores at most $b - 1$ keys. As is usual with B trees, the minimum number of children for an internal node, with the exception of the root, is $\lceil b/2 \rceil$ (the upper integral part of $b/2$).
Each node must be stored in a single contiguous chunk of memory. That is, at most one memory allocation can be retained across an `insert` call (or put differently, all but at most one memory chunk allocated during `insert` must be freed before `insert` returns).

```
template< typename key_t, typename value_t, int max_fanout >
struct bplus
{
```

Insert an element, maintaining the invariants of the B+ tree. Must run in worst-case logarithmic time. Return `true` if the tree was changed.

```
    bool insert( const key_t &, const value_t & );
```

Look up elements. The `at` method should throw `std::out_of_range` if the key is not present in the tree. The indexing operator should insert a default-constructed value if the key is absent, and return a reference to this value.

```
    bool contains( const key_t & ) const;
    value_t &at( const key_t & );
    const value_t &at( const key_t & ) const;
    value_t &operator[]( const key_t & );
```

Look up an element and return the path that leads to it in the tree, i.e. the index of the child node selected during lookup at each level. Return an empty path if the key is not present. The fetch operation then takes a path returned by `path` and fetches the corresponding value from the tree. Please note that the paths must reflect the layout of a correct B+ tree.

```
    using path_t = std::vector< int >;
    path_t path( const key_t & ) const;
    const value_t &fetch( const path_t &path ) const;
};
```

Part T.3.2: [linalg]

In this task, you will implement a few bits of basic linear algebra on top of an arbitrary scalar field, given as a template parameter. Your solution will be tested with the `complex` and `real` classes from task set 2 (which you will submit along with this solution, see below) and also using a reference implementation of those classes.
Implement these 2 data types: `vector` and `matrix` (please try to avoid confusing `vector` with `std::vector`). In addition to methods prescribed below, implement the following operators:

- vector addition and subtraction (operators `+` and `-`),
- multiplication of a vector by a scalar (from both sides, `*`),
- dot product of two vectors (operator `*`),
- matrix addition (operator `+`),
- multiplication of a vector by a matrix (again `*`, both sides),
- multiplication of compatible matrices (again `*`),
- multiplication of matrix by a scalar (`*` once more),
- equality on both vectors and matrices,
- indexing of vectors to get or change their entries,
- indexing of matrices to get or change their rows.

Note: you need to submit a working version of the 'complex' task from the task set 2 along with this one, including the solution of 'natural' if applicable (though 'natural' is not directly required, so only include it if your solution of task 'complex' needs it). Add a copy of the relevant files to this directory before you submit.
Only the exact arithmetic part of 'complex' is required: the approximation part (`abs`, `arg`, `exp`, `log1p`) can be left out. You should also add this (explicit) constructor if you don't have one: `complex( int v )`.

```
// extra files: complex.hpp complex.cpp natural.hpp natural.cpp

#include "complex.hpp" /* required! */

template< typename scalar_ >
struct vector
{
    using scalar = scalar_;

    explicit vector( int dimension ); /* construct a zero vector */
    explicit vector( const std::vector< scalar > & );

    int dim() const; /* return the dimension */
};

template< typename scalar_ >
struct matrix
{
    using scalar = scalar_;
    using vector = ::vector< scalar >;

    matrix( int rows, int columns ); /* construct a zero matrix */
    explicit matrix( const std::vector< vector > &rows );
```

The following two methods give the user direct access to the values stored in the matrix (through column and row vectors). The `n` is a 0-based index, starting from top (row) or left (column). You may return `const` references if appropriate.

```
    vector row( int n ) const;
    vector col( int n ) const;

    int cols() const;
    int rows() const;
```

Compute basic properties of matrices.

```
    int rank() const;
    scalar det() const;       /* determinant */
    matrix inv() const;       /* inverse matrix */
    matrix transpose() const; /* transpose matrix */
```

Performs in-place Gaussian elimination: after the call, the matrix should be in a reduced row echelon form.

```
    void gauss();
};
```

Note: the behaviour is undefined if the `vector` instances passed to a `matrix` constructor are not all of the same dimension and when `det` or `inv` are called on a non-square matrix or `inv` on a singular matrix. Likewise, operations on dimensionally mismatched arguments are undefined. All dimensions must be positive.

Part T.3.3: [lisp]

In this task, you will implement a simple programming language interpreter: the syntax and semantics will be based on LISP. For simplicity, the only data types will be numbers, symbols and cons cells.
Accept a string that corresponds to the number non-terminal as defined below. Store the integer part of the input in value and return the number of characters processed (including the discarded decimal part). If the input is invalid, return 0.

```
int from_string( std::string_view s, int &value );
```

The interpreter itself. The parse and eval methods may be called any number of times on the same instance, in any order, and must not interfere with each other.

```
template< typename number_t_ >
struct lisp
{
    struct error_t {}; /* indicates parse or evaluation error */
    struct nil_t {};   /* empty list */
    struct cons_t;     /* list cell */
    struct lambda_t;   /* lexical closure */

    using number_t = number_t_;
    using symbol_t = std::string;
    using value_t  = std::variant< number_t, symbol_t, error_t,
                                   cons_t, nil_t, lambda_t >;
```

Cons (list) cells are the basic building block of LISP programs. A list is built by putting values in car's and the successive tails of the list in cdr's. The last cdr of a proper list is always nil_t.

```
    struct cons_t
    {
        std::shared_ptr< value_t > car, cdr;
    };
```

Syntax:

```
expr   = { space }, ( atom | list ), { space } ;
list   = '(', expr, { space, expr }, ')' ;
space  = ' ' | ? newline ? ;
atom   = symbol | number ;

number = [ sign ], digits, [ '.', digits ] ;
symbol = s_init, { s_cont } | sign ;

digit  = '0' | '1' | '2' | '3' | '4' |
         '5' | '6' | '7' | '8' | '9' ;
sign   = '+' | '-' ;
digits = digit, { digit } ;

s_init = s_char | s_symb ;
s_char = ? alphabetic character ? ;
s_symb = '!' | '$' | '%' | '&' | '*' | '/' | ':' | '<' |
         '=' | '>' | '?' | '_' | '~' ;
s_cont = s_init | digit | s_spec ;
s_spec = '+' | '-' | '.' | '@' | '#' ;
```

If the input string does not conform to the above grammar, return a value of type error_t. Otherwise, the result is one of number_t, symbol_t, cons_t or nil_t. A list non-terminal is always parsed as a proper list or a nil_t.
Assume that if you have number_t n, it is possible to call from_string("…", n) with the above semantics (possibly extended to also handle the decimal part).

```
    value_t parse( std::string_view expr );
```

Semantics:

- numbers, nils and lambdas evaluate to themselves,
- symbols evaluate to:
  - their bound value in the current lexical environment,
  - error_t if they are unbound,
- lists are evaluated in 3 modes:
  - () to a nil_t,
  - as the if, let or lambda special form,
  - as a closure invocation (symbol arg_1 … arg_n).

Special forms:

- (if cond $expr_1$ $expr_2$):
  - evaluate to $expr_1$ if cond evaluates to non-zero,
  - evaluate to $expr_2$ if cond evaluates to zero,
  - evaluate to error otherwise,
- (let (name value) expr): evaluate to expr, in a lexical environment in which name is bound to value recursively (i.e. if value is a lambda, it may call itself using name),
- (lambda list expr) evaluates to lambda_t (an anonymous closure) with names of formal arguments given by list.

Closure invocation:

- the symbol in the car position of the list must evaluate to a lambda_t (the result is error_t otherwise),
- the entire list then evaluates the body of the lambda in the lexical environment in which the closure was defined,
- extended by binding each formal argument to the corresponding arg_n, evaluated in the current lexical environment;
- if the number of actual arguments does not match the number of formal arguments, the entire list evaluates to error.

The top-level lexical environment is empty with the exception of following builtin functions:

- +, -, *, / which each accepts exactly 2 operands of type number_t and evaluates to the obvious thing,
- car and cdr which expect exactly one value of type cons_t and evaluate to its car or cdr part,
- cons which accepts exactly 2 arguments and constructs a list cell out of them,
- list which accepts arbitrary arguments and returns a cons list of all of them.

