PB161 Programování v C++

Petr Ročkai

Část A: Úvod a organizace ... 1
Část B: Doporučení k zápisu kódu ... 6
Část 1: Hodnoty a funkce ... 7
Část 2: Složené hodnoty ... 14
Část 3: Metody a operátory ... 19
Část 4: Životní cyklus hodnot ... 25
Část S.1: Funkce a hodnoty ... 30
Část 5: Ukazatele ... 32
Část 6: Dědičnost a pozdní vazba ... 37
Část 7: Výjimky a princip RAII ... 47
Část 8: Lambda, pokročilé operátory ... 52
Část S.2: Ukazatele, výjimky, OOP ... 53
Část 9: Součtové typy ... 54
Část 10: Knihovna algoritmů ... 54
Část 11: Řetězce ... 54
Část 12: Vstup a výstup ... 54
Část S.3: Součtové typy, řetězce ... 54
Část T: Technické informace ... 55

Část A: Úvod a organizace

Tento dokument je sbírkou cvičení a komentovaných příkladů zdrojového kódu. Každá kapitola odpovídá jednomu týdnu semestru, a procvičuje látku probranou na přednášce v předchozím týdnu. Cvičení v prvním týdnu semestru („nulté") na žádnou přednášku nenavazuje, je určeno k seznámení se s výukovým prostředím, studijními materiály a základními nástroji ekosystému.

Každá část sbírky (zejména tedy všechny ukázky a příklady) jsou také k dispozici jako samostatné soubory, které můžete upravovat a spouštět. Této rozdělené verzi sbírky říkáme zdrojový balík. Aktuální verzi[1] (ve všech variantách) můžete získat dvěma způsoby:

1. Ve studijních materiálech[2] předmětu v ISu – soubory PDF ve složce `text`, zdrojový balík ve složkách `00` (organizační informace), `01` až `12` (jednotlivé kapitoly = týdny semestru), dále `s1` až `s3` (sady úloh) a konečně ve složce `sol` vzorová řešení. Doporučujeme soubory stahovat dávkově pomocí volby „stáhnout jako ZIP".
2. Po přihlášení na studentský server `aisa` (buď za pomoci `ssh` nebo `putty`) zadáním příkazu `pb161 update`. Všechny výše uvedené složky pak naleznete ve složce `~/pb161`.

Tato kapitola (složka) dále obsahuje závazné organizační pokyny. Než budete pokračovat, pozorně si je prosím přečtěte.

Část A.1: Přehled

Tento předmět sestává z přednášek, cvičení, sad domácích úloh a závěrečné zkoušky. Protože se jedná o „programovací" předmět, většina práce v předmětu – a tedy i jeho hodnocení – se bude zaměřovat na praktické programování. Je důležité, abyste programovali co možná nejvíce, ideálně každý den, ale minimálně několikrát každý týden. K tomu Vám budou sloužit příklady v této sbírce (typicky se bude jednat o velmi malé programy v rozsahu jednotek až desítek řádků, kterých byste měli být v průměru schopni vyřešit několik za hodinu) a domácí úlohy, kterých budou za semestr 3 sady, a budou znatelně většího rozsahu (maximálně malé stovky řádků). V obou případech bude v průběhu semestru stoupat náročnost – je tedy důležité, abyste drželi krok a práci neodkládali na poslední chvíli.

Protože programování je těžké, bude i tento kurz těžký – je zcela nezbytné vložit do něj odpovídající úsilí. Doufáme, že kurz úspěšně absolvujete, a co je důležitější, že se v něm toho naučíte co nejvíce. Je ale nutno podotknout, že i přes svou náročnost je tento kurz jen malým krokem na dlouhé cestě.

A.1.1 Probíraná témata Předmět je rozdělen do 4 bloků (čtvrtý blok patří do zkouškového období). Do každého bloku v semestru patří 4 přednášky (témata) a jim odpovídající 4 cvičení. Pozor! Cvičení jsou vůči přednáškám posunuta o týden: první přednáška druhého bloku je tedy ve stejném týdnu jako poslední cvičení prvního bloku.

bl.		téma	přednáška
1	1.	funkce, jednoduché hodnoty, reference	14.2.
	2.	složené hodnoty	21.2.
	3.	vlastní metody a operátory	28.2.
	4.	životní cyklus hodnot, vlastnictví	7.3.
2	5.	ukazatele	14.3.
	6.	dědičnost, pozdní vazba	21.3.
	7.	výjimky, princip RAII	28.3.
	8.	lexikální uzávěry, další operátory	4.4.
3	9.	součtové typy	11.4.
	10.	knihovna algoritmů	18.4.
	11.	řetězce	25.4.
	12.	vstup a výstup	2.5.
–	13.	bonusy, opakování	9.5.

A.1.2 Organizace sbírky V následujících sekcích naleznete detailnější informace a závazná pravidla kurzu: doporučujeme Vám, abyste se s nimi důkladně seznámili. Zbytek sbírky je pak rozdělen na části, které odpovídají jednotlivým týdnům semestru. Důležité: během prvního týdne semestru už budete řešit přípravy z první kapitoly, přestože první cvičení je ve až v týdnu druhém. Nulté cvičení je volitelné a není nijak hodnoceno.

Kapitoly jsou číslovány podle přednášek, na které navazují: ve druhém týdnu semestru se tedy ve cvičení budeme zabývat tématy, která byla probrána v první přednášce (to je ta, která proběhla v prvním týdnu semestru), a ke kterým jste v prvním týdnu vypracovali a odevzdali přípravy.

A.1.3 Plán semestru Tento kurz vyžaduje značnou aktivitu během semestru. V této sekci naleznete přehled důležitých událostí formou kalendáře. Jednotlivé události jsou značeny takto (bližší informace ke každé naleznete v následujících odstavcích tohoto úvodu):

- „před X" – přednáška ke kapitole č. X,
- „cvič 0" – první den „nultých" cvičení,
- „cvič 1" – první den cvičení ke kapitole 1,
- „X/v" – mezivýsledek verity testů příprav ke kapitole X,
- „X/p" – poslední termín odevzdání příprav ke kapitole X,
- „sX/Y" – Yté kolo verity testů k sadě X.

Nejdůležitější události jsou zvýrazněny: termíny odevzdání příprav a poslední termín odevzdání úloh ze sad (obojí vždy o 23:59 uvedeného dne).

[1] Studijní materiály budeme tento semestr doplňovat průběžně, protože kurz prochází zásadní reorganizací. Než začnete pracovat na přípravách nebo příkladech ze sady, vždy se prosím ujistěte, že máte jejich aktuální verzi. Zadání příprav lze považovat za finální počínaje půlnocí na pondělí odpovídajícího týdne, sady podobně půlnocí na první pondělí odpovídajícího bloku. Pro první týden tedy 13.2.2023 0:00 a první sadu 20.2.2023 0:00.

[2] `https://is.muni.cz/auth/el/fi/jaro2023/PB161/um/`

únor

Po	Út	St	Čt	Pá	So	Ne
13 cvič 0	14 před 1	15	16 01/v	17	18 01/p	19
20 cvič 1	21 před 2	22 s1/2	23 02/v	24 s1/3	25 02/p	26
27 s1/4	28 před 3					

březen

Po	Út	St	Čt	Pá	So	Ne
		1 s1/5	2 03/v	3 s1/6	4 03/p	5
6 s1/7	7 před 4	8 s1/8	9 04/v	10 s1/9	11 04/p	12
13 s1/10	14 před 5	15 s1/11	16 05/v	17 s1/12	18 05/p	19
20 s2/1	21 před 6	22 s2/2	23 06/v	24 s2/3	25 06/p	26
27 s2/4	28 před 7	29 s2/5	30 07/v	31 s2/6		

duben

Po	Út	St	Čt	Pá	So	Ne
					1 07/p	2
3 s2/7	4 před 8	5 s2/8	6 08/v	7 s2/9	8 08/p	9
10 s2/10	11 před 9	12 s2/11	13 09/v	14 s2/12	15 09/p	16
17 s3/1	18 před 10	19 s3/2	20 10/v	21 s3/3	22 10/p	23
24 s3/4	25 před 11	26 s3/5	27 11/v	28 s3/6	29 11/p	30

květen

Po	Út	St	Čt	Pá	So	Ne
1 s3/7	2 před 12	3 s3/8	4 12/v	5 s3/9	6 12/p	7
8 s3/10	9 před 13	10 s3/11	11	12 s3/12	13	14

Část A.2: Hodnocení

Abyste předmět úspěšně ukončili, musíte v každém bloku[3] získat 60 bodů. Žádné další požadavky nemáme.
Výsledná známka závisí na celkovém součtu bodů (splníte-li potřebných 4×60 bodů, automaticky získáte známku alespoň E). Hodnota ve sloupci „předběžné minimum" danou známku zaručuje – na konci semestru se hranice ještě mohou posunout směrem dolů tak, aby výsledná stupnice přibližně odpovídala očekávané distribuci dle ECTS.[4]

známka	předběžné minimum	po vyhodnocení semestru
A	409	90. percentil + 60
B	348	65. percentil + 60
C	301	35. percentil + 60
D	265	10. percentil + 60
E	240	240

Body lze získat mnoha různými způsoby (přesnější podmínky naleznete v následujících sekcích této kapitoly). V blocích 1-3 (probíhají během semestru) jsou to:

- za každou úspěšně odevzdanou přípravu 1 bod (max. 6 bodů každý týden, nebo 24/blok),
- za každou přípravu, která projde „verity" testy navíc 0,5 bodu (max. 3 body každý týden, nebo 12/blok),
- za účast[5] na cvičení získáte 3 body (max. tedy 12/blok),
- za aktivitu ve cvičení 3 body (max. tedy 12/blok).

Za přípravy a cvičení lze tedy získat teoretické maximum 60 bodů. Dále můžete získat:

- 10 bodů za úspěšně vyřešený příklad ze sady domácích úloh (celkem vždy 60/blok).

V blocích 2-4 navíc můžete získat body za kvalitu řešení příkladů ze sady úloh předchozího bloku:

- za kvalitu kódu max. 5 bodů za příklad (celkem 30/blok).

Konečně blok 4, který patří do zkouškového období, nemá ani cvičení ani sadu domácích úloh. Krom bodů za kvalitu kódu ze třetí sady lze získat:

- 15 bodů za každý zkouškový příklad (celkem 90/blok).

Celkově tedy potřebujete:

- blok 1: 60/120 bodů,
- blok 2: 60/150 bodů,
- blok 3: 60/150 bodů,
- blok 4: 60/120 bodů (neplatí pro ukončení zápočtem).

Část A.3: Přípravy

Jak již bylo zmíněno, chcete-li se naučit programovat, musíte programování věnovat nemalé množství času, a navíc musí být tento čas rozložen do delších období – semestr nelze v žádném případě doběhnout tím, že budete týden programovat 12 hodin denně, i když to možná pokryje potřebný počet hodin. Proto od Vás budeme chtít, abyste každý týden odevzdali několik vyřešených příkladů z této sbírky. Tento požadavek má ještě jeden důvod: chceme, abyste vždy v době cvičení už měli látku každý samostatně nastudovanou, abychom mohli řešit zajímavé problémy, nikoliv opakovat základní pojmy.
Také Vás prosíme, abyste příklady, které plánujete odevzdat, řešili vždy samostatně: případnou zakázanou spolupráci budeme trestat (viz také konec této kapitoly).

A.3.1 Odevzdání Každý příklad obsahuje základní sadu testů. To, že Vám tyto testy prochází, je jediné kritérium pro zisk základních bodů za odevzdání příprav. Poté, co příklady odevzdáte, budou tytéž testy na Vašem řešení automaticky spuštěny, a jejich výsledek Vám bude zapsán do poznámkového bloku. Smyslem tohoto opatření je zamezit případům, kdy omylem odevzdáte nesprávné, nebo jinak nevyhovující

[3] Máte-li předmět ukončen zápočtem, čtvrtý blok a tedy ani závěrečný test pro Vás není relevantní. Platí požadavek na 3×60 bodů z bloků v semestru.
[4] Percentil budeme počítat z bodů v semestru (první tři bloky) a bude brát do úvahy všechny studenty, bez ohledu na ukončení, kteří splnili tyto tři bloky (tzn. mají potřebné minimum 3×60 bodů).
[5] V případě, že jste řádně omluveni v ISu, nebo Vaše cvičení odpadlo (např. padlo na státní svátek), můžete body za účast získat buď náhradou v jiné skupině (pro státní svátky dostanete instrukce mailem, individuální případy si domluvte s cvičícími obou dotčených skupin). Nemůžete-li účast nahradit takto, domluvte se se svým cvičícím na vypracování 3 rozšířených příkladů ze sbírky (přesné detaily Vám sdělí cvičící podle konkrétní situace). Neomluvenou neúčast lze nahrazovat pouze v jiné skupině a to max. 1–2× za semestr.

řešení, aniž byste o tom věděli. Velmi silně Vám proto doporučujeme odevzdávat s určitým předstihem, abyste případné nesrovnalosti měli ještě čas vyřešit. Krom základních („sanity") testů pak ve čtvrtek o 23:59 a znovu v sobotu o 23:59 (těsně po konci odevzdávání) spustíme rozšířenou sadu testů („verity").
Za každý odevzdaný příklad, který splnil základní („sanity") testy získáváte jeden bod. Za příklad, který navíc splnil rozšířené testy získáte dalšího 0,5 bodu (tzn. celkem 1,5 bodu). Výsledky testů naleznete v poznámkovém bloku v informačním systému.
Příklady můžete odevzdávat:

1. do odevzdávárny s návem `NN` v ISu (např. `01`),
2. příkazem `pb161 submit` ve složce `~/pb161/NN`.

Podrobnější instrukce naleznete v kapitole B.

A.3.2 Harmonogram Termíny pro odevzdání příprav k jednotlivým kapitolám jsou shrnuty v následující tabulce:

bl.	kapitola	přednáška	verity	termín
1	1	14.2.	16.2.	18.2.
	2	21.2.	23.2.	25.2.
	3	28.2.	2.3.	4.3.
	4	7.3 .	9.3.	11.3.
2	5	14.3.	16.3.	18.3.
	6	24.3.	23.3.	25.3.
	7	28.3.	30.3.	1.4.
	8	4.4.	6.4.	8.4.
3	9	11.4.	13.4.	15.4.
	10	18.4.	20.4.	22.4.
	11	25.4.	27.4.	29.4.
	12	2.5.	4.5.	6.5.

Část A.4: Cvičení

Těžiště tohoto předmětu je jednoznačně v samostatné domácí práci – učit se programovat znamená zejména hodně programovat. Společná cvičení sice nemohou tuto práci nahradit, mohou Vám ale přesto v lecčem pomoct. Smyslem cvičení je:

1. analyzovat problémy, na které jste při samostatné domácí práci narazili, a zejména prodiskutovat, jak je vyřešit,
2. řešit programátorské problémy společně (s cvičícím, ve dvojici, ve skupině) – nahlédnout jak o programech a programování uvažují ostatní a užitečné prvky si osvojit.

Cvičení je rozděleno na dva podobně dlouhé segmenty, které odpovídají těmto bodům. První část probíhá přibližně takto:

- cvičící vybere ty z Vámi odevzdaných příprav, které se mu zdají něčím zajímavé – ať už v pozitivním, nebo negativním smyslu,
  - řešení bude anonymně promítat na plátno a u každého otevře diskusi o tom, čím je zajímavé;
  - Vaším úkolem je aktivně se do této diskuse zapojit (můžete se například ptát proč je daná věc dobře nebo špatně a jak by se udělala lépe, vyjádřit svůj názor, odpovídat na dotazy cvičícího),
  - k promítnutému řešení se můžete přihlásit a ostatním přiblížit, proč je napsané tak jak je, nebo klidně i rozporovat případnou kritiku (není to ale vůbec nutné),
- dále podobným způsobem vybere vzájemné (peer) recenze, které jste v předchozím týdnu psali, a stručně je s Vámi prodiskutuje (celkovou strukturu recenze, proč je který komentář dobrý nebo nikoliv, jestli nějaký komentář chybí, atp.) – opět se můžete (resp. byste se měli) zapojovat,
- na Vaši žádost lze ve cvičení analogicky probrat neúspěšná řešení příkladů (a to jak příprav, tak příkladů z uzavřených sad).

Druhá část cvičení je variabilnější, ale bude se vždy točit kolem bodů za aktivitu (každý týden můžete za aktivitu získat maximálně 3 body). Ve čtvrtém, osmém a dvanáctém týdnu proběhnou „vnitrosemestrálky" kde budete řešit samostatně jeden příklad ze sbírky, bez možnosti hledat na internetu – tak, jak to bude na zkoušce; každé úspěšné řešení (tzn. takové, které splní verity testy) získá ony 3 body za aktivitu pro daný týden.
V ostatních týdnech budete ve druhém segmentu kombinovat různé aktivity, které budou postavené na příkladech typu `r` z aktuální kapitoly (které konkrétní příklady budete ve cvičení řešit vybere cvičící, může ale samozřejmě vzít v potaz Vaše preference):

- můžete se přihlásit k řešení příkladu na plátně, kdy primárně vymýšlíte řešení Vy, ale zbytek třídy Vám bude podle potřeby radit, nebo se ptát co/jak/proč se v řešení děje,
- cvičící Vám může zadat práci ve dvojicích – první dvojice, která se dopracuje k funkčnímu řešení získá možnost své řešení předvést zbytku třídy – vysvětlit jak a proč funguje, odpovědět na případné dotazy, opravit chyby, které v řešení publikum najde, atp. – a získat tak body za aktivitu,
- příklad můžete také řešit společně jako skupina – takto vymyšlený kód bude zapisovat cvičící (body za aktivitu se v tomto případě neudělují).

Část A.5: Sady domácích úloh

Ke každému bloku patří sada 6 domácích úloh, které tvoří významnou část hodnocení předmětu. Na úspěšné odevzdání každé domácí úlohy budete mít 12 pokusů rozložených do 4 týdnů odpovídajícího bloku cvičení. Odevzdávání bude otevřeno vždy v 0:00 prvního dne bloku (tzn. 24h před prvním spuštěním verity testů).
Termíny odevzdání (vyhodnocení verity testů) jsou vždy v pondělí, středu a pátek v 23:59, dle následujícího harmonogramu:

sada	týden	pondělí	středa	pátek
1	1	20.2.	22.2.	24.2.
	2	27.2.	1.3.	3.3.
	3	6.3.	8.3.	10.3.
	4	13.3.	15.3.	17.3.
2	1	20.3.	12.3.	14.3.
	2	27.3.	29.3.	31.3.
	3	3.4.	5.4.	7.4.
	4	10.4.	12.4.	14.4.
3	1	17.4.	19.4.	21.4.
	2	24.4.	26.4.	28.4.
	3	1.5.	3.5.	5.5.
	4	8.5.	10.5.	12.5.

A.5.1 Odevzdávání Součástí každého zadání je jeden zdrojový soubor (kostra), do kterého své řešení vepíšete. Vypracované příklady lze pak odevzdávat stejně jako přípravy:

1. do odevzdávárny s návem `sN_úkol` v ISu (např. `s1_a_queens`),
2. příkazem `pb161 submit sN_úkol` ve složce `~/pb161/sN`, např. `pb161 submit s1_a_queens`.

Podrobnější instrukce naleznete opět v kapitole B.

A.5.2 Vyhodnocení Vyhodnocení Vašich řešení probíhá ve třech fázích, a s každou z nich je spjata sada automatických testů. Tyto sady jsou:

- „syntax" – kontroluje, že odevzdaný program je syntakticky správně, lze jej přeložit a prochází kontrolou programem `clang-tidy`,
- „sanity" – kontroluje, že odevzdaný program se chová „rozumně" na jednoduchých případech vstupu; tyto testy jsou rozsahem a stylem podobné těm, které máte přiložené k příkladům ve cvičení,
- „verity" – důkladně kontrolují správnost řešení, včetně složitých vstupů a okrajových případů, a kontroly paměťových chyb nástro-

jem `valgrind`.

Fáze na sebe navazují v tom smyslu, že nesplníte-li testy v některé fázi, žádná další se už (pro dané odevzdání) nespustí. Pro splnění domácí úlohy je klíčová fáze „verity", za kterou jsou Vám uděleny body. Časový plán vyhodnocení fází je následovný:

- kontrola „syntax" se provede obratem (do cca 5 minut od odevzdání),
- kontrola „sanity" každých 6 hodin počínaje půlnocí (tzn. 0:00, 6:00, 12:00, 18:00),
- kontrola „verity" se provede v pondělí, středu a pátek ve 23:59 (dle tabulky uvedené výše).

Vyhodnoceno je vždy pouze nejnovější odevzdání, a každé odevzdání je vyhodnoceno v každé fázi nejvýše jednou. Výsledky naleznete v poznámkových blocích v ISu (každá úloha v samostatném bloku), případně je získáte příkazem `pb161 status`.

A.5.3 Bodování Za každý domácí úkol, ve kterém Vaše odevzdání v příslušném termínu splní testy „verity", získáte 10 bodů.
Za stejný úkol máte dále možnost získat body za kvalitu kódu, a to vždy v hodnotě max. 5 bodů. Body za kvalitu se počítají v bloku, ve kterém byly uděleny, tzn. body za kvalitu ze sady 1 se započtou do bloku 2.
Maximální bodový zisk za jednotlivé sady:

- sada 1: 60 za funkčnost v bloku 1 + 30 za kvalitu v bloku 2,
- sada 2: 60 za funkčnost v bloku 2 + 30 za kvalitu v bloku 3,
- sada 3: 60 za funkčnost v bloku 3 + 30 za kvalitu v bloku 4 (zkouškovém).

A.5.4 Hodnocení kvality kódu Automatické testy ověřují správnost vašich programů (do takové míry, jak je to praktické – ani nejpřísnější testy nemůžou zaručit, že máte program zcela správně). Správnost ale není jediné kritérium, podle kterého lze programy hodnotit: podobně důležité je, aby byl program čitelný. Programy totiž mimo jiné slouží ke komunikaci myšlenek lidem – dobře napsaný a správně okomentovaný kód by měl čtenáři sdělit, jaký řeší problém, jak toto řešení funguje a u obojího objasnit proč.
Je Vám asi jasné, že čitelnost programu člověkem může hodnotit pouze člověk: proto si každý Váš úspěšně vyřešený domácí úkol přečte opravující a své postřehy Vám sdělí. Přitom zároveň Váš kód oznámkuje podle kritérií podrobněji rozepsaných v kapitole B. Tato kritéria aplikujeme při známkování takto:

- hodnocení A dostane takové řešení, které jasně popisuje řešení zadaného problému, je správně dekomponované na podproblémy, je zapsáno bez zbytečného opakování, a používá správné abstrakce, algoritmy a datové struktury,
- hodnocení B dostane program, který má výrazné nedostatky v jedné, nebo nezanedbatelné nedostatky ve dvou oblastech výše zmíněných, například:
  - je relativně dobře dekomponovaný a zbytečně se neopakuje, ale používá nevhodný algoritmus nebo datovou strukturu a není zapsán příliš přehledně,
  - používá optimální algoritmus a datové struktury a je dobře dekomponovaný, ale lokálně opakuje tentýž kód s drobnými obměnami, a občas používá zavádějící nebo jinak nevhodná jména podprogramů, proměnných atp.,
  - jinak dobrý program, který používá zcela nevhodný algoritmus, nebo velmi špatně pojmenované proměnné, nebo je zapsaný na dvě obrazovky úplně bez dekompozice,
- hodnocení X dostanou programy, u kterých jste se dobrovolně vzdali hodnocení (a to jasně formulovaným komentářem na začátku souboru, např. „Vzdávám se hodnocení."),
- hodnocení C dostanou všechny ostatní programy, zejména ty, které kombinují dvě a více výrazné chyby zmiňované výše.

Známky Vám budou zapsány druhou středu následujícího bloku. Dostanete-li známku B nebo C, budete mít možnost svoje řešení ještě zlepšit, odevzdat znovu, a známku si tak opravit:

- na opravu budete mít 9 dnů (od středy do dalšího pátku),
- na opraveném programu nesmí selhat verity testy,
- testy budou nadále probíhat se stejnou kadencí jako během řádné doby k vypracování (pondělí, středa, pátek o 23:59).

Bude-li opravující s vylepšeným programem spokojen, výslednou známku Vám upraví.

sada	řádný termín	známka	opravný termín	známka
1	17.3.	29.3.	7.4.	14.4.
2	14.4.	26.4.	5.5.	12.5.
3	12.5.	24.5.	2.6.	9.6.

Jednotlivé výsledné známky se promítnou do bodového hodnocení úkolu následovně:

- známka A Vám vynese 5 bodů,
- známka B pak 2 body,
- známka X žádné body neskýtá,
- známka C je hodnocena -1 bodem.

Samotné body za funkcionalitu se při opravě kvality již nijak nemění.

A.5.5 Neúspěšná řešení Příklady, které se Vám nepodaří vyřešit kompletně (tzn. tak, aby na nich uspěla kontrola „verity") nebudeme hodnotit. Nicméně může nastat situace, kdy byste potřebovali na „téměř hotové" řešení zpětnou vazbu, např. proto, že se Vám nepodařilo zjistit, proč nefunguje.
Taková řešení můžou být předmětem společné analýzy ve cvičení, v podobném duchu jako probíhá rozprava kolem odevzdaných příprav (samozřejmě až poté, co pro danou sadu skončí odevzdávání). Máte-li zájem takto rozebrat své řešení, domluvte se, ideálně s předstihem, se svým cvičícím. To, že jste autorem, zůstává mezi cvičícím a Vámi – Vaši spolužáci to nemusí vědět (ke kódu se samozřejmě můžete v rámci debaty přihlásit, uznáte-li to za vhodné). Stejná pravidla platí také pro nedořešené přípravy (musíte je ale odevzdat).
Tento mechanismus je omezen prostorem ve cvičení – nemůžeme zaručit, že v případě velkého zájmu dojde na všechny (v takovém případě cvičící vybere ta řešení, která bude považovat za přínosnější pro skupinu – je tedy možné, že i když se na Vaše konkrétní řešení nedostane, budete ve cvičení analyzovat podobný problém v řešení někoho jiného).

Část A.6: Vzájemné recenze

Jednou z možností, jak získat body za aktivitu, jsou vzájemné (peer) recenze. Smyslem této aktivity je získat praxi ve čtení a hodnocení cizího kódu. Možnost psát tyto recenze se váže na vlastní úspěšné vypracování téhož příkladu.
Příklad: odevzdáte-li ve druhém týdnu 4 přípravy, z toho u třech splníte testy „verity" (řekněme `p1`, `p2`, `p5`), ve třetím týdnu dostanete po jednom řešení těchto příkladů (tzn. budete mít možnost recenzovat po jedné instanci `02/p1`, `02/p2` a `02/p5`). Termín pro odevzdání recenzí na přípravy z druhé kapitoly je shodný s termínem pro odevzdání příprav třetí kapitoly (tzn. sobotní půlnoc).
Vypracování těchto recenzí je dobrovolné. Za každou vypracovanou recenzi získáte jeden bod za aktivitu, počítaný v týdnu, kdy jste recenze psali (v uvedeném příkladu by to tedy bylo ve třetím týdnu semestru, tedy do stejné „kolonky" jako body za příklady `02/r`).
Udělení bodů je podmíněno smysluplným obsahem – nestačí napsat „nemám co dodat" nebo „není zde co komentovat". Je-li řešení dobré, napište proč je dobré (viz též níže). Vámi odevzdané recenze si přečte Váš cvičící a některé z nich může vybrat k diskusi ve cvičení (v dalším týdnu), v podobném duchu jako přípravy samotné.
Pozor, v jednom týdnu lze získat maximálně 3 body za aktivitu, bez ohledu na jejich zdroj (recenze, vypracování příkladu u tabule, atp.).

Toto omezení není dotčeno ani v případě, kdy dostanete k vypracování více než 3 příklady (můžete si ale vybrat, které z nich chcete recenzovat).

A.6.1 Jak recenze psát Jak recenze vyzvednout a odevzdat je blíže popsáno v kapitole B. Své komentáře vkládejte přímo do vyzvednutých zdrojových souborů. Komentáře můžete psát česky (slovensky) nebo anglicky, volba je na Vás. Komentáře by měly být stručné, ale užitečné – Vaším hlavním cílem by mělo být pomoct adresátovi naučit se lépe programovat.
Snažte se aplikovat kritéria a doporučení z předchozí sekce (nejlépe na ně přímo odkázat, např. „tuto proměnnou by šlo jistě pojmenovat lépe (viz doporučení 2.b)"). Nebojte se ani vyzvednout pozitiva (můžete zde také odkázat doporučení, máte-li například za to, že je obzvlášť pěkně uplatněné) nebo poznamenat, když jste se při čtení kódu sami něco naučili.
Komentáře vkládejte vždy před komentovaný celek, a držte se podle možnosti tohoto vzoru (použití `**` pomáhá odlišit původní komentáře autora od poznámek recenzenta):

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

U víceřádkových komentářů:

```
/** 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. **/
```

Při vkládání komentářů neměňte existující řádky (zejména se ujistěte, že máte vypnuté automatické formátování, editujete-li zdrojový kód v nějakém IDE). Jediné povolená operace jsou:

- vložení nových řádků (prázdných nebo s komentářem), nebo
- doplnění komentáře na stávající prázdný řádek.

Část A.7: Zkouška

Zkouška tvoří pomyslný 4. blok a platí pro ni stejné kritérium jako pro všechny ostatní bloky: musíte získat alespoň 60 bodů. Zkouška:

- proběhne v počítačové učebně bez přístupu k internetu nebo vlastním materiálům,
- k dispozici budou oficiální studijní materiály:
  - tato sbírka (bez vzorových řešení příkladů typu `e` a `r`) a
  - offline kopie příručky `cppreference` (bez fulltextového vyhledávání),
- budete moct používat textový editor nebo vývojové prostředí VS Code, překladače `g++` a `clang`, nástroj `clang-tidy` a nástroje `valgrind` a `gdb`.

Na vypracování praktické části budete mít 4 hodiny čistého času, a bude sestávat ze šesti příkladů, které budou hodnoceny automatickými testy, s maximálním ziskem 90 bodů. Příklady jsou hodnoceny binárně (tzn. příklad je uznán za plný počet bodů, nebo uznán není). Kvalita kódu hodnocena nebude, ani nebudeme řešení kontrolovat nástrojem `clang-tidy`. Příklady budou na stejné úrovni obtížnosti jako příklady typu `p`/`r`/`v` ze sbírky.
Zkouška proběhne až po vyhodnocení recenzí za třetí blok (tzn. ve druhé polovině zkouškového období). Plánované termíny[6] jsou tyto (žádné další vypsané nebudou):

- úterý 13.6. 9:00–13:00, 14:00–18:00,
- úterý 20.6. 9:00–13:00, 14:00–18:00,
- úterý 27.6. 9:00–13:00, 14:00–18:00.

[6] Může se stát, že termíny budeme z technických nebo organizačních důvodů posunout na jiný den nebo hodinu. V takovém případě Vám samozřejmě změnu s dostatečným předstihem oznámíme.

A.7.1 Vnitrosemestrálky V posledním týdnu každého bloku, tedy

- týden 4 (13.-17. března),
- týdnu 8 (10.-14. dubna),
- týden 12 (8.-12. května),

proběhne v rámci cvičení programovací test na 40 minut. Tyto testy budou probíhat za stejných podmínek, jako výše popsaná praktická část zkoušky (slouží tedy mimo jiné jako příprava na ni). Řešit budete vždy ale pouze jeden příklad, za který můžete získat 3 body, které se počítají jako body za aktivitu v tomto cvičení.

Část A.8: Opisování

Na všech zadaných problémech pracujte prosím zcela samostatně – toto se týká jak příkladů ze sbírky, které budete odevzdávat, tak domácích úloh ze sad. To samozřejmě neznamená, že Vám zakazujeme společně studovat a vzájemně si pomáhat látku pochopit: k tomuto účelu můžete využít všechny zbývající příklady ve sbírce (tedy ty, které nebude ani jeden z Vás odevzdávat), a samozřejmě nepřeberné množství příkladů a cvičení, které jsou k dispozici online.
Příklady, které odevzdáváte, slouží ke kontrole, že látce skutečně rozumíte, a že dokážete nastudované principy prakticky aplikovat. Tato kontrola je pro Váš pokrok naprosto klíčová – je velice snadné získat pasivním studiem (čtením, posloucháním přednášek, studiem již vypracovaných příkladů) pocit, že něčemu rozumíte. Dokud ale sami nenapíšete na dané téma několik programů, jedná se pravděpodobně skutečně pouze o pocit.
Abyste nebyli ve zbytečném pokušení kontroly obcházet, nedovolenou spolupráci budeme relativně přísně trestat. Za každý prohřešek Vám bude strženo v každé instanci (jeden týden příprav se počítá jako jedna instance, příklady ze sad se počítají každý samostatně):

- 1/2 bodů získaných (ze všech příprav v dotčeném týdnu, nebo za jednotlivý příklad ze sady),
- 10 bodů z hodnocení bloku, do kterého opsaný příklad patří,
- 10 bodů (navíc k předchozím 10) z celkového hodnocení.

