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. Additionally, this section contains the rules and general guidelines that apply to the course as a whole.

Part A.1: Course Overview

Welcome to PB161 Programming in C++. The course consists of lectures, seminars, assignments, and a programming test at the end. Since this is a programming subject, most of the coursework – and grading – will center around actual programming. You will write tiny programs in the seminar (15-20 minutes each), write small programs for homework (a few hundred lines) and there will be a simple (but strict) programming test at the end 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:

- `grading.txt` – what is graded and how; what you need to pass,
- `homework.txt` – general guidelines that govern assignments,
- `reviews.txt` – writing and receiving peer reviews,
- `advisors.txt` – whom to talk to when you need help,
- `test.txt` – about the final programming exam.

Study materials for each seminar are in directories `01` through `13` – one directory per week. Start by reading `intro.txt`. The content for a given week will be made available to you on the Sunday preceding that week. Assignments are in directories `hw1` through `hw6` and will be made available according to the schedule shown in `grading.txt`.

Part A.2: Grading Overview

To pass the subject, you need to:

- collect a total of 20 points (by any means) and
- collect at least 4 activity points and
- pass a programming test (in the exam period)

The points can be obtained as follows (these are upper limits):

- 12 points for homework (6 assignments, 2 points each)
- 6 points for finishing your homework on time/early
- 6 points for activity
- 4 points for writing clean and elegant code
- 3 points for attending seminars
- 3 points for code review

How many points over 20 you have decides your grade:

- 28 points = A
- 26 points = B
- 24 points = C
- 22 points = D
- 20 points = E

If your 'completion type' is 'z' (credit), you will also need 20 points to pass, but you will not need to take the programming test which takes place in the exam period (see below).

A.2.1 Homeworks

There will be 6 assignments, one every two weeks. There will be 8 deadlines for each of them, one week apart and each deadline gives you one chance to pass the automated test suite. If you pass on the first or second deadline, you get 1 extra point for the assignment. For the third and fourth deadlines, the bonus is reduced to 0.5 point. Afterwards, you only get the baseline 2 points.
The deadline schedule is as follows:

	given	try 1	try 2	try 3	try 4	try 5	try 6	try 7	try 8
		3 points		2.5 points		2 points			
hw1	18.2.	25.2.	3.3.	10.3.	17.3.	24.3.	31.3.	7.4.	14.4.
hw2	3.3.	10.3.	17.3.	24.3.	31.3.	7.4.	14.4.	21.4.	28.4.
hw3	17.3.	24.3.	31.3.	7.4.	14.4.	21.4.	28.4.	5.5.	12.5.
hw4	31.3.	7.4.	14.4.	21.4.	28.4.	5.5.	12.5.	19.5.	26.5.
hw5	14.4.	21.4.	28.4.	5.5.	12.5.	19.5.	26.5.	2.6.	9.6.
hw6	28.4.	5.5.	12.5.	19.5.	26.5.	2.6.	9.6.	16.6.	23.6.

The test suite is strictly binary: you either pass or you fail. More details and guidelines are in `homework.txt`.

A.2.2 Activity and Attendance

Most of the time in seminars will be devoted to you (students) demonstrating solutions to small programming problems for your classmates. This is how you earn activity points: each time you demonstrate a problem on the beamer (projector), you earn 2 points. You must do this at least twice during the semester to earn the mandatory points for activity.
Attendance is not mandatory, other than meeting your activity requirement above. Attending the seminar 4 times is worth 1 indivisible point. You can therefore earn 1 point for coming 4-7 times, 2 points for coming 8-11 times or 3 points for coming 12 or more times.

A.2.3 Clean Code

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. Out of the 6 assignments, you will be able to submit two 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. There are two conditions:

- you must have earned at least 2.5 points for the assignment in question
- you must submit the request by the 'try 5' deadline
- the code as submitted for review must pass the testsuite

This means you have an extra week to make your code pretty after the last 2.5 point deadline. However, if you introduce regressions during this time, you will not qualify! For this reason, you may prefer to submit the already-clean code before the 'try 4' deadline.
The points earned here are fractional: you can earn anything between 0 and 1 point for each of the assignments you submit for teacher review. The remaining 2 'clean code' points are reserved for the programming test that you will do at school (details below).