Anything not covered above evaluates to error_t.

```
    value_t eval( value_t expr );
};
```

Part T.3.4: [tree]

In this task, you will write a simple tree iterator, i.e. an iterator that can be used to visit all nodes of a tree, in post-order.
First, implement a tree class, with the given interface. This will be the interface that tree_iterator will use (of course, you can add methods and attributes to tree as you see fit, but tree_iterator must not use them).
The tree is made of nodes, where each node can have an arbitrary number of children and a single value.

```
template< typename value_t_ >
struct tree
{
    using value_t = value_t_;
```

Substitute for any type you like, but make sure it can be copied, assigned and compared for equality; again, tree_iterator must not rely on the details (it can assign and copy values of type node_ref though).

```
    struct node_ref;
```

These functions provide access to the tree and the values stored in nodes.

```
    bool empty() const;
    node_ref root() const;
    node_ref child_at( node_ref, int ) const;
    int child_count( node_ref ) const;

    const value_t &value( node_ref ) const;
    value_t &value( node_ref );
```

Finally, methods for constructing and updating the tree follow. A tree iterator must not use them. The child added last has the highest index.

```
    node_ref make_root( const value_t &value );
    node_ref add_child( node_ref parent, const value_t &value );
```

Remove the entire subtree rooted at node.

```
    void erase_subtree( node_ref node );
};
```

Iterate a given tree in post-order; tree_iterator must be (at least) a forward iterator. Adding nodes to the tree must not invalidate any iterators. Removal of a node invalidates the iterators pointing at that node or at any of its right siblings. Dereferencing the iterator yields the value of the node being pointed at.

The tree given may or may not be an instance of the above class template tree, but it will have a node_ref nested type and the access methods (empty, root, child_at, child_count and value). The value method might return a reference (like in your implementation above) or a value, and the iterator must preserve the return type of value when dereferenced.

Note: You don't have to keep the exact form of the following declarations, but if you decide to replace them, you must make sure that the new declarations are equivalent in this sense:

- it must be possible to declare values using tree_iterator and const_tree_iterator template instances as their type,
- it is permissible to call tree_begin and tree_end on both const and mutable references to the respective tree type and the result type must match the below declarations.

```
template< typename tree > struct tree_iterator;
template< typename tree > struct const_tree_iterator;

template< typename tree >
tree_iterator< tree > tree_begin( tree & );
template< typename tree >
tree_iterator< tree > tree_end( tree & );

template< typename tree >
const_tree_iterator< tree > tree_begin( const tree & );
template< typename tree >
const_tree_iterator< tree > tree_end( const tree & );
```

Part S: Exercise Solutions

Part S.1: Week 1

S.1.1 [01.e1_predicates]

```
#define UNIT e1_predicates
#include "test_main.cpp"

bool all_odd( const std::vector< int > &v )
{
    for ( int x : v )
        if ( x % 2 != 1 )
            return false;
    return true;
}

bool any_odd( const std::vector< int > &v )
{
    for ( int x : v )
        if ( x % 2 == 1 )
            return true;
    return false;
}

bool count_divisible( const std::vector< int > &v, int k, int n )
{
    for ( int x : v )
        if ( x % k == 0 )
            n -= 1;
    return n <= 0;
}
```

S.1.2 [01.e2_palindrome]

```
#define UNIT e2_palindrome
#include "test_main.cpp"

bool is_palindrome( const std::string &s )
{
    for ( int i = 0; i < int( s.size() ); ++i )
        if ( s[ i ] != s[ s.size() - i - 1 ] )
            return false;
    return true;
}
```

S.1.3 [01.e3_pascal]

```
#define UNIT e3_pascal
#include "test_main.cpp"

std::vector< int > pascal( int n )
{
    n --;

    std::vector< int > p;
    p.push_back( 1 ); /* n over 0 */

    for ( int k = 1; k <= n; ++ k ) /* n over 1 … n */
        p.push_back( p.back() * ( n - k + 1 ) / k );

    return p;
}
```

S.1.4 [01.r1_wrap]

```
#define UNIT r1_wrap
#include "test_main.cpp"

std::string fill( const std::string &in, int columns )
{
    std::string out;
    int col = 0;

    for ( char c : in )
        if ( std::isblank( c ) && col >= columns )
            out += '\n', col = 0;
        else if ( c == '\n' )
            out += "\n\n", col = 0;
        else
```

```
            out += c, ++ col;

    return out;
}
```

S.1.5 [01.r2_digits]

```
#define UNIT r2_digits
#include "test_main.cpp"

std::vector< int > digits( int n, int base )
{
    assert( n >= 0 );
    std::vector< int > ds;

    while ( n > 0 )
    {
        ds.push_back( n % base );
        n /= base;
    }

    for ( int i = 0; i < int( ds.size() / 2 ); ++ i )
        std::swap( ds[ i ], ds[ ds.size() - i - 1 ] );

    return ds;
}
```

S.1.6 [01.r3_sieve]

```
#define UNIT r3_sieve
#include <vector>
#include <iostream>

int sieve( int bound )
{
    std::vector< bool > s;
    s.resize( bound + 1, true );

    for ( int i = 2; i <= bound; ++i )
        if ( s[ i ] )
            for ( int j = i + i; j <= bound; j += i )
                s[ j ] = false;

    for ( int i = bound; i > 0; --i )
        if ( s[ i ] )
            return i;

    return 0;
}

#include "test_main.cpp"
```

S.1.7 [01.r4_bsearch]

```
#define UNIT r4_bsearch
#include <vector>

using intvec = std::vector< int >;

intvec::iterator bsearch( intvec &vec, int val )
{
    auto b = vec.begin(), e = vec.end();

    while ( b < e ) /* the search interval is not empty */
    {
        auto mid = b + ( e - b ) / 2;
        if ( val < *mid ) e = mid;     /* must be in [b, mid) */
        if ( val > *mid ) b = mid + 1; /* must be in (mid, e) */
        if ( val == *mid ) return mid; /* we found it */
    }

    return vec.end();
}

#include "test_main.cpp"
```

Part S.2: Week 2

S.2.1 [02.e1_fibonacci]

```
#define UNIT e1_fibonacci
#include <vector>

void fibonacci( std::vector< int > &v, int n )
{
    v.clear();

    if ( n > 0 ) v.push_back( 1 );
    if ( n > 1 ) v.push_back( 1 );

    for ( int i = 2; i < n; ++ i )
        v.push_back( v[ i - 1 ] + v[ i - 2 ] );
}

#include "test_main.cpp"
```

S.2.2 [02.e2_normalize]

```
#define UNIT e2_normalize
#include <utility>   /* swap */
#include <algorithm> /* min, max */

void normalize( int &p, int &q )
{
    int a = std::max( p, q ),
        b = std::min( p, q );

    while ( b > 0 )
    {
        a = a % b;
        std::swap( a, b );
    }

    p /= a;
    q /= a;
}

#include "test_main.cpp"
```

S.2.3 [02.e3_accumulate]

```
#define UNIT e3_accumulate
#include <vector>

auto accumulate = []( auto f, const std::vector< int > &vec )
{
    int sum = 0;

    for ( int x : vec )
        sum += f( x );

    return sum;
};

#include "test_main.cpp"
```

S.2.4 [02.r1_euler]

```
#define UNIT r1_euler
#include "test_main.cpp"

long phi( long n )
{
    long r = n;
    long p = 2;

    while ( p <= n )
    {
        if ( n % p == 0 )
        {
```

```
            r *= p - 1;
            r /= p;
        }

        while ( n % p == 0 )
            n /= p;

        ++ p;
    }

    return r;
}
```

S.2.5 [02.r2_approx]

```
#define UNIT r2_approx
#include <cmath>

auto approx = []( auto f, double initial, double prec )
{
    double x = f( initial ), y;
    do
    {
        y = x;
        x = f( x );
    } while ( std::fabs( x - y ) > prec );

    return x;
};

double golden( double prec )
{
    int a = 1, b = 1;

    auto improve = [&]( double )
    {
        int c = a + b;
        a = b;
        b = c;
        return double( b ) / a;
    };

    return approx( improve, 1, prec );
}

#include "test_main.cpp"
```