Opíšete-li tedy například 2 přípravy ve druhém týdnu a:

- Váš celkový zisk za přípravy v tomto týdnu je 4,5 bodu,
- Váš celkový zisk za první blok je 65 bodů,

jste automaticky hodnoceni známkou X (65 - 2,25 - 10 je méně než potřebných 60 bodů). Podobně s příkladem z první sady (65 - 5 - 10), atd. Máte-li v bloku bodů dostatek (např. 80 - 5 - 10 > 60), ve studiu předmětu pokračujete, ale započte se Vám ještě navíc penalizace 10 bodů do celkové známky. Přestává pro Vás proto platit pravidlo, že 4 splněné bloky jsou automaticky E nebo lepší.
V situaci, kdy:

- za bloky máte před penalizací 77, 62, 61, 64,
- v prvním bloku jste opsali domácí úkol,

budete penalizováni:

- v prvním bloku 10 + 5, tzn. bodové zisky za bloky budou efektivně 62, 62, 61, 64,
- v celkovém hodnocení 10, tzn. celkový zisk 62 + 62 + 61 + 64 - 10 = 239, a budete tedy hodnoceni známkou F.

To, jestli jste příklad řešili společně, nebo jej někdo vyřešil samostatně, a poté poskytl své řešení někomu dalšímu, není pro účely kontroly opisování důležité. Všechny „verze" řešení odvozené ze společného základu budou penalizovány stejně. Taktéž zveřejnění řešení budeme chápat jako pokus o podvod, a budeme jej trestat, bez ohledu na to, jestli někdo stejné řešení odevzdá, nebo nikoliv.
Podotýkáme ještě, že kontrola opisování nespadá do desetidenní lhůty pro hodnocení průběžných kontrol. Budeme se sice snažit opisování kontrolovat co nejdříve, ale odevzdáte-li opsaný příklad, můžete být bodově penalizováni kdykoliv (tedy i dodatečně, a to až do konce zkouš-

kového období).

Část B: Doporučení k zápisu kódu

Tato sekce rozvádí obecné principy zápisu kódu s důrazem na čitelnost a korektnost. Samozřejmě žádná sada pravidel nemůže zaručit, že napíšete dobrý (korektní a čitelný) program, o nic více, než může zaručit, že napíšete dobrou povídku nebo namalujete dobrý obraz. Přesto ve všech těchto případech pravidla existují a jejich dodržování má obvykle na výsledek pozitivní dopad.

Každé pravidlo má samozřejmě nějaké výjimky. Tyto jsou ale výjimkami proto, že nastávají výjimečně. Některá pravidla připouští výjimky častěji než jiná:

1 Dekompozice Vůbec nejdůležitější úlohou programátora je rozdělit problém tak, aby byl schopen každou část správně vyřešit a dílčí výsledky pak poskládat do korektního celku.

A. Kód musí být rozdělen do ucelených jednotek (kde jednotkou rozumíme funkci, typ, modul, atd.) přiměřené velikosti, které lze studovat a používat nezávisle na sobě.
B. Jednotky musí být od sebe odděleny jasným rozhraním, které by mělo být jednodušší a uchopitelnější, než kdybychom použití jednotky nahradili její definicí.
C. Každá jednotka by měla mít jeden dobře definovaný účel, který je zachycený především v jejím pojmenování a případně rozvedený v komentáři.
D. Máte-li problém jednotku dobře pojmenovat, může to být známka toho, že dělá příliš mnoho věcí.
E. Jednotka by měla realizovat vhodnou abstrakci, tzn. měla by být obecná – zkuste si představit, že dostanete k řešení nějaký jiný (ale dostatečně příbuzný) problém: bude Vám tato konkrétní jednotka k něčemu dobrá, aniž byste ji museli (výrazně) upravovat?
F. Má-li jednotka parametr, který fakticky identifikuje místo ve kterém ji používáte (bez ohledu na to, je-li to z jeho názvu patrné), je to často známka špatně zvolené abstrakce. Máte-li parametr, který by bylo lze pojmenovat `called_from_bar`, je to jasná známka tohoto problému.
G. Daný podproblém by měl být vyřešen v programu pouze jednou – nedaří-li se Vám sjednotit různé varianty stejného nebo velmi podobného kódu (aniž byste se uchýlili k taktice z bodu d), může to být známka nesprávně zvolené dekompozice. Zkuste se zamyslet, není-li možné problém rozložit na podproblémy jinak.

2 Jména Dobře zvolená jména velmi ulehčují čtení kódu, ale jsou i dobrým vodítkem při dekompozici a výstavbě abstrakcí.

A. Všechny entity ve zdrojovém kódu nesou anglická jména. Angličtina je univerzální jazyk programátorů.
B. Jméno musí být výstižné a popisné: v místě použití je obvykle jméno náš hlavní (a často jediný) zdroj informací o jmenované entitě. Nutnost hledat deklaraci nebo definici (protože ze jména není jasné, co volaná funkce dělá, nebo jaký má použitá proměnná význam) čtenáře nesmírně zdržuje.[7]
C. Jména lokálního významu mohou být méně informativní: je mnohem větší šance, že význam jmenované entity si pamatujeme, protože byla definována před chvílí (např. lokální proměnná v krátké funkci).
D. Obecněji, informační obsah jména by měl být přímo úměrný jeho rozsahu platnosti a nepřímo úměrný frekvenci použití: globální jméno musí být informativní, protože jeho definice je „daleko“ (takže si ji už nepamatujeme) a zároveň se nepoužívá příliš často (takže si nepamatujeme ani to, co jsme se dozvěděli, když jsme ho potkali naposled).
E. Jméno parametru má dvojí funkci: krom toho, že ho používáme v těle funkce (kde se z pohledu pojmenování chová podobně jako lokální proměnná), slouží jako dokumentace funkce jako celku. Pro parametry volíme popisnější jména, než by zaručovalo jejich použití ve funkci samotné – mají totiž dodatečný globální význam.
F. Některé entity mají ustálené názvy – je rozumné se jich držet, protože čtenář automaticky rozumí jejich významu, i přes obvyklou stručnost. Zároveň je potřeba se vyvarovat použití takovýchto ustálených jmen pro nesouvisející entity. Typickým příkladem jsou iterační proměnné `i` a `j`.
G. Jména s velkým rozsahem platnosti by měla být také zapamatovatelná. Je vždy lepší si přímo vzpomenout na jméno funkce, kterou právě potřebuji, než ho vyhledávat (podobně jako je lepší znát slovo, než ho jít hledat ve slovníku).
H. Použitý slovní druh by měl odpovídat druhu entity, kterou pojmenovává. Proměnné a typy pojmenováváme přednostně podstatnými jmény, funkce přednostně slovesy.
I. Rodiny příbuzných nebo souvisejících entit pojmenováváme podle společného schématu (`table_name`, `table_size`, `table_items` – nikoliv např. `items_in_table`; `list_parser`, `string_parser`, `set_parser`; `find_min`, `find_max`, `erase_max` – nikoliv např. `erase_maximum` nebo `erase_greatest` nebo `max_remove`).
J. Jména by měla brát do úvahy kontext, ve kterém jsou platná. Neopakujte typ proměnné v jejím názvu (`cars`, nikoliv `list_of_cars` ani `set_of_cars`) nemá-li tento typ speciální význam. Podobně jméno nadřazeného typu nepatří do jmen jeho metod (třída `list` by měla mít metodu `length`, nikoliv `list_length`).
K. Dávejte si pozor na překlepy a pravopisné chyby. Zbytečně znesnadňují pochopení a (zejména v kombinaci s našeptávačem) lehce vedou na skutečné chyby způsobené záměnou podobných ale jinak napsaných jmen. Navíc kód s překlepy v názvech působí značně neprofesionálně.

3 Stav a data Udržet si přehled o tom, co se v programu děje, jaké jsou vztahy mezi různými stavovými proměnnými, co může a co nemůže nastat, je jedna z nejtěžších částí programování.
TBD: Vstupní podmínky, invarianty, …

4 Řízení toku Přehledný, logický a co nejvíce lineární sled kroků nám ulehčuje pochopení algoritmu. Časté, komplikované větvení je naopak těžké sledovat a odvádí pozornost od pochopení důležitých myšlenek.
TBD.

5 Volba algoritmů a datových struktur TBD.

6 Komentáře Nejde-li myšlenku předat jinak, vysvětlíme ji doprovodným komentářem. Čím těžší myšlenka, tím větší je potřeba komentovat.

A. Podobně jako jména entit, komentáře které jsou součástí kódu píšeme anglicky.[8]
B. Případný komentář jednotky kódu by měl vysvětlit především „co“ a „proč“ (tzn. jaký plní tato jednotka účel a za jakých okolností ji lze použít).

[7] Nejde zde pouze o samotný fakt, že je potřeba něco vyhledat. Mohlo by se zdát, že tento problém řeší IDE, které nás umí „poslat“ na příslušnou definici samo. Hlavní zdržení ve skutečnosti spočívá v tom, že musíme přerušit čtení předchozího celku. Na rozdíl od počítače je pro člověka „zanořování“ a zejména pak „vynořování“ na pomyslném zásobníku docela drahou operací.

[8] Tato sbírka samotná představuje ústupek z tohoto pravidla: smyslem našich komentářů je naučit Vás poměrně těžké a často nové koncepty, a její cirkulace je omezená. Zkušenost z dřívějších let ukazuje, že pro studenty je anglický výklad značnou bariérou pochopení. Přesto se snažte vlastní kód komentovat anglicky – výjimku lze udělat pouze pro rozsáhlejší komentáře, které byste jinak nedokázali srozumitelně formulovat. V praxi je angličtina zcela běžně bezpodmínečně vyžadovaná.

C. Komentář by také neměl zbytečně duplikovat informace, které jsou k nalezení v hlavičce nebo jiné „nekomentářové“ části kódu – jestli máte například potřebu komentovat parametr funkce, zvažte, jestli by nešlo tento parametr lépe pojmenovat nebo otypovat.
D. Komentář by neměl zbytečně duplikovat samotný spustitelný kód (tzn. neměl by se zdlouhavě zabývat tím „jak“ jednotka vnitřně pracuje). Zejména jsou nevhodné komentáře typu „zvýšíme proměnnou i o jedna“ – komentář lze použít k vysvětlení proč je tato operace potřebná – co daná operace dělá si může každý přečíst v samotném kódu.

7 Formální úprava TBD.

Část 1: Hodnoty a funkce

Vítejte v PB161. Než budete pokračovat, přečtěte si prosím kapitolu A (složku `00` ve zdrojovém balíku). Podrobnější informace jak se soubory v této složce pracovat naleznete v souboru `00/t3_sources.txt`, resp. v sekci T.3.
Cvičení bude tematicky sledovat přednášku z předchozího týdne: první kapitola tak odpovídá první přednášce. Tématy pro tento týden jsou funkce, řízení toku, skalární hodnoty a reference. Tyto koncepty jsou krom přednášky demonstrovány v příkladech typu `d` (ukázkách; naleznete je také v souborech `d?_*.cpp`, např. `d1_fibonacci.cpp`).

1. `fibonacci` – iterativní výpočet Fibonacciho čísel,
2. `comb` – výpočet kombinačního čísla,
3. `hamming` – hammingova vzdálenost dvojkového zápisu,
4. `root` † – výpočet n-té celočíselné odmocniny.

Ve druhé části každé kapitoly pak naleznete tzv. elementární cvičení, která byste měli být schopni relativně rychle vyřešit a ověřit si tak, že jste porozuměli konceptům z přednášky a ukázek. Tyto příklady naleznete v souborech pojmenovaných `e?_*.cpp`. Řešení můžete vepsat přímo do nachystaného souboru se zadáním. Základní testy jsou součástí zadání.
Vzorová řešení těchto příkladů naleznete v kapitole K (klíč) na konci sbírky, resp. ve složce `sol` zdrojového balíku. Mějte na paměti, že přiložená vzorová řešení nemusí být vždy nejjednodušší možná. Také není nutné, aby Vaše řešení přesně odpovídalo tomu vzorovému, nebo bylo založeno na stejném principu. Důležité je, aby pracovalo správně a dodržovalo požadovanou (resp. adekvátní) složitost.
Elementární příklady první kapitoly jsou:

1. `factorial` – spočtěte faktoriál zadaného čísla,
2. `concat` – zřetězení binárního zápisu dvou čísel,
3. `zeros` – počet nul v zápisu čísla.

V další části naleznete o něco složitější příklady, tzv. přípravy. Jejich hlavním účelem je samostatně procvičit látku dané kapitoly, a to ještě předtím, než se o ní budeme bavit ve cvičení. Doporučujeme každý týden vyřešit alespoň 3 přípravy. Abyste byli motivovaní je řešit, odevzdaná řešení jsou bodována (detaily bodování a termíny odevzdání naleznete v kapitole A). Ve zdrojovém balíku se jedná o soubory s názvem `p?_*.cpp`.
Pozor: Diskutovat a sdílet řešení příprav je přísně zakázáno. Řešení musíte vypracovat zcela samostatně (bližší informace naleznete opět v kapitole A).
Přípravy:

1. `nhamming` – Hammingova vzdálenost s libovolným základem,
2. `digitsum` – opakovaný ciferný součet,
3. `parity` – počet jedniček v binárním zápisu,
4. `periodic` – hledání nejkratšího periodického vzoru,
5. `balanced` – ciferné součty ve vyvážených soustavách,
6. `subsetsum` – známý příklad na backtracking.

Další část je tvořena rozšířenými úlohami, které jsou typicky o něco málo složitější, než přípravy. Na těchto úlohách budeme probranou látku dále procvičovat ve cvičení. Tyto úlohy můžete také řešit společně, diskutovat jejich řešení se spolužáky, atp. Svá řešení můžete také srovnat s těmi vzorovými, která jsou k nalezení opět v kapitole K. Tento typ úloh naleznete v souborech pojmenovaných `r?_*.cpp`.

1. `bitwise` – ternární bitové operátory,
2. `euler` – Eulerova funkce (počet nesoudělných čísel),
3. `hamcode` – kód pro detekci chyb Hamming(8,4),
4. `cbc` – cipher block chaining,
5. `cellular` – celulární automat nad celým číslem,
6. `flood` – vyplňování „ploch“ v celém čísle.

Poslední část jsou tzv. volitelné úkoly, které se podobají těm rozšířeným, se dvěma důležitými rozdíly: volitelné úlohy jsou určeny k samostatné přípravě (nebudeme je tedy používat ve cvičení) a nejsou k nim dostupná vzorová řešení. Je totiž důležité, abyste si dokázali sami zdůvodnit a otestovat správnost řešení, aniž byste jej srovnávali s řešením někoho jiného (a přiložený vzor k tomu jistě svádí). Je nicméně povoleno tyto příklady (a jejich řešení, jak abstraktně, tak konkrétně) diskutovat se spolužáky. Přesto velmi důrazně doporučujeme, abyste si řešení zkusili prvně vypracovat sami.

1. `xxx` – …
2. `xxx` – …
3. `xxx` – …

1 Hlavičkové soubory Samotný jazyk, který ve svých řešeních používáte, omezujeme jen minimálně (varování překladače a kontrola nástrojem `clang-tidy` ovšem některé obzvláště problémové konstrukce zamítnou). Trochu významnější omezení klademe na používání standardní knihovny: do svých odevzdaných programů prosím vkládejte pouze ty standardní hlavičky, kterých použití jsme již v předmětu zavedli. Přehled bude vždy uveden v úvodu příslušné kapitoly. Pro tu první jsou to tyto tři:

- `cassert` – umožňuje použití tvrzení `assert`,
- `algorithm` – nabízí funkce `std::min`, `std::max`,
- `cstdint` – celočíselné typy `std::intNN_t` a `std::uintNN_t`.

Omezeno je pouze vkládání hlavičkových souborů: je-li povolena hlavička `algorithm`, můžete v principu používat i jiné algoritmy, které poskytuje. Přesto spíše doporučujeme držet se toho, co jsme Vám zatím ukázali.
Na nic jiného, než vkládání standardních hlaviček, v tomto předmětu preprocesor potřebovat nebudete. Jiné direktivy než `#include` tedy prosím vůbec nepoužívejte.

Část 1.d: Demonstrace (ukázky)

1.d.1 [`fibonacci`] V této ukázce naprogramujeme klasický ukázkový algoritmus, totiž výpočet n-tého Fibonacciho čísla (a použijeme k tomu iterativní algoritmus). Algoritmus bude implementovat podprogram (funkce) `fibonacci`. Definice podprogramu se v jazyce C++ začíná tzv. signaturou neboli hlavičkou funkce, která:

1. popisuje návratovou hodnotu (zejména její typ),
2. udává název podprogramu a
3. jeho formální parametry, opět zejména jejich typy, ale obvykle i názvy.

Signatura může popisovat i další vlastnosti, se kterými se setkáme později.
V tomto případě bude návratovou hodnotou celé číslo (znaménkového typu `int`), podprogram ponese název `fibonacci` a má jeden parametr, opět celočíselného typu `int`.

```
int fibonacci( int n )
```

Po signatuře následuje tzv. tělo, které je syntakticky shodné se složeným příkazem, a je tedy tvořeno libovolným počtem (včetně nuly) příkazů uzavřených do složených závorek. V těle funkce jsou formální parametry (v tomto případě n) ekvivalentní lokálním proměnným pomyslně inicializovaným hodnotou skutečného parametru.

```
{
```

Tělo je tvořeno posloupností příkazů (typický příkaz je ukončen středníkem, ale toto neplatí např. pro složené příkazy, které jsou ukončeny složenou závorkou).
Prvním příkazem podprogramu fibonacci je deklarace lokálních proměnných a, b, opět celočíselného typu int. Deklarace se skládá z:

1. typu, případně klíčového slova auto,
2. neprázdného seznamu deklarovaných jmen (oddělených čárkou), které mohou být doplněny tzv. deklarátory (označují např. reference: uvidíme je v pozdější ukázce),
3. volitelného inicializátoru, který popisuje počáteční hodnotu proměnné.

```
    int a = 1, b = 1, c;
```

Samotný výpočet zapíšeme pomocí tzv. třídílného for cyklu (jinou variantu cyklu for si ukážeme v další kapitole), který má následující strukturu:

1. klíčové slovo for,
2. hlavička cyklu, uzavřená v kulatých závorkách,
   a. inicializační příkaz (výraz, deklarace proměnné, nebo prázdný příkaz) je vždy ukončen středníkem a provede se jednou před začátkem cyklu; deklaruje-li proměnné, tyto jsou platné právě po dobu vykonávání cyklu,
   b. podmínka cyklu (výraz nebo prázdný příkaz) je opět vždy ukončena středníkem a určuje, zda se má provést další iterace cyklu (vyhodnotí-li se na true),
   c. výraz iterace (výraz, který není ukončen středníkem), který je vyhodnocen vždy na konci těla (před dalším vyhodnocením podmínky cyklu),
3. tělo cyklu (libovolný příkaz, často složený).

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

V jazyce C++ je přiřazení výraz, kterého vyhodnocení má vedlejší efekt, a to konkrétně změnu proměnné, která je odkazována levou stranou operátoru = (jedná se o výraz, který se musí vyhodnotit na tzv. l-hodnotu[9] – l od left, protože stojí na levé straně přiřazení). Na pravé straně pak stojí libovolný výraz.

```
        c = a + b;
        a = b;
        b = c;
    }
```

Příkaz návratu z podprogramu return má dvojí význam (podobně jako ve většině imperativních jazyků):

1. určí návratovou hodnotu podprogramu (tato se získá vyhodnocením výrazu uvedeného po klíčovém slově return),
2. ukončí vykonávání podprogramu a předá řízení volajícímu.

```
    return b;
```

[9] Zjednodušeně, l-hodnota je takový výraz, který popisuje identitu resp. lokaci – typicky proměnnou, která je uložena v paměti. L-hodnoty rozlišujeme proto, že smyslem přiřazení je uložit (zapsat) výsledek své pravé strany, a na levé straně tedy musí stát objekt, do kterého lze tuto pravou stranu skutečně zapsat. Nejjednodušší l-hodnotou je název proměnné.

```
}
```

Všechny ukázky v této sbírce obsahují několik jednoduchých testovacích případů, kterých účelem je jednak předvést, jak lze implementovanou funkcionalitu použít, jednak ověřit, že fungování programu odpovídá naší představě. Zkuste si přiložené testy různě upravovat, abyste si ověřili, že dobře rozumíte tomu, jak ukázka funguje.

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

Použití (volání) podprogramu je výraz a jeho vyhodnocení odpovídá naší intuitivní představě: skutečné parametry (uvedené v kulatých závorkách za jménem) se použijí jako pomyslné inicializátory formálních parametrů a s takto inicializovanými parametry se vykoná tělo podprogramu. Po jeho ukončení se výraz volání podprogramu vyhodnotí na návratovou hodnotu.

```
    assert( fibonacci( 1 ) == 1 );
    assert( fibonacci( 2 ) == 1 );
    assert( fibonacci( 7 ) == 13 );
    assert( fibonacci( 20 ) == 6765 );
}
```

1.d.2 [comb] V této ukázce se zaměříme na vlastnosti celočíselných typů. Podíváme se přitom na kombinační čísla, definovaná jako:

$$(n \mid k) = n!/(k! \cdot (n-k)!)$$

kde $k \le n$. Samozřejmě, mohli bychom počítat kombinační čísla přímo z definice, má to ale jeden důležitý problém: celočíselné proměnné mají v C++ pevný rozsah. Výpočet mezivýsledku $n!$ tak může velmi lehce překročit horní hranici použitého typu, a to i v případech, kdy celkový výsledek není problém reprezentovat.
Proto je důležité najít formu výpočtu, která nebude vytvářet zbytečně velké mezivýsledky. Výpočet kombinačního čísla lze navíc provádět na libovolném celočíselném typu (včetně těch bezznaménkových), proto čistou funkci comb definujeme tak, aby fungovala pro všechny takové typy.
Parametr uvedený klíčovým slovem auto může být libovolného typu (použití funkce s takovým typem parametru, pro který tělo funkce není typově správné, překladač zamítne). Něco jiného znamená návratová hodnota deklarovaná jako auto: tento typ se odvodí z příkazů return v těle funkce. Chceme-li toto tzv. odvození návratového typu využít, musí mít výraz u všech příkazů return v těle stejný typ.

```
auto comb( auto n, auto k )
{
```

Kombinační čísla jsou definovaná pouze pro $n \ge k$ a tuto vstupní podmínku si můžeme lehce ověřit tvrzením:

```
    assert( n >= k );
```

Výpočet budeme provádět na stejném typu, jaký má vstupní n. Protože tento typ neznáme, musíme si pomoct konstrukcí decltype, která nám umožní vytvořit proměnnou stejného typu, jako nějaká existující. Pozor! Je-li původní proměnná referencí, bude i nová proměnná referencí. Pozor! Nemůžeme zde použít `auto result = 1`. Proč?

```
    decltype( n ) result = 1;
```

Jak jistě víte, faktoriál je definován takto:

$$n! = \prod_{i=1}^{n} i$$

A tedy:

$$n!/k! = \prod_{i=1}^{n} i / \prod_{i=1}^{k} i = \prod_{i=k+1}^{n} i$$

Tento výpočet bychom jednoduše zapsali do for cyklu v příslušných mezích. Ve skutečnosti ale můžeme výpočet ještě znatelně zlepšit. Klíčové pozorování je, že ani zbývající $(n-k)!$ není potřeba vyčíslovat. Víme jistě, že výsledek bude celé číslo, tzn. všechny faktory $(n-k)!$ se musí pokrátit s nějakými faktory $n!/k!$. Jedna možnost je seřadit faktory čitatele sestupně a faktory jmenovatele vzestupně a mezivýsledek střídavě násobit a dělit příslušným faktorem (celočíselnost mezivýsledků je zde zaručena tím, že jsou to opět kombinační čísla, jak lze nahlédnout např. rozšířením příslušných zlomků vhodným faktoriálem).
Toto řešení je optimální v počtu aritmetických operací, není ale optimální ve velikosti mezivýsledku. Přesněji, je-li $\binom{n}{h}$ největší kombinační číslo s daným n, největší mezivýsledek při výpočtu $\binom{n}{k}$ bude $h \cdot \binom{n}{h}$. Využijeme-li navíc symetrie $\binom{n}{k} = \binom{n}{n-k}$, můžeme tuto mez zlepšit na $k \cdot \binom{n}{k}$ a zároveň zabezpečit, že $k \leq h$. Je nicméně zřejmé, že výpočet nám může přetéct i v případě, kdy celkový výsledek reprezentovat lze.
Proměnné nom_f a denom_f budou reprezentovat aktuální faktory v čitateli a jmenovateli. Opět budou stejného typu jako vstupní n.

```
    decltype( n ) nom_f = n, denom_f = 1;
```

Cyklus provedeme pro nom_f v klesajícím rozsahu $\langle n, k)$ resp. $\langle n, n-k)$, podle toho která spodní mez je větší. Protože jednotlivé mezihodnoty na spodní hranici iterace nezávisí, je jistě výhodnější provést méně iterací.

```
    while ( nom_f > std::max( k, n - k ) )
    {
```

Máme-li cyklus zapsaný správně, faktor jmenovatele nemůže překročit menší z hodnot k nebo $n-k$. O tomto se opět ujistíme tvrzením.

```
        assert( denom_f <= std::min( k, n - k ) );
```

Dále provedeme samotný krok výpočtu. Tvrzením se ujistíme, že provádíme skutečně pouze celočíselná dělení beze zbytku (kdyby tomu tak nebylo, výpočet by byl nesprávný!).

```
        result *= nom_f;
        assert( result % denom_f == 0 );
        result /= denom_f;
```

Nakonec upravíme iterační proměnné a pokračujeme další iterací.

```
        --nom_f;
        ++denom_f;
    }

    return result;
}

int main() /* demo */
{
    assert( comb( 1, 1 ) == 1 );
    assert( comb( 2, 1 ) == 2 );
    assert( comb( 5, 2 ) == 10 );
```

Postup implementovaný podprogramem comb nám umožňuje spočítat, za pomoci 64-bitových proměnných, všechna kombinační čísla pro $n \leq 60$, a to i přesto, že nejen $60! \approx 1,1 \cdot 2^{272}$, ale $60!/30! \approx 1,34 \cdot 2^{164}$ a tedy ani toto mnohem menší číslo se do 64-bitové proměnné v žádném případě nevejde.
Poznámka: typ std::int64_t je právě 64-bitový celočíselný typ se znaménkem. Abychom ho zde mohli použít, museli jsme výše vložit hlavičku cstdint.

```
    for ( std::int64_t i = 1; i < 60; ++i )
        for ( std::int64_t k = 1; k < i; ++k )
            assert( comb( i + 1, k + 1 ) ==
                    comb( i, k ) + comb( i, k + 1 ) );
    return 0;
}
```

1.d.3 [hamming] Tato ukázka přináší výstupní parametry a s nimi reference. Naším úkolem bude naprogramovat podprogram hamming, který spočítá tzv. Hammingovu vzdálenost dvou nezáporných čísel a, b. Hammingova vzdálenost se tradičně definuje jako počet znaků, ve kterých se vstupní hodnoty liší.
Abychom tedy mohli mluvit o vzdálenosti čísel, musíme je nějak zapsat – v této ukázce k tomu zvolíme dvojkovou soustavu. Protože Hammingova vzdálenost je navíc definovaná pouze pro stejně dlouhá slova, je-li některý dvojkový zápis kratší, doplníme ho pro účely výpočtu levostrannými nulami.
Krom samotné vzdálenosti nás bude zajímat také řád nejvyšší číslice, ve které se vstupní čísla liší (rozmyslete si, že taková existuje právě tehdy, je-li výsledná vzdálenost nenulová). Pro tento dodatečný výsledek (který navíc nemusí být vždy definovaný) použijeme již zmiňovaný výstupní parametr. V případech, kdy definovaný není, nebudeme hodnotu výstupního parametru měnit.
Výstupní parametr realizujeme referencí, kterou zapíšeme za pomoci deklarátoru & – reference na celočíselnou hodnotu typu int bude tedy int &.[10] Takto deklarovaná reference zavádí nové jméno pro již existující objekt. Objeví-li se tedy reference ve formálním parametru, takto zavedené jméno se přímo váže k hodnotě skutečného parametru.
Pro tento skutečný parametr platí stejná omezení, jako pro levou stranu přiřazení (musí tedy být l-hodnotou). Je to proto, že takto zavedený parametr je pouze novým jménem pro skutečný parametr. Zejména tedy platí, že kdykoliv se formální parametr objeví na levé straně přiřazení, toto přiřazení má efekt na skutečný parametr. Díky tomu můžeme uvnitř těla změnit hodnotu skutečného parametru. Je-li tedy skutečný parametr např. jméno lokální proměnné ve volající funkci, hodnota této proměnné se může provedením volané funkce změnit. Rozmyslete si, že u běžných parametrů (které nejsou referencemi) tomu tak není.

```
int hamming( auto a, auto b, int &order )
{
```

Jako obvykle nejprve ověříme vstupní podmínku.

```
    assert( a >= 0 && b >= 0 );
```

Protože pracujeme s dvojkovou reprezentací, můžeme si výpočet zjednodušit použitím vhodných bitových operací. Operátor ^ (xor, exclusive or) nastaví na jedničku právě ty bity výsledku, ve kterých se jeho operandy liší. Hledaná Hammingova vzdálenost je tedy právě počet jedniček v binární reprezentaci čísla x.
Všimněte si, že pro lokální proměnnou x neuvádíme typ – podobně jako v deklaraci parametrů a, b jsme zde použili zástupné slovo auto. Typ takto deklarované proměnné se odvodí z jejího inicializátoru (v tomto případě a ^ b). Na rozdíl od konstrukce decltype je takto deklarovaná proměnná vždy hodnotou, i v případě, že pravá strana je referenčního typu.[11]

```
    auto x = a ^ b;
```

Pro výsledek si zavedeme pomocnou proměnnou result, do které sečteme počet nenulových bitů. Pozor! proměnné jednoduchých typů je nutné inicializovat i v případě, že má být jejich počáteční hodnota nulová. Bez inicializátoru vznikne neinicializovaná proměnná kterou

[10] To, že se jedná o referenci, je součástí typu takto zavedené proměnné (resp. parametru) – projeví se to např. při použití konstrukce decltype. Zároveň ale platí, že ve většině případů jsou reference a hodnoty záměnné: je to mimo jiné proto, že na výsledek libovolného výrazu lze nahlížet jako na určitý druh reference. Blíže se budeme referencemi zabývat ve čtvrté kapitole.