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 deadlines for requesting or providing peer reviews
- writing a review is worth 0.3 points and you can write at most 10

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 Programming Test

To pass the subject, you need to demonstrate your ability to write programs on your own. The goal of the programming test is exactly this. You will get:

- a simple programming assignment
- a computer without internet access
- a selection of basic test cases
- an offline copy of `cppreference.com`
- a C++17 compiler
- a selection of text editors and IDEs in their default configuration
- 120 minutes of time

The programming test will be evaluated using automated tests, just like homework. You must pass those tests in order to succeed. If you fail the tests but you believe either the tests are in error, or you failed due to a very minor mistake, you can appeal to a human, who can overrule the test suite.
If you fail, you get an F and you can try again according to the standard rules for repeating exams. If you really don't want the F to blemish your study record, you can trade in 3 points that you already earned to have the result erased.
If you pass, your program will be reviewed by a human, who can still fail you if your code is unacceptable for some reason that was not detected by the automated testsuite. Otherwise (and this is the expected outcome) they will give you 0-2 points for style. You can keep collecting points after passing the exam, if you want.
To take the test, you need to already have 18 points. For more details, see also `test.txt`.

A.2.6 Plagiarism

Copying someone else's work or letting someone else copy yours will earn you -8 points per instance and a chat with the disciplinary committee. 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). All parties in a copying incident will be treated equally.

Part A.3: Homework

The general principles outlined here apply to all assignments. The first and most important rule is, use your brain – 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 week. In the real world, the first delivered version of your product will rarely be perfect, or even acceptable, despite your best effort to fulfill every customer requirement.
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 come to discuss the issue in person (see `advisors.txt` for details).

A.3.1 Submitting Solutions

The easiest way to submit a solution is this:

```
$ ssh aisa.fi.muni.cz
$ cd ~/pb161/hw1
<edit files until satisfied>
$ pb161 submit
```

If you prefer to work in some other directory, you may need to specify which homework you wish to submit, like this: `pb161 submit hw1`. The number of times you submit is not limited (but see also below).
NB. Only the files listed in the assignment will be submitted and evaluated. Please put your entire solution into existing files.
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
```

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 homework, use the `make` command: each `hwX` directory has a `makefile` in it. Typing `make` in the homework directory will first compile your homework into an executable binary and then run `clang-tidy` for you. 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 passes clang-tidy.
The next step is 'sanity' and runs every midnight. 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 a week – every Tuesday at midnight, right after the deadline. If you pass the verity suite, the assignment is considered complete and you are awarded the corresponding number of points. The verity suite will not run unless the code passes 'sanity'. Please note that any memory errors (including memory leaks, as reported by `valgrind`) will cause 'verity' to fail.
If you pass on the first or the second run of the full test suite (7 or 14 days after the assignment is given), you are entitled to a bonus point. If you pass at one of the next 2 attempts, you are entitled to half a bonus point. After that, you have 4 more attempts to get it right. See `grading.txt` for more details.
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 per assignment).

Part A.4: Peer Reviews

You can optionally participate in peer reviews, both as a reviewer and as a review recipient. Reviewers get points for their effort, the recipients do not, but instead 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 hw1
```

Substitute other assignments for `hw1` as appropriate. You can request a review on an assignment 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 they agreed to do the review. 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/hw1.from.xlogin`. To read the review, look at the files in `~/pb161/reviews/hw1.from.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 from other students), add `--teacher` to the command:

```
$ pb161 review --request hw1 --teacher
```