S.2.6 [02.r3_solve]

```
#define UNIT r3_solve
#include "test_main.cpp"
#include <algorithm>

bool recurse( int pos, std::vector< bool > &visited,
              const std::vector< int > &jumps )
{
    if ( pos == int( jumps.size() ) )
    {
        int cnt = std::count( visited.begin(), visited.end(),
                              true );
        return int( jumps.size() ) == cnt;
    }

    if ( pos < 0 || pos >= int( visited.size() ) || visited[ pos ] )
        return false;

    visited[ pos ] = true;
    bool won = recurse( pos - jumps[ pos ], visited, jumps ) ||
               recurse( pos + jumps[ pos ], visited, jumps );
    visited[ pos ] = false;
    return won;
}

bool solve( std::vector< int > jumps )
{
    std::vector< bool > visited( jumps.size(), false );
    return recurse( 0, visited, jumps );
}
```

Part S.3: Week 3

S.3.1 [03.e1_unique]

```
#define UNIT e1_unique
#include <set>
#include "test_main.cpp"

std::vector< int > unique( const std::vector< int > &v )
{
    std::vector< int > out;
    std::set< int > seen;

    for ( int x : v )
        if ( !seen.count( x ) )
        {
            out.push_back( x );
            seen.insert( x );
        }

    return out;
}
```

S.3.2 [03.e2_reflexive]

```
#define UNIT e2_reflexive
#include "test_main.cpp"

relation reflexive( const relation &r )
{
    relation out = r;

    for ( auto [ x, y ] : r )
    {
        out.emplace( x, x );
        out.emplace( y, y );
    }

    return out;
}
```

S.3.3 [03.e3_normalize]

```
#define UNIT e3_normalize
#include "test_main.cpp"

signal_t normalize( const signal_t &s )
{
    double m = 0;
    signal_t out;

    for ( double x : s )
        m = std::max( m, x );

    if ( m == 0 )
        m = 1;

    for ( double x : s )
        out.push_back( x / m );

    return out;
}
```

S.3.4 [03.r1_mode]

```
#define UNIT r1_mode
#include "test_main.cpp"
#include <map>

int mode( const std::vector< int > &in )
```

```
{
    std::map< int, int > freq;
    int max_val = 0, max_freq = 0;

    for ( int x : in )
        freq[ x ] ++;

    for ( auto [ v, f ] : freq )
        if ( f > max_freq )
        {
            max_val = v;
            max_freq = f;
        }

    return max_val;
}
```

S.3.5 [03.r2_buckets]

```
#define UNIT r2_buckets
#include "test_main.cpp"
#include <map>

std::vector< int > sort( const std::vector< int > &stones,
                         const std::vector< bucket > &buckets )
{
    std::vector< int > out( buckets.size(), 0 );

    for ( int s : stones )
        for ( size_t i = 0; i < buckets.size(); ++ i )
        {
            auto [ min, max ] = buckets[ i ];

            if ( s >= min && s <= max )
                out[ i ] += s;
        }

    return out;
}
```

S.3.6 [03.r3_shortest]

```
#define UNIT r3_shortest
#include "test_main.cpp"
#include <queue>

std::map< int, int > shortest( const graph &g, int initial )
{
    std::map< int, int > dist;
    std::queue< int > queue;
    queue.push( initial );
    dist[ initial ] = 0;

    while ( !queue.empty() )
    {
        int from = queue.front();
        queue.pop();

        for ( auto to : g.at( from ) )
        {
            if ( dist.count( to ) )
                continue;

            dist[ to ] = dist[ from ] + 1;
            queue.push( to );
        }
    }

    return dist;
}
```

Part S.4: Week 4

S.4.1 [04.e1_diameter]

```
#define UNIT e1_diameter
#include <cmath>

struct point
{
    double x, y;
    point( double x, double y ) : x( x ), y( y ) {}
};

struct circle_radius
{
    point center;
    double radius;
    circle_radius( point c, double r ) : center( c ), radius( r ) {}
};

struct circle_point
{
    point center, perimeter;
    circle_point( point c, point p )
        : center( c ), perimeter( p )
    {}
};

double diameter( const circle_radius &c )
{
    return c.radius * 2;
}

double diameter( const circle_point &c )
{
    double dx = c.center.x - c.perimeter.x;
    double dy = c.center.y - c.perimeter.y;
    return 2 * std::sqrt( dx * dx + dy * dy );
}

#include "test_main.cpp"
```

S.4.2 [04.e2_circle]

```
#define UNIT e2_circle
#include <cmath>

struct point
{
    double x, y;
    point( double x, double y ) : x( x ), y( y ) {}
};

struct circle
{
    point center;
    double radius;

    circle( point c, double r )
        : center( c ), radius( r )
    {}

    circle( point c, point p )
        : center( c ),
          radius( std::sqrt( std::pow( p.x - c.x, 2 ) +
                             std::pow( p.y - c.y, 2 ) ) )
    {}
};

#include "test_main.cpp"
```

S.4.3 [04.e3_index]

```
#define UNIT e3_index
#include <vector>
#include <utility>

int &element( std::vector< int > &v, int idx )
```

```
{
    return v[ idx ];
}

int element( const std::vector< int > &v, int idx )
{
    return v[ idx ];
}

int &element( std::pair< int, int > &v, int idx )
{
    return idx == 0 ? v.first : v.second;
}

int element( const std::pair< int, int > &v, int idx )
{
    return idx == 0 ? v.first : v.second;
}

int size( const std::pair< int, int > & ) { return 2; }
int size( const std::vector< int > &v ) { return v.size(); }

#include "test_main.cpp"
```

S.4.4 [04.r1_complex]

```
#define UNIT r1_complex
#include <cmath>

struct angle { double v; };

struct complex
{
    double real, imag;

    complex( double r, double i )
        : real( r ), imag( i )
    {}

    complex( double m, angle phi )
        : real( m * std::cos( phi.v ) ),
          imag( m * std::sin( phi.v ) )
    {}
};

double magnitude( double x )
{
    return std::fabs( x );
}

double norm( complex c )
{
    return c.real * c.real + c.imag * c.imag;
}

double magnitude( complex c )
{
    return std::sqrt( norm( c ) );
}

double reciprocal( double x )
{
    return 1 / x;
}

complex reciprocal( complex c )
{
    return complex(  c.real / norm( c ),
                    -c.imag / norm( c ) );
}

double arg( complex c )
{
    return std::atan2( c.real, c.imag );
}

double real( complex c ) { return c.real; }
double imag( complex c ) { return c.imag; }

#include "test_main.cpp"
```

S.4.5 [04.r3_search]

```
#define UNIT r3_search
#include <vector>

struct node
{
    int value;
    int left = -1, right = -1;
    node( int v ) : value( v ) {}
};

using node_pool = std::vector< node >;

class node_ref
{
    const node_pool &pool;
    int idx;
    friend class tree;
public:
    node_ref( const node_pool &p, int i ) : pool( p ), idx( i ) {}
    node_ref left()  const { return { pool, pool[ idx ].left  }; }
    node_ref right() const { return { pool, pool[ idx ].right }; }
    int value() const { return pool[ idx ].value; }
    bool valid() const { return idx >= 0; }
};

class tree
{
    node_pool _pool;
    int _root = -1;

public:
    node_ref root() const { return { _pool, _root }; }
    bool empty() const { return _root == -1; }
    node &get( node_ref n ) { return _pool[ n.idx ]; }

    void insert( node_ref what, node_ref where, int &attach )
    {
        if ( !where.valid() )
            attach = what.idx;
        else if ( what.value() < where.value() )
            return insert( what, where.left(),  get( where ).left );
        else
            return insert( what, where.right(), get( where ).right
);
    }

    void insert( int v )
    {
        int id = _pool.size();
        _pool.emplace_back( v );
        return insert( { _pool, id }, root(), _root );
    }

};

#include "test_main.cpp"
```

Part S.5: Week 5

S.5.1 [05.e1_cartesian]

```
#define UNIT e1_cartesian
```

This is a solution that uses the friend syntax. For a solution which uses the method syntax, see complex.alt.cpp.

```
class complex
{
    double real, imag;
public:
    complex( double r, double i ) : real( r ), imag( i ) {}

    friend complex operator+( complex a, complex b )
    {
```

You may not know this syntax yet. In a return statement, braces without a type name call the constructor of the return type. I.e. `{ a, b }` in this context is the same as `complex( a, b )`.

```
        return { a.real + b.real, a.imag + b.imag };
    }

    friend complex operator-( complex a, complex b )
    {
        return { a.real - b.real, a.imag - b.imag };
    }

    friend complex operator-( complex a )
    {
        return { -a.real, -a.imag };
    }

    friend bool operator==( complex a, complex b )
    {
        return a.real == b.real && a.imag == b.imag;
    }
};
```

