Hugsaðu eins og tölvunarfræðingur

Allen B. Downey
Þýtt og staðfært af Hrafni Loftssyni

2014-2015

Hugsaður eins og tölvunarfræðingur

C++ útgáfa

The original form of this book is LaTeX source code. Compiling this code has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

This book was typeset by the author using latex, dvips and ps2pdf, among other free, open-source programs. The LaTeX source for this book is available from `http://greenteapress.com/thinkcpp` and from the SVN repository `http://code.google.com/p/thinkcpp`.

Efnisyfirlit

Kafli 1

Forritun

Megin markmiðið með þessari bók er að kenna þér, lesandi góður, að hugsa eins og tölvunarfræðingur. Tölvunarfræðileg hugsun sameinar marga af bestu eiginleikum úr stærðfræði, verkfræði og náttúruvísindum. Tölvunarfræðingar nota, eins og stærðfræðingar, formleg mál til að lýsa hugmyndum (sérstaklega útreikningum/vinnslu (e. computation)). Eins og verkfræðingar, þá hanna þeir hluti, setja þá saman til að mynda kerfi og meta kosti og galla mismunandi valkosta. Eins og vísindamenn, þá fylgjast þeir með hegðun flókinna kerfa, setja fram tilgátur og prófa þær.

Einn mikilvægasti eiginleiki tölvunarfræðinga er hæfni til að **leysa vandamál/verkefni**. Hér er átt við að geta skilgreint verkefni, hugsað á skapandi hátt um lausnir og tjáð lausnaraðferðir á skýran og nákvæman máta. Það vill einmitt svo vel til að það að læra að forrita er mjög góð leið til að þróa með sér hæfni til að leysa vandamál.

Þessi bók er hugsuð sem kennslubók í fyrsta forritunarnámskeiðinu (Forritun) við tölvunarfræðideild Háskólans í Reykjavík (HR). Annað markmið bókarinnar er því að kenna þér grundvallaratriði í forritun – grunn sem er nauðsynlegur fyrir önnur námskeið í tölvunarfræðináminu, eins og Gagnaskipan, Reiknirit og Forritunarmál.

1.1 Hvað er forritunarmál?

Í þessari bók er forritun kennd með því að nota forritunarmálið C++ sem er mjög útbreitt forritunarmál. C++ er **æðra forritunarmál** (e. high-level language) en önnur sambærileg forritunarmál sem þú hefur kannski heyrt um eru t.d. Java, C# og Python. Einkenni æðri forritunarmála er að smáatriði varðandi innri virkni tölvunnar hafa verið dregin út (e. abstracted away) sem merkir að forritarinn getur hugsað um verkefnið á "æðra plani" í stað þess að þurfa að hugsa um eiginleika tiltekins vélbúnaðar.

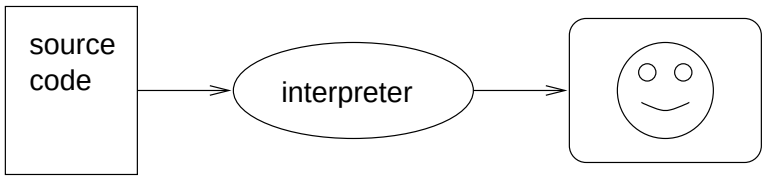
Andstæðan við æðri forritunarmál eru **lágtæknimál** (e. low-level languages), sem stundum eru einnig kölluð vélarmál (e. machine languages) eða smala-

mál (e. assembly languages). Í stuttu máli sagt þá geta tölvur aðeins keyrt forrit sem skrifuð eru í vélarmáli. Þess vegna þarf að þýða (e. compile) forrit sem skrifað er í æðra forritunarmáli yfir á vélarmál tiltekinnar tölvu áður en forritið er keyrt. Ef hægt er að tala um einhvern ókost við æðri forritunarmál þá er hann sá að umrædd þýðing getur tekið nokkurn tíma.