The output from `pb161 status` will indicate the assignments 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 assignments 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. **/
```

You can write up to 10 reviews, each for a maximum of 0.3 points (and a total of 3 points). The limit is applied at checkout time: once you agree to do a particular review, you cannot change your mind and 'uncheckout' it to reclaim one of the 10 slots. If you have questions, your first (and best) option is to come to see one of the advisors for the subject. They can help you with understanding the assignments, they can help explain failed tests and they should be able to help you with C++ problems. If you cannot make it to any of the slots listed below, try asking your tutor after the seminar you attend. If that fails, use the discussion board in the IS.
Everything else failing (i.e. you didn't get a satisfactory answer using any of the above methods), come to me directly during my office hours (every Tuesday and Thursday between 2pm and 3pm in B421).
You can find the advisers in A417 at the following times (every week, unless noted otherwise), starting Tuesday 25th of February (inclusive).
Location: A417

advisor	day	from	until
Roman Lacko	Monday	4pm	5pm
Anna Řechtáčková	Tuesday	12pm	1pm
Peter Navrátil	Wednesday	6pm	7pm
Vladimír Ulman	Friday	10am	11am

Except: Wednesday 25th (cancelled due to illness)

Part A.5: Programming Test

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, and that the lack of any personalisation of tools that you get may slow you down. For these reasons, we will be striving to give you problems that can be done in an hour, but you will get 2 hours of time to work them out.
For the test, you get:

- a simple programming assignment
- a computer without internet access
- a selection of basic testcases
- an offline copy of cppreference.com
- a C++17 compiler
- a selection of text editors and IDEs in their default configuration
- 120 minutes of time

You will also get a chance for a 'rehearsal': near the end of the semester, we will publish one or two mock programming tests. You should work them out on your own and try to do so in an hour, to account for the adverse conditions of an actual exam. When you are done, you will be able to submit them (using the same commands that you use for submitting homework) and you will get test results back. You can

submit multiple times for the mock test, but please keep in mind that this will not be possible at the actual exam – you will have to get it right the first time around.

Finally, the rehearsal tests will not be graded in any way, you will just get your test results and that's it. It is your responsibility to use this mechanism wisely.

Part 1: Strings and Classes

Welcome to PB161. If you haven't read the rules and guidelines under `../info`, please make a note to read them as soon as possible after the seminar. Your tutor will only highlight the most important parts for you.
The exercises today 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. The introduction:

1. (to be done)

The second part of the study materials for the week gives you a couple of „warm-up" exercises: you can do these at home and then compare your solution with a commented solution which is enclosed. This week, the exercises are:

2. `bulbs.cpp` – light bulb control center
3. (to be done)

The next section is the exercises that we will work out together in the seminar. You should read them and perhaps think about how you would go about solving them before the seminar, but the expectation is that you do the bulk of the work during the seminar. That said, the main set of exercises for today is:

4. `counting.cpp` – count words and lines in a string
5. `wrap.cpp` – wrap long lines into paragraphs of a given width
6. `words.cpp` – break a string into a vector of one-word strings
7. `account.cpp` – encapsulation of state, `const` methods
8. `shapes.cpp` – object composition
9. `contacts.cpp` – collections of your own objects

There's also some bonus work, in case you are finished early with the above, or want to practice a bit more:

10. `sieve.cpp` – find prime numbers
11. `bsearch.cpp` – binary search in an `std::vector`

To build any of the exercises, use `make` and the name of the exercise:

```
$ make words
$ ./words
```

Part 1.1: `counting`

We will start by working 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.

```
#include <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 & );
```

Part 1.2: `wrap`

We will first look at using std::string. Our first goal will be to implement a simple word wrapping (paragraph filling) algorithm.

```
#include <string>
```

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 );
```

Part 1.3: `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.

```
#include <vector>
#include <string>
```

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 a contains a single word from `in`.

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

Part 1.4: `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;
```

Part 1.5: `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).

```
#include <cmath>
```

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;
```

Part 1.6: `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).

```
#include <vector>
#include <string>
```

We need `contact` to posses 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 1.7: `sieve`

Implement the Sieve of Eratosthenes for quickly finding the largest prime smaller than or equal to a given bound.

```
#include <vector>

int sieve( int bound );
```

Part 1.8: `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).

```
#include <vector>

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

Part 2: References and Lambdas

There will be 6 exercises again, but the focus will shift a little: in the first part, we will work with references (both constant and mutable) and in the second, we will look at writing higher-order functions in C++.

1. `rewrap.cpp` – word wrapping redux, this time in-place
2. `fib.cpp` – basic uses of output parameters
3. `divisors.cpp` – collections as in/out parameters
4. `midpoints.cpp` – in/out parameters of custom types
5. `higher.cpp` – higher-order function primer: `map` and `zip`
6. `approx.cpp` – generalize the approximation routine from `fib.cpp`

And again, there are 2 bonus exercises:

1. `solve.cpp` – a very simple game solver (fits in 20 lines of code)
2. `newton.cpp` – even more general form of numeric approximation

Part 2.1: `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.