To avoid having a copy of the tests, we `#include` the original `.cpp` file here. You won't be able to compile this solution if you add your implementation to the original `.cpp` file, but you can probably trust us that the solution above works.

```
#include "test_main.cpp"
```

S.5.2 [05.e2_force]

```
#define UNIT e2_force
#include <cmath>

class force
{
    double x, y, z; /* cartesian components of the force */
public:
    force( double x, double y, double z )
        : x( x ), y( y ), z( z ) {}
```

We only define multiplication by a scalar (`double`) from left, since we only need that here, but it would be equally valid to flip the operand types (and define scalar multiplication on the right).

```
    friend force operator*( double s, force f )
    {
        return { s * f.x, s * f.y, s * f.z };
    }
```

Bog-standard vector addition.

```
    friend force operator+( force a, force b )
    {
        return { a.x + b.x, a.y + b.y, a.z + b.z };
    }
```

Fuzzy vector equality. Two vectors are equal when all their components are equal.

```
    friend bool operator==( force a, force b )
    {
        return std::fabs( a.x - b.x ) < 1e-10 &&
               std::fabs( a.y - b.y ) < 1e-10 &&
               std::fabs( a.z - b.z ) < 1e-10;
    }
};

#include "test_main.cpp"
```

S.5.3 [05.e3_forcefmt]

```
#define UNIT e3_forcefmt
#include <sstream>
#include <cmath>

class force
{
    double x = 0, y = 0, z = 0;
public:
    force( double x, double y, double z )
        : x( x ), y( y ), z( z )
    {}

    force() = default;

    bool operator==( const force &f ) const
    {
        return std::fabs( f.x - x ) < 1e-10 &&
               std::fabs( f.y - y ) < 1e-10 &&
               std::fabs( f.z - z ) < 1e-10;
    }

    friend std::ostream &operator<<( std::ostream &o,
                                     const force &f )
    {
        return o << "[" << f.x << " " << f.y << " " << f.z << "]";
    }

    friend std::istream &operator>>( std::istream &i, force &f )
    {
        char ch;

        if ( !( i >> ch ) || ch != '[' )
            i.setstate( i.failbit );

        i >> f.x >> f.y >> f.z;

        if ( !( i >> ch ) || ch != ']' )
            i.setstate( i.failbit );

        return i;
    }
};

#include "test_main.cpp"
```

S.5.4 [05.r1_poly]

```
#define UNIT r1_poly
#include <vector>

class poly
{
    std::vector< int > cs;
public:
    void set( int p, int c )
    {
        cs.resize( std::max( degree(), p + 1 ), 0 );
        cs[ p ] = c;
    }

    int get( int p ) const
    {
        return p < degree() ? cs[ p ] : 0;
    }

    int degree() const { return cs.size(); }
```

```
    poly operator+( const poly &o ) const
    {
        poly rv;
        for ( int i = 0; i < std::max( degree(), o.degree() ); ++i )
            rv.set( i, get( i ) + o.get( i ) );
        return rv;
    }

    poly operator*( const poly &o ) const
    {
        poly rv;
        for ( int i = 0; i < degree(); ++i )
            for ( int j = 0; j < o.degree(); ++j )
                rv.set( i + j,
                        rv.get( i + j ) + get( i ) * o.get( j ) );
        return rv;
    }

    bool operator==( const poly &o ) const
    {
        for ( int i = 0; i < std::max( degree(), o.degree() ); ++i )
            if ( get( i ) != o.get( i ) )
                return false;
        return true;
    }
};

#include "test_main.cpp"
```

S.5.5 [05.r2_csv]

```
#define UNIT r2_csv
```

It is probably easiest to implement this using `std::getline` to fetch both lines and individual cells. Other approaches are certainly possible though.

```
#include <sstream>
#include <iostream>
#include <vector>

class bad_format {};

class csv
{
    std::vector< std::vector< int > > data;
public:
```

Process a single line, with some rudimentary format validation. The `std::stoi` call will throw if the number cannot be parsed, but will not complain about trailing garbage.

```
    void process_line( const std::string &line, int cols )
    {
        std::istringstream i_line( line );
        std::string cell;

        data.emplace_back();

        for ( int i = 0; i < cols; ++i )
        {
            if ( !std::getline( i_line, cell, ',' ) )
                throw bad_format();
            data.back().push_back( std::stoi( cell ) );
        }

        i_line.get();

        if ( !i_line.eof() )
            throw bad_format();
    }
```

The constructor, fetches lines until it reaches the end of the file and processes each of them using the above.

```
    csv( std::istream &i, int cols )
    {
        std::string line;

        while ( std::getline( i, line ) )
            process_line( line, cols );
    }
```

The indexing operator. Since we want `[ x ][ y ]` to work, we need to return something with an indexing operator of its own here. The easiest thing to do is to return the underlying `vector` in which we store the row. It would be possible to return a proxy object too.

```
    std::vector< int > &operator[]( int i )
    {
        return data[ i ];
    }
};

#include "test_main.cpp"
```

S.5.6 [05.r3_set]

```
#define UNIT r3_set
#include <set>
#include <algorithm>

class set
{
    std::set< int > s;
public:
    void add( int x ) { s.insert( x ); }
    bool has( int x ) const { return s.find( x ) != s.end(); }

    set operator|( const set &o ) const
    {
        set r;
        std::set_union( s.begin(), s.end(),
                        o.s.begin(), o.s.end(),
                        std::inserter( r.s, r.s.begin() ) );
        return r;
    }

    set operator&( const set &o ) const
    {
        set r;
        std::set_intersection( s.begin(), s.end(),
                               o.s.begin(), o.s.end(),
                               std::inserter( r.s, r.s.begin() ) );
        return r;
    }

    set operator-( const set &o ) const
    {
        set r;
        std::set_difference( s.begin(), s.end(),
                             o.s.begin(), o.s.end(),
                             std::inserter( r.s, r.s.begin() ) );
        return r;
    }

    bool operator<=( const set &o ) const
    {
        return ( *this - o ).s.empty();
    }
};

#include "test_main.cpp"
```

Part S.6: Week 6

S.6.1 [06.e1_default]

```
#define UNIT e1_default
#include <string>
#include <stdexcept>

int stoi_or( std::string s, int def )
{
    try
    {
        return std::stoi( s );
    }
    catch ( std::out_of_range & )
    {
        return def;
    }
    catch ( std::invalid_argument & )
    {
        return def;
    }
}

#include "test_main.cpp"
```

S.6.2 [06.e2_counter]

```
#define UNIT e2_counter

struct counted
{
    counted();
    counted( const counted & );
    ~counted();
};

#include "test_main.cpp"

counted::counted()
{
    ++ counter;
}

counted::counted( const counted & )
{
    ++ counter;
}

counted::~counted()
{
    -- counter;
}
```

S.6.3 [06.e3_coffee]

```
#define UNIT e3_coffee

class token
{
    bool valid = false;
    friend class machine;

public:
    token() = default;
    token( const token & ) = delete;
    token( token &&o ) : valid( o.valid )
    {
        o.valid = false;
    }

    token &operator=( const token & ) = delete;
    token &operator=( token &&o ) noexcept
    {
        valid = o.valid;
        o.valid = false;
        return *this;
    }
};

class machine
{
    bool busy = false;
public:
    token make();
    void fetch( token &t );
};

#include "test_main.cpp"

token machine::make()
{
    if ( busy )
        throw ::busy();
    token t;
    t.valid = true;
    busy = true;
    return t;
}

void machine::fetch( token &t )
{
    assert( busy );

    if ( !t.valid )
        throw invalid();

    t.valid = false;
}
```

Part S.7: Week 7

S.7.1 [07.e1_dynarray]

```
#define UNIT e1_dynarray
#include <memory>
#include <algorithm>

class dynarray
{
    std::unique_ptr< int[] > _data;
    int _size;
public:

    dynarray( int size )
        : _data( std::make_unique< int[] >( size ) ),
          _size( size )
    {}

    void resize( int size )
    {
        auto d_new = std::make_unique< int[] >( size );
        auto d_old = _data.get();
        std::copy( d_old, d_old + std::min( size, _size ),
                   d_new.get() );
        _data = std::move( d_new );
        _size = size;
    }

    int &operator[]( int i )
    {
        return _data[ i ];
    }
};

#include "test_main.cpp"
```