Á hinn bóginn eru kostirnir miklir. Í fyrsta lagi er miklu auðveldara að forrita í æðra forritunarmáli heldur en í lágtæknimáli. Með "auðveldara" er hér átt við að forritunin tekur minni tíma, forritið er styttra og læsilegra og mun líklegra til að vera rétt. Í öðru lagi eru forrit sem skrifuð eru í æðri forritunarmál oftast **færanleg** (e. portable), í þeim skilningi að hægt er að keyra þau á mismunandi vélum með engum eða litlum breytingum. Forrit sem skrifuð eru í lágtæknimáli er aftur á móti eingöngu hægt að keyra á einni tiltekinni vél og þarfnast endurskriftar til keyrslu á annarri vélartegund.

Vegna þessara kosta eru langflest forrit í dag skrifuð í æðri forritunarmálum. Lágtæknimál eru notuð við sérstakar aðstæður, t.d. þegar forritað er á móti tilteknum örgjörvum.

Það eru tvær leiðir til að umbreyta forriti, sem skrifað er í æðra forritunarmáli, þannig að hægt sé að keyra það á tiltekinni vél: **túlkun** (e. interpretation) eða **þýðing** (e. compilation). *Túlkur* (e. interpreter) er forrit sem les annað forrit S, sem skrifað er í æðra forritunarmáli, og líkir eftir sérhverri skipun í S. Túlkur les forrit línu fyrir línu, varpar sérhverri skipun í S yfir í skipun/skipanir í því máli sem túlkurinn er skrifaður í, og framkvæmir síðan þær skipanir. Úttakið úr túlkinum er úttakið úr S.



The interpreter reads the source code...

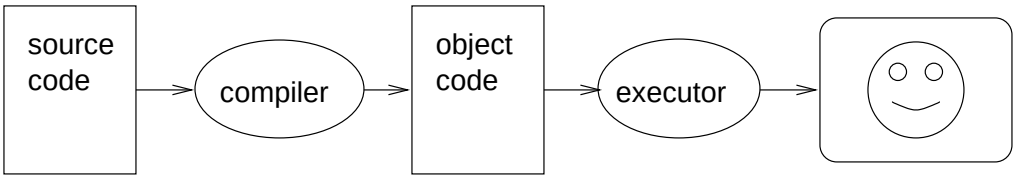
... and the result appears on the screen.

Þýðandi er forrit sem les annað forrit (sem oftast er skrifað í æðra forritunarmáli) og þýðir það yfir á annað mál (oftast lágtæknimál). Hið þýdda forrit er þá tilbúið til keyrslu síðar meir. Inntakið í þýðandann er kallað **frumkóði** (e. source code) og úttakið er **markkóði** (e. target code).

Gefum okkur t.d. að þú skrifir forrit í C++. Þú gætir notað textaritil (e. text editor) eða sérstakt C++ þróunarumhverfi (e. development environment) til að skrifa forritið og vistað það í skrá með nafninu `program.cpp`. Hér er `program` eitthvað nafn sem þú gefur forritinu og viðskeytið `.cpp` gefur til kynna að umrædd skrá innihaldi C++ frumkóða.

Þegar forritið hefur verið skrifað þá gætir þú keyrt þýðanda á það. Þýðandinn myndi lesa frumkóðann, þýða hann og búa til nýja skrá með nafninu `program.o` fyrir markkóðann, og/eða `program.exe` fyrir hina endanlegu keyrsluskrá (e. ex-

ecutable file).



The compiler reads the source code...

... and generates object code.

You execute the program (one way or another)...

... and the result appears on the screen.

Þegar keyrsluskráin er síðan keyrð þá sér undirliggjandi stýrikerfi um að hlaða forritinu upp í minni (afrita það af diski yfir í minni) og lætur síðan tölvuna byrja að keyra forritið.

Þetta ferli virðist vera flókið en sem betur fer þá eru einstök skref í ferlinu framkvæmd á sjálfvirkan máta í flestum þróunarumhverfum. Yfirleitt þarft þú aðeins að skrifa forritið og ýta síðan á hnapp (eða skrifa skipun) sem setur þýðingu og keyrslu af stað. Á hinn bóginn er mikilvægt fyrir þig að vita hvaða einstöku hlutar eiga sér stað í ferlinu þannig að þú getir lagfært þá ef eitthvað bjátar á.