```
#include <string>

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

Part 2.2: `fib`

```
#include <cmath>
```

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 $\varphi = 1.618...$
- 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 );
```

Part 2.3: `divisors`

```
#include <vector>
```

```
#include <algorithm>
```

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 );
```

Part 2.4: midpoints

```
#include <vector>
#include <cmath>
```

A familiar class: add a 2-parameter constructor and x(), y() accessors.

```
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() ); }
```

Part 2.5: higher

```
#include <vector>
```

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.

Part 2.6: approx

Remember fib.cpp? We can do a bit better. Let's decompose our golden() function differently this time.

```
#include <cmath>
```

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 );
```

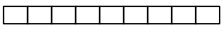
The Babylonian (Heron) method to compute square roots. Please take note, you may find it helpful later. This is how approx is supposed to be used.

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

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

Part 2.7: 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 number of cells to jump in the chosen direction in each round is given in the input vector jumps. The objective is to visit each cell. The size of the field is jumps.size() + 1.

```
#include <vector>

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

Part 2.8: newton

This exercise is as far as we'll venture with regards to numeric approximation. We will implement the 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.

```
#include <cmath>

// auto newton = []( auto f, auto df, double init, double prec )

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 );
```

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.
Approximate 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;

    for ( ; pow / fact > prec / 10; fact *= ++i, pow *= x )
        r += coeff( i ) * pow / fact;

    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 );
}
```

Part 3: Containers

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

1. freq.cpp – a word frequency histogram
2. dfs.cpp – reachability using recursive depth-first search
3. dag.cpp – check whether a graph is acyclic (dfs again)
4. rel.cpp – a tiny bit of relational algebra
5. numbers.cpp – a slightly enriched set of numbers
6. bfs.cpp – bipartiteness checking using BFS

And the bonus exercises:

7. magic.cpp – solve 4x4 semimagic squares
8. (tbd)

Part 3.1: freq

Build up a histogram of word appearances. Should be default-constructible and provide 2 methods: process, which adds 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.

```
#include <string>

class freq;
```

Part 3.2: dfs

```
#include <vector>
#include <map>

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

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 );
```

Part 3.3: dag

```
#include <map>

using graph = std::multimap< int, int >;
bool is_dag( const graph &g ); /* false iff ‹g› contains a cycle */
```

Part 3.4: rel

```
#include <tuple>
#include <set>
#include <string>
```

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 );
```

Part 3.5: 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 */

```
#include <initializer_list>

class numbers;
```

Part 3.6: bfs

```
#include <map>
#include <vector>

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 3.7: magic

A semi-magic square is an n × n grid of natural numbers 1–n^2, such that all rows and columns 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 semi-magic property. In this exercise, we will only deal with 4x4 squares (which have magic constant 34).
The interface is as follows: set takes 3 arguments: the x/y coordinates and the value of the designated cell, get takes the x/y coordinates and returns the cell value and solve fills in any unset cells.

```
class magic;
```

Returns true if the square is magic. Assumes that get returns 0 if the value has not been set. Feel free to re-use this in your solver. If strict is false, only returns false if the square has no hope of becoming magic by filling in more zeroes.