[11] To opět souvisí s tím, že každý výraz lze interpretovat jako referenci. Chování tohoto typu deklarace je uzpůsobeno tomu, že obvykle chceme deklarovat lokální proměnné – lokální reference jsou mnohem vzácnější.

je zakázáno číst (níže použitý operátor ++ „zvětši hodnotu o jedna" samozřejmě svůj operand přečíst musí).

```
    int result = 0;
```

Číslo, které obsahuje alespoň jeden nenulový bit je jistě nenulové – cyklus se tedy bude provádět tak dlouho, dokud jsou v čísle x nenulové bity.

```
    for ( int i = 0; x != 0; ++i, x >>= 1 )
```

V těle cyklu budeme zkoumat nejnižší bit hodnoty x, přitom proměnná i obsahuje jeho původní řád. Všimněte si, že for cyklus je poměrně flexibilní, a že je důležité si jeho hlavičku dobře přečíst: v tomto případě se např. proměnná i vůbec neobjevuje v podmínce.
Naopak výraz iterace má dvě části (oddělené operátorem čárka, který vyhodnotí svůj první operand pouze pro jeho vedlejší efekt – jeho hodnotu zapomene). Efekt na i je celkem zřejmý, zajímavější je efekt na x: výraz x >>= 1 provede bitový posun proměnné x o jeden bit doprava. Původní druhý nejnižší bit se tak stane nejnižším, atd., až nejvyšší bit se doplní nulou. Příklad: posunem osmibitové hodnoty 10011001 o jednu pozici doprava vznikne hodnota 01001100.
Celý cyklus bychom samozřejmě mohli zapsat jako while cyklus a vyhnuli bychom se tím relativně komplikované hlavičce. Výhodou cyklu for v tomto případě ale je, že veškeré informace o změnách iteračních proměnných jsou uvedeny na jeho začátku. Čtenář tak už od začátku ví, jaký mají tyto proměnné význam (i se každou iterací zvýší o jedna, x se bitově posune o jednu pozici doprava) a nemusí tuto informaci zdlouhavě hledat v těle.

```
    {
```

V každé iteraci tak budeme zkoumat nejnižší bit hodnoty x (který odpovídá i-tému bitu vstupních parametrů a, b). S výhodou k tomu použijeme operaci bitové konjunkce (and; na jedničku jsou nastaveny právě ty bity výsledku, které mají hodnotu 1 v obou operandech). Tento operátor zapisujeme znakem & (pozor, nezaměňujte s deklarátorem reference!).

```
        if ( x & 1 )
        {
```

Je-li nejnižší bit nastavený, zvýšíme hodnotu proměnné result a do výstupního parametru order zapíšeme jeho původní řád (který si udržujeme v proměnné i).
Protože bity procházíme v pořadí od nejnižšího k nejvyššímu, poslední zápis do parametru order přesně odpovídá nejvyššímu rozdílnému bitu. Podmínka cyklu nám navíc zaručuje, že do proměnné order zapíšeme pouze v případě, že takový bit existuje.

```
            ++result;
            order = i;
        }
    }
```

Po ukončení cyklu platí, že jsme zpracovali všechny nenulové bity x, a tedy všechny bity, ve kterých se hodnoty a, b lišily. Nezbývá, než nastavit návratovou hodnotu a podprogram ukončit.

```
    return result;
```

Všimneme si ještě, že hodnotu parametru order jsme nečetli. Definující vlastností výstupních parametrů je, že chování podprogramu nezávisí na jejich počáteční hodnotě. V případě, že jediná operace, kterou s výstupním parametrem provedeme, je přiřazení do něj, je tento požadavek triviálně splněn.

```
}

int main() /* demo */
{
    int order = 3;
```

Protože skutečný parametr order předáváme referencí (odpovídající formální parametr je referenčního typu), změny, které v něm podprogram hamming provede, jsou viditelné i navenek. Nejprve ovšem ověříme, že při nulové vzdálenosti se hodnota order nemění.

```
    assert( hamming( 0, 0, order ) == 0 && order == 3 );
    assert( hamming( 1, 1, order ) == 0 && order == 3 );
```

Hodnoty v dalším příkladě se liší ve dvou bitech (osmém a devátém) a proto očekáváme, že po provedení funkce hamming bude mít proměnné order hodnotu 9.

```
    assert( hamming( 512, 256, order ) == 2 && order == 9 );
    assert( hamming( 0, 1, order ) == 1 && order == 0 );

    assert( hamming( 0xffffff, 0, order ) == 24 );
    assert( hamming( 0xffffffffff, 0, order ) == 40 );
    assert( hamming( 0xf000000000, 0xf, order ) == 8 );
    assert( hamming( 0xf000000000, 0xe000000000, order ) == 1 );

    return 0;
}
```

1.d.4 [root] † V této poslední ukázce bude naším cílem spočítat celočíselnou n-tou odmocninu zadaného nezáporného čísla k. Nejprve ale budeme potřebovat dvě pomocné funkce: celočíselný dvojkový logaritmus a n-tou mocninu. Vyzbrojeni těmito funkcemi pak budeme schopni odmocninu vypočítat tzv. Newtonovou-Raphsonovou metodou.
Celočíselný dvojkový logaritmus čísla n definujeme jako největší celé číslo k takové, že $2^k \leq n$. Za povšimnutí stojí, že pro $n < 1$ takové k neexistuje – proto tuto funkci definujeme pouze pro kladná n.

```
auto int_log2( auto n )
{
```

Jako obvykle nejprve ověříme vstupní podmínku.

```
    assert( n > 0 );
```

Výpočet budeme provádět v pomocné proměnné, která bude stejného typu jako n.

```
    decltype( n ) result = 0;
```

Princip výpočtu je jednoduchý, uvážíme-li dvojkový rozvoj čísla $n = \sum a_i 2^i$, který obsahuje člen 2^i pro každý nenulový bit a_i. Dvojkovým logaritmem bude právě nejvyšší mocnina dvojky, která se v tomto rozvoji objeví: jistě pak platí, že $2^k \leq n$, stačí se ujistit, že takto získané k je největší možné. Uvažme tedy, že existuje nějaké $l > k$ a zároveň $2^l \leq n$. Pak se ale musí 2^l nutně objevit ve dvojkovém rozvoji čísla n, což je spor s tím, že k byla nejvyšší mocnina v témže rozvoji.
Stačí nám tedy nalézt řád nejvyššího jedničkového bitu. To provedeme tak, že budeme provádět bitové posuny doprava tak dlouho, až n vynulujeme. Počet takto provedených posunů je pak hledaný řád.

```
    while ( n > 1 )
    {
        ++result;
        n >>= 1;
    }

    return result;
}
```

Jazyk C++ na rozdíl od některých jiných neposkytuje zabudovanou operaci mocnění pro celočíselné typy. Výpočet provedeme známým algoritmem binárního umocňování (anglicky známého popisněji jako

„exponentiation by squaring“).[12] Klíčovou vlastností tohoto algoritmu je, že jeho složitost je lineární k počtu bitů exponentu – naivní algoritmus opakovaným násobením má naproti tomu složitost exponenciální (složitost je v tomto případě přímo úměrná hodnotě exponentu, nikoliv délce jeho zápisu).

```
auto int_pow( auto n, auto exp )
{
```

Operaci budeme definovat pouze pro kladné exponenty. Vyhneme se tak mimo jiné nutnosti definovat hodnotu pro 0^0.

```
    assert( exp >= 1 );
```

Pomocná proměnná, která bude udržovat „liché“ mocniny. Její význam je přesněji vysvětlen níže.

```
    decltype( n ) odd = 1;
```

Výpočet je založený na pozorování, že pro sudý exponent k platí $n^k = n^{2l} = (n^2)^l$ kde $l = k/2$. Za cenu výpočtu jedné druhé mocniny – n^2 – tak můžeme exponent snížit na polovinu (cyklus se provede právě tolikrát, kolik bitů je v zápisu hodnoty exp).

```
    while ( exp > 1 )
    {
```

Musíme ovšem ještě vyřešit situaci, kdy je exponent lichý. Zde je potřebný vztah trochu složitější: $n^k = n \cdot (n^{2l})$ pro $l = \lfloor k/2 \rfloor$. V rekurzivním zápisu bychom mohli tento vztah přímo použít, v tom iterativním ale nastane drobný problém: faktor n před závorkou nevstupuje do výpočtu druhé mocniny v další iteraci. Asi nejjednodušším řešením je použití pomocného střadače, který bude udržovat tyto „přebývající“ faktory. Je-li exp liché, přinásobíme tedy faktor n do proměnné odd. Na konci ovšem nesmíme zapomenout, že ve výsledném n tyto faktory chybí.

```
        if ( exp % 2 == 1 )
            odd *= n;
```

Dále je výpočet pro sudé i liché exponenty stejný: hodnotu proměnné n umocníme na druhou a exponent vydělíme dvěma.

```
        n *= n;
        exp /= 2;
    }
```

Na závěr si vzpomeneme, že některé faktory celkového výsledku jsme si „odložili“ do proměnné odd.

```
    return n * odd;
```

Pro ilustraci uvažme výpočet 3^{10}:

iterace	n	exp	odd
0.	3	10	1
1.	3·3	5	1
2.	(3·3)·(3·3)	2	3·3
3.	(3·3)·(3·3)·(3·3)·(3·3)	1	3·3

V proměnné n jsme sesbírali 8 faktorů, zatímco proměnná odd získala 2, celkem jich je tedy potřebných 10. Rekurzivní výpočet by naproti tomu dopadl takto:

$$(3 \cdot (3 \cdot 3) \cdot (3 \cdot 3)) \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3))$$

[12] Popis algoritmu na české wikipedii je v době psaní tohoto textu zcela nepoužitelný. Podívejte se raději do té anglické.

Uvažme ještě výpočet 3^{11}. Je zejména důležité si uvědomit, že faktor, který na daném řádku přidáváme do odd (je-li exp na předchozím řádku liché) je právě hodnota n z tohoto předchozího řádku.

iterace	n	exp	odd
0.	3	11	1
1.	3·3	5	3
2.	(3·3)·(3·3)	2	3·(3·3)
3.	(3·3)·(3·3)·(3·3)·(3·3)	1	3·(3·3)

Stejný výpočet rekurzivně:

$$3 \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3)) \cdot (3 \cdot (3 \cdot 3) \cdot (3 \cdot 3))$$

```
}
```

Tím se dostáváme k poslední části: samotnému výpočtu celočíselné odmocniny. Budeme ji opět definovat jako největší s takové, že $s^n \le k$.

```
auto int_nth_root( auto n, auto k )
{
```

Pro jednoduchost budeme uvažovat pouze odmocniny nezáporných čísel.

```
    assert( k >= 0 );
    assert( n >= 1 );
```

Jednoduché případy vyřešíme zvlášť, protože by nám v obecném výpočtu níže působily určité potíže.

```
    if ( n == 1 || k == 0 )
        return k;
```

Na podrobný popis Newtonovy-Raphsonovy metody (známé též jako metoda tečen) zde nemáme prostor: možná ji znáte z kurzu matematické analýzy, případně si ji můžete vyhledat např. na wikipedii. Pro nás jsou klíčové její základní vlastnosti:

1. metoda nám umožní rychle nalézt x takové, že pro zadané f platí $f(x) = 0$,
2. potřebujeme k tomu samozřejmě definici f,
3. dále její první derivaci f'
4. a počáteční odhad hledané hodnoty x_0.

Výpočet opakovaně zlepšuje aktuální odhad x, a to pomocí vzorce:

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

Vyvstává otázka, jak nám hledání nuly pomůže ve výpočtu odmocniny. K tomu musíme problém přeformulovat. Uvažme

$$f(s) = s^n - k$$

Je-li $f(s) = 0$, pak jistě $s^n = k$, což je ale přesně definice n-té odmocniny (prozatím té reálné). Potřebujeme ještě derivaci, která je naštěstí velmi jednoduchá:

$$f'(s) = n \cdot s^{n-1}$$

protože n je celočíselná konstanta (pro $n = 1$ bychom ovšem narazili na problém). Celkem tedy:

$$s_{i+1} = s_i - (s_i^n - k)/(n \cdot s_i^{n-1})$$

Nebo výhodněji (přechodem na společný jmenovatel a krácením mocnin s_i):

$$s_{i+1} = t_i \ / \ n$$
$$t_i \ \ = (n-1) \cdot s_i + k/s^{n-1}$$

Zbývá počáteční odhad, který potřebujeme spočítat rychle (a samozřejmě potřebujeme, aby byl co nejblíže výsledné hodnotě s). Využijeme k tomu dříve definovaný dvojkový logaritmus. Protože $\log(a^b) = b \cdot \log(a)$, můžeme hledanou odmocninu odhadnout jako 2^{l+1} pro $l = \lfloor \log_2(k)/n \rfloor$. Také si všimneme, že tento odhad leží na stejné straně jediného stacionárního bodu funkce f (totiž $s = 0$) jako skutečné řešení. Nemůže se nám tedy stát, že by výpočet divergoval.

```
auto s = 2 << ( int_log2( k ) / n );
```

Samotná iterace je po zdlouhavé přípravě už velmi jednoduchá. Zbývají nám k vyřešení dva (související) problémy: kdy iteraci ukončit a jak výpočet provést nad celými čísly. Protože $k > 0$, je funkce f v kritické oblasti konvexní (tečny leží pod grafem). Po první iteraci bude náš odhad tedy celkem jistě příliš velký – průsečík tečny s osou x najdeme vpravo od skutečné nuly – a tato situace se už nezmění.
V tuto chvíli ale do hry vstupuje skutečnost, že pracujeme s celými a nikoliv reálnými čísly. Výpočet jsme uspořádali tak, aby byl výpočet průsečíku přesný – konkrétně je výsledkem výpočtu dolní celá část jeho reálné hodnoty. Tato dolní celá část sice může být menší, než skutečná hodnota reálné odmocniny, nemůže ale být menší než námi definovaná celočíselná odmocnina.
Tato pozorování nám konečně umožní formulovat podmínku ukončení: najdeme-li skutečnou celočíselnou odmocninu, další odhad může být buď stejný nebo větší než ten předchozí. Tato situace zároveň nemůže nastat dříve:

1. z konvexnosti plyne, že odhad, který je příliš velký, se musí ke skutečnému výsledku provedením iterace přiblížit,
2. protože předchozí vypočtený odhad je vždy celé číslo, musí sebemenší posun na reálné ose směrem doleva vést ke snížení jeho dolní celé části alespoň o jedničku.

Celkově tedy cyklus skončí přesně ve chvíli, kdy začne platit $s^n \leq k$.

```
    while ( true )
    {
        const auto t = ( n - 1 ) * s + k / int_pow( s, n - 1 );
        const auto s_next = t / n;

        if ( s_next >= s )
            return s;
        else
            s = s_next;
    }
}

int main() /* demo */
{
    assert( int_log2( 1 ) == 0 );
    assert( int_log2( 2 ) == 1 );

    assert( int_pow( 2, 2 ) == 4 );

    assert( int_nth_root( 1, 2 ) == 2 );
    assert( int_nth_root( 2, 4 ) == 2 );
    assert( int_nth_root( 3, 8 ) == 2 );

    for ( int k = 0; k < 1000; ++k )
        for ( int n = 1; n < 20; ++n )
        {
            auto root = int_nth_root( n, k );
            assert( int_pow( root, n ) <= k );
            assert( int_pow( root + 1, n ) > k );
        }

    return 0;
}
```

Část 1.e: Elementární příklady

1.e.1 [factorial] Vypočtěte faktoriál zadaného nezáporného čísla.

```
int factorial( int n );
```

1.e.2 [concat] Najděte a vraťte číslo, které vznikne zapsáním (nezáporných) celých čísel a, b v binárním zápisu za sebe (zápis čísla a bude vlevo, zápis čísla b vpravo a bude doplněn levostrannými nulami na délku b_bits bitů). Je-li zápis čísla b delší než b_bits, výsledek není definován.
Příklad: concat(1, 1, 2) vrátí hodnotu 0b101 = 5.

```
std::uint64_t concat( std::uint64_t a,
                      std::uint64_t b, int b_bits );
```

1.e.3 [zeros] Zapišme nezáporné číslo n v soustavě o základu base. Určete kolik (nelevostranných) nul se v tomto zápisu objeví. Do výstupního parametru order uložte řád nejvyšší z nich. Není-li v zápisu nula žádná, hodnotu order nijak neměňte.

```
int zeros( int n, int base, int &order );
```

1.e.4 [normalize] Write a function to normalize a fraction, that is, find the greatest common divisor of the numerator and denominator and divide it out. Both numbers are given as in/out parameters.

```
// void normalize( … )
```

Část 1.p: Přípravy

1.p.1 [nhamming] Nezáporná čísla a, b zapíšeme v poziční soustavě o základu base. Spočtěte hammingovu vzdálenost těchto dvou zápisů (přitom zápis kratšího čísla podle potřeby doplníme levostrannými nulami).

```
int hamming( int a, int b, int base );
```

1.p.2 [digitsum] Funkce digit_sum sečte cifry nezáporného čísla num v zápisu o základu base. Je-li výsledek ve stejném zápisu víceciferný, sečte cifry tohoto výsledku, atd., až je výsledkem jediná cifra, kterou vrátí jako svůj výsledek.

```
int digit_sum( int num, int base );
```

1.p.3 [parity] Funkce parity zjistí, je-li počet jedniček na vstupu sudý (výsledkem je false) nebo lichý (výsledkem je true).
Jedničky počítáme v binárním zápisu vstupního bezznaménkového čísla word. Je-li navíc chksum na začátku true, počítá se jako další jednička. Celkový výsledek jednak uložte do parametru chksum, jednak ho vraťte jako návratovou hodnotu.

```
bool parity( auto word, bool &chksum );
```

1.p.4 [periodic] Najděte nejmenší nezáporné číslo n takové, že 64-bitový zápis čísla word lze získat zřetězením nějakého počtu binárně zapsaných kopií n. Protože potřebný zápis n může obsahovat levostranné nuly, do výstupního parametru length uložte jeho délku v bitech. Je-li možné word sestavit z různě dlouhých zápisů nějakého n, vyberte ten nejkratší možný.
Příklad: pro word = 0x100000001 bude hledané $n = 1$, protože word lze zapsat jako dvě kopie čísla 1 zapsaného do 32 bitů.

```
std::uint64_t periodic( std::uint64_t word, int &length );
```

1.p.5 [balanced] V této úloze budeme opět počítat ciferný součet, ale v takzvaných vyvážených ciferných soustavách, které mají jak záporné tak kladné číslice. Budeme uvažovat pouze liché základy symet-

ricky rozložené kolem nuly (tzn. trojkovou s číslicemi $-1, 0, 1$, pětkovou $-2, -1, 0, 1, 2$, atd.). Vaším úkolem je napsat predikát `is_balanced`, který rozhodne, má-li zadané číslo `n` ve vyvážené soustavě zadané svým základem `base` nulový ciferný součet.
Výpočet cifer čísla n ve vyvážené soustavě o základu b probíhá podobně, jako v té klasické se stejným základem. Nejprve si připomeneme klasický algoritmus. Nastavíme $n_0 = n$ a opakujeme:

1. cifru c_i získáme jako zbytek po dělení n_i základem b,
2. spočítáme n_{i+1} tak, že vydělíme n_i základem b,
3. je-li výsledek nenulový, pokračujeme bodem 1, jinak skončíme.

Abychom získali vyvážený zápis místo toho klasického, musíme vyřešit situaci, kdy c_i není povolenou číslicí. Všimneme si, že musí po každém kroku platit (přímo z definice použitých operací):

$$n_i = c_i + bn_{i+1}$$

Tuto rovnost musíme zachovat, ale zároveň potřebujeme, aby c_i bylo platnou číslicí. To zajistíme jednoduše tak, že od c_i odečteme b a přičteme místo toho jedničku k n_{i+1} (tím se součet jistě nezmění, protože jsme jedno b ubrali a jedno přidali).

```
bool is_balanced( int n, int base );
```

1.p.6 [`subsetsum`] Vstupem pro problém subset sum je množina povolených čísel A a hledaný součet n. Řešením je pak podmnožina $B \subseteq A$ taková, že součet jejích prvků je právě n.
V tomto příkladu budeme pracovat pouze s množinami A, které obsahují kladná čísla menší nebo rovna 64, a které lze tedy reprezentovat jediným bezznaménkovým číslem z rozsahu `0` (prázdná množina) až $2^{64} - 1$ (obsahuje všechna čísla $1, 2, \ldots, 64$). Číslo `1` pak reprezentuje množinu $\{1\}$, číslo 2 množinu $\{2\}$, číslo 3 množinu $\{1, 2\}$ atd.
Vaším úkolem je napsat funkci `subset_sum`, které výsledkem bude `true`, má-li zadaná instance řešení. Toto řešení zároveň zapíše do výstupního parametru. V případě, že řešení neexistuje, hodnotu `solution` nijak neměňte.

```
bool subset_sum( int n, std::uint64_t allowed,
                 std::uint64_t &solution );
```

Část 1.r: Řešené úlohy

1.r.1 [`bitwise`] Předmětem této úlohy jsou ternární bitové operace: vstupem jsou 3 čísla a kód operace. Každý bit výsledku je určen příslušnými 3 bity operandů. Tyto 3 vstupní bity lze chápat jako pravdivostní hodnoty – potom je celkem jasné, že operaci můžeme zadat klasickou pravdivostní tabulkou, která pro každou kombinaci 3 bitů určí výsledek. Mohla by vypadat např. takto:

a	b	c	výsledek
0	0	0	r_0
0	0	1	r_1
0	1	0	r_2
0	1	1	r_3
1	0	0	r_4
1	0	1	r_5
1	1	0	r_6
1	1	1	r_7

Snadno se nahlédne, že zafixujeme-li tuto tabulku a zadáme hodnoty r_0 až r_7, kýžená operace bude jednoznačně určena. Protože potřebujeme 8 pravdivostních hodnot, můžeme je například předat jako jednobajtovou hodnotu typu `std::uint8_t`. Hodnotu r_0 předáme v nejnižším a r_7 v nejvyšším bitu.
Operace musí fungovat pro libovolný bezznaménkový celočíselný typ.
Můžete předpokládat, že hodnoty `a`, `b`, `c` jsou stejných typů (ale také se můžete zamyslet, jak řešit situaci, kdy stejné nejsou).

```
auto bitwise( std::uint8_t opcode, auto a, auto b, auto c );
```

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

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

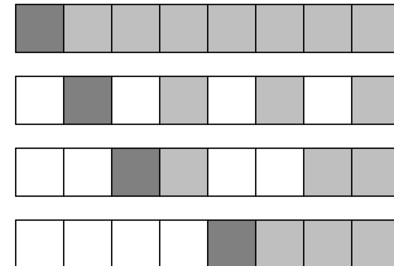
1.r.3 [`hamcode`] V této úloze budeme programovat dekodér pro kód Hamming(8, 4) – jedná se o variaci běžnějšího Hamming(7, 4) s dodatečným paritním bitem.
Vstupní blok je zadán osmibitovým číslem bez znaménka:

p_0	p_1	p_2	d_1	p_3	d_2	d_3	d_4
7	6	5	4	3	2	1	0

Blok je správně utvořený, platí-li tyto vztahy:

$$\begin{aligned} p_1 &= P(d_1, d_2, d_4) \\ p_2 &= P(d_1, d_3, d_4) \\ p_3 &= P(d_2, d_3, d_4) \\ p_0 &= P(p_1, p_2, p_3, d_1, d_2, d_3, d_4) \end{aligned}$$

kde P značí paritu. Graficky (zvýrazněný bit je paritním pro vyznačené bity):



Rozmyslete si, jaký mají uvedené vztahy dopad na paritu celých označených oblastí. Poté napište funkci `h84_decode`, která na vstupu dostane jeden blok, ověří správnost paritních bitů, a je-li vše v pořádku, vrátí `true` a do výstupního parametru `out` zapíše dekódovanou půlslabiku (d v nejnižším bitu). Jinak vrátí `false` a hodnotu `out` nemění.

```
bool h84_decode( std::uint8_t data, std::uint8_t &out );
```

1.r.5 [`cellular`] Napište čistou funkci, která simuluje jeden krok výpočtu jednorozměrného buněčného automatu (cellular automaton). Stav budeme reprezentovat celým číslem bez znaménka – jednotlivé buňky budou bity tohoto čísla. Stav osmibitového automatu by mohl vypadat například takto:

0	1	1	0	1	0	0	1
7	6	5	4	3	2	1	0

Pro zjednodušení použijeme pevnou sadu pravidel (+1, 0, -1 jsou relativní pozice bitů vůči tomu, který právě počítáme):

+1	0	-1	nová
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Pravidla určují, jakou hodnotu bude mít buňka v následujícím stavu, v závislosti na okolních buňkách stavu nynějšího. Na krajích stavu interpretujeme chybějící políčko jako nulu.
Výpočet s touto sadou pravidel tedy funguje takto:

1

0	1	1	0	0	0	1	0
0							

001 → 0

2

0	1	1	0	0	0	1	0
0	0						

011 → 0

3

0	1	1	0	0	0	1	0
0	0	1					

110 → 1

atd.

7

0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	

010 → 0

8

0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	1

100 → 1

Výpočet kroku by mělo být možné provést na libovolně širokém celočíselném typu.

```
auto cellular_step( auto w );
```

Část 2: Složené hodnoty

V této kapitole samozřejmě pokračujeme s použitím funkcí, skalárů a referencí, a přidáváme složené hodnoty: standardní kontejnery (`std::vector`, `std::set`, `std::map`, `std::array`) a součinové (produktové) typy `struct` a `std::tuple`.
Ukázky:

1. `stats` – záznamové typy, zjednodušený `for` cyklus
2. `primes` – vkládání prvků do hodnoty typu `std::vector`
3. `iterate` – vytvoření posloupnosti iterací funkce
4. `dfs` – dosažitelnost vrcholu v grafu
5. `bfs` † – nejkratší vzdálenost v neohodnoceném grafu

Elementary exercises:

1. `fibonacci` – stará posloupnost, nová signatura
2. `reflexive` – reflexivní uzávěr zadané relace
3. `unique` – odstraněni duplicit ve vektoru

Preparatory exercises:

1. `minsum` – dělení posloupnosti čísel podle součtu
2. `connected` – rozklad grafu na komponenty souvislosti
3. `divisors` – kontejner jako vstupně-výstupní parametr
4. `midpoint` – kontejner s prvky složeného typu
5. `dag` † – hledání cyklu v orientovaném grafu
6. `bipartite` – rozhodování o bipartitnosti grafu

Regular exercises:

1. `mode` – nalezněte mód číselné posloupnosti
2. `sssp` – nejkratší cesty z pevně zvoleného vrcholu
3. `solve` – solver pro velmi jednoduchou hru
4. `buckets` – řazení kamenů do kyblíčků podle váhy
5. `permute` – permutace číslic
6. `flood` – semínkové vyplňování s vektorem

1 Hlavičkové soubory Tato kapitola přidává řadu nových povolených hlavičkových souborů:

- `tuple` – definice N-tic `std::tuple` a pomocných funkcí,
- `vector` – definice dynamického pole `std::vector`,
- `set` – podobně, ale pro `std::set` a `std::multiset`,
- `map` – umožňuje použití kontejnerů `std::map`, `std::multimap`,
- `deque` – definuje oboustrannou frontu `std::deque`,
- `queue` – definuje klasickou frontu `std::queue`,
- `stack` – podobně ale zásobník `std::stack`,
- `utility` – různé pomocné funkce, `std::pair`,
- `ranges` – prozatím zejména `std::ranges::subrange`,
- `numeric` – funkce pro práci (zejména) s číselnými sekvencemi,
- `cmath` – funkce pro práci s čísly s plovoucí desetinnou čárkou.

Část 2.d: Demonstrace (ukázky)

2.d.1 [`stats`] V této ukázce spočítáme několik jednoduchých statistických veličin – míry polohy (průměr, medián) a variance (směrodatnou odchylku). Využijeme k tomu záznamové typy a sekvenční typ `std::vector`. Nejprve si definujeme typ pro výsledek naší jednoduché analýzy – použijeme k tomu záznamový typ, který deklarujeme klíčovým slovem `struct`, názvem, a seznamem deklarací složek uzavřeným do složených závorek (a jako všechny deklarace, ukončíme i tuto středníkem):

```
struct stats
{
    double median = 0.0;
    double mean = 0.0;
    double stddev = 0.0;
};
```

Tím máme definovaný nový typ s názvem `stats`, který můžeme dále používat jako libovolný jiný typ (zabudovaný, nebo ze standardní knihovny). Zejména můžeme vytvářet hodnoty tohoto typu, předávat je jako parametry nebo vracet jako výsledky podprogramů.
V této ukázce si zadefinujeme čistou funkci `compute_stats`, která potřebné veličiny spočítá a vrátí je jako hodnotu typu `stats`. Vstupní parametr `data` předáme konstantní referencí: hodnoty nebudeme nijak měnit (programujeme čistou funkci a `data` považujeme za vstupní para-

metr). Zároveň nepotřebujeme vytvořit kopii vstupních dat – budeme je pouze číst, taková kopie by tedy byla celkem zbytečná a potenciálně drahá (dat, které chceme zpracovat, by mohlo být mnoho).

```
stats compute_stats( const std::vector< double > &data )
{
    int n = data.size();
    double sum = 0, square_error_sum = 0;
    stats result;
```

Na tomto místě se nám bude hodit nový prvek řízení toku, kterému budeme říkat stručný `for` cyklus (angl. „range `for`"). Jeho účelem je procházet posloupnost hodnot uloženou v iterovatelném typu (použitelnost hodnoty ve stručném `for` cyklu lze chápat přímo jako definici iterovatelného typu). Do závorky uvádíme deklaraci proměnné cyklu (můžeme zde použít zástupné slovo `auto`) a dvojtečkou oddělený výraz. Tento výraz musí být iterovatelného typu a výsledná iterovatelná hodnota je ta, kterou budeme cyklem procházet.

```
    for ( double x_i : data )
```

Cyklus se provede pro každý prvek předané iterovatelné hodnoty. Tento prvek je pokaždé uložen do proměnné cyklu (která může být referencí – v takovém případě tato reference odkazuje přímo na prvek, v opačném případě se jedná o kopii).

```
        sum += x_i;
```

K položkám hodnoty záznamového typu přistupujeme výrazem `expr.field`, kde:

- `expr` je výraz záznamového typu (zejména to tedy může být název proměnné), následovaný
- tečkou (technicky se jedná o operátor s vysokou prioritou),
- `field` je jméno atributu (tzn. na pravé straně tečky nestojí výraz).

Je-li výraz `expr` l-hodnotou, je l-hodnotou i výraz přístupu k položce jako celek a lze do něj tedy přiřadit hodnotu.

```
    result.mean = sum / n;
```

Medián získáme dobře známým postupem. Za povšimnutí stojí indexace vektoru `data` zápisem indexu do hranatých závorek. Obecněji jsou-li `x` a `i` výrazy, je výraz také `x[ i ]` kde `x` je indexovatelného typu (omezení na typ `i` závisí na typu `x`). Je-li `x` navíc l-hodnota, je l-hodnotou i výraz `x[ i ]` jako celek.[13] *

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

Konečně spočítáme směrodatnou odchylku. K tomu budeme potřebovat dříve vypočítaný průměr.

```
    for ( double x_i : data )
        square_error_sum += std::pow( x_i - result.mean, 2 );

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

    return result;
}

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

    sample.push_back( 1100 );
    s = compute_stats( sample );

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

2.d.2 [`primes`] Krom jednoduchých výstupních parametrů (kterými jsme se zabývali v předchozí kapitole) lze uvažovat i o výstupních parametrech složených typů. V této ukázce naprogramujeme funkci `primes`, která na konec předaného objektu typu `std::vector` vloží všechna prvočísla ze zadaného rozsahu.

O parametru `out` budeme mluvit jako o výstupním parametru, i když situace je zde o něco složitější: operace, které můžeme se složenou hodnotou provádět, se totiž neomezují pouze na čtení a přiřazení. Musíme tedy vědět, jak závisí chování operací, které chceme provést, na počáteční hodnotě.

Například operace vložení prvku na konec vektoru bude fungovat stejně pro libovolný vektor.[14] Protože žádnou jinou operaci s parametrem `out` provádět nebudeme, je jeho označení za výstupní parametr opodstatněné.

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

Rozhodování o prvočíselnosti kandidáta provedeme naivně, zkusmým dělením.

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

Konečně, je-li kandidát skutečně prvočíslem, vložíme ho na konec vektoru odkazovaného parametrem `out` (protože `out` je referenčního typu, `out` je pouze nové jméno pro původní objekt uvedený ve skutečném parametru).

Novým prvkem je zde ale zejména volání metody. Syntakticky se podobá přístupu k položce (viz předchozí ukázka), ale je následováno kulatými závorkami a seznamem parametrů, stejně jako volání běžného podprogramu. Výraz před tečkou se použije jako skrytý parametr metody (ta tedy s výslednou hodnotou může pracovat – zde například volání `out.push_back( x )` modifikuje objekt `out`). O metodách si toho povíme více v následující kapitole.

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

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

[13] V některých případech je `x[ i ]` l-hodnotou i v případě, že `x` samotné l-hodnota není (opačná implikace tedy obecně neplatí). Výslednou l-hodnotu ale stejně nelze smysluplně použít.

[14] Situaci, kdy je vektor „plný" (obsahuje tolik prvků, že další nelze přidat, i kdyby to kapacita paměti umožnila) můžeme zanedbat: na 64b počítači, který skutečně existuje, nemůže nastat.

```
                p15 = { 2, 3, 5, 7, 11, 13 };

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

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

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

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

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

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

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

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

```
    return true;
}
```

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

```
bool is_symmetric( const relation &r )
{
    for ( auto [ x, y ] : r )
        if ( !r.contains( { y, x } ) )
            return false;
    return true;
}
```

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

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

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

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

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

2.d.4 [dfs] V této ukázce se budeme zabývat prohledáváním orientovaného grafu. Asi nejjednodušším vhodným algoritmem je rekurzivní prohledávání do hloubky. Konkrétně nás bude zajímat odpověď na otázku „je vrchol b dosažitelný z vrcholu a?“. Budeme navíc požadovat, aby byla příslušná cesta neprázdná (tzn. a budeme považovat za dosažitelné z a pouze leží-li na cyklu).
Vstupní graf bude zadaný za pomoci seznamů následníků: typ `graph` udává pro každý vrchol grafu jeho následníky. Asociativní kontejner `std::map` ukládá dvojice klíč-hodnota a umožňuje mimo jiné efektivně (v logaritmickém čase) nalézt hodnotu podle zadaného klíče.
Všimněte si také, že množina vrcholů nemusí nutně sestávat z nepřerušené posloupnosti, nebo jen z malých čísel (proto používáme `std::map` a nikoliv `std::vector`).

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

Krom samotného grafu budeme potřebovat reprezentaci pro množinu navštívených vrcholů. V grafu s cykly by algoritmus, který si takovou množinu neudržuje, vedl na nekonečnou rekurzi (nebo nekonečný cyklus). Navíc i v acyklickém grafu bude takový algoritmus vyžadovat (v nejhorším případě) exponenciální čas.
Protože sémanticky se jedná o množinu, není asi velkým překvapením, že pro její reprezentaci použijeme asociativní kontejner `std::set`. Vyhledání prvku (resp. test na přítomnost prvku) v `std::set` má logaritmickou časovou složitost. Podobně tak vložení prvku.