S.7.2 [07.e2_list]

```
#define UNIT e2_list
#include <memory>
```

```
class list
{
    int _head;
    std::unique_ptr< list > _tail;
public:

    list( const list &o )
        : _head( o._head ),
          _tail( o._tail ? std::make_unique< list >( *o._tail )
                         : nullptr )
    {}

    list( int h, const list &t )
        : _head( h ),
          _tail( std::make_unique< list >( t ) )
    {}

    list() = default;

    bool empty() const { return !_tail; }
    int head() const { return _head; }
    const list &tail() const { return *_tail; }
};

#include "test_main.cpp"
```

S.7.3 [07.r1_circular]

```
#define UNIT r1_circular
```

The solution proceeds along the lines of queue.cpp: we use a singly-linked list. The solution is simpler because we do not need iteration (which was replaced by rotate.

```
#include <memory>
```

A node of the data structure, bog standard.

```
struct circular_node
{
    using pointer = std::unique_ptr< circular_node >;
    pointer next;
    int value;
};
```

Like before, we remember the head of the list (as a unique_ptr) and a pointer to the last node, which we need to implement rotate.

```
class circular
{
    std::unique_ptr< circular_node > head;
    circular_node *last = nullptr;
public:

    bool empty() const { return !last; }
```

In this case, the push method works at the head, since we use the list in a stack-like order. We have already seen move assignment, using the std::move helper function.

```
    void push( int v )
    {
        auto new_head = std::make_unique< circular_node >();
        new_head->value = v;
        new_head->next = std::move( head );
        head = std::move( new_head );
        if ( !last ) last = head.get();
    }
```

Popping items at the head is quite simple.

```
    void pop()
    {
        head = std::move( head->next );
        if ( !head ) last = nullptr;
    }
```

Access to the top element.

```
    int  top() const { return head->value; }
    int &top()       { return head->value; }
```

And the rotate operation: we pop a node off the head and chain it to the list at the tail end. Must not forget to update the last pointer. Does not work on empty list.

```
    void rotate()
    {
        auto next_head = std::move( head->next );
        last->next = std::move( head );
        last = last->next.get();
        head = std::move( next_head );
    }
};

#include "test_main.cpp"
```

S.7.4 [07.r2_zipper]

```
#define UNIT r2_zipper
#include <memory>
#include <cassert>

struct node
{
    using ptr = std::unique_ptr< node >;
    int value;
    ptr next;
    node( int v, ptr n ) : value( v ), next( std::move( n ) ) {}
};

class zipper
{
    int _focus;
    using node_ptr = std::unique_ptr< node >;
    node_ptr _left, _right;

public:
    zipper( int f ) : _focus( f ) {}

    bool shift( node_ptr &a, node_ptr &b )
    {
        auto new_b = std::move( b->next );
        auto new_a = std::move( b );
        new_a->next = std::move( a );

        std::swap( new_a->value, _focus );

        b = std::move( new_b );
        a = std::move( new_a );

        return true;
    }

    void push( node_ptr &p, int v )
    {
        p = std::make_unique< node >( v, std::move( p ) );
    }

    bool shift_left()
    {
        return _left  ? shift( _right, _left ) : false;
    }

    bool shift_right()
    {
        return _right ? shift( _left, _right ) : false;
    }
```

```
    void push_left( int v )  { push( _left,  v ); }
    void push_right( int v ) { push( _right, v ); }

    int &focus()       { return _focus; }
    int  focus() const { return _focus; }
};

#include "test_main.cpp"
```

S.7.5 [07.r3_segment]

```
#define UNIT r3_segment
#include <memory>
#include <utility>

struct segment_map
{
    struct node
    {
        std::unique_ptr< node > l, r;
        int div;
        node( int d ) : div( d ) {}
    };

    std::unique_ptr< node > root;
    int min, max;

    segment_map( int l, int u ) : min( l ), max( u ) {}

    std::pair< int, int > query( int i, node *n, int l, int u )
const
    {
        if ( !n ) return { l, u };
        if ( i <  n->div ) return query( i, n->l.get(), l, n->div );
        if ( i >= n->div ) return query( i, n->r.get(), n->div, u );
        abort();
    }

    std::pair< int, int > query( int i ) const
    {
        return query( i, root.get(), min, max );
    }

    void split( int n )
    {
        auto old_root = std::move( root );
        root = std::make_unique< node >( n );
        if ( !old_root )
            return;

        if ( old_root->div > n )
            root->r = std::move( old_root );
        else
            root->l = std::move( old_root );
    }
};

#include "test_main.cpp"
```

S.7.6 [07.r4_diff]

```
#define UNIT r4_diff
#include <memory>
#include <cmath>

struct node
{
    using ptr = std::shared_ptr< node >;
    enum op_t { cnst, var, add, mul, exp } op;
    int num = 0;
    ptr l, r;
};

class expr
{
public:
    node::ptr ptr;
    expr() : ptr( std::make_shared< node >() ) {}
    expr( int c ) : expr() { ptr->num = c; ptr->op = node::cnst; }
    expr( node::op_t o, node::ptr l = nullptr,
          node::ptr r = nullptr )
        : expr()
    {
        ptr->op = o;
        ptr->l = l;
        ptr->r = r;
    }
    expr( node::ptr p ) :ptr( p ) {}

    friend expr expnat( expr e )
    {
        return { node::exp, e.ptr };
    }

    friend expr operator+( expr a, expr b )
    {
        return { node::add, a.ptr, b.ptr };
    }

    friend expr operator*( expr a, expr b )
    {
        return { node::mul, a.ptr, b.ptr };
    }
};

const expr x{ node::var };

double eval( expr e, double v )
{
    switch ( e.ptr->op )
    {
        case node::cnst: return e.ptr->num;
        case node::var: return v;
        case node::add: return eval( e.ptr->l, v ) + eval( e.ptr->r,
v );
        case node::mul: return eval( e.ptr->l, v ) * eval( e.ptr->r,
v );
        case node::exp: return std::exp( eval( e.ptr->l, v ) );
    }
    abort();
}

expr diff( expr e )
{
    switch ( e.ptr->op )
    {
        case node::cnst: return { 0 };
        case node::var:  return { 1 };
        case node::add:
            return diff( e.ptr->l ) + diff( e.ptr->r );
        case node::mul:
            return diff( e.ptr->l ) * e.ptr->r +
                   diff( e.ptr->r ) * e.ptr->l;
        case node::exp:
            return e * diff( e.ptr->l );
    }
    abort();
}

#include "test_main.cpp"
```

Part S.8: Week 8

S.8.1 [08.e1_resistance]

```
#define UNIT e1_resistance
```

```
class segment
{
public:
    virtual double r() const = 0;
};

class series : public segment
{
    double total = 0;
public:
    void add( double r ) { total += r; }
    void add( const segment &s ) { total += s.r(); }
    double r() const override { return total; }
};

class parallel : public segment
{
    double recip = 0;
public:
    void add( double r ) { recip += 1.0 / r; }
    void add( const segment &s ) { recip += 1.0 / s.r(); }
    double r() const override { return 1.0 / recip; }
};

double resistance( const segment &s )
{
    return s.r();
}

#include "test_main.cpp"
```

S.8.2 [08.e2_perimeter]

```
#define UNIT e2_perimeter
#include <cmath>

class shape
{
public:
    virtual double perimeter() const = 0;
};

class circle : public shape
{
    double _radius;
public:
    circle( double r ) : _radius( r ) {}
    double perimeter() const override
    {
        return 8 * std::atan( 1 ) * _radius;
    }
};

class rectangle : public shape
{
    double _width, _height;
public:
    rectangle( double w, double h ) : _width( w ), _height( h ) {}
    double perimeter() const override
    {
        return 2 * _width + 2 * _height;
    }
};

#include "test_main.cpp"
```

S.8.3 [08.e3_fight]

```
#define UNIT e3_fight

class rock;
class paper;
class scissors;

class gesture
{
public:
    virtual bool visit( const rock & ) const = 0;
    virtual bool visit( const paper & ) const = 0;
    virtual bool visit( const scissors & ) const = 0;
    virtual bool fight( const gesture & ) const = 0;
};

class rock : public gesture
{
    bool visit( const rock & )     const override { return false; }
    bool visit( const paper & )    const override { return true;  }
    bool visit( const scissors & ) const override { return false; }

    bool fight( const gesture &g ) const override
    {
        return g.visit( *this );
    }
};

class paper : public gesture
{
    bool visit( const rock & )     const override { return false; }
    bool visit( const paper & )    const override { return false; }
    bool visit( const scissors & ) const override { return true;  }

    bool fight( const gesture &g ) const override
    {
        return g.visit( *this );
    }
};

class scissors : public gesture
{
    bool visit( const rock & )     const override { return true; }
    bool visit( const paper & )    const override { return false; }
    bool visit( const scissors & ) const override { return false; }

    bool fight( const gesture &g ) const override
    {
        return g.visit( *this );
    }
};

#include "test_main.cpp"
```