```
bool check( const magic &m, bool strict )
{
    auto sum = [&]( int c, int r, bool dir, int s = 0 )
    {
        for ( int i = 0; i < 4; ++i )
            s += m.get( dir ? c : i, dir ? i : r );
        return s;
    };

    for ( bool dir : { true, false } )
        for ( int x = 0; x < 4; ++x )
            if ( int s = sum( x, x, dir );
                    ( strict && s != 34 ) || s > 34 )
                return false;
    return true;
}

#include <vector>
```

Part 4: Object Lifecycle, Function Overloading

This week, we will practice writing constructors and destructors and we will see a few cases of function (and constructor) overloading. The basic exercise set is as follows:

1. format.cpp – method overloading 101
2. loan.cpp – banking with simple constructors and destructors
3. area.cpp – geometry with function and ctor overloads
4. zipper.cpp – const method overloading on a zipper
5. rpn.cpp – postfix arithmetic with more overloading
6. radix.cpp – smart references, ownership and destructors

Bonus exercises:

1. eval.cpp – infix evaluation with more ownership

Part 4.1: 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;
```

Part 4.2: 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. 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?

```
class account;
class loan;
```

Part 4.3: area

Implement 2 classes which represent 2D shapes: 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) /* C */
- 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 diameter)

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

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

Part 4.4: 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;
```

Part 4.5: `rpn`

Write a simple stack-based evaluator for numeric expressions in an RPN form. The opertions: `push` takes a number and pushes it onto the working stack. The `apply` operation 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. And like with the zipper, a `top` method should give access to the current top of the stack, including the possibility of changing the value. Finally, add `pop` which also returns the popped value and `empty` which returns a `bool`.

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

class eval;
```

Part 4.6: `radix`

In this exercise, we will implement a non-lookup variant of a radix tree: a data structure where a single key (string) is stored along the entire path from the root to one of the leaves. The `radix` class will own all the data and provide a `root` method and a `make` method (see below), while the `node` class will ensure that node storage is freed up when nodes go out of scope. Make sure `node` cannot be copied or assigned. A `node` has a const method `key` which retrieves the entire key.

```
#include <forward_list>

struct node_storage
{
    const node_storage *parent = nullptr;
    std::string key;
};

using radix_storage = std::forward_list< node_storage >;

class node;

class radix_tree
{
    radix_storage _nodes;
    node _root;
public:
    radix_tree();
    const node &root() const;
    node make( const node &parent, std::string key );

    bool empty() const
    {
        return std::next( _nodes.begin() ) == _nodes.end();
    }
};
```

Part 5: Operators and Exceptions

Welcome to PB161, Covid-19 edition. To make up for missing (live) lectures and seminar narration, the next four weeks (at minimum) will use extended self-study materials. The files in directories 05 through 08 or maybe 09 will contain additional explanations and examples and it is recommended that you follow them carefully. This should be your primary material: lecture slides and lecture videos from an earlier run of the subject will be supplementary. The homework schedule is unaffected, and all constructs that will be required in assignments that fall into the quarantine month will be explained in these notes.
That said, the main topics for week 5 are operator overloading (which will build on what we learned about function and method overloading in week 4) and exceptions. The first set of files to look at are annotated sources that introduce the concepts for this week:

1. `arithmetic.cpp` – introduction to operator overloading,
2. `relational.cpp` – implementing equality and ordering,
3. `access.cpp` – dereference, indexing and other access ops,
4. `exceptions.cpp` – throwing and catching exceptions.
5. `convert.cpp` – conversion and assignment,

Now that we have explained the new concepts, let's try a couple warm-up exercises. For these, you can compare your solution to a reference implementation, which is always in a file called `x.sol.cpp`, or in the PDF version at the end of the section.

6. `cartesian.cpp` – complex numbers in algebraic form,
7. `force.cpp` – composing and scaling forces.

Then comes the usual block of 6 exercises with basic test-cases included (you have to work out these for yourself, no reference solutions will be provided):

8. `polar.cpp` – complex numbers in polar form,
9. `rational.cpp` – rational numbers with ordering,
10. `set.cpp` – a set of integers with set operators,
11. `nibble.cpp` – a pointer-like class for sub-byte access,
12. `invest.cpp` – we further stretch the banking story,
13. `fixnum.cpp` – more numbers, this time with a parser.

Bonus exercises (more difficult, also no solutions):

14. `poly.cpp` – polynomials with addition and multiplication
15. `pretty.cpp` – construct strings from expressions

Part 5.1: `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`.

Part 5.2: `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.

```
#include <cstdint>
```

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.