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

Hlavní rekurzivní podprogram bude potřebovat 2 pomocné parametry: již zmiňovanou množinu navštívených vrcholů, a navíc pravdivostní hodnotu `moved`, která řeší případ, kdy potřebujeme zjistit, zda je vrchol dosažitelný sám ze sebe. Naivní řešení by totiž pro dvojici (a, a) vždy vrátilo `true` (v rozporu s naším zadáním). Proto si v tomto parametru budeme pamatovat, zda jsme se již podél nějaké hrany posunuli.
Tento podprogram bude tedy odpovídat na otázku „existuje cesta, která začíná ve vrcholu `from`, neprochází žádným vrcholem v `seen`, a zároveň končí ve vrcholu `to`?“ Všimněte si ale, že množinu `seen` předáváme odkazem (referencí) – existuje pouze jediná množina navštívených vrcholů, sdílená všemi rekurzivními aktivacemi podprogramu. Jakmile tedy vrchol potkáme na libovolné cestě, bude vyloučen ze zkoumání ve všech ostatních větvích výpočtu (tedy i v těch sourozeneckých, nikoliv jen v potomcích toho současného).

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

První bázový případ je situace, kdy jsme cílový vrchol našli – protože je velmi jednoduchý, vyřešíme jej první. Všimněte si kontroly parametru `moved`.

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

Hlavní cyklus pokrývá zbývající případy:

1. druhý bázový případ, kdy žádný nenavštívený potomek již neexistuje (tzn. nacházíme se ve slepé větvi a výsledkem je `false`), a
2. případ, kdy existuje dosud nenavštívený soused – pak lze ale problém vyřešit rekurzí, protože současný vrchol jsme z problému vyloučili a zbývající problém je tedy menší.

Výsledkem volání metody `at` je reference na hodnotu přidruženou klíči, který jsme předali v parametru. Proměnná `next` tedy nabývá hodnot, které odpovídají přímým následníkům vrcholu `from`.

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

V případě, že jsme nalezli nenavštívený vrchol, nejprve ho označíme za navštívený a poté provedeme rekurzivní volání. Protože jsme se právě posunuli po hraně `from, next`, nastavujeme parametr `moved` na `true`.

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

Skončí-li cyklus jinak, než návratem z podprogramu, znamená to, že jsme vyčerpali všechny možnosti, aniž bychom našli přípustnou cestu, která vrcholy `from` a `to` spojuje.

```
    return false;
}
```

Konečně doplníme jednoduchou funkci, která doplní potřebné hodnoty pomocným parametrům. Odpovídá na otázku „lze se do vrcholu `to` dostat podél jedné nebo více hran, začneme-li ve vrcholu `to`?".
Za povšimnutí také stojí, že `is_reachable` je čistou funkcí (a to i přesto, že `is_reachable_rec` čistou funkcí není).

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

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

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

2.d.5 [bfs] † The goal of this demonstration will be to find the shortest distance in an unweighted, directed graph:

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

The canonical 'shortest path' algorithm in this setting is breadth-first search. The algorithm makes use of two data structures: a queue and a set, which we will represent using the standard C++ containers named, appropriately, `std::queue`[15] and `std::set`.
In the previous demonstration, we have represented the graph explicitly using adjacency list encoded as instances of `std::vector`. Here, we will take a slightly different approach: we well use `std::multimap` – as the name suggests, it is related to `std::map` with one crucial difference: it can associate multiple values to each key. Which is exactly what we need to represent an directed graph – the values associated with each key will be the successors of the vertex given by the key.

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

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

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

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

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

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

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

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

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

Follows the standard BFS loop:

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

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

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

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

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

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

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

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

about), we extract the second element:

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

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

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

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

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

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

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

```
    return -1;
}

int main() /* demo */
{
    graph g{ { 1, 2 }, { 1, 6 }, { 2, 4 }, { 2, 5 }, { 6, 4 } },
          h{ { 8, 2 }, { 8, 6 }, { 2, 4 }, { 2, 5 }, { 5, 8 } },
          i{ { 2, 4 }, { 4, 2 } };

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

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

Část 2.e: Elementární příklady

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

```
// void fibonacci( … )
```

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

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

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

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

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

Část 2.p: Přípravy

2.p.1 [minsum] Na vstupu dostanete posloupnost celočíselných hodnot (jako instanci kontejneru std::vector). Vaším úkolem je rozdělit je na kratší posloupnosti tak, že každá posloupnost je nejkratší možná, ale zároveň je její součet alespoň sum. Výjimku tvoří poslední posloupnost, pro kterou nemusí nutně existovat potřebné prvky.
Pořadí prvků musí být zachováno, tzn. zřetězením všech posloupností na výstupu musí vzniknout původní posloupnost numbers.

```
auto minsum( const std::vector< int > &numbers, int sum );
```

2.p.2 [connected] Rozložte zadaný neorientovaný graf na souvislé komponenty (výsledné komponenty budou reprezentované množinou svých vrcholů). Graf je zadaný jako symetrická matice sousednosti. Vrcholy jsou očíslované od 1 do n, kde n je velikost vstupní matice.
V grafu je hrana $\{u, v\}$ přítomna právě tehdy, je-li na řádku u ve sloupci v hodnota true.

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

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

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

2.p.3 [divisors] Nalezněte všechny prvočíselné dělitele čísla num a vložte je do vektoru divs. Počáteční hodnota parametru divs:

- obsahuje právě všechny prvočíselné dělitele všech čísel ostře menších než num,
- je vzestupně seřazená.

Výstupní podmínkou pro vektor divs je:

- obsahuje všechna čísla, která obsahoval na vstupu,
- zároveň obsahuje všechny prvočíselné dělitele čísla num,
- je vzestupně seřazený.

Funkce musí pracovat efektivně. Určit vhodnou časovou složitost je v této úloze součástí zadání.

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

2.p.4 [midpoints] Strukturu point doplňte tak, aby měla složky x a y, kde obojí jsou čísla s plovoucí desetinnou čárkou, a to tak, že deklarace point p; vytvoří bod se souřadnicemi 0, 0.

```
struct point;
```

Nyní uvažme uzavřenou lomenou čáru. Nahraďte každou úsečku A takovou, která začíná prostředním bodem úsečky A a končí prostředním bodem úsečky B, kde B v obrazci následuje bezprostředně po A. Vstup je zadán jako sekvence bodů (kde každý bod náleží dvěma úsečkám). Poslední úsečka jde z posledního bodu do prvního, čím se obrazec uzavře.

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

2.p.5 [dag] † Budeme opět pracovat s orientovaným grafem – tentokrát budeme hledat cykly. Existuje na výběr několik algoritmů, ty založené na prohledávání do hloubky jsou nejjednodušší. Graf je zadaný jako hodnota typu std::multimap – více se o této reprezentaci dozvíte v ukázce d5_bfs.
Čistá funkce is_dag nechť vrátí false právě když g obsahuje cyklus. Pozor, graf nemusí být souvislý.

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

2.p.6 [bipartite] Rozhodněte, zda je zadaný neorientovaný graf bipartitní (tzn. jeho vrcholy lze rozdělit do dvou množin A, B tak, že každá hrana má jeden vrchol v A a jeden v B). Protože graf je neorientovaný, seznamy sousedů na vstupu jsou symetrické.

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

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

Část 2.r: Řešené úlohy

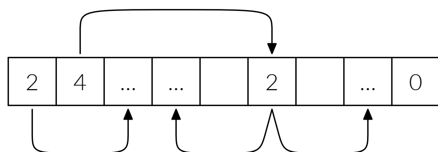
2.r.1 [mode] Find the mode (most common value) in a non-empty vector and return it. If there are more than one, return the smallest.

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

2.r.2 [sssp] Compute single-source shortest path distances for all vertices in an unweighted directed graph. The graph is given using adjacency (successor) lists. The result is a map from a vertex to its shortest distance from initial. Vertices which are not reachable from initial should not appear in the result.

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

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

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



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

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

2.r.4 [buckets] Sort stones into buckets, where each bucket covers a range of weights; the range of stone weights to put in each bucket is given in an argument – a vector with one element per bucket, each element a pair of min/max values (inclusive). Assume the bucket ranges do not overlap. Stones are given as a vector of weights. Throw away stones which do not fall into any bucket. Return the weights of individual buckets.

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

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

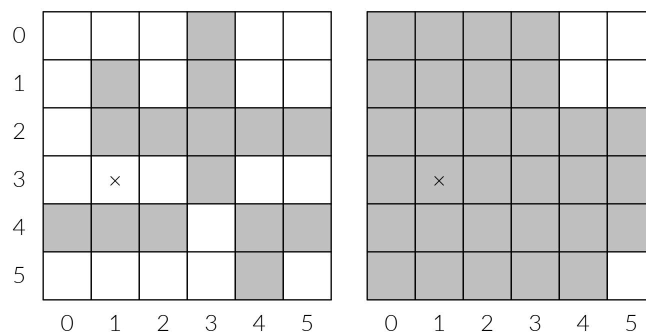
2.r.5 [colour] Write a function to decide whether a given graph can be 3-colored. A correct colouring is an assignment of colours to vertices such that no edge connects vertices with the same colour. The graph is given as a set of edges. Edges are represented as pairs; assume that if (u, v) is a part of the graph, so is (v, u).

```
using graph = std::set< std::pair< int, int > >;
bool has_3colouring( const graph &g );
```

2.r.6 [flood] V tomto cvičení implementujeme tzv. semínkové vyplňování, obvykle popsané algoritmem, který:

1. dostane na vstupu bitmapu (odélníkovou síť pixelů),
2. počátečnou pozici v síti,
3. barvu výplně (cílovou barvu),

a změní celou souvislou jednobarevnou plochu, která obsahuje počáteční pozici, na cílovou barvu.
Budeme uvažovat monochromatický případ – pixely jsou černé nebo bílé, resp. 0 nebo 1. Navíc nebudeme měnit vstupní bitmapu, ale pouze spočítáme, kolik políček změní barvu a tuto hodnotu vrátíme. Příklad (prázdná políčka mají hodnotu 0, vybarvená hodnotu 1, startovní pozice má souřadnice 1, 3):



Všimněte si, že „záplava" se šíří i po diagonálách (např. z pozice (2, 3) na (3, 4) a dále na (4, 3)). Vstupní bitmapa je zadaná jako jednorozměrný vektor a šířka. Chybí-li nějaké pixely v posledním řádku, uvažujte jejich hodnotu nulovou. Je-li poslední řádek kompletní, nic nepřidávejte. Je-li startovní pozice, x0 nebo y0 mimo meze bitmapy (tzn. její šířku a výšku), žádné pixely barvu nezmění. Poslední parametr, fill, udává cílovou barvu. Je-li startovní pozice cílové barvy, podobně se nic nebude měnit.

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

int flood( const grid &pixels, int width,
           int x0, int y0, bool fill );
```

Část 3: Metody a operátory

Ukázky:

1. freq – analýza frekvence n-gramů
2. lemmings – modelujeme figurky z počítačové hry
3. arithmetic – přetěžování aritmetických operátorů
4. relational – přetěžování relačních operátorů

Elementární příklady:

2. cartesian – komplexní čísla

Přípravy:

1. area – výpočet plochy jednoduchých útvarů
2. rational – racionální čísla (zlomky)
3. mountains – „rekurzivní" pohoří
4. polar – komplexní čísla podruhé
5. numset – mírně vylepšená množina čísel
6. continued – řetězové zlomky

Regular exercises:

1. poly – polynomy se sčítáním a násobením
2. xxx
3. set – množina čísel s operátory
4. xxx
5. xxx
6. xxx

Část 3.d: Demonstrace (ukázky)

3.d.1 [freq] V této ukáce budeme počítat histogram (číselných) n-gramů, tzn. bloků n po sobě jdoucích čísel v nějaké delší sekvenci. Jednotlivé n-gramy se mohou překrývat (n-gram tedy může začínat na libovolném offsetu).
Naším úkolem je navrhnout typ, který bude tuto frekvenci počítat „za běhu" – počítáme totiž s tím, že vstupních dat bude hodně a budou přicházet po blocích. Zároveň předpokládáme, že různých n-gramů

bude řádově méně než vstupních dat.
Budeme implementovat dvě metody:

1. count, která pro zadaný n-gram vrátí počet jeho dosavadních výskytů, a metodu
2. process, která zpracuje další blok dat.

```
struct freq
{
    std::size_t ngram_size = 3;
```

Budeme potřebovat dvě datové složky: samotné počítadlo výskytů implementujeme pomocí standardního asociativního kontejneru std::map. Klíčem bude std::vector potřebné délky (reprezentuje n-gram), hodnotou pak počet výskytů tohoto n-gramu.

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

Druhou složkou bude posuvné okno, ve kterém budeme uchovávat poslední zpracovaný n-gram. Je to proto, že některé n-gramy budou rozděleny mezi dva bloky (nebo i více, pokud se objeví velmi krátký blok). Pro jednoduchost budeme toto okno používat pro všechny n-gramy, a realizovat ho budeme jako instanci std::deque[16].

```
    std::deque< int > _window;
```

Nejprve implementujeme pomocnou metodu, která zpracuje jedno číslo. Je-li okno již plné, odstraníme z něj nejstarší hodnotu. Je-li po vložení čísla okno dostatečně plné, výsledný n-gram započítáme. Vzpomeňte si, že indexovací operátor kontejneru std::map podle potřeby vloží novou dvojici klíč-hodnota (s hodnotou inicializovanou „na nulu“).

```
    void add( int value )
    {
        if ( _window.size() == ngram_size )
            _window.pop_front();

        _window.push_back( value );

        if ( _window.size() == ngram_size )
        {
            std::vector< int > ngram;
            for ( int v : _window )
                ngram.push_back( v );
            ++ _counter[ ngram ];
        }
    }
```

Protože metoda add kompletně řeší jak správu okna, tak počítadlo, zpracování bloku už je jednoduché.

```
    void process( const std::vector< int > &block )
    {
        for ( int value : block )
            add( value );
    }
```

Metodu count, která pouze vrací informace a aktuální objekt nijak nemění, bychom rádi označili jako const. Jako drobný problém se jeví, že indexace položky _counter ale není konstantní operace: jak jsme zmiňovali, operátor indexace může do kontejneru vložit novou dvojici, a tím ho změnit.
Nemůžeme také přímo použít metodu at, protože musíme být schopni správně odpovídat i v případě, že dotazovaný n-gram se na vstupu dosud neobjevil, a tedy takový klíč v kontejneru _counter není přítomen.
Zbývá tedy metoda find, která nám dá jak informaci o tom, jestli je klíč přítomen (hledání vyžaduje logaritmický čas), a pokud ano, tak nám k němu přímo umožní přístup (již v konstantním čase). Použití s inicializační sekcí podmíněného příkazu if sze považovat za idiomatické.

```
    int count( const std::vector< int > &ngram ) const
    {
        if ( auto it = _counter.find( ngram ); it != _counter.end() )
            return it->second;
        else
            return 0;
    }
};

int main() /* demo */
{
    freq f{ .ngram_size = 3 };
```

Vytvoříme si na f také konstantní referenci, abychom se ujistili, že metodu count skutečně lze volat na konstantní hodnotu.

```
    const freq &cf = f;

    assert( cf.count( { 1, 1, 1 } ) == 0 );

    f.process( { 1, 1, 2, 1, 1 } );
    assert( cf.count( { 1, 1, 1 } ) == 0 );
    assert( cf.count( { 1, 1, 2 } ) == 1 );
    assert( cf.count( { 1, 2, 1 } ) == 1 );

    f.process( { 1 } );
    assert( cf.count( { 1, 1, 1 } ) == 1 );
    assert( cf.count( { 1, 2, 1 } ) == 1 );

    f.process( { 1 } );
    f.process( { 2, 2 } );
    assert( cf.count( { 1, 1, 1 } ) == 2 );
    assert( cf.count( { 1, 2, 2 } ) == 1 );
    assert( cf.count( { 2, 2, 1 } ) == 0 );
}
```

3.d.2 [lemmings] While we are talking about computer games, you might have heard about a game called Lemmings (but it's not super important if you didn't). In each level of the game, lemmings start spawning at a designated location, and immediately start to wander about, fall off cliffs, drown and generally get hurt. The player is in charge of saving them (or rather as many as possible), by giving them tasks like digging tunnels, or stopping and redirecting other lemmings.
Let's try to design a type which will capture the state of a single lemming:

```
struct lemming
{
```

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

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

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

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

Finally, they might be assigned a task, which they will immediately start performing. The exact meaning of the number is not very important.

```
    int _task = 0;
```

[16] Tato volba reprezentace není úplně nejefektivnější, ale pro naše účely dostatečná. Asymptoticky jí není co vytknout.

Let us define a few (mostly self-explanatory) methods:

```
    void start_digging() { _task = 1; }

    bool busy()  const { return _task != 0; }
    bool alive() const { return _alive; }

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

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

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

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

```
    int count = 0;
```

Of course we can iterate a vector of lemmings like any other vector, and call methods on the individual lemmings (inside the vector, since we are using a reference).

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

    return count;
}

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

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

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

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

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

We can also modify the lemmings in a range `for` loop – notice the absence of `const`; this time, we use a mutable reference – the lemmings are modified in place inside the container.

```
    for ( lemming &l : lemmings )
    {
        assert( l.alive() );
        l.start_digging();
    }

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

3.d.3 [`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).

```
struct gf7
{
    int value;
```

We can name a constant by wrapping it in a method. There are other ways to achieve the same effect, but we don't currently have the necessary pieces of syntax.

```
    int modulus() const { return 7; }
```

A helper method to normalize an integer. We would really prefer to enforce the normalization (such that all values of type `gf7` would have their `value` field in the range $\langle 0, 7)$, but we currently do not have the mechanisms to do that either. This will improve in the next chapter.

```
    gf7 normalize() const { return { value % modulus() }; }
```

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 compound type in this case – the trick is that we can write it here anyway. The implementation relies on the simple fact that $[a]_7 + [b]_7 = [a + b]_7$.

```
    friend gf7 operator+( gf7 a, gf7 b )
    {
        return gf7{ a.value + b.value }.normalize();
    }
```

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). Note that normally, you would use the same form for all symmetric operators for any given type – we mix them here to highlight the difference. We again use the fact that $[a]_7 \cdot [b]_7 = [a \cdot b]_7$.

```
    gf7 operator*( gf7 b ) const
    {
        return gf7{ value * b.value }.normalize();
    }
```

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 % modulus(); }
};
```

Operators can be also overloaded using 'normal' top-level functions, like this unary minus (which finds the additive inverse of the given element).

```
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 */
}
```

3.d.4 [relational] In this example, we will show relational operators, which are very similar to the arithmetic operators from previous example, except for their return types, which are bool values.
The example which we will use in this case are sets of small natural numbers (1-64) with inclusion as the order. We will use three-way comparison to obtain most of the comparison operators 'for free'.
NB. Standard ordered containers like std::set and std::map require the operator less-than to define a strict weak ordering (which is corresponds to a total order). The comparison operators in this example do not define a total order (some sets are incomparable).

```
struct 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 = 0;
```

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). Note that defining a separate operator != is not required in C++20.

```
    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 { 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. We will write a single comparison operator, then use it to implement three-way comparison, which the compiler will then use to derive all the remaining comparison operators.

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

In addition to getting all other comparisons for free, the three-way comparison also allows us to declare the properties of the ordering.

```
    friend std::partial_ordering operator<=>( set a, set b )
    {
        if ( a == b )
            return std::partial_ordering::equivalent;
        if ( a <= b )
            return std::partial_ordering::less;
        if ( b <= a )
            return std::partial_ordering::greater;

        return std::partial_ordering::unordered;
    }
};

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 );
}
```

Část 3.e: Elementární příklady

3.e.2 [cartesian] V tomto příkladu implementujeme typ cartesian, který reprezentuje komplexní číslo pomocí reálné a imaginární části. Takto realizovaná čísla umožníme sčítat, odečítat, získat číslo opačné (unárním mínus) a určit jejich rovnost (zamyslete se, má-li smysl definovat na tomto typu uspořádání; proč ano, proč ne?).

```
struct cartesian;
```

Implementujte také čistou funkci make_cartesian, která vytvoří hodnotu typu cartesian se zadané reálné a imaginární složky.

```
cartesian make_cartesian( double, double );
```

Část 3.p: Přípravy

3.p.1 [area] Doplňte definice typů point, polygon a circle tak, abyste pak mohli s jejich pomocí možné implementovat tyto čisté funkce:

- make_polygon, která přijme jako parametr celé číslo (počet stran) a dále:
  - 2 body (střed a některý vrchol), nebo
  - 1 bod (střed) a 1 reálné číslo (poloměr opsané kružnice),
- make_circle které přijme jako parametry:
  - 2 body (střed a bod na kružnici), nebo
  - 1 bod a 1 reálné číslo (střed a poloměr),
- area, které přijme polygon nebo circle a vrátí plochu odpovídajícího útvaru.

Typ point nechť má složky x a y (reálná čísla).

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

3.p.2 [rational] V tomto příkladu budeme programovat jednoduchá racionální čísla (taková, že je lze reprezentovat dvojicí celých čísel typu int). Hodnoty typu rat lze:

- vytvořit čistou funkcí make_rat(p, q) kde p, q jsou hodnoty typu int (čitatel a jmenovatel) a q > 0,
- použít jako operandy základních aritmetických operací:

- sčítání `+`,
- odečítání (`-`),
- násobení (`*`) a
- dělení (`/`),

- libovolně srovnávat (`==`, `!=`, `<=`, `<`, `>`, `>=`).

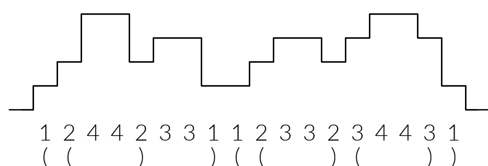
Vzpomeňte si, jak se jednotlivé operace nad racionálními čísly zavádí. Jsou-li $a = p_1/q_1$ a $b = p_2/q_2$ zlomky v základním tvaru, můžete se u libovolné operace $a \diamond b$ spolehnout, že žádný ze součinů $p_1 \cdot q_2$, $p_2 \cdot q_1$, $p_1 \cdot p_2$ a $q_1 \cdot q_2$ nepřeteče rozsah `int`-u.

```
struct rat;

rat make_rat( int, int );
```

3.p.3 [`mountains`] Vaším úkolem je vytvořit typ `mountain_range`, který bude reprezentovat rekurzivní pohoří. Rekurzivní pohoří má tento tvar:

1. levý svah (může být prázdný), který může na každém kroku libovolně stoupat,
2. libovolný počet (i nula) vnitřních pohoří stejného typu (první z nich začíná ve výšce, na které skončil levý svah),
3. pravý svah, který je zrcadlovým obrazem toho levého.

Například (hlavní pohoří má prázdný svah; závorky naznačují začátky a konce jednotlivých vnitřních pohoří):



Je-li `outer` hodnota typu `mountain_range`, nechť:

1. `outer.get( i )` vrátí výšku `i`-tého pole pohoří `outer`, a
2. `outer.set_slope( slope )` pro zadaný vektor čísel `slope` nastaví oba svahy tak, aby ten levý odpovídal výškám v `slope`,
3. `outer.insert( inner )` vloží nové vnitřní pohoří zadané hodnotou `inner` typu `mountain_range`, a to těsně před pravý svah.

Dobře si rozmyslete vhodnou reprezentaci. Požadujeme:

- metoda `get` musí mít konstantní složitost,
- metoda `set_slope` může být vůči argumentu lineární, ale nesmí záviset na délce vnitřních pohoří,
- metoda `insert` může být vůči vkládanému pohoří (`inner`) lineární, vůči tomu vnějšímu (`outer`) ale musí být amortizovaně konstantní.

Nově vytvořená hodnota typu `mountain_range` reprezentuje prázdné pohoří (prázdný svah a žádná vnitřní pohoří).

```
struct mountain_range;
```

3.p.4 [`polar`] Vaším úkolem je implementovat typ `polar`, který realizuje polární reprezentaci komplexního čísla. Protože tato podoba zjednodušuje násobení a dělení, implementujeme v této úloze právě tyto operace (sčítání jsme definovali v příkladu `e2_cartesian`).
Krom násobení a dělení nechť je možné pro hodnoty typu polar určit jejich rovnost operátory `==` a `!=`. Rovnost implementujte s ohledem na nepřesnost aritmetiky s plovoucí desetinnou čárkou. V tomto příkladě můžete pro reálná čísla (typu `double`) místo `x == y` použít `std::fabs( x - y ) < 1e-10`.
Pozor! Argument komplexního čísla je periodický: buďto jej normalizujte tak, aby ležel v intervalu $[0, 2\pi)$, nebo zajistěte, aby platilo `polar( 1, x ) == polar( 1, x + 2π )`.

```
struct polar;

polar make_polar( double, double );
```

3.p.5 [`numset`] Navrhněte typ `numset`, kterého hodnoty budou reprezentovat množiny čísel. Jsou-li `ns₁`, `ns₂` hodnoty typu `numset` a dále `i`, `j` jsou hodnota typu `int`, požadujeme následující operace:

- `ns₁.add( i )` – vloží do `ns₁` číslo `i`,
- `ns₁.del( i )` – odstraní z `ns₁` číslo `i`,
- `ns₁.del_range( i, j )` – odstraní z `ns₁` všechna čísla, která spadají do uzavřeného intervalu $\langle i, j \rangle$,
- `ns₁.merge( ns₂ )` – přidá do `ns₁` všechna čísla přítomná v `ns₂`,
- `ns₁.has( i )` – rozhodne, zda je `i` přítomné v `ns₁`.

Složitost:

- `del_range` a `merge` musí mít nejvýše lineární složitost,
- ostatní operace nejvýše logaritmickou.

```
struct numset;
```

3.p.6 [`continued`] Předmětem tohoto příkladu jsou tzv. řetězové zlomky. Typ `fraction` bude reprezentovat racionální číslo, které lze:

- zadat posloupností koeficientů řetězového zlomku (přesněji popsáno níže) pomocí metody `set_coefficients( cv )` kde `cv` je vektor hodnot typu `int`,
- sčítat (operátorem `+`),
- násobit (operátorem `*`),
- srovnávat (všemi relačními operátory, tzn. `==`, `!=`, `<`, `<=`, `>`, `>=`).

Řetězový zlomek reprezentuje racionální číslo q_0 jako součet celého čísla a_0 a převrácené hodnoty nějakého dalšího racionálního čísla, q_1, které je samo zapsáno pomocí řetězového zlomku. Tedy

$$q_0 = a_0 + 1/q_1$$
$$q_1 = a_1 + 1/q_2$$
$$q_2 = a_2 + 1/q_3$$

a tak dále, až než je nějaké q_i celé číslo, kterým sekvence končí (pro racionální číslo se to jistě stane po konečném počtu kroků). Hodnotám $a_0, a_1, a_2, \ldots$ říkáme koeficienty řetězového zlomku – jeho hodnota je jimi jednoznačně určena.
Rozmyslete si vhodnou reprezentaci vzhledem k zadanému rozhraní. Je důležité jak to, které operace jsou požadované, tak to, které nejsou.

```
struct fraction;
```

Část 3.r: Řešené úlohy

3.r.1 [`poly`] Cílem cvičení je naprogramovat typ, který bude reprezentovat polynomy s celočíselnými koeficienty, s operacemi sčítání (jednoduché) a násobení (méně jednoduché). Polynom je výraz tvaru $7x^4 + 0x^3 + 0x^2 + 3x + x^0$ – tzn. součet nezáporných celočíseslných mocnin proměnné x, kde u každé mocniny stojí pevný (konstantní) koeficient.
Součet polynomů má u každé mocniny součet koeficientů příslušné mocniny sčítanců. Součin je složitější, protože:

- každý monom (sčítanec) prvního polynomu musí být vynásoben každým monomem druhého polynomu,
- některé z těchto součinů vedou na stejnou mocninu x a tedy jejich koeficienty musí být sečteny.

Pro každý polynom existuje nějaké n takové, že všechny mocniny větší než n mají nulový koeficient. Tento fakt nám umožní polynomy snadno reprezentovat.
Implicitně sestrojená hodnota typu `poly` nechť má všechny koeficienty nulové. Krom sčítání a násobení (formou operátorů) implementujte také rovnost a metodu `set`, která má dva parametry: mocninu x a koeficient, obojí celá čísla.

```
struct poly;
```

3.r.2 [qsort] Implementujte typ array, který reprezentuje pole čísel a bude mít metody:

- get(i) – vrátí hodnotu na indexu i,
- append(x) – vloží hodnotu x na konec pole,
- partition(p, l, r) – přeuspořádá část pole v rozsahu indexů $\langle l, r)$ tak, aby hodnoty menší než p předcházely hodnotě rovné p a tato předcházela zbývající hodnoty větší nebo rovné p (nejsou-li l a r uvedeny, přeuspořádá celé pole; vstupní podmínkou je, že p je v uvedeném rozsahu přítomno),
- sort() – seřadí pole metodou quicksort (bez dodatečné paměti mimo místa na zásobníku potřebného pro rekurzi).

Algoritmus quicksort pracuje takto:

1. má-li pole žádné nebo 1 prvek, je již seřazené: konec;
2. jinak jeden z prvků vybereme jako pivot,
3. přeuspořádáme pole na dvě menší partice (viz popis metody partition výše),
4. rekurzivně aplikuje algoritmus quicksort na levou a pravou partici (vynechá přitom hodnoty rovné pivotu).

Užitečný invariant: po každé partici jsou prvky rovné vybranému pivotu na pozicích, které budou mít ve výsledném uspořádaném poli.
Viz též: https://xkcd.com/1185/

```
struct array;
```

3.r.3 [ttt] Naprogramujte typ tictactoe, který bude reprezentovat stav této jednoduché hry (piškvorky na ploše 3×3). Stav hry má tyto složky:

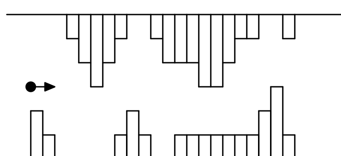
- který hráč je na tahu,
- který hráč zabral která políčka.

V nově vytvořené hře je plocha prázdná a na tahu je hrač s křížky. Metody:

- play(x, y) umístí symbol aktivního hráče na souřadnice x, y (platnost souřadnic i tahu je vstupní podmínkou, roh má souřadnice 0, 0),
- read(x, y) vrátí hodnotu zadaného pole:
  - -1 je křížek,
  - 0 je prázdné pole, konečně
  - 1 je kolečko,
- winner() vrátí podobně -1/0/1 podle toho, který hráč vyhrál (0 značí, že hra buď ještě neskončila, nebo skončila remízou).

```
struct tictactoe;
```

3.r.4 [flight] Vaší úlohou je naprogramovat jednoduchý simulátor hry, kde hráč ovládá létající objekt („loď") v bočním pohledu. Cílem hráče je nenarazit do hranic „jeskyně" ve které se pohybuje, a která je zadaná jako seznam dvojic, kde čísla na indexu i udávají vždy souřadnice y spodní a horní meze příslušného sloupce – pole – v rozsahu souřadnice $x \in \langle i, i + 1)$. Například:



Loď lze ovládat nastavením stoupání c (pro každý posun o l jednotek doprava se loď zároveň posune $c \cdot l$ jednotek nahoru; je-li c záporné, posouvá se dolů). Hra má tyto 4 metody:

- append(y₁, y₂) přidá na pravý konec hracího pole novou dvojici překážek (zadanou čísly s plovoucí desetinnou čárkou),
- move(l) posune loď o l jednotek doprava (l je celé číslo; při posunu dojde také k příslušné změně výšky podle aktuálního nastavení) a vrátí true v případě, že při tomto posunu nedošlo ke kolizi,
- set_climb(c) nastaví aktuální stoupání na c (číslo s plovoucí desetinnou čárkou),
- finished() vrátí true nachází-li se loď na pravém konci hracího pole.