S.8.4 [08.r1_bom]

```
#define UNIT r1_bom
#include <memory>
#include <string>
#include <vector>
```

The base class. It remembers the part number and provides the required interface: `description` and `part_no`. Do not forget the `virtual` destructor!

```
class part
{
    std::string _part_no;
public:
    part( std::string pn ) : _part_no( pn ) {}
    virtual std::string description() const = 0;
    std::string part_no() const { return _part_no; }
    virtual ~part() = default;
};
```

The two derived classes, 80 % boilerplate.

```
class resistor : public part
{
    int _resistance;
```

```
public:
    resistor( std::string pn, int r )
        : part( pn ), _resistance( r )
    {}

    std::string description() const override
    {
        return std::string( "resistor " ) +
               std::to_string( _resistance ) + "Ω";
    }
};

class capacitor : public part
{
    int _capacitance;
public:
    capacitor( std::string pn, int c )
        : part( pn ), _capacitance( c )
    {}

    std::string description() const override
    {
        return std::string( "capacitor " ) +
               std::to_string( _capacitance ) + "µF";
    }
};
```

The smart pointer to hold and own instances of `part`.

```
using part_ptr = std::unique_ptr< part >;
```

The `bom` class itself holds the parts using the above pointer. It would be possible to use `std::map` too (and also more efficient for longer part lists). Here, we use an `std::vector` of pairs, where the pair holds the part pointer and the quantity. When the item with the given order number is not on the list, we throw an exception.

```
class bom
{
    using item = std::pair< part_ptr, int >;
    std::vector< item > _parts;
```

Find the item in the list: the common parts of `find` and `qty`.

```
    const item &_find( std::string pn ) const
    {
        for ( const auto &part : _parts )
            if ( part.first->part_no() == pn )
                return part;
        throw std::runtime_error( "part not found" );
    }

public:
```

We don't bother with duplicates. Notice the `std::move` though – we have to transfer the ownership of the `part` instance to the vector (via the pair).

```
    void add( part_ptr p, int c )
    {
        _parts.emplace_back( std::move( p ), c );
    }

    const part &find( std::string pn ) const
    {
        return *_find( pn ).first;
    }

    int qty( std::string pn ) const { return _find( pn ).second; }
};

#include "test_main.cpp"
```

S.8.5 [08.r2_circuit]

```
#define UNIT r2_circuit
```

The base class. We keep track of the inputs using raw pointers, since we do not own them. We use a `protected virtual` method to implement the 'business logic' that changes from class to class, while the outside interface is defined entirely using standard (non-virtual) methods.

```
class component
{
    component *left = nullptr,
              *right = nullptr;

protected:
    virtual bool eval( bool, bool ) = 0;

public:
    void connect( int n, component &c )
    {
        ( n ? right : left ) = &c;
    }

    bool read()
    {
        return eval( left ? left->read() : false,
                     right ? right->read() : false );
    }

    virtual ~component() = default;
};
```

The NAND gate and the `source` component are trivial enough.

```
class nand : public component
{
    bool eval( bool x, bool y ) override { return !( x && y ); }
};

class source : public component
{
    bool eval( bool, bool ) override { return true; }
};
```

The `delay` component provides one bit of memory. Reading the component will cause the value to be updated (`read` always calls `eval` internally). This class is also the reason why `eval` cannot be marked `const`.

```
class delay : public component
{
    bool _value = false;
    bool eval( bool x, bool ) override
    {
        bool rv = _value;
        _value = x;
        return rv;
    }
};

#include "test_main.cpp"
```

S.8.6 [08.r3_loops]

```
#define UNIT r3_loops

class component
{
    int left_i, right_i;
    component *left = nullptr,
              *right = nullptr;

protected:
    virtual bool eval_0( bool, bool ) { return false; }
    virtual bool eval_1( bool, bool ) { return false; }
```

```
    bool get_left() const { return left ? left->read( left_i ) :
false; }
    bool get_right() const { return right ? right->read( right_i ) :
false; }

public:
    void connect( int i, int o, component &c )
    {
        ( i ? right_i : left_i ) = o;
        ( i ? right : left ) = &c;
    }

    virtual bool read( int o )
    {
        auto l = get_left();
        auto r = get_right();
        if ( o == 0)
            return eval_0( l, r );
        else
            return eval_1( l, r );
    }

    virtual ~component() = default;
};

class cnot : public component
{
    bool eval_0( bool x, bool ) override { return x; }
    bool eval_1( bool x, bool y ) override
    {
        if ( x )
            return y;
        else
            return !y;
    }
};

class nand : public component
{
    bool eval_0( bool x, bool y ) override { return !( x && y ); }
    bool eval_1( bool x, bool y ) override { return x && y; }
};

class eq : public component
{
    bool eval_0( bool x, bool y ) override { return x == y; }
    bool eval_1( bool x, bool y ) override { return x != y; }
};

class delay : public component
{
    bool _x = false, _y = false;
    bool _in_read = false;

    bool read( int o ) override
    {
        bool out = o ? _y : _x;

        if ( _in_read )
            return out;

        _in_read =true;
        _x = get_left();
        _y = get_right();
        _in_read = false;

        return out;
    }
};

class latch : public component
{
    bool _value = false;

    bool eval_0( bool x, bool y ) override { return  eval( x, y ); }
    bool eval_1( bool x, bool y ) override { return !eval( x, y ); }

    bool eval( bool x, bool y )
    {
        if ( !x && y ) _value = true;
        if ( x ) _value = false;

        return _value;
    }
};

#include "test_main.cpp"
```

Part S.9: Week 9

S.9.1 [09.e1_iota]

```
#define UNIT e1_iota

template< typename fun_t >
void iota( fun_t f, int start, int end )
{
    for ( int i = start; i < end; ++ i )
        f( i );
}

#include "test_main.cpp"
```

S.9.2 [09.e2_quot]

```
#define UNIT e2_quot

template< typename id_t > /* id for integral domain */
class rat
{
    id_t p, q;
public:
    rat( id_t p, id_t q ) : p( p ), q( q ) {}

    bool operator==( rat r ) const { return p * r.q == r.p * q; }

    friend rat operator+( rat a, rat b )
    {
        return { a.p * b.q + b.p * a.q, a.q * b.q };
    }

    rat operator*( rat r ) const { return { p * r.p, q * r.q }; }
    rat operator/( rat r ) const { return { p * r.q, q * r.p }; }
};

class gauss
{
    int r, i;
public:
    gauss( int r, int i ) : r( r ), i( i ) {}

    gauss operator+( gauss b ) const
    {
        return { r + b.r, i + b.i };
    }

    gauss operator*( gauss b ) const
    {
        return { r * b.r - i * b.i, r * b.i + i * b.r };
    }

    bool operator==( gauss b ) const
    {
        return r == b.r && i == b.i;
    }
};
```

```
#include "test_main.cpp"
```

S.9.3 [09.e3_split]

```
#define UNIT e3_split
#include "test_main.cpp"

split_view split( std::string_view s, char delim )
{
    size_t idx = s.find( delim );
    if ( idx == s.npos )
        return { s, "" };
    else
        return { s.substr( 0, idx ), s.substr( idx + 1, s.npos ) };
}
```

S.9.4 [09.r1_tfold]

```
#define UNIT r1_tfold
template< typename value_t >
struct tree;

template< typename fun_t, typename value_t >
value_t tfold( fun_t f, const tree< value_t > &t );

#include "test_main.cpp"

template< typename fun_t, typename value_t >
value_t tfold( fun_t f, const tree< value_t > &t )
{
    if ( !t.left )
        return t.value;

    auto left  = tfold( f, *t.left ),
         right = tfold( f, *t.right );

    return f( f( t.value, left ), right );
}
```