Part 5.3: 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).

```
#include <vector>
```

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 ) { 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`.

Part 5.4: 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;
```

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() : _ptr( nullptr ) {}
    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 ( 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 ( null_pointer_exception ) { caught = true; }

    assert( caught );
}
```

Part 5.5: 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).

```
#include <stdexcept>
#include <new>
```

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 ( 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 ( 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 ( 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
}
```

And finally the last example: conversion operators and the assignment operator, in convert.cpp.

Part 5.6: cartesian

In this exercise, we will implement complex numbers with addition, subtraction, unary minus and equality.
The class should be called cartesian (unfortunately for us, complex is a keyword in C). The constructor should take 2 real numbers (the real and imaginary parts).

```
class cartesian;
```

Part 5.7: 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;
```

Part 5.8: 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 */
```

Part 5.9: rational

In this exercise, we will represent rational numbers (fractions) with addition and ordering. The constructor of rat should take the numer-

ator 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 */
```

Part 5.10: 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 5.11: 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: treat everything as mutable.
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 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 */
```

Part 5.12: 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 */
```

Part 5.13: 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 integers 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 5.14: 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 */
```

Part 5.15: pretty

TBD

Part 5.16: Exercise Solutions

Sample solutions for the warm-up exercises follow.

5.16.1 cartesian (solution)

This is a solution that uses the friend syntax. For a solution which uses the method syntax, see cartesian.alt.cpp.

```
class cartesian
{
    double real, imag;
public:
    cartesian( double r, double i ) : real( r ), imag( i ) {}

    friend cartesian operator+( cartesian a, cartesian 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 cartesian(a, b).

```
        return { a.real + b.real, a.imag + b.imag };
    }

    friend cartesian operator-( cartesian a, cartesian b )
    {
        return { a.real - b.real, a.imag - b.imag };
    }

    friend cartesian operator-( cartesian a )
    {
        return { -a.real, -a.imag };
    }

    friend bool operator==( cartesian a, cartesian 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 "cartesian.cpp"
```

5.16.2 cartesian (alternate solution)

This is a solution that uses the friend syntax. For a solution which uses the method syntax, see cartesian.alt.cpp.

```
class cartesian
{
    double real, imag;
public:
    cartesian( double r, double i ) : real( r ), imag( i ) {}

    cartesian operator+( cartesian b ) const
    {
        return { real + b.real, imag + b.imag };
    }

    cartesian operator-( cartesian b )
    {
        return { real - b.real, imag - b.imag };
    }

    cartesian operator-()
    {
        return { -real, -imag };
    }

    bool operator==( cartesian b ) const
    {
        return real == b.real && imag == b.imag;
    }
};

#include "cartesian.cpp"
```

5.16.3 force (solution)

```
#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 "force.cpp"
```

Part 6: Memory, IO

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`.
The second half of the seminar will be about IO: we will look at formatted input and output and at reading and writing files. That said, here are some introductory examples:

1. `queue.cpp` – a queue with stable references
2. `files.cpp` – opening files, reading and writing strings
3. `streams.cpp` – from values to strings and back
4. `format.cpp` – overloading formatting operators

Exercises with solutions:

5. `circular.cpp` – a singly-linked circular list
6. `csv.cpp` – parse comma-separated numeric data

Standard exercises:

7. `unrolled.cpp` – a linked list of arrays
8. `bittrie.cpp` – bitwise tries (radix trees)
9. `force.cpp` – vectors redux, this time with IO
10. `grep.cpp` – print matching lines
11. `solid.cpp` – efficient storage of optional data
12. `tmpfile.cpp` – an auto-erasing temporary file

Bonus exercises:

13. `chartrie.cpp` – binary tree for holding string keys
14. `parser.cpp` – parse a very simple lisp-like language

Part 6.1: 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.

```
#include <memory>
```

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 /* C */
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() );
}
```

Part 6.2: `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.

```
#include <fstream>
```

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 output. 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( "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( "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( "files.out" );
    assert( check == line );
}
```

Part 6.3: 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.

```
#include <sstream>
#include <cmath>
```

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" );
```

```
}
```

Part 6.4: 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.