V případě, že dojde k pokusu o posun lodě za konec pole, loď zůstane na posledním definovaném poli. Dojde-li ke kolizi, další volání move již nemají žádný efekt a vrací false.
Implicitně sestrojený stav hry má hrací pole délky 1 s překážkami $(-10, +10)$ a počáteční výška i stoupání lodě jsou 0.

```
struct flight;
```

3.r.5 [qfield] Uvažme těleso $\mathbb{Q}$ a číslo j takové, že $j^2 = 2$ (tedy zejména $j \notin \mathbb{Q}$). Těleso $\mathbb{Q}$ můžeme rozšířit na tzv. (algebraické) číselné těleso (v tomto případě konkrétně kvadratické těleso) $\mathbb{Q}(j)$, kterého prvky jsou tvaru $a + bj$, kde $a, b \in \mathbb{Q}$.
Vaším úkolem je naprogramovat typ, který prvky tohoto tělesa reprezentuje, a umožňuje je sčítat, násobit a rozhodovat jejich rovnost. Rozmyslete si vhodnou reprezentaci; uvažte zejména jak bude vypadat výsledek násobení $(a + bj) \cdot (x + yj)$.

```
struct qf;
```

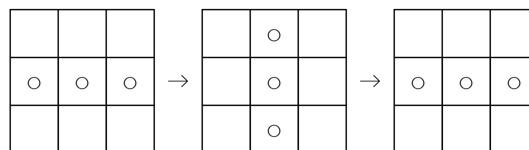
Protože k dispozici máme pouze celá čísla, k zápisu jednoho prvku $\mathbb{Q}(j)$ budeme potřebovat 4 (vystačili bychom si se třemi? pokud ano, jaké to má výhody a nevýhody?).

```
qf make_qf( int a_nom, int a_den, int b_nom, int b_den );
```

3.r.6 [life] Hra života je dvourozměrný buněčný automat: buňky tvoří čtvercovou síť a mají dva možné stavy: živá nebo mrtvá. V každé generaci (kroku simulace) spočítáme novou hodnotu pro každou buňku, a to podle těchto pravidel (výsledek závisí na současném stavu buňky a na tom, kolik z jejích 8 sousedů je živých):

stav	živí sousedi	výsledek
živá	0–1	mrtvá
živá	2–3	živá
živá	4–8	mrtvá
mrtvá	0–2	mrtvá
mrtvá	3	živá
mrtvá	4-8	mrtvá

Příklad krátké periodické hry:



Napište funkci, která dostane na vstupu množinu živých buněk a vrátí množinu buněk, které jsou živé po n generacích. Živé buňky jsou zadané svými souřadnicemi, tzn. jako dvojice x, y.

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

grid life( const grid &, int );
```

Část 4: Životní cyklus hodnot

Ukázky:

1. xxx
2. numbers – a list of numbers which remember their type
3. refs – overloading with references
4. pool – ownership and indirect references

Elementary exercises:

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

Preparatory exercises:

1. distance – vzdálenost v rovině
2. least – nejmenší prvek bez kopií
3. loan – bankovní půjčka
4. zipper – jednoduchá datová struktura
5. rpn – postfixová aritmetika s přetěžováním
6. eval – infixová aritmetika s vlastnictvím zdrojů

Regular exercises:

1. complex – přetěžování konstruktorů
2. xxx
3. search – vyhledávací strom s uzly v poli
4. bitptr – odkaz na jednotlivý bit
5. xxx
6. xxx

Část 4.d: Demonstrace (ukázky)

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

```
struct number
{
    bool is_real;
    int integer = 0;
    double real = 0;
```

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

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

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

```
class numbers
{
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. first should work correctly when we call it on a constant,

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

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

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

The case of `int_vector` is completely analogous:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This is an ubiquitous problem, and if you think about it, we have encountered it a lot, but did not have to think about it yet: the items in an `std::vector` or `std::map` or other containers have the same property: their lifetime must match the lifetime of the instance which owns them.
This is one of the most important differences between C and C++: if we do C++ right, most of the time, we do not need to manage object lifetimes manually. This is achieved through two main mechanisms:

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

In general, destroying objects at an appropriate time is the job of the owner of the object – whether the owner is a function (this is the case with by-value arguments and local variables) or another object (attributes, elements of a container and so on). Additionally, this happens transparently for the user: the compiler takes care of inserting the right calls at the right places to ensure everything is destroyed at the right time.
The end result is modular or composable resource management – well-behaved objects can be composed into well-behaved composites without any additional glue or boilerplate.
To make things easy for now, we will take advantage of existing con-

tainers to do resource management for us, which will save us from writing destructors (the proverbial glue, which is boring to write and easy to get wrong). In the next chapter, we will see how we can use smart pointers for the same purpose.
We will be keeping the nodes of our binary tree in an `std::vector` – this means that each node has an index which we can use to refer to that node. In other words, in this demo (and in some of this week's exercises) indices will play the role of pointers. Since 0 is a valid index, we will use -1 to indicate an invalid ('null') reference. Besides 'pointers' to the left and right child, the node will contain a single integer value.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    int make( int value )
    {
        _pool.emplace_back();
        _pool.back().value = value;
        return _pool.size() - 1;
    }

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

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

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

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

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

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

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

Main entry point for adding nodes.

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

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

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

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

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

Část 4.e: Elementární příklady

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

```
struct point;
```

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

```
struct circle;
```

Část 4.p: Přípravy

4.p.1 [distance] V této úloze se budeme pohybovat v dvourozměrné ploše a počítat při tom uraženou vzdálenost. Typ walk nechť má tyto metody:

- line(p) – přesuneme se do bodu p po úsečce,
- arc(p, radius) – přesuneme se do bodu p po kružnicovém oblouku s poloměrem radius,[17] přitom radius je alespoň polovina vzdálenosti do bodu p po přímce,
- backtrack() – vrátíme se po vlastních stopách do předchozího bodu (vzdálenost se přitom bude zvětšovat),
- distance() – vrátí celkovou dosud uraženou vzdálenost.

Metody nechť je možné libovolně řetězit, tzn. je-li w typu walk, následovný výraz musí být dobře utvořený:

```
w.line( { 1, 1 } )
 .line( { 2, 1 } )
 .backtrack()
 .arc( { 4, 1 }, 7 );
```

Hodnoty typu walk lze sestrojit zadáním počátečního bodu, nebo implicitně – začínají pak z bodu (0, 0).

```
struct walk;
```

4.p.2 [least] Uvažme typ element hodnot, které (z nějakého důvodu) nelze kopírovat. Našim cílem bude naprogramovat funkci, která vrátí nejmenší prvek ze zadaného vektoru hodnot typu element. Definici tohoto typu nijak neměňte.

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

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

Naprogramujte funkci (nebo rodinu funkcí) least tak, že volání least(d) vrátí nejmenší prvek zadaného vektoru d typu data. Dobře si rozmyslete platnost (délku života) dotčených objektů.
Nápověda: Protože nemůžete přímo manipulovat hodnotami typu element, zkuste využít k zapamatování si dosud nejlepšího kandidáta iterátor.

4.p.3 [loan] V tomto příkladu se budeme zabývat (velmi zjednodušenými) bankovními půjčkami. Navrhneme 2 třídy: account, která bude mít obvyklé metody deposit, withdraw, balance, a které konstruktoru lze předat počáteční zůstatek (v opačném případě bude implicitně nula). Druhá třída bude loan – její konstruktor přijme referenci na instanci třídy account a velikost půjčky (int). Sestrojením hodnoty typu loan se na přidružený účet připíše vypůjčená částka. Třída loan nechť má metodu repay, která zařídí (případně částečné – určeno volitelným parametrem typu int) navrácení půjčky. Proběhne-li vše v pořádku, metoda vrátí true, jinak false.
Zůstatek účtu může být záporný, hodnota půjčky nikoliv. Půjčka musí být vždy plně splacena (tzn. zabraňte situaci, kdy se informace o dluhu „ztratí“ aniž by byl tento dluh splacen).

```
struct account;
struct loan;
```

4.p.4 [zipper] V tomto příkladu implementujeme jednoduchou datovou strukturu, které se říká zipper – reprezentuje sekvenci prvků, přitom právě jeden z nich je aktivní (angl. focused). Abychom se nemuseli zabývat generickými datovými typy, vystačíme si celočíselnými položkami. Typ zipper nechť má toto rozhraní:

- konstruktor vytvoří jednoprvkový zipper z celého čísla,
- shift_left (shift_right) aktivuje prvek vlevo (vpravo) od toho dosud aktivního, a to v čase O(1); metody vrací true bylo-li posun možné provést (jinak nic nezmění a vrátí false),
- insert_left (insert_right) přidá nový prvek těsně vlevo (vpravo) od právě aktivního prvku (opět v čase O(1))
- focus zpřístupní aktivní prvek (pro čtení i zápis)
- volitelně metody erase_left (erase_right) které odstraní prvek nalevo (napravo) od aktivního, v čase O(1), a vrátí true bylo-li to možné

```
struct zipper;
```

4.p.5 [rpn] Naprogramujte jednoduchý zásobníkový evaluátor aritmetických výrazů zapsaných v RPN (postfixové notaci). Operace:

- push vloží na vrch pracovního zásobníku konstantu,
- apply přijme hodnotu jednoho ze tří níže definovaných typů, které reprezentují operace a příslušnou operaci provede,
- metoda top poskytne přístup k aktuálnímu vrcholu pracovního zásobníku, včetně možnosti změnit jeho hodnotu,
- pop odstraní jednu hodnotu z vrcholu zásobníku a vrátí ji,
- empty vrátí true je-li pracovní zásobník prázdný.

Podobně jako v příkladu distance zařiďte, aby bylo možné metody push a apply libovolně řetězit. Všechny tři operace uvažujeme jako binární.

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

struct eval;
```

4.p.6 [eval] V tomto cvičení naprogramujeme vyhodnocování infixových aritmetických výrazů. Zároveň zabezpečíme, aby bylo lze sdílet společné podvýrazy (tzn. uložit je jenom jednou a při dalším výskytu je pouze odkázat). Proto budeme uzly ukládat ve společném úložišti.

[17] Potřebný středový úhel naleznete například vyřešením rovnoramenného trojúhelníku s délkou ramene radius a základnou určenou vzdáleností spojovaných bodů.

```
const int op_mul = 1;
const int op_add = 2;
const int op_num = 3;

struct node
{
```

Operace, kterou tento uzel reprezentuje (viz konstanty definované výše). Pouze uzly typu `mul` a `add` mají potomky.

```
    int op;
```

Položky `left` a `right` jsou indexy, přičemž hodnota -1 značí neplatný odkaz. Položka `is_root` je nastavena na `true` právě tehdy, když tento uzel není potomkem žádného jiného uzlu.

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

Hodnota uzlu, je-li tento uzel typu `op_num`.

```
    int value = 0;
};

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

Dočasná reference na uzel, kterou lze použít při procházení stromu, ale která je platná pouze tak dlouho, jako hodnota typu `eval`, která ji vytvořila. Přidejte (konstantní) metody `left` a `right`, kterých výsledkem je nová hodnota typu `node_ref` popisující příslušný uzel, a dále metodu `compute`, která vyhodnotí podstrom začínající v aktuálním uzlu. Konečně přidejte metodu `update`, která upraví hodnotu v aktuálním uzlu. Metodu `update` je dovoleno použít pouze na uzly typu `op_num`.

```
struct node_ref;
```

Typ `eval` reprezentuje výraz jako celek. Umožňuje vytvářet nové výrazy ze stávajících (pomocí metod `add`, `mul` a `num`) a procházet strom výrazů (počínaje z kořenů, které lze získat metodou `roots`).

```
struct eval
{
    node_pool _pool;

    std::vector< node_ref > roots();

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

Část 4.r: Řešené úlohy

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

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

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

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

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

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

4.r.2 [`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;
```

4.r.3 [`search`] Implement a binary search tree, i.e. a binary tree which maintains the search property. That is, a value of each node is:

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

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

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

The `node_ref` class then ought to provide:

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

Calling `root` on an empty tree is undefined.

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

using node_pool = std::vector< node >;

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

4.r.4 [`bitptr`] Implement 2 classes, `bitptr` and `const_bitptr`, which provide access to a single (mutable or constant) bit. Instances of these classes should behave as pointers in principle, though we don't yet have tools to make them behave this way syntactically (that comes next week). In the meantime, let's use the following interface:

- `bool get()` – read the pointed-to bit,
- `void set( bool )` – write the same,
- `void advance()` – move to the next bit,
- `void advance( int )` – move by a given number of bits,
- `bool valid()` – is the pointer valid?

A default-constructed `bitptr` is not valid. Moving an invalid `bitptr` results in another invalid `bitptr`. Otherwise, a `bitptr` is constructed from a `std::byte` pointer and an `int` with value between 0 and 7 (with 0 being the least-significant bit). A `bitptr` constructed this way is always considered valid, regardless of the value of the `std::byte` pointer passed to it.

```
class bitptr;
class const_bitptr;
```

4.r.6 [`sort`] Implement `sort` which works both on vectors (`std::vector`) and linked lists (`std::list`) of integers. The former should use in-place quicksort, while the latter should use merge sort (it's okay to use the `splice` and `merge` methods on lists, but not `sort`). Feel free to refer back to `01/r5` for the quicksort.

Část S.1: Funkce a hodnoty

Tato sada obsahuje příklady zaměřené na zápis jednoduchých podprogramů (zejména čistých funkcí) a na práci s hodnotami (jak skalárními, tak složenými).

1. a_queens – problém osmi dam,
2. b_city – panorama města,
3. c_magic – doplnění magického čtverce,
4. d_reversi – třírozměrná verze hry Reversi,
5. e_cellular – celulární automat na kružnici,
6. f_natural – přirozená čísla se sčítáním a násobením.

Úlohu a byste měli zvládnout vyřešit hned po první přednášce. Příklady b, c vyžadují znalosti nejvýše z druhé přednášky, příklad d si vystačí s třetí přednáškou a konečně při řešení příkladů e, f můžete potřebovat i základní znalosti z přednášky čtvrté.
Pozor! Řešení některých příkladů z této sady může být potřebné pro vyřešení příkladů v sadách pozdějších. Doporučujeme prolistovat si i zadání pozdějších sad.

Část S.1.a: queens

V této úloze budete programovat řešení tzv. problému osmi královen (osmi dam). Vaše řešení bude predikát, kterého vstupem bude jediné 64-bitové bezznaménkové číslo (použijeme typ uint64_t), které popisuje vstupní stav šachovnice: šachovnice 8×8 má právě 64 polí, a pro reprezentaci každého pole nám stačí jediný bit, který určí, je-li na tomto políčku umístěna královna.
Políčka šachovnice jsou uspořádána počínaje levým horním rohem (nejvyšší, tedy 64. bit) a postupují zleva doprava (druhé pole prvního řádku je uloženo v 63. bitu, tj. druhém nejvyšším) po řádcích zleva doprava (první pole druhého řádku je 56. bit), atd., až po nejnižší (první) bit, který reprezentuje pravý dolní roh.
Predikát nechť je pravdivý právě tehdy, není-li žádná královna na šachovnici ohrožena jinou. Program musí pracovat správně i pro případy, kdy je na šachovnici jiný počet královen než 8. Očekávaná složitost je v řádu 64^2 operací – totiž $O(n^2)$ kde n představuje počet políček.
Poznámka: preferované řešení používá pro manipulaci se šachovnicí pouze bitové operace a zejména nepoužívá standardní kontejnery. Řešení, které bude nevhodně používat kontejnery (spadá sem např. jakékoliv použití std::vector) nemůže získat známku A.

```
bool queens( std::uint64_t board );
```

Část S.1.b: city

V tomto úkolu budeme pracovat s dvourozměrnou „mapou města", kterou reprezentujeme jako čtvercovou síť. Na každém políčku může stát budova (tvaru kvádru), která má barvu a celočíselnou výšku (budova výšky 1 má tvar krychle). Pro práci s mapou si zavedeme:

- typ building, který reprezentuje budovu,
- typ coordinates, který určuje pozici budovy a nakonec
- typ city, který reprezentuje mapu jako celek.

Jihozápadní (levý dolní) roh mapy má souřadnice $(0, 0)$, x-ová souřadnice roste směrem na východ, y-ová směrem na sever.

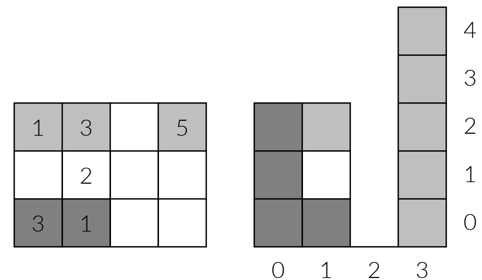
```
struct building
{
    int height;
    int colour;
};

using coordinates = std::tuple< int, int >;
using city = std::map< coordinates, building >;
```

Nejsou-li nějaké souřadnice v mapě přítomny, znamená to, že na tomto místě žádná budova nestojí.
Vaším úkolem je podle zadané mapy spočítat pravoúhlý boční pohled na město (panorama), které vznikne při pohledu z jihu, a které bude popsáno typy:

- column, který reprezentuje jeden sloupec a pro každou viditelnou jednotkovou krychli obsahuje jedno číslo, které odpovídá barvě této krychle,
- skyline, které obsahuje pro každou x-ovou souřadnici mapy jednu hodnotu typu column, kde index příslušného sloupce odpovídá jeho x-ové souřadnici.

```
using column = std::vector< int >;
using skyline = std::vector< column >;
```

Vstup a odpovídající výstup si můžete představit např. takto:



Napište čistou funkci compute_skyline která výpočet provede. Počet prvků každého sloupce musí být právě výška nejvyšší budovy s danou x-ovou souřadnicí.

```
skyline compute_skyline( const city & );
```

Část S.1.c: magic

Magický čtverec je čtvercová síť o rozměru $n \times n$, kde

1. každé políčko obsahuje jedno z čísel 1 až n^2 (a to tak, že se žádné z nich neopakuje), a
2. má tzv. magickou vlastnost: součet každého sloupce, řádku a obou diagonál je stejný. Tomuto součtu říkáme „magická konstanta".

Částečný čtverec je takový, ve kterém mohou (ale nemusí) být některá pole prázdná. Vyřešením částečného čtverce pak myslíme doplnění případných prázdných míst ve čtvercové síti tak, aby měl výsledný čtverec obě výše uvedené vlastnosti. Může se samozřejmě stát, že síť takto doplnit nelze.

```
using magic = std::vector< std::int16_t >;
```

Vaším úkolem je naprogramovat backtrackující solver, který čtverec doplní (je-li to možné), nebo rozhodne, že takové doplnění možné není. Napište podprogram magic_solve, o kterém platí:

- návratová hodnota (typu bool) indikuje, bylo-li možné vstupní čtverec doplnit,
- parametr in specifikuje částečný čtverec, ve kterém jsou prázdná pole reprezentována hodnotou 0, a který je uspořádaný po řádcích a na indexu 0 je levý horní roh,
- je-li výsledkem hodnota true, zapíše zároveň doplněný čtverec do výstupního parametru out (v opačném případě parametr out nezmění),
- vstupní podmínkou je, že velikost vektoru in je druhou mocninou, ale o stavu předaného vektoru out nic předpokládat nesmíte.

Složitost výpočtu může být až exponenciální vůči počtu prázdných polí, ale solver nesmí prohledávat stavy, o kterých lze v čase $O(n^2)$

rozhodnout, že je doplnit nelze. Prázdná pole vyplňujte počínaje levým horním rohem po řádcích (alternativou je zajistit, že výpočet v jiném pořadí nebude výrazně pomalejší).

```
bool magic_solve( const magic &in, magic &out );
```

Část S.1.d: reversi

Předmětem tohoto úkolu je hra Reversi (známá také jako Othello), avšak ve třírozměrné verzi. Hra se tedy odehrává v kvádru, který se skládá ze sudého počtu polí (krychlí) v každém ze tří základních směrů (podle os x, y a z). Dvě taková pole můžou sousedit stěnou (6 směrů), hranou (12 směrů) nebo jediným vrcholem (8 směrů). Pole může být prázdné, nebo může obsahovat černý nebo bílý hrací kámen.
Hru hrají dva hráči (černý a bílý, podle barvy kamenů, které jim patří) a pravidla hry jsou přímočarým rozšířením těch klasických dvourozměrných:

- každý hráč má na začátku 4 kameny, rozmístěné kolem prostředního bodu kvádru (jedná se tedy o 8 polí, které tento bod sdílí), a to tak, že žádná dvě obsazená pole stejné barvy nesdílí stěnu, přičemž pole s nejmenšími souřadnicemi ve všech směrech obsahuje bílý kámen,
- hráči střídavě pokládají nový kámen do volného pole; je-li na tahu bílý hráč, pokládá bílý kámen do pole, které musí být nepřerušeně spojeno[18] černými kameny s alespoň jedním stávajícím bílým kamenem (černý hráč hraje analogicky),
- po položení nového kamene se barva všech kamenů, které leží na libovolné takové spojnici, změní na opačnou (tzn. přebarví se na barvu právě položeného kamene).

Začíná bílý hráč. Hra končí, není-li možné položit nový kámen (ani jedné barvy). Vyhrává hráč s více kameny na ploše.

```
struct reversi
{
```

Metoda start začne novou hru na ploše zadané velikosti. Případná rozehraná partie je tímto voláním zapomenuta. Po volání start je na tahu bílý hráč.

```
    void start( int x_size, int y_size, int z_size );
```

Metoda size vrátí aktuální velikost hrací plochy.

```
    std::tuple< int, int, int > size() const;
```

Metoda play položí kámen na souřadnice zadané parametrem. Barva kamene je určena tím, který hráč je právě na tahu. Byl-li tah přípustný, metoda vrátí true a další volání položí kámen opačné barvy. V opačném případě se hrací plocha nezmění a stávající hráč musí provést jiný tah. Není určeno, co se má stát v případě, že hra ještě nezačala, nebo již skončila (tzn. nebyla zavolána metoda start, nebo by metoda finished vrátila true).

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

Nemůže-li aktivní hráč provést platný tah, zavolá metodu pass. Tato vrátí true, jedná-li se o korektní přeskočení tahu (má-li hráč k dispozici jakýkoliv jiný platný tah, musí nějaký provést – volání pass v takovém případě vrátí false a aktivní hráč se nemění).
Platí stejná omezení na stav hry jako u metody play.

```
    bool pass();
```

[18] Uvažujme dvojicí polí (krychlí) A, B a úsečku u, která spojuje jejich středy, a která prochází středem stěny, hrany nebo vrcholem pole A. Nepřerušeným spojením myslíme všechna pole, které úsečka u protíná, vyjma A a B samotných. Dvojici polí, pro které potřebná úsečka u neexistuje, nelze nepřerušeně spojit.

Metoda-predikát finished vrací true právě tehdy, nemůže-li ani jeden z hráčů provést platný tah a hra tedy skončila. Výsledek volání není určen pro hru, která dosud nezačala (nedošlo k volání metody start).

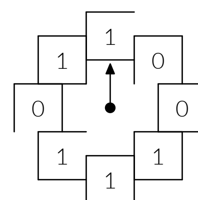
```
    bool finished() const;
```

Metodu result je povoleno zavolat pouze v případě, že hra skončila (tzn. volání finished by vrátilo true). Její návratovou hodnotou je rozdíl v počtu kamenů mezi bílým a černým hráčem – kladné číslo značí výhru bílého hráče, záporné výhru černého hráče a nula značí remízu.

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

Část S.1.e: cellular

Vaším úkolem bude naprogramovat jednoduchý simulátor jednorozměrného celulárního automatu. Implementace bude sestávat ze dvou struktur, automaton_state a automaton, které jsou popsané níže. Zadané rozhraní je nutné dodržet.
Definujte strukturu, automaton_state, která reprezentuje stav automatu definovaného na kružnici, s buňkami číslovanými po směru hodinových ručiček od indexu 0. Stav si můžete představit takto:



Platným indexem je libovolné celé číslo – určuje o kolik políček od indexu 0 se posuneme (kladná čísla po směru, záporná proti směru hodinových ručiček).
Jsou-li s, t hodnoty typu automaton_state, dále i, n jsou hodnoty typu int a v je hodnota typu bool, tyto výrazy musí být dobře utvořené:

- automaton_state(s) vytvoří nový stav, který je stejný jako stav s,
- automaton_state(n) vytvoří nový stav o n buňkách, které jsou všechny nastaveny na false (pro n ≤ 0 není definováno),
- s.size() vrátí aktuální počet buněk stavu s,
- s.get(i) vrátí hodnotu buňky na indexu i,
- s.set(i, v) nastaví buňku na indexu i na hodnotu v,
- s.extend(n) vloží n nových buněk nastavených na hodnotu false, a to tak, že nové buňky budou na indexech -1 až $-n$ (je-li n záporné, chování není definováno),
- s.reduce(n) odstraní n buněk proti směru hodinových ručiček, počínaje indexem -1 (je-li n ≥ s.size() nebo je n záporné, chování není definováno),
- t = s upraví stav t tak, aby byl stejný jako s,
- t == s je true právě když jsou stavy s a t stejné,
- t != s je true právě když se stavy s a t liší,
- t <= s se vyhodnotí na true právě když pro všechny indexy i platí s.get(i) || !t.get(i),
- t < s se vyhodnotí na true právě když t <= s && t != s.

Je-li to možné, výrazy musí pracovat správně i v případech, kdy jsou s a/nebo t konstantní. Metody size, get a set musí pracovat v konstantním čase, vše ostatní v čase nejvýše lineárním.

```
struct automaton_state;
```

Struktura automaton reprezentuje samotný automat. Třída si udržuje interní stav, na kterém provádí výpočty (tzn. například volání metody step() změní tento interní stav).
Následovné výrazy musí být dobře utvořené (kde a, b jsou hodnoty typu automaton, s je hodnota typu automaton_state, a konečně n a rule jsou hodnoty typu int):

- `automaton( rule, n )` sestrojí automat s `n` buňkami nastavenými na `false` (chování pro `n ≤ 0` není definováno), a s pravidlem `rule` zadaným tzv. Wolframovým kódem (chování je definováno pouze pro `rule` v rozsahu 0 až 255 včetně),
- `automaton( rule, s )` sestrojí nový automat a nastaví jeho vnitřní stav tak, aby byl stejný jako `s` (význam parametru `rule` je stejný jako výše),
- `a.state()` umožní přístup k internímu stavu, a to tak, že je možné jej měnit, není-li samotné `a` konstantní (např. `a.state().set( 3, true )` nastaví buňku interního stavu s indexem 3 na hodnotu `true`),
- `a = b` nastaví automat `a` tak, aby byl stejný jako automat `b` (zejména tedy upraví nastavené pravidlo a vnitřní stav),
- `a.step()` provede jeden krok výpočtu na vnitřním stavu (jeden krok nastaví všechny buňky vnitřního stavu na další generaci),
- `a.reset( s )` přepíše vnitřní stav kopií stavu `s`.

Hodnoty, které vstupují do výpočtu nové generace buňky podle zadaného Wolframova kódu, čteme po směru hodinových ručiček (tzn. ve směru rostoucích indexů).
Krok výpočtu musí mít nejvýše lineární (časovou i paměťovou) složitost.

```
struct automaton;
```

Část S.1.f: `natural`

Vaším úkolem je tentokrát naprogramovat strukturu, která bude reprezentovat libovolně velké přirozené číslo (včetně nuly). Tyto hodnoty musí být možné:

- sčítat (operátorem `+`),
- odečítat (`x - y` je ovšem definováno pouze za předpokladu `x ≥ y`),
- násobit (operátorem `*`),
- libovolně srovnávat (operátory `==`, `!=`, `<`, atd.),
- mocnit na kladný exponent typu `int` metodou `power`,
- sestrojit z libovolné nezáporné hodnoty typu `int`.

Implicitně sestrojená hodnota typu `natural` reprezentuje nulu. Všechny operace krom násobení musí být nejvýše lineární vůči počtu dvojkových cifer většího z reprezentovaných čísel. Násobení může mít v nejhorším případě složitost přímo úměrnou součinu $m \cdot n$ (kde m a n jsou počty cifer operandů).

```
struct natural;
```

Část 5: Ukazatele

Before you dig into the demonstrations and exercises, do not forget to read the extended introduction below. That said, the units for this week are, starting with demonstrations:

1. `queue` – a queue with stable references
2. `finexp` – like regexps but finite
3. `expr` – expressions with operators and shared pointers
4. `family` – genealogy with weak pointers

Elementary exercises:

1. `dynarray` – a simple array with a dynamic size
2. `list` – a simple linked list with minimal interface
3. `iota`

Preparatory exercises:

1. `unrolled` – a linked list of arrays
2. `bittrie` – bitwise tries (radix trees)
3. `solid` – efficient storage of optional data
4. `chartrie` – binary tree for holding string keys
5. `bdd` – binary decision diagrams
6. `rope` – a string-like structure with cheap concatenation

Regular exercises:

1. `circular` – a singly-linked circular list
2. `zipper` – implementing zipper as a linked list
3. `segment` – a binary tree of disjoint intervals
4. `diff` – automatic differentiation
5. `critbit` – more efficient version of binary tries
6. `refcnt` † – implement a simple reference-counted heap

Část 5.A: Exclusive Ownership

So far, we have managed to almost entirely avoid thinking about memory management: standard containers manage memory behind the scenes. We sometimes had to think about copies (or rather avoiding them), because containers could carry a lot of memory around and copying all that memory without a good reason is rather wasteful (this is why we often pass arguments as `const` references and not as values).
This week, we will look more closely at how memory management works and what we can do when standard containers are inadequate to deal with a given problem. In particular, we will look at building our own pointer-based data structures and how we can retain automatic memory management in those cases using `std::unique_ptr`.
TODO

Část 5.B: Shared Ownership

While `unique_ptr` is very useful and efficient, it only works in cases where the ownership structure is clear, and a given object has a single owner. When ownership of a single object is shared by multiple entities (objects, running functions or otherwise), we cannot use `unique_ptr`.
To be slightly more explicit: shared ownership only arises when the lifetime of the objects sharing ownership is not tied to each other. If A owns B and A and B both need references to C, we can assign the ownership of C to object A: since it also owns B, it must live at least as long as B and hence there ownership is not actually shared.
However, if A needs to be able to transfer ownership of B to some other, unrelated object while still retaining a reference to C, then C will indeed be in shared ownership: either A or B may expire first, and hence neither can safely destroy the shared instance of C to which they both keep references. In many modern languages, this problem is solved by a garbage collector, but alas, C++ does not have one.
Of course, it is usually better to design data structures in a way that allows for clear, 1:1 ownership structure. Unfortunately, this is not always easy, and sometimes it is not the most efficient solution either. Specifically, when dealing with large immutable (or persistent, in the functional programming sense) data structures, shared ownership can save considerable amount of memory, without introducing any ill side-effects, by only storing common sub-structures once, instead of cloning them. Of course, there are also cases where shared mutable state is the most efficient solution to a problem.

Část 5.d: Demonstrace (ukázky)

5.d.1 [queue] In this example, we will demonstrate the use of `std::unique_ptr`, which is an RAII class for holding (owning) values dynamically allocated from the heap. We will implement a simple one-way, non-indexable queue. We will require that it is possible to erase elements from the middle in O(1), without invalidating any other itera-

tors. The standard containers which could fit:

- std::deque fails the erase in the middle requirement,
- std::forward_list does not directly support queue-like operation, hence using it as a queue is possible but awkward; wrapping std::forward_list would be, however, a viable approach to this task, too,
- std::list works well as a queue out of the box, but has twice the memory overhead of std::forward_list.