S.9.5 [09.r2_tmap]

```
#define UNIT r2_tmap
#include <type_traits>
#include <vector>

template< typename value_t >
struct tree;

template< typename fun_t, typename val_t >
using mapped_tree = tree< std::invoke_result_t< fun_t, val_t > >;

template< typename fun_t, typename val_t >
using mapped_vec =
    std::vector< std::invoke_result_t< fun_t, val_t > >;

template< typename fun_t, typename value_t >
mapped_tree< fun_t, value_t >
tmap( fun_t f, const tree< value_t > &t );

#include "test_main.cpp"

template< typename fun_t, typename value_t >
auto map( fun_t f, const std::vector< value_t > &vec )
{
    mapped_vec< fun_t, value_t > out;
    for ( const auto &v : vec )
        out.push_back( f( v ) );
    return out;
}

template< typename fun_t, typename value_t >
mapped_tree< fun_t, value_t >
tmap( fun_t f, const tree< value_t > &t )
{
    using mt = mapped_tree< fun_t, value_t >;

    auto map_sub = [&]( const auto &subtree )
    {
        return tmap( f, subtree );
    };

    return mt( f( t.value ), map( map_sub, t.children ) );
}
```

S.9.6 [09.r3_monoid]

```
#define UNIT r3_monoid
#include <string>

template< typename hom_t >
struct elem
{
    std::string v;
    hom_t h;

    elem( std::string s, hom_t h ) : v( s ), h( h ) {}
    elem operator*( const elem &e ) const { return { v + e.v, h }; }
    bool operator==( const elem &e ) const { return h( v ) == h( e.v
); }
};

template< typename hom_t >
class monoid
{
    hom_t h;
public:
    monoid( hom_t h ) : h( h ) {}
    ::elem< hom_t > elem( std::string s ) { return { s, h }; }
};

#include "test_main.cpp"
```

Part S.10: Week 10

S.10.1 [10.e1_format]

```
#define UNIT e1_format
#include <vector>
#include <set>
#include <sstream>

template< typename T >
std::string format( const T &coll, char b, char e )
{
    int i = 0;
    std::ostringstream str;
    for ( const auto &e : coll )
        str << ( i++ ? ',' : b ) << " " << e;
    if ( i ) str << " " << e; else str << b << e;
    return str.str();
}

template< typename T >
std::string format( const std::vector< T > &s )
{
    return format( s, '[', ']' );
}

template< typename T >
std::string format( const std::set< T > &s )
{
    return format( s, '{', '}' );
}

#include "test_main.cpp"
```

S.10.2 [10.e2_concat]

```
#define UNIT e2_concat
```

```
#include <vector>

template< typename seq1_t, typename seq2_t >
auto concat( const seq1_t &s1, const seq2_t &s2 )
{
    std::vector< typename seq1_t::value_type > out;

    for ( const auto &x : s1 )
        out.push_back( x );
    for ( const auto &x : s2 )
        out.push_back( x );

    return out;
}

#include "test_main.cpp"
```

S.10.3 [10.e3_select]

```
#define UNIT e3_select
#include <vector>
#include <variant>

template< typename seq1_t, typename seq2_t >
auto select( const seq1_t &s1, const seq2_t &s2,
             const std::vector< bool > &bmp )
{
    using variant = std::variant< typename seq1_t::value_type,
                                  typename seq2_t::value_type >;
    std::vector< variant > out;

    auto i = s1.begin();
    auto j = s2.begin();

    for ( bool first : bmp )
    {
        out.emplace_back( first ? variant( *i ) : variant( *j ) );
        ++ i, ++ j;
    }

    return out;
}

#include "test_main.cpp"
```

S.10.4 [10.r1_icons]

```
#define UNIT r1_icons
struct null;

template< typename cdr_t >
struct cons
{
    int car;
    cdr_t cdr;
    cons( int car, const cdr_t &cdr ) : car( car ), cdr( cdr ) {}
};

int sum( null );

template< typename cons_t >
int sum( const cons_t & c )
{
    return c.car + sum( c.cdr );
}

#include "test_main.cpp"

int sum( null )
{
    return 0;
}
```

S.10.5 [10.r2_sorted]

```
#define UNIT r2_sorted
#include <any>

struct check_sorted
{
    std::any last;
    bool mismatch = false;

    bool was_sorted() const { return !mismatch; }

    template< typename value_t >
    void operator()( const value_t &v )
    {
        if ( last.has_value() )
        {
            if ( std::any_cast< value_t >( last ) > v )
                mismatch = true;
        }

        last = v;
    }
};

#include "test_main.cpp"
```

S.10.6 [10.r3_fsm]

```
#define UNIT r3_fsm
#include <string_view>
#include <map>

template< typename letter_t >
class fsm
{
    std::map< letter_t, const fsm * > _next;
    bool _accept;

public:
    explicit fsm( bool a = false ) : _accept( a ) {}
    void next( letter_t c, const fsm &n ) { _next[ c ] = &n; }

    template< typename seq_t >
    bool accept( const seq_t &s ) const
    {
        return accept( s.begin(), s.end() );
    }

    template< typename iter_t >
    bool accept( iter_t b, iter_t e ) const
    {
        if ( b == e ) return _accept;

        if ( auto n = _next.find( *b ); n != _next.end() )
            return n->second->accept( ++b, e );
        else
            return false;
    }
};

#include "test_main.cpp"
```

Part S.11: Week 11

S.11.1 [11.e1_iota]

```
#define UNIT e1_iota

struct iota_iterator
{
    using iterator = iota_iterator;
    int _val;
    bool operator==( iterator o ) const { return _val == o._val; };
    bool operator!=( iterator o ) const { return _val != o._val; };
    iota_iterator &operator++() { ++ _val; return *this; }
```

```
    int operator*() const { return _val; }
};

class iota
{
    int _start, _end;
public:
    iota_iterator begin() const { return { _start }; }
    iota_iterator end()   const { return { _end }; }
    iota( int s, int e ) : _start( s ), _end( e ) {}
};

#include "test_main.cpp"
```

S.11.2 [11.e2_view]

```
#define UNIT e2_view

template< typename iter_t >
class view
{
    iter_t _begin, _end;
public:
    view( iter_t b, iter_t e ) : _begin( b ), _end( e ) {}
    iter_t begin() const { return _begin; }
    iter_t end()   const { return _end; }
};

#include "test_main.cpp"
```

S.11.3 [11.e3_skip]

```
#define UNIT e3_skip

template< typename iter_t >
class skip
{
    iter_t _begin, _end;
    int _skip;
public:

    struct iterator
    {
        iter_t iter, end;
        int skip;

        decltype( auto ) operator*()       { return *iter; }
        decltype( auto ) operator*() const { return *iter; }

        bool operator==( iterator o ) const
        {
            return iter == o.iter;
        }

        bool operator!=( iterator o ) const
        {
            return iter != o.iter;
        }

        iterator operator++( int )
        {
            iterator i = *this;
            ++*this;
            return i;
        }

        iterator &operator++()
        {
            for ( int i = 0; i < skip; ++ i )
                if ( iter != end )
                    ++iter;
            return *this;
        }
    };

    skip( iter_t b, iter_t e, int s )
        : _begin( b ), _end( e ), _skip( s )
    {}

    iterator begin() const { return { _begin, _end, _skip }; }
    iterator end()   const { return { _end,   _end, _skip }; }
};

#include "test_main.cpp"
```

S.11.4 [11.r1_map]

```
#define UNIT r1_map
```

We first define the iterator. It is convenient to take the underlying iterator as a type parameter (instead of the container), though the latter would also work. The other type parameter is the lambda to call on each dereference.

```
template< typename iterator_t, typename fun_t >
struct map_iterator
{
    iterator_t it;
    const fun_t &fun;
```

Construct an iterator. The signature makes template argument deduction work, which we will use to our advantage below.

```
    map_iterator( iterator_t it, const fun_t &fun )
        : it( it ), fun( fun )
    {}
```

The dereference operator first dereferences the underlying iterator, applies fun to it and returns the result. The return type of the dereference operator is tricky, so we let the compiler figure it out for us.

```
    auto operator*() const { return fun( *it ); }
```

Pre-increment simply calls the underlying pre-increment.

```
    map_iterator &operator++() { ++it; return *this; }
```

Same thing with inequality.

```
    bool operator!=( const map_iterator &o ) const
    {
        return it != o.it;
    }
};
```

The map class template. Here we take the underlying container and the type of the lambda as parameters, since those are what the user will supply as arguments to the constructor. This way, template argument deduction will work for users as expected.

```
template< typename container_t, typename fun_t >
struct map
{
```