```
#include <cmath>
#include <sstream>
```

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 over-

loads 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 6.5: `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;
```

Part 6.6: `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.

```
#include <sstream>
```

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;
```

Part 6.7: `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 */
```

Part 6.8: `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.

```
#include <vector>
```

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 */
```

Part 6.9: force

This week in the physics department, we will deal with formatting and parsing vectors (forces, just to avoid confusion with `std::vector`... for now).

```
#include <sstream>
```

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 6.10: 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.

```
#include <sstream>
#include <string>
```

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 & );
```

Part 6.11: 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.

```
#include <vector>
```

To make testing easier, we declare a global counter for 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;
```

Part 6.12: 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.

```
#include <fstream>  /* fstream */
#include <cstdio>   /* remove */
#include <unistd.h> /* access */
```

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;
```

Part 6.13: Exercise Solutions

Sample solutions for the warm-up exercises follow.

6.13.1 `circular` (solution)

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 "circular.cpp"
```

6.13.2 `csv` (solution)

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 "csv.cpp"
```

Part 7: 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 comments:

1. `account.cpp` – a simple inheritance example
2. `shapes.cpp` – polymorphism and late dispatch
3. `expr.cpp` – dynamic and static types, more polymorphism
4. `destroy.cpp` – virtual destructors
5. `factory.cpp` – polymorphic return values

As usual, we will continue with a couple exercises with solutions:

6. `bom.cpp` – polymorphism and collections
7. `circuit.cpp` – calling virtual methods within the class

The basic set of exercises:

8. `prisoner.cpp` – the famous dilemma
9. `bexpr.cpp` – boolean expressions with variables
10. `sexpr.cpp` – a tree made of lists (lisp style)
11. `network.cpp` – a network of counters
12. `filter.cpp` – filter items from a data source
13. `geometry.cpp` – shapes and visitors

Bonus exercises:

14. `loops.cpp` – circuits with loops
15. `intersect.cpp` – a different take on geometry

Part 7.1: 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 );
}
```

Part 7.2: 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.

```
#include <cmath>
#include <utility>
```

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 ) );
}
```

Part 7.3: 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.

```
#include <stdexcept>
```

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[1] 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:

```
    int value() const override { return _value; }
};
```

[1] As long as we pretend that the `volatile` keyword does not exist, which is an entirely reasonable thing to do. */

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 ( 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 ( std::logic_error & ) {}
}
```

Part 7.4: destroy

In this (entirely synthetic, sorry) example, we will look at object destruction, especially in the context of polymorphism.

```
#include <memory>
```

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.

```
}
```

Part 7.5: 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.

```
#include <memory>
#include <sstream>
```

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
    {
        return std::string( "cog with " ) +
               std::to_string( teeth ) + " teeth";
    }
};

class axle : public part
{
public:
    std::string description() const
    {
        return "axle";
    }
};

class screw : public part
{
    int _thread, _length;
public:

    screw( int t, int l ) : _thread( t ), _length( l ) {}

    std::string description() const
    {
        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 7.6: 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.

```
#include <string>
```

```
#include <memory>
```

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;
```

Part 7.7: 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) 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;
```

Part 7.8: 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 );
```

Part 7.9: 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.

```
#include <map>
```

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 */
```

Part 7.10: 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.

```
#include <memory>
```

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;
```

Part 7.11: 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;
```

Part 7.12: 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.

```
#include <set>
#include <memory>
```

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;
```

Part 7.13: 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).

```
#include <cmath>
```

First some definitions: the familiar point, but also a helper class slope 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.

```
using point = std::pair< double, double >;

bool close( double a, double b )
{
    return std::fabs( a - b ) < 1e-10;
}

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 );
    }
};
```

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 );
```

Now we can define the class `object`, which will have a `virtual` method `intersects` with two overloads: one that accepts a `const` reference to a `circle` and another that accepts a `const` reference to a `line`.

```
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 7.14: Exercise Solutions

Sample solutions for the warm-up exercises follow.

7.14.1 `bom` (solution)

```
#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 "bom.cpp"
```

7.14.2 `circuit` (solution)

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 "circuit.cpp"
```