As usual, since we do not yet understand templates, we will only implement a queue of integers, but it is not hard to imagine we could generalize to any type of element.
Since we are going for a custom, node-based structure, we will need to first define the class to represent the nodes. For sake of simplicity, we will not encapsulate the attributes.

```
struct queue_node
{
```

We do not want to handle all the memory management ourselves. To rule out the possibility of accidentally introducing memory leaks, we will use std::unique_ptr to manage allocated memory for us. Whenever a unique_ptr is destroyed, it will free up any associated memory. An important limitation of unique_ptr is that each piece of memory managed by a unique_ptr must have exactly one instance of unique_ptr pointing to it. When this instance is destroyed, the memory is deallocated.

```
    std::unique_ptr< queue_node > next;
```

Besides the structure itself, we of course also need to store the actual data. We will store a single integer per node.

```
    int value;
};
```

We will also need to be able to iterate over the queue. For that, we define an iterator, which is really just a slightly generalized pointer (you may remember nibble_ptr from last week). We need 3 things: pre-increment, dereference and inequality.

```
struct queue_iterator
{
    queue_node *node;
```

The queue will need to create instances of a queue_iterator. Let's make that convenient.

```
    queue_iterator( queue_node *n ) : node( n ) {}
```

The pre-increment operator simply shifts the pointer to the next pointer of the currently active node.

```
    queue_iterator &operator++()
    {
        node = node->next.get();
        return *this;
    }
```

Equality is very simple (we need this because the condition of iteration loops is it != c.end(), including range for loops). We could implement != directly, but == is usually more natural, and given ==, the compiler will derive != for us automatically.

```
    bool operator==( const queue_iterator &o ) const
    {
        return o.node == node;
    }
```

And finally the dereference operator: this is what will be called when *it is evaluated. Also notice the const/non-const overloads (for completeness, it is often preferable to return a const reference from the const overload; this depends on the element type).

```
    int &operator*()       { return node->value; }
    int  operator*() const { return node->value; }
};
```

This class represents the queue itself. We will have push and pop to add and remove items, empty to check for emptiness and begin and end to implement iteration.

```
class queue
{
```

We will keep the head of the list in another unique_ptr. An empty queue will be represented by a null head. Also worth noting is that when using a list as a queue, the head is where we remove items. The end of the queue (where we add new items) is represented by a plain pointer because it does not own the node (the node is owned by its predecessor).

```
    std::unique_ptr< queue_node > first;
    queue_node *last = nullptr;
public:
```

As mentioned above, adding new items is done at the 'tail' end of the list. This is quite straightforward: we simply create the node, chain it into the list (using the last pointer as a shortcut) and point the last pointer at the newly appended node. We need to handle empty and non-empty lists separately because we chose to represent an empty list using null head, instead of using a dummy node.

```
    void push( int v )
    {
        if ( last ) /* non-empty list */
        {
            last->next = std::make_unique< queue_node >();
            last = last->next.get();
        }
        else /* empty list */
        {
            first = std::make_unique< queue_node >();
            last = first.get();
        }

        last->value = v;
    }
```

Reading off the value from the head is easy enough. However, to remove the corresponding node, we need to be able to point first at the next item in the queue.
Unfortunately, we cannot use normal assignment (because copying unique_ptr is not allowed). We will have to use an operation that is called move assignment and which is written using a helper function in from the standard library, called std::move.
Operations which move their operands invalidate the moved-from instance. In this case, first->next is the moved-from object and the move will turn it into a null pointer. In any case, the next pointer which was invalidated was stored in the old head node and by rewriting first, we lost all pointers to that node. This means two things:

1. the old head's next pointer, now null, is no longer accessible
2. memory allocated to hold the old head node is freed

```
    int pop()
    {
        int v = first->value;
        first = std::move( first->next );
```

Do not forget to update the last pointer in case we popped the last item.

```
        if ( !first ) last = nullptr;
```

```
        return v;
    }
```

The emptiness check is simple enough.

```
    bool empty() const { return !last; }
```

Now the begin and end methods. We start iterating from the head (since we have no choice but to iterate in the direction of the next pointers). The end method should return a so-called past-the-end iterator, i.e. one that comes right after the last real element in the queue. For an empty queue, both begin and end should be the same. Conveniently, the next pointer in the last real node is nullptr, so we can use that as our end-of-queue sentinel quite naturally. You may want to go back to the pre-increment operator of queue_iterator just in case.

```
    queue_iterator begin() { return { first.get() }; }
    queue_iterator end()   { return { nullptr }; }
```

And finally, erasing elements. Since this is a singly-linked list, to erase an element, we need an iterator to the element before the one we are about to erase. This is not really a problem, because erasing at the head is done by pop. We use the same move assignment construct that we have seen in pop earlier.

```
    void erase_after( queue_iterator i )
    {
        assert( i.node->next );
        i.node->next = std::move( i.node->next->next );
    }
};

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

We start by constructing an (empty) queue and doing some basic operations on it. For now, we only try to insert and remove a single element.

```
    queue q;
    assert( q.empty() );
    q.push( 7 );
    assert( !q.empty() );
    assert( q.pop() == 7 );
    assert( q.empty() );
```

Now that we have emptied the queue again, we add a few more items and try erasing one and iterating over the rest.

```
    q.push( 1 );
    q.push( 2 );
    q.push( 7 );
    q.push( 3 );
```

We check that erase works as expected. We get an iterator that points to the value 2 from above and use it to erase the value 7.

```
    queue_iterator i = q.begin();
    ++ i;
    assert( *i == 2 );
    q.erase_after( i );
```

We can use instances of queue in range for loops, because they have begin and end, and the types those methods return (i.e. iterators) have dereference, inequality and pre-increment.

```
    int x = 1;
    for ( int v : q )
        assert( v == x++ );
```

That went rather well, let's just check that the order of removal is the same as the order of insertion (first in, first out). This is how queues should behave.

```
    assert( q.pop() == 1 );
    assert( q.pop() == 2 );
    assert( q.pop() == 3 );
    assert( q.empty() );
}
```

5.d.3 [expr] In this example program, we will look at using shared pointers and operator overloading to get a nicer version of our expression examples, this time with sub-structure sharing: that is, doing something like a + a will not duplicate the sub-expression a.
Like in week 7, we will define an abstract base class to represent the nodes of the expression tree.

```
struct expr_base
{
    virtual int eval() const = 0;
    virtual ~expr_base() = default;
};
```

Since we will use (shared) pointers to expr_base quite often, we can save ourselves some typing by defining a convenient type alias: expr_ptr sounds like a reasonable name.

```
using expr_ptr = std::shared_ptr< expr_base >;
```

We will have two implementations of expr_base: one for constant values (nothing much to see here),

```
struct expr_const : expr_base
{
    const int value;
    expr_const( int v ) : value( v ) {}
    int eval() const override { return value; }
};
```

and another for operator nodes. Those are more interesting, because they need to hold references to the sub-expressions, which are represented as shared pointers.

```
struct expr_op : expr_base
{
    enum op_t { add, mul } op;
    expr_ptr left, right;
    expr_op( op_t op, expr_ptr l, expr_ptr r )
        : op( op ), left( l ), right( r )
    {}

    int eval() const override
    {
        if ( op == add ) return left->eval() + right->eval();
        if ( op == mul ) return left->eval() * right->eval();
        assert( false );
    }
};
```

In principle, we could directly overload operators on expr_ptr, but we would like to maintain the illusion that expressions are values. For that reason, we will implement a thin wrapper that provides a more natural interface (and also takes care of operator overloading). Again, the expr class essentially provides Java-like object semantics – which is quite reasonable for immutable objects like our expression trees here.

```
struct expr
{
    expr_ptr ptr;
    expr( int v ) : ptr( std::make_shared< expr_const >( v ) ) {}
    expr( expr_ptr e ) : ptr( e ) {}
    int eval() const { return ptr->eval(); }
};
```

The overloaded operators simply construct a new node (of type expr_op and wrap it up in an expr instance.

```
expr operator+( expr a, expr b )
{
    return { std::make_shared< expr_op >( expr_op::add,
                                          a.ptr, b.ptr ) };
}

expr operator*( expr a, expr b )
{
    return { std::make_shared< expr_op >( expr_op::mul,
                                          a.ptr, b.ptr ) };
}

int main() /* demo */
{
    expr a( 3 ), b( 7 ), c( 2 );
    expr ab = a + b;
    expr bc = b * c;
    expr abc = a + b * c;

    assert( a.eval() == 3 );
    assert( b.eval() == 7 );
    assert( ab.eval() == 10 );
    assert( bc.eval() == 14 );
    assert( abc.eval() == 17 );
}
```

Část 5.e: Elementární příklady

5.e.1 [dynarray] Implement a dynamic array of integers with 2 operations: element access (using methods `get( i )` and `set( i, v )`) and `resize( n )`. The constructor takes the initial size as its only parameter.

```
struct dynarray;
```

5.e.2 [list] Implement a linked list of integers, with `head`, `tail` (returns a reference) and `empty`. Asking for a `head` or `tail` of an empty list has undefined results. A default-constructed list is empty. The other constructor takes an int (the value of head) and a reference to an existing list. It will should make a copy of the latter.

```
class list;
```

5.e.3 [iota] Write a class `iota`, which can be iterated using a `range` for to yield a sequence of numbers in the range `start`, `end - 1` passed to the constructor.

```
class iota;
```

Část 5.p: Přípravy

5.p.1 [unrolled] Předmětem tohoto cvičení je datová struktura, tzv. „rozbalený“ zřetězený seznam. Typ, který bude strukturu zastřešovat, by měl mít metody `begin`, `end`, `empty` a `push_back`. Ukládat budeme celá čísla.
Rozdíl mezi běžným zřetězeným seznamem a rozbaleným seznamem spočívá v tom, že ten rozbalený udržuje v každém uzlu několik hodnot (pro účely tohoto příkladu 4). Samozřejmě, poslední uzel nemusí být zcela zaplněný. Aby měla taková struktura smysl, požadujeme, aby byly hodnoty uloženy přímo v samotném uzlu, bez potřeby další alokace paměti.
Návratová hodnota metod `begin` a `end` bude „pseudo-iterátor“: bude poskytovat prefixový operátor zvětšení o jedničku (pre-increment), rovnost a operátor dereference. Více informací o tomto typu objektu naleznete například v ukázce `d1_queue`.
V tomto příkladu není potřeba implementovat mazání prvků.

```
struct unrolled_node;
struct unrolled_iterator;
struct unrolled;
```

5.p.2 [bittrie] Binární trie je binární stom, který kóduje množinu bitových řetězců, s rychlým vkládáním a vyhledáváním. Každá hrana kóduje jeden bit.
Klíč chápeme jako sekvenci bitů – každý bit určuje, kterým směrem budeme ve stromě pokračovat (0 = doleva, 1 = doprava). Bitový řetězec budeme chápat jako přítomný v reprezentované množině právě tehdy, kdy přesně popisuje cestu k listu. Pro jednoduchost budeme klíče reprezentovat jako vektor hodnot typu `bool`.

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

struct trie_node;
```

Pro jednoduchost nebudeme programovat klasickou metodu `insert`. Místo toho umožníme uživateli přímo vystavět trie pomocí metod `root` (zpřístupní kořen trie) a `make` (vloží nový uzel: parametry určí rodiče a směr – 0 nebo 1 – ve kterém bude uzel vložen). V obou případech je výsledkem odkaz na uzel, který lze předat metodě `make`.
Hlavní část úkolu tedy spočívá v implementaci metody `has`, která pro daný klíč rozhodne, je-li v množině přítomen.

```
struct trie;
```

5.p.3 [solid] V tomto cvičení se zaměříme na typy (v tomto cvičení typ `solid`) s volitelnými složkami (typ `transform_matrix`). Budou nás zejména zajímat situace, kdy je relativně častý případ, že volitelná data nejsou potřebná, a zároveň jsou dostatečně velká aby mělo smysl je oddělit do samostatného objektu (v samostatně alokované oblasti paměti). Zároveň budeme požadovat, aby logicky hodnoty hlavního typu (`solid`) vystupovaly jako jeden celek a nepřítomnost volitelných dat byla vnějšímu světu podle možnosti skrytá.
Typ `solid` bude reprezentovat nějaké třírozměrné těleso, zatímco typ `transform_matrix` bude popisovat třírozměrnou lineární transformaci takového tělesa, a bude tedy reprezentován devíti čísly s plovoucí desetinnou čárkou (3 řádky × 3 sloupce). Tyto hodnoty nechť jsou (přímo nebo nepřímo) položkami typu `transform_matrix` (bez jakékoliv další pomocné paměti). Implicitně sestrojená hodnota nechť reprezentuje identitu (hodnoty na hlavní diagonále rovné 1, mimo diagonálu 0).

```
struct transform_matrix;
```

Typ `solid` bude reprezentovat společné vlastnosti pevných těles (které nezávisí na konkrétním tvaru nebo typu tělesa). Měl by mít tyto metody:

- `pos_x`, `pos_y` a `pos_z` určí polohu těžiště v prostoru,
- `transform_entry( int r, int c )` udává koeficient transformační matice na řádku `r` a sloupci `c`,
- `transform_set( int r, int c, double v )` nastaví příslušný koeficient na hodnotu `v`,
- konstruktor přijme 3 parametry typu `double` (vlastní souřadnice x, y a z).

Výchozí transformační maticí je opět identita. Paměť pro tuto matici alokujte pouze v případě, že se oproti implicitnímu stavu změní některý koeficient.

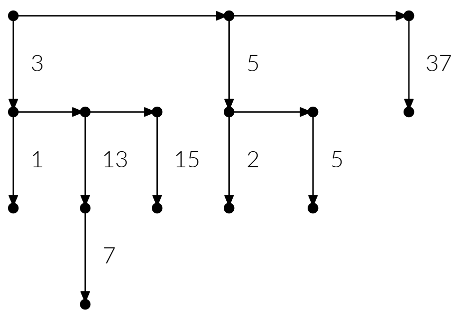
```
struct solid;
```

5.p.4 [inttrie] V tomto cvičení rozšíříme binární trie z `p2` – místo posloupnosti bitů budeme za klíče brát posloupnosti celých čísel typu `int`. Vylepšíme také rozhraní – místo ruční správy uzlů poskytneme přímo operaci vložení zadaného klíče.
Množiny budeme nadále kódovat do binárního stromu:

- levý potomek uzlu rozšiřuje reprezentovaný klíč o jedno celé číslo (podobně jako tomu bylo u binární trie) – toto číslo je tedy součástí levé hrany,
- pravý „potomek“ uzlu je ve skutečnosti jeho sourozenec, a hrana není nijak označená (přechodem doprava se klíč nemění),

* řetěz pravých potomků tvoří de-facto zřetězený seznam, který budeme udržovat seřazený podle hodnot na odpovídajících levých hranách.

Příklad: na obrázku je znázorněná trie s klíči [3, 1], [3, 13, 7], [3, 15], [5, 2], [5, 5], [37]. Levý potomek je pod svým rodičem, pravý je od něj napravo.



Můžete si představit takto reprezentovanou trie jako 2^{32}-ární, které by bylo zcela jistě nepraktické přímo implementovat. Proto reprezentujeme virtuální uzly pomyslného 2^{32}-árního stromu jako zřetězené seznamy pravých potomků ve fyzicky binárním stromě.

```
using key = std::vector< int >;
struct trie_node;
```

Rozhraní typu trie je velmi jednoduché: má metodu add, která přidá klíč a metodu has, která rozhodne, je-li daný klíč přítomen. Obě jako parametr přijmou hodnotu typu key. Prefixy vložených klíčů nepovažujeme za přítomné.

```
struct trie;
```

5.p.5 [bdd] Binární rozhodovací diagram je úsporná reprezentace booleovských funkcí více parametrů. Lze o nich uvažovat jako o orientovaném acyklickém grafu s dodatečnou sémantikou: každý vrchol je buď:

1. proměnná (parametr) a má dva následníky, kteří určí, jak pokračovat ve vyhodnocení funkce, je-li daná proměnná pravdivá resp. nepravdivá;
2. krom proměnných existují dva další uzly, které již žádné následníky nemají, a reprezentují výsledek vyhodnocení funkce; označujeme je jako 0 a 1.

Implementujte tyto metody:

* konstruktor má jeden parametr typu char – název proměnné, kterou reprezentuje kořenový uzel,
* one vrátí „pravdivý“ uzel (tzn. uzel 1),
* zero vrátí „nepravdivý“ uzel (tzn. uzel 0),
* root vrátí počáteční (kořenový) uzel,
* add_var přijme char a vytvoří uzel pro zadanou proměnnou; k jedné proměnné může existovat více než jeden uzel
* add_edge přijme rodiče, hodnotu typu bool a následníka,
* eval přijme map z char do bool a vyhodnotí reprezentovanou funkci na parametrech popsaných touto mapou (tzn. bude procházet BDD od kořene a v každém uzlu se rozhodne podle zadané mapy, až než dojde do koncového uzlu).

Chování není definováno, obsahuje-li BDD uzel, který nemá nastavené oba následníky.

```
struct bdd_node;
struct bdd;
```

5.p.6 [rope] Lano je datová struktura, která reprezentuje sekvenci, implementovaná jako binární strom, který má v listech klasická pole a ve vnitřních uzlech udržuje celočíselné váhy. Sdílení podstromů je dovolené a očekávané.
Váhou uzlu se myslí celková délka sekvence reprezentovaná jeho levým podstromem. Díky tomu lze lana spojovat v konstantním čase, a indexovat v čase lineárním k hloubce stromu.
Naprogramujte:

* konstruktor, který vytvoří jednouzlové lano z vektoru,
* konstruktor, který spojí dvě stávající lana,
* metodu get(i), která získá i-tý prvek,
* a set(i, value), která i-tý prvek nastaví na value.

Pro účely tohoto příkladu není potřeba implementovat žádnou formu vyvažování.

```
struct rope;
```

Část 5.r: Řešené úlohy

5.r.1 [circular] In this exercise, we will implement a slightly unusual data structure: a circular linked list, but instead of the usual access operators and iteration, it will have a rotate method, which rotates the entire list. We require that rotation does not invalidate any references to elements in the list.
If you think of the list as a stack, you can think of the rotate operation as taking an element off the top and putting it at the bottom of the stack. It is undefined on an empty list.
To add and remove elements, we will implement push and pop which work in a stack-like manner. Only the top element is accessible, via the top method. This method should allow both read and write access. Finally, we also want to be able to check whether the list is empty. As always, we will store integers in the data structure.

```
class circular;
```

5.r.2 [zipper] Implement our favourite data structure – a zipper of integers – this time using a unique_ptr-linked list extending both ways from the focus. Methods:

* (constructor) constructs a singleton zipper from an integer,
* shift_left and shift_right move the point of focus, in O(1), to the nearest left (right) element; they return true if this was possible, otherwise they return false and do nothing,
* push_left and push_right add a new element just left (just right) of the current focus, again in O(1),
* focus access the current item (read and write).

5.r.3 [segment] In this exercise, we will go back to building data structures, in this particular case a simple binary tree. The structure should represent a partitioning of an interval with integer bounds into a set of smaller, non-overlapping intervals.
Implement class segment_map with the following interface:

* the constructor takes two integers, which represent the limits of the interval to be segmented,
* a split operation takes a single integer, which becomes the start of a new segment, splitting the existing segment in two,
* query, given an integer n, returns the bounds of the segment that contains n, as an std::pair of integers.

The tree does not need to be self-balancing: the order of splits will determine the shape of the tree.

5.r.4 [diff] In this exercise, we will implement automatic differentiation of simple expressions. You will need the following rules:

* linearity: $(a \cdot f(x) + b \cdot g(x))' = a \cdot f'(x) + b \cdot g'(x)$
* the Leibniz rule: $(f(x) \cdot g(x))' = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
* chain rule: $(f(g(x)))' = f'(g(x)) \cdot g'(x)$
* derivative of exponential: $exp'(x) = exp(x)$

Define a type, expr (from expression), such that values of this type can be constructed from integers, added and multiplied, and exponentiated using function expnat (to avoid conflicts with the exp in the standard

library).

```
class expr; /* ref: 29 + 7 lines */
expr expnat( expr );
```

Implement function diff that accepts a single expr and returns the derivative (again in the form of expr). Define a constant x of type expr such that diff(x) is 1.

```
expr diff( expr ); /* ref: 11 lines */
// const expr x;
```

Finally, implement function eval which takes an expr and a double and it substitutes for x and computes the value of the expression.

```
double eval( expr, double ); /* ref: 11 lines */
```

Část 6: Dědičnost a pozdní vazba

This week will be about objects in the OOP (object-oriented programming) sense and about inheritance-based polymorphism. In OOP, classes are rarely designed in isolation: instead, new classes are derived from an existing base class (the derived class inherits from the base class). The derived class retains all the attributes (data) and methods (behaviours) of the base (parent) class, and usually adds something on top, or at least modifies some of the behaviours.
So far, we have worked with composition (though we rarely called it that). We say objects (or classes) are composed when attributes of classes are other classes (e.g. standard containers). The relationship between the outer class and its attributes is known as 'has-a': a circle has a center, a polynomial has a sequence of coefficients, etc.
Inheritance gives rise to a different type of relationship, known as 'is-a': a few stereotypical examples:

- a circle is a shape,
- a ball is a solid, a cube is a solid too,
- a force is a vector (and so is velocity).

This is where polymorphism comes into play: a function which doesn't care about the particulars of a shape or a solid or a vector can accept an instance of the base class. However, each instance of a derived class is an instance of the base class too, and hence can be used in its place. This is known as the Liskov substitution principle.
An important caveat: this does not work when passing objects by value, because in general, the base class and the derived class do not have the same size. Languages like Python or Java side-step this issue by always passing objects by reference. In C++, we have to do that explicitly if we want to use inheritance-based polymorphism. Of course, this also works with pointers (including smart ones, like std::unique_ptr).
With this bit of theory out of the way, let's look at some practical examples: the rest of theory (late binding in particular) will be explained in demonstrations:

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

Elementary exercises:

1. resistance – compute resistance of a simple circuit
2. perimeter – shapes and their perimeter length
3. fight – rock, paper and scissors

Preparatory exercises:

1. prisoner – the famous dilemma
2. bexpr – boolean expressions with variables
3. sexpr – a tree made of lists (lisp style)
4. network – a network of counters
5. filter – filter items from a data source
6. geometry – shapes and visitors

Regular exercises:

1. bom – polymorphism and collections
2. circuit – calling virtual methods within the class
3. loops – circuits with loops
4. xxx
5. xxx
6. while – interpreting while programs using an AST

Část 6.d: Demonstrace (ukázky)

6.d.1 [account] In this example, we will demonstrate the syntax and most basic use of inheritance. Polymorphism will not enter the picture yet (but we will get to that very soon: in the next example). We will consider bank accounts (a favourite subject, surely).
We will start with a simple, vanilla account that has a balance, can withdraw and deposit money. We have seen this before.

```
class account
{
```

The first new piece of syntax is the protected keyword. This is related to inheritance: unlike private, it lets subclasses (or rather subclass methods) access the members declared in a protected section. We also notice that the balance is signed, even though in this class, that is not strictly necessary: we will need that in one of the subclasses (yes, the system is already breaking down a little).

```
protected:
    int _balance;

public:
```

We allow an account to be constructed with an initial balance. We also allow it to be default-constructed, initializing the balance to 0.

```
    account( int initial = 0 )
        : _balance( initial )
    {}
```

Standard stuff.

```
    bool withdraw( int sum )
    {
        if ( _balance > sum )
        {
            _balance -= sum;
            return true;
        }

        return false;
    }

    void deposit( int sum ) { _balance += sum; }
    int balance() const { return _balance; }
};
```

With the base class in place, we can define a derived class. The syntax for inheritance adds a colon, :, after the class name and a list of classes to inherit from, with access type qualifiers. We will always use public inheritance. Also, did you know that naming things is hard?

```
class account_with_overdraft : public account
```

```
{
```

The derived class has, ostensibly, a single attribute. However, all the attributes of all base classes are also present automatically. That is, there already is an `int _balance` attribute in this class, inherited from `account`. We will use it below.

```
protected:
    int _overdraft;

public:
```

This is another new piece of syntax that we will need: a constructor of a derived class must first call the constructors of all base classes. Since this happens before any attributes of the derived class are constructed, this call comes first in the initialization section. The derived-class constructor is free to choose which (overloaded) constructor of the base class to call. If the call is omitted, the default constructor of the base class will be called.

```
    account_with_overdraft( int initial = 0, int overdraft = 0 )
        : account( initial ), _overdraft( overdraft )
    {}
```

The methods defined in a base class are automatically available in the derived class as well (same as attributes). However, unlike attributes, we can replace inherited methods with versions more suitable for the derived class. In this case, we need to adjust the behaviour of `withdraw`.

```
    bool withdraw( int sum )
    {
        if ( _balance + _overdraft > sum )
        {
            _balance -= sum;
            return true;
        }

        return false;
    }
};
```

Here is another example based on the same language features.

```
class account_with_interest : public account
{
protected:
    int _rate; /* percent per annum */

public:

    account_with_interest( int initial = 0, int rate = 0 )
        : account( initial ), _rate( rate )
    {}
```

In this case, all the inherited methods can be used directly. However, we need to add a new method, to compute and deposit the interest. Since naming things is hard, we will call it `next_year`. The formula is also pretty lame.

```
    void next_year()
    {
        _balance += ( _balance * _rate ) / 100;
    }
};
```

The way objects are used in this exercise is not super useful: the goal was to demonstrate the syntax and basic properties of inheritance. In modern practice, code re-use through inheritance is frowned upon (except perhaps for mixins, which are however out of scope for this subject). The main use-case for inheritance is subtype polymorphism, which we will explore in the next unit, `shapes.cpp`.

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

We first make a normal account and check that it behaves as expected. Nothing much to see here.

```
    account a( 100 );
    assert( a.balance() == 100 );
    assert( a.withdraw( 50 ) );
    assert( !a.withdraw( 100 ) );
    a.deposit( 10 );
    assert( a.balance() == 60 );
```

Let's try the first derived variant, an account with overdraft. We notice that it's possible to have a negative balance now.

```
    account_with_overdraft awo( 100, 100 );
    assert( awo.balance() == 100 );
    assert( awo.withdraw( 50 ) );
    assert( awo.withdraw( 100 ) );
    awo.deposit( 10 );
    assert( awo.balance() == -40 );
```

And finally, let's try the other account variant, with interest.

```
    account_with_interest awi( 100, 20 );
    assert( awi.balance() == 100 );
    assert( awi.withdraw( 50 ) );
    assert( !awi.withdraw( 100 ) );
    awi.deposit( 10 );
    assert( awi.balance() == 60 );
    awi.next_year();
    assert( awi.balance() == 72 );
}
```

6.d.2 [shapes] The inheritance model in C++ is an instance of a more general notion, known as subtyping. The defining characteristic of subtyping is the Liskov substitution principle: a value which belongs to a subtype (a derived class) can be used whenever a variable stores, or a formal argument expects, a value that belongs to a supertype (the base class). As mentioned earlier, in C++ this only extends to values passed by reference or through pointers.
We will first define a couple useful type aliases to represent points and bounding boxes.

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

Subtype polymorphism is, in C++, implemented via late binding: the decision which method should be called is postponed to runtime (with normal functions and methods, this happens during compile time). The decision whether to use early binding (static dispatch) or late binding (dynamic dispatch) is made by the programmer on a method-by-method basis. In other words, some methods of a class can use static dispatch, while others use dynamic dispatch.

```
class shape
{
public:
```

To instruct the compiler to use dynamic dispatch for a given method, put the keyword `virtual` in front of that method's return type. Unlike normal methods, a `virtual` method may be left unimplemented: this is denoted by the `= 0` at the end of the declaration. If a class has a method like this, it is marked as abstract and it becomes impossible to create instances of this class: the only way to use it is as a base class, through inheritance. This is commonly done to define interfaces. In our case, we will declare two such methods.

```
    virtual double area() const = 0;
    virtual bounding_box box() const = 0;
```

A class which introduces virtual methods also needs to have a destructor marked as virtual. We will discuss this in more detail in a later unit. For now, simply consider this to be an arbitrary rule.

```
    virtual ~shape() = default;
};
```

As soon as the interface is defined, we can start working with arbitrary classes which implement this interface, even those that have not been defined yet. We will start by writing a simple polymorphic function which accepts arbitrary shapes and computes the ratio of their area to the area of their bounding box.

```
double box_coverage( const shape &s )
{
```

Hopefully, you remember structured bindings (if not, revisit e.g. 03/rel.cpp).

```
    auto [ ll, ur ] = s.box();
    auto [ left, bottom ] = ll;
    auto [ right, top ] = ur;

    return s.area() / ( ( right - left ) * ( top - bottom ) );
}
```

Another function: this time, it accepts two instances of shape. The values it actually receives may be, however, of any type derived from shape. In fact, a and b may be each an instances of a different derived class.

```
bool box_collide( const shape &sh_a, const shape &sh_b )
{
```

A helper function (lambda) to decide whether a point is inside (or on the boundary) of a bounding box.

```
    auto in_box = []( const bounding_box &box, const point &pt )
    {
        auto [ x, y ] = pt;
        auto [ ll, ur ] = box;
        auto [ left, bottom ] = ll;
        auto [ right, top ] = ur;

        return x >= left && x <= right && y >= bottom && y <= top;
    };

    auto [ a, b ] = sh_a.box();
    auto box = sh_b.box();
```

The two boxes collide if either of the corners of one is in the other box.

```
    return in_box( box, a ) || in_box( box, b );
}
```

We now have the interface and two functions that are defined in terms of that interface. To make some use of the functions, however, we need to be able to make instances of shape, and as we have seen earlier, that is only possible by deriving classes which provide implementations of the virtual methods declared in the base class. Let's start by defining a circle.

```
class circle : public shape
{
    point _center;
    double _radius;
public:
```

The base class has a default constructor, so we do not need to explicitly call it here.

```
    circle( point c, double r ) : _center( c ), _radius( r ) {}
```

Now we need to implement the virtual methods defined in the base class. In this case, we can omit the virtual keyword, but we should specify that this method overrides one from a base class. This informs the compiler of our intention to provide an implementation to an inherited method and allows it (the compiler) to emit a warning in case we accidentally hide the method instead, by mistyping the signature. The most common mistake is forgetting the trailing const. Please always specify override where it is applicable.

```
    double area() const override
    {
        return 4 * std::atan( 1 ) * std::pow( _radius, 2 );
    }
```

Now the other virtual method.

```
    bounding_box box() const override
    {
        auto [ x, y ] = _center;
        double r = _radius;
        return { { x - r, y - r }, { x + r, y + r } };
    }
};
```

And a second shape type, so we can actually make some use of polymorphism. Everything is the same as above.

```
class rectangle : public shape
{
    point _ll, _ur; /* lower left, upper right */
public:

    rectangle( point ll, point ur ) : _ll( ll ), _ur( ur ) {}

    double area() const override
    {
        auto [ left, bottom ] = _ll;
        auto [ right, top ] = _ur;
        return ( right - left ) * ( top - bottom );
    }

    bounding_box box() const override
    {
        return { _ll, _ur };
    }
};

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

We cannot directly construct a shape, since it is abstract, i.e. it has unimplemented pure virtual methods. However, both circle and rectangle provide implementations of those methods which we can use.

```
    rectangle square( { 0, 0 }, { 1, 1 } );
    assert( square.area() == 1 );
    assert( square.box() == bounding_box( { 0, 0 }, { 1, 1 } ) );
    assert( box_coverage( square ) == 1 );

    circle circ( { 0, 0 }, 1 );
```

Check that the area of a unit circle is π, and the ratio of its area to its bounding box is π / 4.

```
    double pi = 4 * std::atan( 1 );
    assert( std::fabs( circ.area() - pi ) < 1e-10 );
    assert( std::fabs( box_coverage( circ ) - pi / 4 ) < 1e-10 );
```

The two shapes quite clearly collide, and if they collide, their bounding boxes must also collide. A shape should always collide with itself, and collisions are symmetric, so let's check that too.

```
    assert( box_collide( square, circ ) );
    assert( box_collide( circ, square ) );
    assert( box_collide( square, square ) );
```

```
    assert( box_collide( circ, circ ) );
```

Let's make a shape a bit further out and check the collision detection with that.

```
    circle c1( { 2, 3 }, 1 ), c2( { -1, -1 }, 1 );
    assert( !box_collide( circ, c1 ) );
    assert( !box_collide( c1, c2 ) );
    assert( !box_collide( c1, square ) );
    assert(  box_collide( c2, square ) );
}
```

6.d.3 [expr] To better understand polymorphism, we will need to set up some terminology, particularly:

- the notion of a static type, which is, essentially, the type written down in the source code, and of a
- dynamic type (also known as a runtime type), which is the actual type of the value that is stored behind a given reference (or pointer).

The relationship between the static and dynamic type may be:

- the static and dynamic type are the same (this was always the case until this week), or
- the dynamic type may be a subtype of the static type (we will see that in a short while).

Anything else is a bug.
We will use a very simple representation of arithmetic expressions as our example here. An expression is a tree, where each node carries either a value or an operation. We will want to explicitly track the type of each node, and for that, we will use an enumerated type. Those work the same as in C, but if we declare them using `enum class`, the enumerated names will be scoped: we use them as `type::sum`, instead of just `sum` as would be the case in C.

```
enum class type { sum, product, constant };
```

Now for the class hierarchy. The base class will be `node`.

```
class node
{
public:
```

The first thing we will implement is a `static_type` method, which tells us the static type of this class. The base class, however, does not have any sensible value to return here, so we will just throw an exception.

```
    type static_type() const
    {
        throw std::logic_error( "bad static_type() call" );
    }
```

The 'real' (dynamic) type must be a `virtual` method, since the actual implementation must be selected based on the dynamic type: this is exactly what late binding does. Since the method is `virtual`, we do not need to supply an implementation if we can't give a sensible one.

```
    virtual type dynamic_type() const = 0;
```

The interesting thing that is associated with each node is its value. For operation nodes, it can be computed, while for leaf nodes (type `constant`), it is simply stored in the node.

```
    virtual int value() const = 0;
```

We also observe the virtual destructor rule.

```
    virtual ~node() = default;
};
```

We first define the (simpler) leaf nodes, i.e. constants.

```
class constant : public node
{
    int _value;
public:
```

The leaf node constructor simply takes an integer value and stores it in an attribute.

```
    constant( int v ) : _value( v ) {}
```

Now the interface common to all `node` instances:

```
    type static_type() const { return type::constant; }
```

In methods of class `constant`, the static type of `this` is always[19] 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; }
};
```

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; }
```

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

And now the (dynamic-dispatch) interface mandated by the (indirect) base class node. We can use the same approach that we used in constant for dynamic_type:

```
    type dynamic_type() const override { return static_type(); }
```

And finally the logic. The static return type of left and right is const node &, but the method we call on each, value, uses dynamic dispatch (it is marked virtual in class node). Therefore, the actual method which will be called depends on the dynamic type of the respective child node.

```
    int value() const override
    {
        return left().value() + right().value();
    }
};
```

Basically a re-run of sum.

```
class product : public operation
{
public:
```

We will use a trick which will allow us to not type out the (boring and redundant) constructor. If all we want to do is just forward arguments to the parent class, we can use the following syntax. You do not have to remember it, but it can save some typing if you do.