There are two ways to go about building the iterator type. One is explicitly, by figuring out the type of the underlying iterator (i.e. the iterator of container_t and creating an explicit instance of map_iterator. We will use this in begin.

```
    using underlying = typename container_t::const_iterator;
    using iterator = map_iterator< underlying, fun_t >;

    const container_t &container;
    const fun_t &fun;
```

The begin method needs to construct a suitable map_iterator: we built the correct type above, so we can use that as the return type of begin, then use return with braces to call the constructor.

```
    iterator begin() const { return { container.begin(), fun }; }
```

An alternative, which does not need to mention the type of the underlying iterator, but instead relies on the argument deduction that we were careful to build into the constructor of `map_iterator`.

```
    auto end() const
    {
        return map_iterator( container.end(), fun );
    }
```

Finally the constructor of `map` which lets us conveniently create instances through template argument deduction.

```
    map( const container_t &c, const fun_t &f )
        : container( c ), fun( f )
    {}
};

#include "test_main.cpp"
```

S.11.5 [11.r2_range]

```
#define UNIT r2_range
#include <algorithm>
#include <memory>

template< typename container_t >
class range
{
    using data_ptr = std::shared_ptr< container_t >;
    using iterator = typename container_t::const_iterator;
    data_ptr _data;
    iterator _b, _e;

public:
    range( container_t c )
        : _data( std::make_shared< container_t >( std::move( c ) ) ),
          _b( _data->begin() ), _e( _data->end() )
    {}

    range( data_ptr c, iterator b, iterator e )
        : _data( c ), _b( b ), _e( e )
    {}

    auto begin() const { return _b; }
    auto end()   const { return _e; }

    range take( int n ) const
    {
        return { _data, _b, std::next( _b, n ) };
    }

    range drop( int n ) const
    {
        return { _data, std::next( _b, n ), _e };
    }

    template< typename C >
    friend bool operator==( range a, range< C > b )
    {
        return std::equal( a.begin(), a.end(), b.begin(), b.end() );
    }
};

#include "test_main.cpp"
```

S.11.6 [11.r3_permute]

```
#define UNIT r3_permute
#include <vector>
#include <algorithm>

struct permutations
{
    struct permutation
    {
        std::vector< int > p;
        auto begin() const { return p.begin(); }
        auto end()   const { return p.end(); }
        int operator[]( int i ) const { return p[ i ]; }
    };

    struct iterator
    {
        std::vector< int > p;

        iterator &operator++()
        {
            if ( !std::next_permutation( p.begin(), p.end() ) )
                p.clear();
            return *this;
        }

        permutation operator*() const { return { p }; }

        bool operator!=( const iterator &o ) const
        {
            return p != o.p;
        }

        bool operator==( const iterator &o ) const
        {
            return p == o.p;
        }
    };

    std::vector< int > first;

    permutations( std::vector< int > v ) : first( std::move( v ) )
    {
        std::sort( first.begin(), first.end() );
    }

    iterator begin() const { return { first }; }
    iterator end()   const { return {}; }
};

#include "test_main.cpp"
```

Part S.12: Week 12

S.12.1 [12.e1_digraph]

```
#define UNIT e1_digraph
#include <map>
#include <string>

struct strmap
{
    std::map< std::string, int > m;

    int operator[]( std::string s ) const
    {
        return m.count( s ) ? m.find( s )->second : 0;
    }

    void add( std::string s )
    {
        m[ s ] ++;
    }
};

strmap digraph_freq( const std::string &s )
{
```

```
    strmap m;

    for ( size_t i = 0; i < s.size() - 1; ++i )
        if ( std::isalpha( s[ i ] ) && std::isalpha( s[ i + 1 ] ) )
            m.add( s.substr( i, 2 ) );

    return m;
}

#include "test_main.cpp"
```

S.12.2 [12.e2_spelling]

```
#define UNIT e2_spelling
#include <fstream>
#include <set>
#include <string>

class spell
{
    std::set< std::string > _words;
public:

    spell( const char *fn )
    {
        std::ifstream words( fn );
        std::string word;
        while ( std::getline( words, word ) )
            _words.insert( word );
    }

    bool check( const std::string s ) const
    {
        return _words.count( s );
    }
};

#include "test_main.cpp"
```

S.12.3 [12.e3_ternary]

```
#define UNIT e3_ternary

struct tristate
{
    bool val, det;
};

const tristate yes  { true,  true };
const tristate no   { false, true };
const tristate maybe{ false, false };

tristate operator&&( tristate a, tristate b )
{
    if ( a.det && b.det )
        return a.val && b.val ? yes : no;
    if ( ( a.det && !a.val ) || ( b.det && !b.val ) )
        return no;
    else
        return maybe;
}

tristate operator||( tristate a, tristate b )
{
    if ( a.det && b.det )
        return a.val || b.val ? yes : no;
    if ( ( a.det && a.val ) || ( b.det && b.val ) )
        return yes;
    else
        return maybe;
}

bool operator==( tristate a, tristate b )
{
    if ( a.det && b.det )
        return a.val == b.val;
    else
        return a.det == b.det;
}

#include "test_main.cpp"
```

S.12.4 [12.r1_trie]

```
#define UNIT r1_trie
#include <memory>
#include <vector>
#include <cassert>

using key = std::vector< bool >;

struct node
{
    std::shared_ptr< node > l, r;
    std::weak_ptr< node > up;
    node( std::shared_ptr< node > u )
        : up( u )
    {}

    bool val() const
    {
        assert( up.lock()->l.get() == this ||
                up.lock()->r.get() == this );
        return up.lock()->r.get() == this;
    }

    ::key key() const
    {
        ::key k;
        if ( up.lock() )
        {
            k = up.lock()->key();
            k.push_back( val() );
        }
        return k;
    }
};

class trie
{
    using ptr = std::shared_ptr< node >;
    ptr r;
    using ref = node &;

public:
    trie() : r( std::make_shared< node >( nullptr ) ) {}

    auto make( ptr u ) { return std::make_shared< node >( u ); }

    ptr add( ptr n, bool l )
    {
        return ( l ? n->r : n->l ) = make( n );
    }

    ptr add_amb( ptr n )
    {
        return n->r = n->l = make( n );
    }

    ptr find( key k, int idx, ptr n ) const
    {
        if ( idx == int( k.size() ) ) return n;
        return find( k, idx + 1, k[ idx ] ? n->r : n->l );
    }

    ptr find( key k ) const { return find( k, 0, r ); }
    ptr root() const { return r; }
};
```

```
#include "test_main.cpp"
```

S.12.5 [12.r2_cooking]

```
#define UNIT r2_cooking
#include <map>
#include <string>

class pantry
{
public:
    std::map< std::string, int > stuff;
    int count( std::string s ) const { return stuff.at( s ); }
    void add( std::string s, int v ) { stuff[ s ] += v; }
};

class recipe
{
public:
    std::map< std::string, std::pair< int, int > > stuff;

    void add( std::string s, int v, int o = 0 )
    {
        stuff[ s ].first  += v;
        stuff[ s ].second += o;
    }
};

bool cook( pantry &p, const recipe &r, int qty )
{
    for ( const auto &[ k, v ] : r.stuff )
        if ( qty * v.first > p.count( k ) )
            return false;

    for ( const auto &[ k, v ] : r.stuff )
    {
        p.stuff[ k ] -= qty * v.first;
        if ( p.count( k ) > qty * v.second )
            p.stuff[ k ] -= qty * v.second;
    }

    return true;
}

#include "test_main.cpp"
```

S.12.6 [12.r3_cards]

```
#define UNIT r3_cards
#include <sstream>

class card
{
    char suit, rank;
public:

    friend std::ostream &operator<<( std::ostream &o, card c )
    {
        return o << c.rank << c.suit;
    }

    friend std::istream &operator>>( std::istream &i, card &c )
    {
        char ch;
        i >> ch;

        if ( ( !std::isdigit( ch ) &&
                ch != 'A' && ch != 'J' && ch != 'Q' &&
                ch != 'K' && ch != 'T' ) ||
             ( i.peek() != 'D' && i.peek() != 'S' &&
               i.peek() != 'H' && i.peek() != 'C' ) ||
             ch == '0' )
        {
            i.unget();
            i.setstate( i.failbit );
            return i;
        }

        c.rank = ch;
        return i >> c.suit;
    }
};

#include "test_main.cpp"
```