```
    using operation::operation;
```

Now the interface methods.

```
    type static_type() const { return type::product; }
    type dynamic_type() const override { return static_type(); }

    int value() const override
    {
        return left().value() * right().value();
    }
};

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

Instances of class constant are quite straightforward. Let's declare some.

```
    constant const_1( 1 ),
             const_2( 2 ),
             const_m1( -1 ),
             const_10( 10 );
```

The constructor of sum accepts two instances of node, passed by reference. Since constant is a subclass of node, it is okay to use those, too.

```
    sum sum_0( const_1, const_m1 ),
        sum_3( const_1, const_2 );
```

The product constructor is the same. But now we will also try using instances of sum, since sum is also derived (even if indirectly) from node and therefore sum is a subtype of node, too.

```
    product prod_4( const_2, const_2 ),
            prod_6( const_2, sum_3 ),
            prod_40( prod_4, const_10 );
```

Let's also make a sum instance which has children of different types.

```
    sum sum_9( sum_3, prod_6 );
```

For all variables which hold values (i.e. not references), static type = dynamic type. To make the following code easier to follow, the static type of each of the above variables is explicitly mentioned in its name. Clearly, we can call the value method on the variables directly and it will call the right method.

```
    assert( const_1.value() == 1 );
    assert( const_2.value() == 2 );
    assert( sum_0.value() == 0 );
    assert( sum_3.value() == 3 );
    assert( prod_4.value() == 4 );
    assert( prod_6.value() == 6 );
    assert( prod_40.value() == 40 );
    assert( sum_9.value() == 9 );
```

However, the above results should already convince us that dynamic dispatch works as expected: the results depend on the ability of sum::value and product::value to call correct versions of the value method on their children, even though the static types of the references stored in operation are const node. We can however explore the behaviour in a bit more detail.

```
    const node &sum_0_ref = sum_0, &prod_6_ref = prod_6;
```

Now the static type of sum_0_ref is const node &, but the dynamic type of the value to which it refers is sum, and for prod_6_ref the static type is const node & and dynamic is product.

```
    assert( sum_0_ref.value() == 0 );
    assert( prod_6_ref.value() == 6 );
```

Let us also check the behaviour of left and right.

```
    assert( sum_0.left().value()  ==  1 );
    assert( sum_0.right().value() == -1 );
```

The static type through which we call left and right does not matter, because neither product nor sum provide a different implementation of the method.

```
    const operation &op = sum_0;
    assert( op.left().value() == 1 );
    assert( op.right().value() == -1 );
```

The final thing to check is the static_type and dynamic_type methods. By now, we should have a decent understanding of what to expect. Please note that sum_0 and sum_0_ref refer to the same instance and hence they have the same dynamic type, even though their static types differ.

```
    assert( sum_0.dynamic_type() == type::sum );
    assert( sum_0_ref.dynamic_type() == type::sum );

    assert( sum_0.static_type() == type::sum );

    try { sum_0_ref.static_type(); assert( false ); }
    catch ( const std::logic_error & ) {}
```

And the same is true about prod_6 and prod_6_ref.

```
    assert( prod_6.dynamic_type() == type::product );
    assert( prod_6_ref.dynamic_type() == type::product );
    assert( prod_6.static_type() == type::product );

    try { prod_6_ref.static_type(); assert( false ); }
    catch ( const std::logic_error & ) {}
}
```

6.d.4 [destroy] In this (entirely synthetic, sorry) example, we will look at object destruction, especially in the context of polymorphism. We first set up a few counters to track constructor and destructor calls.

```
static int bad_base_counter = 0, bad_derived_counter = 0,
           good_base_counter = 0, good_derived_counter = 0;

class bad_base
{
```

```
public:
    virtual int bad_dummy() { return 0; }

    bad_base() { bad_base_counter ++; }
```

We will knowingly break the virtual destructor rule here, to see why the rule exists.

```
    ~bad_base() { bad_base_counter --; }
};

class good_base
{
public:
    virtual int good_dummy() { return 0; }

    good_base() { good_base_counter ++; }
```

Notice the virtual.

```
    virtual ~good_base() { good_base_counter --; }
};
```

Let's add some innocent derived classes.

```
class bad_derived : public bad_base
{
public:
    bad_derived() { bad_derived_counter ++; }
    ~bad_derived() { bad_derived_counter --; }
};

class good_derived : public good_base
{
public:
    good_derived() { good_derived_counter ++; }
```

It is good practice to also add override to destructors of derived classes. This will tell the compiler we expect the base class to have a virtual destructor which we are extending. The compiler will emit an error if the base class destructor is (through some unfortunate accident) not marked as virtual.

```
    ~good_derived() override { good_derived_counter --; }
};

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

For regular variables, everything works as expected: constructors and destructors of all classes in the hierarchy are called.

```
    {
        bad_base bb;
        assert( bad_base_counter == 1 );
        bad_derived bd;
        assert( bad_base_counter == 2 );
        assert( bad_derived_counter == 1 );
    }

    assert( bad_base_counter == 0 );
    assert( bad_derived_counter == 0 );
```

Same thing with virtual destructors.

```
    {
        good_base gb;
        assert( good_base_counter == 1 );
        good_derived gd;
        assert( good_base_counter == 2 );
        assert( good_derived_counter == 1 );
    }

    assert( good_base_counter == 0 );
    assert( good_derived_counter == 0 );
```

However, problems start if an instance is destroyed through a pointer whose static type disagrees with the dynamic type. This cannot happen with references (unless the destructor is called explicitly), but it is entirely plausible with pointers, including smart pointers. Let's first demonstrate the case that works: good_derived.

```
    using good_ptr = std::unique_ptr< good_base >;
```

Please make good note of the fact, that the static type of the pointer refers to good_base, but the actual value stored in it has dynamic type good_derived.

```
    {
        good_ptr gp = std::make_unique< good_derived >();
        assert( good_base_counter == 1 );
        assert( good_derived_counter == 1 );
    }
```

Since the unique_ptr went out of scope, the instance stored behind it was destroyed. The counters should be both zero again.

```
    assert( good_base_counter == 0 );
    assert( good_derived_counter == 0 );
```

Let's observe what happens with the bad_base and bad_derived combination.

```
    using bad_ptr = std::unique_ptr< bad_base >;

    {
        bad_ptr bp = std::make_unique< bad_derived >();
        assert( bad_base_counter == 1 );
        assert( bad_derived_counter == 1 );
    }
```

The pointer went out of scope. Since the destructor was called using static dispatch, only the base class destructor was called. This is of course very problematic, since resources were leaked and invariants broken.

```
    assert( bad_base_counter == 0 );
    assert( bad_derived_counter == 1 );
```

Please note that some compilers (recent clang versions) will emit a warning if this happens. Unfortunately, this is not the case with gcc 9.2 which we are using (and which is a rather recent compiler). It is therefore unadvisable to rely on the compiler to catch this type of problem. Stay vigilant.

```
    // cxxflags: -Wno-delete-non-abstract-non-virtual-dtor
    // clang-tidy: -clang-diagnostic-delete-non-abstract-non-virtual-dtor
}
```

6.d.5 [factory] TODO: do not use strings here
As we have seen, subtype polymorphism allows us to define an interface in terms of virtual methods (that is, based on late dispatch) and then create various implementations of this interface.
It is sometimes useful to create instances of multiple different derived classes based on runtime inputs, but once they are created, to treat them uniformly. The uniform treatment is made possible by subtype polymorphism: if the entire interaction with these objects is done through the shared interface, the instances are all, at the type level, interchangeable with each other. The behaviour of those instances will of course differ, depending on their dynamic type.
When a system is designed this way, the entire program uses a single static type to work with all instances from the given inheritance hierarchy – the type of the base class. Let's define such a base class.

```
class part
{
public:
```

```
    virtual std::string description() const = 0;
    virtual ~part() = default;
};
```

Let's add a simple function which operates on generic parts. Working with instances is easy, since they can be passed through a reference to the base type. For instance the following function which formats a single line for a bill of materials (bom).

```
std::string bom_line( const part &p, int count )
{
    return std::to_string( count ) + "x " + p.description();
}
```

However, creation of these instances poses a somewhat unique challenge in C++: memory management. In languages like Java or C#, we can create the instance and return a reference to the caller, and the garbage collector will ensure that the instance is correctly destroyed when it is no longer used. We do not have this luxury in C++.
Of course, we could always do memory management by hand, like it's 1990. Fortunately, modern C++ provides smart pointers in the standard library, making memory management much easier and safer. Recall that a unique_ptr is an owning pointer: it holds onto an object instance while it is in scope and destroys it afterwards. Unlike objects stored in local variables, though, the ownership of the instance held in a unique_ptr can be transferred out of the function (i.e. an instance of unique_ptr can be legally returned, unlike a reference to a local variable).
This will make it possible to define a factory: a function which constructs instances (parts) and returns them to the caller. Of course, to actually define the function, we will need to define the derived classes which it is supposed to create.

```
using part_ptr = std::unique_ptr< part >;
part_ptr factory( std::string );
```

In the program design outlined earlier, the derived classes change some of the behaviours, or perhaps add data members (attributes) to the base class, but apart from construction, they are entirely operated through the interface defined by the base class.

```
class cog : public part
{
    int teeth;
public:
    cog( int teeth ) : teeth( teeth ) {}

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

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

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

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

    std::string description() const override
    {
        return std::to_string( _length ) + "mm M" +
               std::to_string( _thread ) + " screw";
    }
};
```

Now that we have defined the derived classes, we can finally define the factory function.

```
part_ptr factory( std::string desc )
{
```

We will use std::istringstream (first described in 06/streams.cpp) to extract a description of the instance that we want to create from a string. The format will be simple: the type of the part, followed by its parameters separated by spaces.

```
    std::istringstream s( desc );
    std::string type;
    s >> type; /* extract the first word */

    if ( type == "cog" )
    {
        int teeth;
        s >> teeth;
        return std::make_unique< cog >( teeth );
    }

    if ( type == "axle" )
        return std::make_unique< axle >();

    if ( type == "screw" )
    {
        int thread, length;
        s >> thread >> length;
        return std::make_unique< screw >( thread, length );
    }

    throw std::runtime_error( "unexpected part description" );
}

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

Let's first use the factory to make some instances. They will be held by part_ptr (i.e. unique_ptr with the static type part.

```
    part_ptr ax = factory( "axle" ),
             m7 = factory( "screw 7 50" ),
             m3 = factory( "screw 3 10" ),
             c8 = factory( "cog 8" ),
             c9 = factory( "cog 9" );
```

From the point of view of the static type system, all the parts created above are now the same. We can call the methods which were defined in the interface, or we can pass them into functions which work with parts.

```
    assert( ax->description() == "axle" );
    assert( m7->description() == "50mm M7 screw" );
    assert( m3->description() == "10mm M3 screw" );
    assert( c8->description() == "cog with 8 teeth" );
    assert( c9->description() == "cog with 9 teeth" );
```

Let's try using the bom_line function which we have defined earlier.

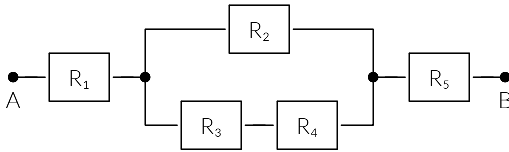
```
    assert( bom_line( *ax, 3 ) == "3x axle" );
    assert( bom_line( *m7, 20 ) == "20x 50mm M7 screw" );
```

At the end of the scope, the objects are destroyed and all memory is automatically freed.

```
}
```

Část 6.e: Elementární příklady

6.e.1 [resistance] We are given a simple electrical circuit made of resistors and wires, and we want to compute the total resistance between two points. The circuit is simple in the sense that in any given section, all its immediate sub-sections are either connected in series or in parallel. Here is an example:



The resistance that we are interested in is between the points A and B. Given R_1 and R_2 connected in series, the total resistance is $R = R_1 + R_2$. For the same resistors connected in parallel, the resistance is given by $1/R = 1/R_1 + 1/R_2$.
You will implement 2 classes: series and parallel, each of which represents a single segment of the circuit. Both classes shall provide a method add, that will accept either a number (double) which will add a single resistor to that segment, or a const reference to the opposite class (i.e. an instance of series should accept a reference to parallel and vice versa).

```
class series;
class parallel;
```

Then add a top-level function resistance, which accepts either a series or a parallel instance and computes the total resistance of the circuit described by that instance. The exact prototype is up to you.

6.e.2 [perimeter] Implement a simple inheritance hierarchy – the base class will be shape, with a pure virtual method perimeter, the 2 derived classes will be circle and rectangle. The circle is constructed from a radius, while the rectangle from a width and height, all of them floating-point numbers.

```
class shape;
class circle;
class rectangle;

bool check_shape( const shape &s, double p )
{
    return std::fabs( s.perimeter() - p ) < 1e-8;
}
```

6.e.3 [fight] There should be 4 classes: the base class gesture and 3 derived: rock, paper and scissors. Class gesture has a (pure virtual) method fight which takes another gesture (via a const reference) and returns true if the current gesture wins.
To do this, add another method, visit, which has 3 overloads, one each for rock, paper and scissors. Then override fight in each derived class, to simply call visit(*this) on the opposing gesture. The visit method knows the type of both this and the opponent (via the overload) – simply indicate the winner by returning an appropriate constant.

```
class rock;
class paper;
class scissors;
```

Keep the forward declarations, you will need them to define the overloads for visit.

```
class gesture;
```

Now define the 3 derived classes.

Část 6.p: Přípravy

6.p.1 [prisoner] Another exercise, another class hierarchy. The abstract base class will be called prisoner, and the implementations will be different strategies in the well-known game of (iterated) prisoner's dilemma.
The prisoner class should provide method betray which takes a boolean (the decision of the other player in the last round) and returns the decision of the player for this round. In general, the betray method should not be const, because strategies may want to remember past decisions (though we will not implement a strategy like that in this exercise).

```
class prisoner;
```

Implement an always-betray strategy in class traitor, the tit-for-tat strategy in vengeful and an always-cooperate in benign.

```
class traitor;
class vengeful;
class benign;
```

Implement a simple strategy evaluator in function play. It takes two prisoners and the number of rounds and returns a negative number if the first one wins, 0 if the game is a tie and a positive number if the second wins (the absolute value of the return value is the difference in scores achieved). 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 );
```

6.p.2 [bexpr] Boolean expressions with variables, represented as binary trees. Internal nodes carry a logical operation on the values obtained from children while leaf nodes carry variable references.
To evaluate an expression, we will need to supply values for each of the variables that appears in the expression. We will identify variables using integers, and the assignment of values will be done through the type input defined below. It is undefined behaviour if a variable appears in an expression but is not present in the provided input value.

```
using input = std::map< int, bool >;
```

Like earlier in expr.cpp, the base class will be called node, but this time will only define a single method: eval, which accepts a single input argument (as a const reference).

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

Internal nodes are all of the same type, and their constructor takes an unsigned integer, table, and two node references. Assuming bit zero is the lowest-order bit, the node operates as follows:

- false false → bit 0 of table
- false true → bit 1 of table
- true false → bit 2 of table
- true true → bit 3 of table

```
class operation; /* ref: 16 lines */
```

The leaf nodes carry a single integer (passed in through the constructor) – the identifier of the variable they represent.

```
class variable; /* ref: 7 lines */
```

6.p.3 [sexpr] An s-expression is a tree in which each node has an

arbitrary number of children. To make things a little more interesting, our s-expression nodes will own their children.
The base class will be called node (again) and it will have single (virtual) method: value, with no arguments and an int return value.

```
class node;
using node_ptr = std::unique_ptr< node >;
```

There will be two types of internal nodes: sum and product, and in this case, they will compute the sum or the product of all their children, regardless of their number. A sum with no children should evaluate to 0 and a product with no children should evaluate to 1.
Both will have an additional method: add_child, which accepts (by value) a single node_ptr and both should have default constructors. It is okay to add an intermediate class to the hierarchy.

```
class sum;
class product;
```

Leaf nodes carry an integer constant, given to them via a constructor.

```
class constant;
```

6.p.4 [network] In this exercise, we will define a network of counters, where each node has its own counter which starts at zero, and events which affect the counters propagate in the network. Different node types react differently to the events.
There are three basic events which can propagate through the network: reset will set the counter to 0, increment and decrement add and subtract 1, respectively.

```
enum class event { reset, increment, decrement };
```

The abstract base class, node, will define the polymorphic interface. Methods:

- react with a single argument of type event,
- connect which will take a reference to another node instance: the connection thus created starts in this and extends to the node given in the argument,
- read, a const method that returns the current value of the counter.

Think carefully about which methods need to be virtual and which don't. The counter is signed and starts at 0. Each node can have an arbitrary number of both outgoing and incoming connections.

```
class node;
```

Now for the node types. Each node type first applies the event to its own counter, then propagates (or not) some event along all outgoing connections. Implement the following node types:

- forward sends the same event it has received
- invert sends the opposite event
- gate resends the event if the new counter value is positive

```
class forward;
class invert;
class gate;
```

6.p.5 [filter] This exercise will be yet another take on a set of numbers. This time, we will add a capability to filter the numbers on output. It will be possible to change the filter applied to a given set at runtime. The base class for representing filters will contain a single pure virtual method, accept. The method should be marked const.

```
class filter;
```

The set (which we will implement below) will own the filter instance and hence will use a unique_ptr to hold it.

```
using filter_ptr = std::unique_ptr< filter >;
```

The set should have standard methods: add and has, the latter of which will respect the configured filter (i.e. items rejected by the filter will always test negative on has). The method set_filter should set the filter. If no filter is set, all numbers should be accepted. Calling set_filter with a nullptr argument should clear the filter.
Additionally, set should have begin and end methods (both const) which return very simple iterators that only provide dereference to an int (value), pre-increment and inequality. It is a good idea to keep two instances of std::set< int >::iterator in attributes (in addition to a pointer to the output filter): you will need to know, in the pre-increment operator, that you ran out of items when skipping numbers which the filter rejected.

```
class set_iterator;
class set;
```

Finally, implement a filter that only accepts odd numbers.

```
class odd;
```

6.p.6 [geometry] We will go back to a bit of geometry, this time with circles and lines: in this exercise, we will be interested in planar intersections. We will consider two objects to intersect when they have at least one common point. On the C++ side, we will use a bit of a trick with virtual method overloading (in a slightly more general setting, the trick is known as the visitor pattern).
First some definitions: the familiar point.

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

Check whether two floating-point numbers are 'essentially the same' (i.e. fuzzy equality).

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

We will need to use forward declarations in this exercise, since methods of the base class will refer to the derived types.

```
struct circle;
struct line;
```

These two helper functions are already defined in this file and may come in useful (like the slope class above).

```
double dist( point, point );
double dist( const line &, point );
```

A helper class which is constructed from two points. Two instances of slope compare equal if the slopes of the two lines passing through the respective point pairs are the same.

```
struct slope : std::pair< double, double >
{
    slope( point p, point q )
        : point( ( q.first  - p.first  ) / dist( p, q ),
                 ( q.second - p.second ) / dist( p, q ) )
    {}

    bool operator==( const slope &o ) const
    {
        auto [ px, py ] = *this;
        auto [ qx, qy ] = o;

        return ( close( px, qx ) && close( py, qy ) ) ||
               ( close( px, -qx ) && close( py, -qy ) );
    }

    bool operator!=( const slope &o ) const
    {
```

```
        return !( *this == o );
    }
};
```

Now we can define the class object, which will have a virtual method intersects with 3 overloads: one that accepts a const reference to a circle, another that accepts a const reference to a line and finally one that accepts any object.

```
class object;
```

Put definitions of the classes circle and line here. A circle is given by a point and a radius (double), while a line is given by two points. NB. Make the line attributes public and name them p and q to make the dist helper function work.

```
struct circle; /* ref: 18 lines */
struct line;   /* ref: 18 lines */
```

Definitions of the helper functions.

```
double dist( point p, point q )
{
    auto [ px, py ] = p;
    auto [ qx, qy ] = q;
    return std::sqrt( std::pow( px - qx, 2 ) +
                      std::pow( py - qy, 2 ) );
}

double dist( const line &l, point p )
{
    auto [ x2, y2 ] = l.q;
    auto [ x1, y1 ] = l.p;
    auto [ x0, y0 ] = p;

    return std::fabs( ( y2 - y1 ) * x0 - ( x2 - x1 ) * y0 +
                      x2 * y1 - y2 * x1 ) /
           dist( l.p, l.q );
}
```

Část 6.r: Řešené úlohy

6.r.1 [bom] TODO: do not use strings here
Let's revisit the idea of a bill of materials that made a brief appearance in factory.cpp, but in a slightly more useful incarnation.
Define the following class hierarchy: the base class, part, should have a (pure) virtual method description that returns an std::string. It should also keep an attribute of type std::string and provide a getter for this attribute called part_no() (part number). Then add 2 derived classes:

- resistor which takes the part number and an integral resistance as its constructor argument and provides a description of the form "resistor ?Ω" where ? is the provided resistance,
- capacitor which also takes a part number and an integral capacitance and provides a description of the form "capacitor ?μF" where ? is again the provided value.

```
class part;
class resistor;
class capacitor;
```

We will also use owning pointers, so let us define a convenient type alias for that:

```
using part_ptr = std::unique_ptr< part >;
```

That was the mechanical part. Now we will need to think a bit: we want a class bom which will remember a list of parts, along with their quantities and will own the part instances it holds. The interface:

- a method add, which accepts a part_ptr by value (it will take ownership of the instance) and the quantity (integer)
- a method find which accepts an std::string and returns a const reference to the part instance with the given part number,
- a method qty which returns the associated quantity, given a part number.

```
class bom;
```

6.r.2 [circuit] In this exercise, we will look at calling virtual methods from within the class, in an 'inverted' approach to inheritance. Most of the implementation will be part of the base class, in terms of a few (or in this case one) protected virtual methods.
We will implement a simple class hierarchy to represent a logical circuit: a bunch of components connected with wires. Each component will have at most 2 inputs and a single output (all of which are boolean values). Implement the following (non-virtual) methods:

- connect which takes an integer (0 or 1, the index of the input to connect) and a reference to another component and connects the output of the given component to the input of this component
- read with no arguments, which returns the current output of the component (this will of course depend on the state of the input components).

Both inputs start out unconnected. Unconnected inputs always read out false. Behaviour is undefined if there is a loop in the circuit (but see also loops.cpp).

```
class component;
```

The derived classes should be as follows:

- nand for which the output is the NAND logical function of the two inputs,
- source which ignores both inputs and reads out true,
- delay which behaves as follows: first time read is called, it always returns zero; subsequent read calls return a value that the input 0 had at the time of the previous call to read.

```
class nand;
class source;
class delay;
```

6.r.3 [loops] Same basic idea as circuit.cpp: we model a circuit made of components. Things get a bit more complicated in this version:

- loops are allowed
- parts have 2 inputs and 2 outputs each

The base class, with the following interface:

- read takes an integer (decides which output) and returns a boolean,
- connect takes two integers and a reference to a component (the first integer defines the input of this and the second integer defines the output of the third argument to connect).

There is more than one way to resolve loops, some of which require read to be virtual (that's okay). Please note that each loop must have at least one delay in it (otherwise, behaviour is still undefined). NB. Each component should first read input 0 and then input 1: the ordering will affect the result.

```
class component; /* ref: 30 lines */
```

A delay is a component that reads out, on both outputs, the value it has obtained on the corresponding input on the previous call to read.

```
class delay; /* ref: 20 lines */
```

A latch remembers one bit of information (starting at false):

- if both inputs read false, the remembered bit remains unchanged,

- if input 0 is false while input 1 is true the remembered bit is set to true,
- in all other cases, the remembered bit is set to false.

The value on output 0 is the new value of the remembered bit: there is no delay. The value on output 1 is the negation of output 0.

```
class latch; /* 15 lines */
```

Finally, the cnot gate, or a controlled not gate has the following behaviour:

- output 0 always matches input 0, while
- output 1 is set to:
  - input 1 if input 0 is true
  - negation of input 1 if input 0 is false

```
class cnot; /* ref: 11 lines */
```

6.r.6 [while] TODO: use single-character variables
Consider an abstract syntax tree of a very simple imperative programming language with structured control flow. Let there be 3 types of statements:

1. a variable increment, a ++,
2. a while loop of the form while (a != b) stmt, and finally
3. a block, which is a sequence of statements.

```
class statement;
using stmt_ptr = std::unique_ptr< statement >;
```

We will represent variables as strings. Provide an eval method which takes variable assignment as an argument and returns the same as a result. Variables used in the program but not given in the input start as 0. The variable assignment (i.e. the state of the program) is represented as an std::map from strings to integers.

```
using state = std::map< std::string, int >;
```

The constructors should be as follows:

- stmt_inc takes the name of the variable to be incremented,
- stmt_while takes 2 variable names and an stmt_ptr for the body, and
- stmt_block is default-constructed as a noop, but provides an append method to insert a stmt_ptr at the end.

```
class stmt_inc;
class stmt_while;
class stmt_block;
```

Část 7: Výjimky a princip RAII

Demonstrace:

1. exceptions – vyhazování a chytání výjimek
2. stdexcept – typy výjimek ve standardní knihovně
3. semaphore – automatická správa zdrojů

Elementární příklady:

1. xxx
2. counter – jednoduché počítadlo instancí
3. coffee – model automatu na kávu

Preparatory exercises:

1. fd – POSIX file descriptors
2. loan – database-style transactions with resources
3. library – borrowing books
4. parse – a simple parser which throws exceptions
5. invest – we further stretch the banking story
6. linear – linear equations, with some exceptions

Regular exercises:

1. printing – printing with a monthly budget
2. bsearch – a key-value vector which throws on failure
3. enzyme – cellular chemistry with RAII
4. tinyvec † – a vector in a fixed memory buffer
5. lock – a movable mutual exclusion token
6. bounded – a bounded queue that throws when full

Část 7.d: Demonstrace (ukázky)

7.d.1 [exceptions] Exceptions are, as their name suggests, a mechanism for handling unexpected or otherwise exceptional circumstances, typically error conditions. A canonic example would be trying to open a file which does not exist, trying to allocate memory when there is no free memory left and the like. Another common circumstance would be errors during processing user input: bad format, unexpected switches and so on.
NB. Do not use exceptions for 'normal' control flow, e.g. for terminating loops. That is a really bad idea (even though try blocks are cheap, throwing exceptions is very expensive).
This example will be somewhat banal. We start by creating a class which has a global counter of instances attached to it: i.e. the value of counter tells us how many instances of counted exist at any given time. Fair warning, do not do this at home.

```
int counter = 0;

struct counted
{
    counted()  { ++ counter; }
    ~counted() { -- counter; }

    counted( const counted & ) = delete;
    counted( counted && ) = delete;
    counted &operator=( const counted & ) = delete;
    counted &operator=( counted && ) = delete;
};
```

A few functions which throw exceptions and/or create instances of the counted class above. Notice that a throw statement immediately stops the execution and propagates up the call stack until it hits a try block (shown in the main function below). The same applies to a function call which hits an exception: the calling function is interrupted immediately.

```
int f() { counted x; return 7; }
int g() { counted x; throw std::bad_alloc(); assert( 0 ); }
int h() { throw std::runtime_error( "h" ); }
int i() { counted x; g(); assert( 0 ); }

int main() /* demo */
{
    bool caught = false;
```

A try block allows us to detect that an exception was thrown and react, based on the type and attributes of the exception. Otherwise, it is a regular block with associated scope, and behaves normally.

```
    try
    {
        counted x;
```

```
        assert( counter == 1 );
        f();
        assert( counter == 1 );
    }
```

One or more catch blocks can be attached to a try block: those describe what to do in case an exception of a matching type is thrown in one of the statements of the try block. The catch clause behaves like a prototype of a single-argument function – if it could be 'called' with the thrown exception as an argument, it is executed to handle the exception.
This particular catch block is never executed, because nothing in the associated try block above throws a matching exception (or rather, any exception at all):

```
    catch ( const std::bad_alloc & ) { assert( false ); }
```

The counted instance x above went out of scope:

```
    assert( counter == 0 );
```

Let's write another try block. This time, the i call in the try block throws, indirectly (via g) an exception of type std::bad_alloc.

```
    try { i(); }
```

To demonstrate how catch blocks are selected, we will first add one for std::runtime_error, which will not trigger (the 'prototype' does not match the exception type that was thrown):

```
    catch ( const std::runtime_error & ) { assert( false ); }
```

As mentioned above, each try block can have multiple catch blocks, so let's add another one, this time for the bad_alloc that is actually thrown. If the catch matches the exception type, it is executed and propagation of the exception is stopped: it is now handled and execution continues normally after the end of the catch sequence.

```
    catch ( const std::bad_alloc & ) { caught = true; }
```

Execution continues here. We check that the catch block was actually executed:

```
    assert( caught );
    assert( counter == 0 ); // no <counted> instances were leaked
}
```

7.d.2 [stdexcept] It is possible to sub-class standard exception classes. For most uses, std::runtime_error is the most appropriate base class.

```
class custom_exception : public std::runtime_error
{
public:
    custom_exception() : std::runtime_error( "custom" ) {}
};
```

This demo simply demonstrates some of the standard exception types (i.e. those that are part of the standard library, and which are thrown by standard functions or methods; as long as those methods or functions are not too arcane).

```
int main() /* demo */
{
    try
    {
        throw custom_exception();
        assert( false );
    }
```

As per standard rules, it's possible to catch exceptions of derived classes (of course including user-defined types) via a catch clause which accepts a reference to a superclass.

```
    catch ( const std::exception & ) {}

    try
    {
        std::vector x{ 1, 2 };
```

Attempting out-of-bounds access through at gives std::out_of_range

```
        x.at( 7 );
        assert( false );
    }
    catch ( const std::out_of_range & ) {}

    try
    {
```

If the string passed to stoi is not a number, we get back an exception of type std::invalid_argument.

```
        std::stoi( "foo" );
        assert( false );
    }
    catch ( const std::invalid_argument & ) {}

    try
    {
```

If an integer is too big to fit the result type, stoi throws std::out_of_range.

```
        std::stoi( "123456123456123456" );
        assert( false );
    }
    catch ( const std::out_of_range & ) {}

    try
    {
```

System-interfacing functions may throw std::system_error. Here, for instance, trying to detach a thread which was not started.

```
        std::thread().detach();
        assert( false );
    }
    catch ( const std::system_error & ) {}

    try
    {
```

Throwing a system_error is the appropriate reaction when dealing with a failure of a POSIX function which sets errno.

```
        int fd = ::open( "/does/not/exist", O_RDONLY );
        if ( fd < 0 )
            throw std::system_error( errno, std::system_category(),
                                     "opening /does/not/exist" );
        assert( false );
    }
    catch ( const std::system_error & ) {}

    try
    {
```

Passing a size that is more than max_size() when constructing or resizing an std::string or an std::vector gives us back an std::length_error. Note that the -1 turns into a really big number in this context.

```
        std::string x( -1, 'x' );
        assert( false );
    }
    catch ( const std::length_error & ) {}

    try
    {
```

```
        std::bitset< 128 > x;
        x[ 100 ] = true;
```

Trying to convert an std::bitset to an integer type may throw std::overflow_error, if there are bits set that do not fit into the target integer type.

```
        x.to_ulong();
        assert( false );
    }
    catch ( const std::overflow_error & ) {}
}
```

7.d.3 [semaphore] In this demo, we will implement a simple semaphore. A semaphore is a device which guards a resource of which there are multiple instances, but the number of instances is limited. It is a slight generalization of a mutex (which guards a singleton resource). Internally, semaphore simply counts the number of clients who hold the resource and refuses further requests if the maximum is reached. In a multi-threaded program, semaphores would typically block (wait for a slot to become available) instead of refusing. In a single-threaded program (which is what we are going to use for a demonstration), this would not work. Hence our get method returns a bool, indicating whether acquisition of the lock succeeded.

```
class semaphore
{
    int _available;
public:
```

When a semaphore is constructed, we need to know how many instances of the resource are available.

```
    explicit semaphore( int max ) : _available( max ) {}
```

Classes which represent resource managers (in this case 'things that can be locked' as opposed to 'locks held') have some tough choices to make. If they are impossible to copy/move/assign, users will find that they must not appear as attributes in their classes, lest those too become un-copyable (and un-movable) by default. However, this is how the standard library deals with the problem, see std::mutex or std::condition_variable. While it is the safest option, it is also the most annoying. Nonetheless, we will do the same.

```
    semaphore( const semaphore & ) = delete;
    semaphore &operator=( const semaphore & ) = delete;
```

We allow would-be lock holders to query the number of resource instances currently available. Perhaps if none are left, they can make do without one, or they can perform some other activity in the hopes that the resource becomes available later.

```
    int available() const
    {
        return _available;
    }
```

Finally, what follows is the 'low-level' interface to the semaphore. It is completely unsafe, and it is inadvisable to use it directly, other than perhaps in special circumstances. This being C++, such interfaces are commonly made available. Again see std::mutex for an example.
However, it would also be an option to be strict about it, make the following 2 methods private, and declare the RAII class defined below, semaphore_lock, to be a friend of this one.

```
    bool get()
    {
        if ( _available > 0 )
            return _available --;
        else
            return false;
    }

    void put()
    {
        ++ _available;
    }
};
```

We will want to write a RAII 'lock holder' class. However, since get above might fail, we need a way to indicate the failure in the RAII class as well. But constructors don't return values: it is therefore a reasonable choice to throw an exception. It is reasonable as long as we don't expect the failure to be a common scenario.

```
class resource_exhausted : public std::runtime_error
{
public:
    resource_exhausted()
        : std::runtime_error( "semaphore full" )
    {}
};
```

Now the RAII class itself. It will need to hold a reference to the semaphore for which it holds a lock (good thing the semaphore is not movable, so we don't have to think about its address changing). Of course, it must not be possible to make a copy of the resource class: we cannot duplicate the resource, which is a lock being held. However, it does make sense to move the lock to a new owner, if the client so wishes. Hence, both a move constructor and move assignment are appropriate.

```
class semaphore_lock
{
    semaphore *_sem = nullptr;
public:
```

To construct a semaphore lock, we understandably need a reference to the semaphore which we wish to lock. You might be wondering why the attribute is a pointer and the argument is a reference. The main difference between references and pointers (except the syntactic sugar) is that references cannot be null. In a correct program, all references always refer to valid objects. It does not make sense to construct a semaphore_lock which does not lock anything. Hence the reference. Why the pointer in the attributes? That will become clear shortly.
Before we move on, notice that, as promised, we throw an exception if the locking fails. Hence, no noexcept on this constructor.

```
    semaphore_lock( semaphore &s ) : _sem( &s )
    {
        if ( !_sem->get() )
            throw resource_exhausted();
    }
```

As outlined above, semaphore locks cannot be copied or assigned. Let's make that explicit.

```
    semaphore_lock( const semaphore_lock & ) = delete;
    semaphore_lock &operator=( const semaphore_lock & ) = delete;
```

The new object (the one initialized by the move constructor) is quite unremarkable. The interesting part is what happens to the 'old' (source) instance: we need to make sure that when it is destroyed, it does not release the resource (i.e. the lock held) – the ownership of that has been transferred to the new instance. This is where the pointer comes in handy: we can assign nullptr to the pointer held by the source instance. Then we just need to be careful when we release the resource (in the destructor, but also in the move assignment operator) – we must first check whether the pointer is valid.
Also notice the noexcept qualifier: even though the 'normal' constructor throws, we are not trying to obtain a new resource here, and there is

nothing in the constructor that might fail. This is good, because move constructors, as a general rule, should not throw.

```
    semaphore_lock( semaphore_lock &&src ) noexcept
        : _sem( src._sem )
    {
        src._sem = nullptr;
    }
```

We now define a helper method, release, which frees up (releases) the resource held by this instance. It will do this by calling put on the semaphore. However, if the semaphore is null, we do nothing: the instance has been moved from, and no longer owns any resources. Why the helper method? Two reasons:

1. it will be useful in both the move assignment operator and in the destructor,
2. the client might need to release the resource before the instance goes out of scope or is otherwise destroyed 'naturally' (compare `std::fstream::close()`).

```
    void release() noexcept
    {
        if ( _sem )
            _sem->put();
    }
```

Armed with release, writing both the move assignment and the destructor is easy. The move assignment is also noexcept, which is usually a pretty good idea.

```
    semaphore_lock &operator=( semaphore_lock &&src ) noexcept
    {
```

Self-move is not very useful in this case, forbid it.

```
        assert( &src != this );
```

First release the resource held by the current instance. We cannot hold both the old and the new resource at the same time.

```
        release();
```

Now we reset our _sem pointer and update the src instance – the resource is now in our ownership.

```
        _sem = src._sem;
        src._sem = nullptr;
        return *this;
    }

    ~semaphore_lock() noexcept
    {
        release();
    }
};

int main() /* demo */
{
    semaphore sem( 3 );
    sem.get();
    semaphore_lock l1( sem );
    bool l4_made = false;

    try
    {
        semaphore_lock l2( sem );
        assert( sem.available() == 0 );
        semaphore_lock l3 = std::move( l2 );
        assert( sem.available() == 0 );
        semaphore_lock l4 = std::move( l1 );
        assert( sem.available() == 0 );
        l4_made = true;
        semaphore_lock l5( sem );
        assert( false );
    }
    catch ( const resource_exhausted & ) {}

    assert( l4_made );
    assert( sem.available() == 2 );

    // clang-tidy: -clang-analyzer-deadcode.DeadStores
}
```

Část 7.e: Elementární příklady

7.e.2 [counter]

```
int counter = 0;
```

Přidejte konstruktory a destruktor typu counted tak, aby počítadlo counter vždy obsahovalo počet existujících hodnot typu counted. Nezapomeňte na pravidlo pěti (rule of five).

```
struct counted;
```

7.e.3 [coffee] Implement a coffee machine which gives out a token when the order is placed and takes the token back when it is done… at most one order can be in progress.
Throw this when the machine is already busy making coffee.

```
class busy {};
```

And this when trying to use a default-constructed or already-used token.

```
class invalid {};
```

Fill in the two classes. Besides constructors and assignment operators, add methods make and fetch to machine, to create and redeem tokens respectively.

```
class machine;
class token;
```

7.e.4 [lock] TBD lock a resource, with ownership transfer but no copy

Část 7.p: Přípravy

7.p.1 [fd] In POSIX systems, opening a file or a file-like resource gives us a file descriptor, a small number that can be passed to system calls such as read or write. The descriptor must be closed when it is no longer needed, by calling close on it exactly once (it is important not to close the same descriptor twice). Write a class which safely wraps a file descriptor so that we can't accidentally lose it or close it twice.
It should be possible to move-construct and move-assign file descriptors. A new valid descriptor can be created in 2 ways: by calling `fd::open( "file", flags )` or `fd::dup( raw_fd )` where flags and raw_fd are both int. Use POSIX functions open and dup to implement this. Run `man 2 open` and `man 2 dup` on aisa for details about these POSIX functions.
Add methods read and write to the fd class, the first will simply take an integer, read the given number of bytes and return an `std::string`. The latter will take an `std::string` and write it into the descriptor. Again see `man 2 read` and `man 2 write` on aisa for advice.
If open, read or write fails, throw `std::system_error`. Attempting to call read or write on an invalid descriptor (one that was default-constructed or already closed) should throw `std::invalid_argument`.

7.p.2 [queue] Naprogramujte typ queue, který bude reprezentovat omezenou frontu celých čísel (velikost fronty je zadaná jako parametr konstruktoru), s metodami:

- pop() – vrátí nejstarší vložený prvek,
- push(x) – vloží nový prvek x na konec fronty,
- empty() vrátí true je-li fronta prázdná.

Metody pop a push nechť v případě selhání skončí výjimkou queue_empty resp. queue_full. Všechny operace musí mít složitost O(1).

```
struct queue_empty;
struct queue_full;
struct queue;
```

7.p.3 [partition] Napište generický podprogram partition(seq, p), který přeuspořádá zadanou sekvenci tak, že všechny prvky menší než p budou předcházet všem prvkům rovným p budou předcházet všem prvkům větším než p. Sekvence seq má tyto metody:

- size() vrátí počet prvků uložených v sekvenci,
- swap(i, j) prohodí prvky na indexech i a j,
- compare(i, p) srovná prvek na pozici i s hodnotou p:
  - výsledek -1 znamená, že hodnota na indexu i je menší,
  - výsledek 0, že je stejná, a konečně
  - výsledek +1, že je větší než p.

Metoda compare může skončit výjimkou. V takovém případě vraťte sekvenci seq do původního stavu a výjimku propagujte dál směrem k volajícímu. Hodnoty typu seq nelze kopírovat, máte ale povoleno použít pro výpočet dodatečnou paměť. Metody size ani swap výjimkou skončit nemohou.

7.p.4 [buckets] Napište generický podprogram buckets(list, vec), který dostane na vstupu:

- referenci na seznam tokenů list, který má tyto metody:
  - front() – vrátí referenci na první token seznamu,
  - drop() – odstraní první token v seznamu,
  - empty() – vrátí true je-li senam prázdný,
- referenci na hodnotu vec neurčeného typu, který má metodu size(), a který lze indexovat celými čísly. Uvnitř vec jsou uloženy kontejnery typu, který poskytuje metodu emplace_back, která zkonstruuje nový token (parametry předané metodě emplace_back se přepošlou 1:1 konstruktoru typu token, stejně jako u knihovních metod emplace).

Tokeny nelze kopírovat, ani přiřazovat kopií, pouze přesouvat.
Podprogram bude ze vstupního seznamu list postupně odebírat tokeny, které vždy přesune metodou emplace_back na konec kontejneru vec[i % vec.size()], kde i je pořadové číslo odebraného tokenu v původním seznamu list (počínaje nulou).
Zabezpečte, aby byl výsledek konzistentní, a to i v případě, kdy selže alokace paměti (metoda emplace_back může selhat s výjimkou std::bad_alloc). Zejména se nesmí žádný token ztratit (tzn. do vec musí být přidány právě ty prvky, ktere byly odebrány z list).
Můžete se spolehnout, že dojde-li k výjimce std::bad_alloc, konstruktor hodnoty token dosud nebyl zavolán.

```
struct token
{
    token( int v ) : value( v ) {}
    token( const token & ) = delete;
    token( token &&o ) noexcept
        : value( o.value )
    {
        o.value = 0;
    }

    token &operator=( const token & ) = delete;
    token &operator=( token &&o )
    {
        value = o.value;

        if ( &o != this )
            o.value = 0;
        return *this;
    }
private:
    int value;
};
```

7.p.5 [invest] We will revisit 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 */
```

7.p.6 [linear] Write a solver for linear equations in 2 variables. The interface will be a little unconventional: overload operators +, * and == and define global constants x and y of suitable types, so that it is possible to write the equations as shown in main below.
Note that the return type of == does not have to be bool. It can be any type you like, including of course custom types. For solve, I would suggest looking up Cramer's rule.
ref: class eqn 25 lines, solve 8 lines, x and y 2 lines
If the system has no solution, throw an exception of type no_solution. Derive it from std::exception.

Část 7.r: Řešené úlohy

7.r.1 [printing] Jobs need resources (printing credits, where 1 page = 1 credit) which must be reserved when the job is queued, but are only consumed at actual printing time; jobs can be moved between queues (printers) by the system, and jobs that are still in the queue can be aborted.
The class job represents a document to be printed, along with resources that have already been earmarked for its printing.

- The constructor should take a numeric identifier, the name of the user who owns the job, and the number of pages (= credits allocated for the job),
- method owner should return the name of the owner,
- method page_count should return the number of pages.

```
class job;
```

A single queue instance represents a printer. It should have the following methods:

- dequeue: consume (print) the oldest job in the queue and return its id,
- enqueue: add a job to the queue,
- release(id): remove the job given by id from the queue and re-

turn it to the caller,
- page_count: number of pages in the queue.

You can assume that oldest job has the lowest id.

```
class queue;
```

7.r.2 [bsearch] Let's revisit the perennial favourite: binary search. We will implement a container similar to std::map. Let's assume that it'll be used in a search-heavy scenario, so the cost of insertion is much less important than the cost of lookup. A good candidate, then, would be a sorted vector (lookup is logarithmic like with a tree, but the data is stored much more compactly).
Implement at least emplace, an indexing operator, and at, with semantics familiar from std::map, except emplace should simply return a bool (we will not write iterators).
Finally, since we still don't know how to write generic classes, use strings for keys and token instances for values.

```
class token
{
    int _value;
public:
    token( int i ) : _value( i ) {}
    token( const token & ) = delete;
    token &operator=( const token & ) = delete;
    token( token && ) = default;
    token &operator=( token && ) = default;
    token &operator=( int v ) { _value = v; return *this; }
    bool operator==( int v ) const { return _value == v; }
};

class flat_map;
```

7.r.3 [enzyme] TBD. Reactions tie up enzymes, which return to the pool after the reaction is done. Different reactions need different sets of enzymes present, and a given enzyme cannot be used by more than one reaction at a time.

7.r.4 [tinyvec] † Implement tiny_vector, a class which works like a vector, but instead of allocating memory dynamically, it uses a fixed-size buffer (32 bytes) which is part of the object itself (use e.g. an std::array of bytes). Like earlier, we will use token as the value type. Provide the following methods:

- insert (take an index and an rvalue reference),
- erase (take an index),
- back and front, with appropriate return types.

In this exercise (unlike in most others), you are allowed to use reinterpret_cast.

```
class token
{
    int _value;
    bool _robbed = false;
public:
    static int _count;

    token( int i ) : _value( i ) { ++ _count; }
    ~token() { if ( !_robbed ) -- _count; }

    token( const token & ) = delete;
    token( token &&o ) : _value( o._value ) { o._robbed = true; }

    token &operator=( const token & ) = delete;
    token &operator=( token &&o )
    {
        if ( !_robbed && o._robbed ) -- _count;
        _value = o._value;
        _robbed = o._robbed;
        o._robbed = true;
        return *this;
    }

    token &operator=( int v )
    {
        _value = v;
        _robbed = false;
        return *this;
    }

    bool operator==( int v ) const
    {
        assert( !_robbed );
        return _value == v;
    }
};
```

Throw this if insert is attempted but the element wouldn't fit into the buffer.

```
class insufficient_space {};
```

Hint: Use uninitialized_* and destroy(_at) functions from the memory header.

```
class tiny_vector;

int token::_count = 0;
```

7.r.5 [lock] Implement class lock which holds a mutex locked as long as it exists. The lock instance can be moved around. For simplicity, the mutex itself is immovable.

```
class mutex
{
    bool _locked = false;
public:
    ~mutex() { assert( !_locked ); }

    mutex() = default;
    mutex( const mutex & ) = delete;
    mutex( mutex && ) = delete;
    mutex &operator=( const mutex & ) = delete;
    mutex &operator=( mutex && ) = delete;

    void lock() { assert( !_locked ); _locked = true; }
    void unlock() { assert( _locked ); _locked = false; }
    bool locked() const { return _locked; }
};

class lock;
```

7.r.6 [bounded] Implement a simple FIFO of integers. The constructor takes the maximum number of items in the queue as its sole argument. Add methods push, pop, full, empty and next (the last of which simply returns the next value, without changing anything). If the queue is full, push should throw insufficient_space.
Try to make the implementation efficient (i.e. no deque).

```
class insufficient_space;
class bounded_queue;
```

Část 8: Lambda, pokročilé operátory

Kapitola se připravuje.

Část S.2: Ukazatele, výjimky, OOP

Druhá sada přináší příklady zaměřené na objektově-orientované programování, na práci s ukazately a výjimkami a v neposlední řadě na správu zdrojů.

1. `a_natural` – rozšíření `s1/f_natural` o dělení,
2. `b_treap` – jednoduchý vyhledávací strom,
3. `c_robots` – programujeme roboty na mapě,
4. `d_network` – simulátor počítačové sítě s přepínači,
5. `e_query` – hledání v heterogenním stromě,
6. `f_scrap` – hra o zdrojích a zajímání.

Příklad `a` si vystačí se znalostmi z prvního bloku, příklad `b` lze vyřešit po nastudování 5. kapitoly, příklady `c` až `d` vyžadují znalost 6. kapitoly a konečně příklady `e`, `f` vyžadují znalost 7. kapitoly. Opět platí, že řešení nějakého příkladu z této sady může být potřebné pro vyřešení příkladu z poslední sady.

Část S.2.a: `natural`

Tento úkol rozšiřuje `s1/f_natural` o tyto operace (hodnoty `m` a `n` jsou typu `natural`):

- konstruktor, který přijme nezáporný parametr typu `double` a vytvoří hodnotu typu `natural`, která reprezentuje dolní celou část parametru,
- operátory `m / n` a `m % n` (dělení a zbytek po dělení; chování pro `n = 0` není definované),
- metodu `m.digits( n )` která vytvoří `std::vector`, který bude reprezentovat hodnotu `m` v soustavě o základu `n` (přitom nejnižší index bude obsahovat nejvýznamnější číslici),
- metodu `m.to_double()` která vrátí hodnotu typu `double`, která co nejlépe aproximuje hodnotu `m` (je-li $l = log_2(m) - 52$ a `d = m.to_double()`, musí platit $m - 2^l \leq$ `natural( d )` a zároveň `natural( d )` $\leq m + 2^l$; je-li `m` příliš velké, než aby šlo typem `double` reprezentovat, chování je nedefinované).

Převody z/na typ `double` musí být lineární v počtu bitů operandu. Dělení může mít složitost nejvýše kvadratickou v počtu bitů levého operandu. Metoda `digits` smí vůči počtu bitů `m`, `n` provést nejvýše lineární počet aritmetických operací (na hodnotách `m`, `n`).

```
struct natural;
```

Část S.2.b: `treap`

Datová struktura treap kombinuje binární vyhledávací strom a binární haldu. Samozřejmě jedna struktura nemůže splnit obě vlastnosti na stejném klíči:

- vyhledávací strom požaduje, aby hodnoty v levém podstromě byly menší (a hodnoty v pravém větší) než hodnota v kořenu,
- maximová halda vyžaduje, aby hodnoty v obou podstromech byly menší, než hodnota v kořenu.

Proto bude treap uchovávat dvojici hodnot: klíč a prioritu. Strom bude uspořádán tak, aby tvořil vzhledem ke klíči vyhledávací strom a binární haldu vzhledem k prioritě.
Smyslem haldové části struktury je udržet strom přibližně vyvážený. Algoritmus vložení prvku pracuje takto:

1. na základě klíče vložíme uzel na vhodné místo tak, aby nebyla porušena vlastnost vyhledávacího stromu,
2. je-li porušena vlastnost haldy, budeme rotacemi přesouvat nový uzel směrem ke kořenu, a to až do chvíle, než se tím vlastnost haldy obnoví.

Budou-li priority přiděleny náhodně, vložení uzlu do větší hloubky vede i k vyšší pravděpodobnosti, že tím bude vlastnost haldy porušena; navíc rotace, které obnovují vlastnost haldy, zároveň snižují maximální hloubku stromu.
Implementujte typ `treap`, který bude reprezentovat množinu pomocí datové struktury treap a poskytovat tyto operace (`t` je hodnota typu `treap`, `k`, `p` jsou hodnoty typu `int`):

- implicitně sestrojená instance `treap` reprezentuje prázdnou množinu,
- `t.insert( k, p )` vloží klíč `k` s prioritou `p` (není-li uvedena, použije se náhodná); metoda vrací `true` pokud byl prvek skutečně vložen (dosud nebyl přítomen),
- `t.erase( k )` odstraní klíč `k` a vrátí `true` byl-li přítomen,
- `t.contains( k )` vrátí `true` je-li klíč `k` přítomen,
- `t.priority( k )` vrátí prioritu klíče `k` (není-li přítomen, chování není definováno),
- `t.clear()` smaže všechny přítomné klíče,
- `t.size()` vrátí počet uložených klíčů,
- `t.copy( v )`, kde `v` je reference na `std::vector< int >`, v lineárním čase vloží na konec `v` všechny klíče z `t` ve vzestupném pořadí,
- metodu `t.root()`, které výsledkem je ukazatel `p`, pro který:
  - `p->left()` vrátí obdobný ukazatel na levý podstrom,
  - `p->right()` vrátí ukazatel na pravý podstrom,
  - `p->key()` vrátí klíč uložený v uzlu reprezentovaném `p`,
  - `p->priority()` vrátí prioritu uzlu `p`,
  - je-li příslušný strom (podstrom) prázdný, `p` je `nullptr`.
- konečně hodnoty typu `treap` nechť je možné přesouvat, kopírovat a také přiřazovat (a to i přesunem).[20]

Metody `insert`, `erase` a `contains` musí mít složitost lineární k výšce stromu (při vhodné volbě priorit tedy očekávaně logaritmickou k počtu klíčů). Metoda `erase` nemusí nutně zachovat vazbu mezi klíči a prioritami (tzn. může přesunout klíč do jiného uzlu aniž by zároveň přesunula prioritu).

```
struct treap;
```

Část S.2.c: `robots`

V této úloze budete programovat jednoduchou hru, ve které se ve volném prostoru pohybují robotické entity tří typů.
Navrhněte a naprogramujte tyto třídy:

- `game` – reprezentuje třírozměrné hrací pole, ve kterém se pohybují všechny herní objekty:
  - metoda `tick` posune čas o 1/60 sekundy, přepočítá všechny pozice a provede všechny aplikovatelné akce,
  - metoda `run` simuluje hru až do jejího konce.
- `proximity_mine` – reprezentuje minu, která detekuje přítomnost libovolného robota ve svém okolí

Část S.2.d: `network`

Vaším úkolem bude tentokrát naprogramovat simulátor počítačové sítě, s těmito třídami, které reprezentují propojitelné síťové uzly:

- `endpoint` – koncový uzel, má jedno připojení k libovolnému jinému uzlu,
- `bridge` – propojuje 2 nebo více dalších uzlů,
- `router` – podobně jako bridge, ale každé připojení musí být v jiném

[20] Verze s přesunem můžete volitelně vynechat (v takovém případě je označte jako smazané). Budou-li přítomny, budou testovány. Implementace přesunu je podmínkou hodnocení kvality známkou A.

segmentu.

Dále bude existovat třída network, která reprezentuje síťový segment jako celek. Každý uzel patří právě jednomu segmentu. Je-li segment zničen, musí být zničeny (a odpojeny) i všechny jeho uzly.
Třída network bude mít tyto metody pro vytváření uzlů:

- add_endpoint() – vytvoří nový (zatím nepřipojený) koncový uzel, převezme jeho vlastnictví a vrátí vhodný ukazatel na něj,
- add_bridge(p) – podobně pro p-portový bridge,
- add_router(i) – podobně pro směrovač s i rozhraními.

Jsou-li m a n libovolné typy uzlů, musí existovat vhodné metody:

- m->connect(n) – propojí 2 uzly. Metoda je symetrická v tom smyslu, že m->connect(n) a n->connect(m) mají tentýž efekt. Metoda vrátí true v případě, že se propojení podařilo (zejména musí mít oba uzly volný port).
- m->disconnect(n) – podobně, ale uzly rozpojí (vrací true v případě, že uzly byly skutečně propojené).
- m->reachable(n) – ověří, zda může uzel m komunikovat s uzlem n (na libovolné vrstvě, tzn. včetně komunikace skrz routery; jedná se opět o symetrickou vlastnost; vrací hodnotu typu bool).

Konečně třída network bude mít tyto metody pro kontrolu (a případnou opravu) své vnitřní struktury:

- has_loops() – vrátí true existuje-li v síti cyklus,
- fix_loops() – rozpojí uzly tak, aby byl výsledek acyklický, ale pro libovolné uzly, mezi kterými byla před opravou cesta, musí platit, že po opravě budou nadále vzájemně dosažitelné.

Cykly, které prochází více sítěmi (a tedy prohází alespoň dvěma směrovači), neuvažujeme.

```
class endpoint;
class bridge;
class router;
class network;
```

Část S.2.e: query

Část S.2.f: scrap

Část 9: Součtové typy

Kapitola se připravuje.

Část 10: Knihovna algoritmů

Kapitola se připravuje.

Část 11: Řetězce

Kapitola se připravuje.

Část 12: Vstup a výstup

Kapitola se připravuje.

Část S.3: Součtové typy, řetězce

1. a_machine – jednoduchý virtuální stroj s pamětí,
2. b_chess – hrajeme šach,
3. c_real – reálná čísla (dále rozšiřuje s2/a_natural),
4. d_json – reprezentace JSON-u použitím std::variant,
5. e_robots – rozšíření s2/c_robots o načtení vstupu,
6. f_network – načítání vstupu pro simulátor z s2/d_network.

V příkladech a až c využijete zejména znalosti z prvních dvou bloků, vyžadují navíc pouze výčtové typy (enum) z 9. kapitoly. Příklad d vyžaduje znalosti 9. kapitoly a příklady e, f vyžadují znalost 11. kapitoly.

Část S.3.a: machine

In this task, you will implement a simple register machine (i.e. a simple model of a computer). The machine has an arbitrary number of integer registers and byte-addressed memory. Registers are indexed from 1 to INT_MAX. Each instruction takes 2 register numbers (indices) and an 'immediate' value (an integral constant). Each register can hold a single value of type int32_t (i.e. the size of the machine word is 4 bytes). In memory, words are stored MSB-first. The entire memory and all registers start out as 0.
The machine has the following instructions (whenever reg_x is used in the description, it means the register itself (its value or storage location), not its index; the opposite holds in the column reg_2 which always refers to the register index).

opcode	reg_2	description
mov	≥ 1	copy a value from reg_2 to reg_1
	= 0	set reg_1 to immediate
add	≥ 1	store reg_1 + reg_2 in reg_1
	= 0	add immediate to reg_1
mul	≥ 1	store reg_1 * reg_2 in reg_1
jmp	= 0	jump to the address stored in reg_1
	≥ 1	jump to reg_1 if reg_2 is nonzero
load	≥ 1	copy value from addr. reg_2 into reg_1
stor	≥ 1	copy reg_1 to memory address reg_2
halt	= 0	halt the machine with result reg_1
	≥ 1	same, but only if reg_2 is nonzero

Each instruction is stored in memory as 4 words (addresses increase

from left to right). Executing a non-jump instruction increments the program counter by 4 words.

opcode	immediate	reg_1	reg_2

Executing one instruction should take constant time. The memory required for the computation should be at most proportional to the sum of the highest address and the highest register index used by the program.

```
enum class opcode { mov, add, mul, jmp, load, stor, hlt };

struct machine
{
```

Read and write memory, one word at a time.

```
    std::int32_t get( std::int32_t addr ) const;
    void         set( std::int32_t addr, std::int32_t v );
```

Start executing the program, starting from address zero. Return the value of reg_1 given to the hlt instruction which halted the computation.

```
    std::int32_t run();
};
```

Část S.3.b: chess

Část S.3.c: real

Část S.3.d: json

Část S.3.e: robots

Část S.3.f: network

Část T: Technické informace

Tato kapitola obsahuje informace o technické realizaci předmětu, a to zejména:

- jak se pracuje s kostrami úloh,
- jak sdílet obrazovku (terminál) ve cvičení,
- jak se odevzdávají úkoly,
- kde najdete výsledky testů a jak je přečtete,
- kde najdete hodnocení kvality kódu (učitelské recenze),
- jak získáte kód pro vzájemné recenze.

Část T.1: Informační systém

TBD.

Část T.2: Studentský server aisa

Na server aisa se přihlásíte programem ssh, který je k dispozici v prakticky každém moderním operačním systému (v OS Windows skrze WSL[21] – Windows Subsystem for Linux). Konkrétní příkaz (za xlogin doplňte ten svůj):

```
$ ssh xlogin@aisa.fi.muni.cz
```

Program se zeptá na heslo: použijte to fakultní (to stejné, které používáte k přihlášení na ostatní fakultní počítače, nebo např. ve fadmin-u nebo fakultním gitlab-u).

T.2.1 Pracovní stanice Veškeré instrukce, které zde uvádíme pro použití na stroji aisa platí beze změn také na libovolné školní UNIX-ové pracovní stanici (tzn. z fakultních počítačů není potřeba se hlásit na stroj aisa, navíc mají sdílený domovský adresář, takže svoje soubory z tohoto serveru přímo vidíte, jako by byly uloženy na pracovní stanici).

T.2.2 Stažení koster Aktuální zdrojový balík stáhnete příkazem:

```
$ pb161 update
```

Stažené soubory pak naleznete ve složce ~/pb161. Je bezpečné tento příkaz použít i v případě, že ve své kopii již máte rozpracovaná řešení – systém je při aktualizaci nepřepisuje. Došlo-li ke změně kostry u příkladu, který máte lokálně modifikovaný, aktualizovanou kostru naleznete v souboru s dodatečnou příponou .pristine, např. 01/e2_concat.cpp.pristine. V takovém případě si můžete obě verze srovnat příkazem diff:

```
$ diff -u e2_concat.cpp e2_concat.cpp.pristine
```

Případné relevantní změny si pak již lehce přenesete do svého řešení. Krom samotného zdrojového balíku Vám příkaz pb161 update stáhne i veškeré recenze (jak od učitelů, tak od spolužáků). To, že máte k dispozici nové recenze, uvidíte ve výpisu. Recenze najdete ve složce ~/pb161/reviews.

T.2.3 Odevzdání řešení Odevzdat vypracované (nebo i rozpracované) řešení můžete ze složky s relevantními soubory takto:

```
$ cd ~/pb161/01
$ pb161 submit
```

Přidáte-li přepínač --wait, příkaz vyčká na vyhodnocení testů fáze „syntax“ a jakmile je výsledek k dispozici, vypíše obsah příslušného poznámkového bloku. Chcete-li si ověřit co a kdy jste odevzdali, můžete použít příkaz

```
$ pb161 status
```

nebo se podívat do informačního systému (blíže popsáno v sekci T.2).

T.2.4 Sdílení terminálu Řešíte-li příklad typu r ve cvičení, bude se Vám pravděpodobně hodit režim sdílení terminálu s cvičícím (který tak bude moct promítat Váš zdrojový kód na plátno, případně do něj jednoduše zasáhnout).
Protože se sdílí pouze terminál, budete se muset spokojit s negrafickým textovým editorem (doporučujeme použít micro, případně vim umíte-li ho ovládat). Spojení navážete příkazem:

```
$ pb161 beamer
```

Protože příkaz vytvoří nové sezení, nezapomeňte se přesunout do správné složky příkazem cd ~/pb161/NN.

T.2.5 Recenze Příkaz pb161 update krom zdrojového balíku stahuje také:

1. zdrojové kódy, které máte možnost recenzovat, a to do složky ~/pb161/to_review,
2. recenze, které jste na svůj kód obdrželi (jak od spolužáků, tak od vyučujících), a to do stávajících složek zdrojového balíku (tzn. recenze

[21] Jako alternativu, nechcete-li z nějakého důvodu WSL instalovat, lze použít program putty.

na příklady z první kapitoly se Vám objeví ve složce ~/pb161/01 – že se jedná o recenzi poznáte podle jména souboru, který bude začínat uživatelským jménem autora recenze, např. xrockai.00123.p1_nhamming.cpp).

Chcete-li vypracované recenze odeslat:

1. přesuňte se do složky ~/pb161/to_review a
2. zadejte příkaz pb161 submit, případně doplněný o seznam souborů, které hodláte odeslat (jinak se odešlou všechny, které obsahují jakýkoliv přidaný komentář).

Část T.3: Kostry úloh

Pracujete-li na studentském serveru aisa, můžete pro překlad jednotlivých příkladů použít přiložený soubor makefile, a to zadáním příkazu

```
$ make příklad
```

kde příklad je název souboru bez přípony (např. tedy make e1_factorial). Tento příkaz postupně:

1. přeloží Vaše řešení překladačem g++,
2. provede kontrolu nástrojem clang-tidy,
3. spustí přiložené testy,
4. spustí kontrolu nástrojem valgrind.

Selže-li některý krok, další už se provádět nebudou. Povede-li se překlad v prvním kroku, v pracovním adresáři naleznete spustitelný soubor s názvem příklad, se kterým můžete dále pracovat (např. ho ladit/krokovat nástrojem gdb).
Existující přeložené soubory můžete smazat příkazem make clean (vynutíte tak jejich opětovný překlad a spuštění všech kontrol).

T.3.1 Textový editor Na stroji aisa je k dispozici jednoduchý editor micro, který má podobné ovládání jako klasické textové editory, které pracují v grafickém režimu, a který má slušnou podporu pro práci se zdrojovým kódem. Doporučujeme zejména méně pokročilým. Další možností jsou samozřejmě pokročilé editory vim a emacs.
Mimo lokálně dostupné editory si můžete ve svém oblíbeném editoru, který máte nainstalovaný u sebe, nastavit režim vzdálené editace (použitím protokolu ssh). Minimálně ve VS Code je takový režim k dispozici a je uspokojivě funkční.

T.3.2 Vlastní prostředí Každý příklad je zcela obsažen v jednom standardním zdrojovém souboru, proto je jejich překlad velmi jednoduchý. Pravděpodobně každé IDE zvládne s příklady bez problémů pracovat (spouštět, ladit, atp.) – dejte si pouze pozor na to, že potřebujete překladač s podporou standardu C++20 (nejstarší verze, které lze rozumně použít, jsou gcc verze 10 a clang verze 13). Úroveň podpory standardu C++20 v našeptávačích, zabudovaných kontrolách atp. je různá. YMMV.
Pracujete-li na POSIX-ovém systému (včetně WSL), můžete také použít dodaný soubor makefile, pouze si v nadřazené složce (tzn. vedle složek 01, 02, atd.) vytvořte soubor local.mk, ve kterém nastavíte, jak se na Vašem systému spouští potřebné příkazy. Na typickém systému bude fungovat:

```
CXX = g++
TIDY = clang-tidy
VALGRIND = valgrind
```

V opačném případě si budete muset potřebné kontroly spouštět ručně.