1.2 Hvað er forrit?

Forrit er röð skipana sem segja til um hvernig framkvæma á tiltekna vinnslu. Vinnslan gæti t.d. verið stærðfræðileg, eins og að leysa jöfnuhneppi eða að finna rætur margliðu. Hún gæti jafnframt líka verið táknræn (e. symbolic), t.d. að leita að og skipta út texta í skrá eða að þýða forrit!

Skipanir eða setningar líta mismunandi út í mismunandi forritunarmálum en nokkrar grunnaðgerðir eru sameiginlegar flestum málum:

inntak (e. input): Sækja gögn af lyklaborði, úr skrá eða af einhverju öðru tæki.

úttak (e. output): Sýna gögn á skjánum, skrifa gögn í skrá eða í annað tæki.

stærðfræðilegar aðgerðir: Framkvæma stærðfræðilegar grunnaðgerðir eins og samlagningu og margföldun.

prófun (e. test): Athuga hvort tiltekið skilyrði er satt/ósatt og framkvæma síðan viðeigandi röð setninga.

endurtekning (e. repitition): Endurtaka tilteknar aðgerðir, yfirleitt með einhverjum frávikum á milli endurtekninga.

Það kann að hljóma ótrúlega en þetta er í raun allt og sumt! Sérhvert forrit, sem þú hefur notað/keyrt, samanstendur af aðgerðum sem þessum. Forritun

er því í raun það ferli að brjóta stórt, flókið verkefni upp í smærri og smærri einingar þangað til að sérhver eining er orðin það einföld að hægt er að leysa hana með einhverjum af þessum aðgerðum.

1.3 Hvað er kembing?

Forritun er flókið ferli sem framkvæmt er mönnum (en ekki vélum) og þess vegna eiga forritunarvillur oft sér stað. Af sögulegum ástæðum eru forritunarvillur oft kallaðar **böggar** (e. bugs) og ferlið við að finna og leiðrétta þær er kallað **kembing** eða **aflúsun** (e. debugging).

Mismunandi forritunarvillur geta komið upp í forriti og það er mikilvægt að gera greinarmun á þeim til að geta fundið þær og leiðrétt hratt.

1.3.1 Þýðingarvillur

Þýðandinn getur aðeins þýtt (búið til markkóða) fyrir forrit sem er setningafræðilega rétt. Að öðrum kosti mistekst þýðingin og þá er ekki hægt að keyra forritið. **Málskipan** (e. syntax) vísar til hvernig forrit er samansett og hvaða reglur gilda um samsetninguna.

Í íslensku þarf sérhver setning t.d. að byrja á stórum staf og enda á punkti. þessi setning inniheldur málskipunarvillu. Einnig þessi hér

Flestir lesendur eiga ekki í erfiðleikum með að skilja texta sem inniheldur nokkrar málskipunarvillur. Við getum t.d. lesið ljóð eftir e e cummings án þess að spýja út villuskilaboðum!

Þýðendur líta aftur á móti málskipunarvillur alvarlegum augum. Ef þýðandi finnur t.d. aðeins eina villu í forritinu þínu þá mun hann skrifa út villuskilaboð og hætta án þess að búa til keyrsluskrá. Í því tilviki munt þú ekki geta keyrt forritið þitt.

Til að gera illt verra þá eru margar málskipunarreglur í C++ og villuskilaboðin sem koma frá þýðandanum eru stundum ekki mjög hjálpleg. Á fyrstu vikum forritunarferils þíns munt þú líklega eyða verulegum tíma í að finna **þýðingarvillur** (e. compile-time errors). Með reynslunni munt þú hins vegar gera færri villur og finna þær mun hraðar.

1.3.2 Keyrsluvillur

Önnur tegund af villum eru svokallaðar **keyrsluvillur** (e. run-time errors), þ.e. villur sem koma ekki upp fyrr en við keyrslu forrits.

Í þeim einföldu forritum sem við munum skrifa á næstu vikum verða keyrsluvillur sjaldgæfar þannig að líklega mun líða dálítill tími þangað til þú sérð þess konar villur.

1.3.3 Rökvillur og merking

Rökvillur (e. logical errors) eða **merkingarvillur** (e. semantic errors) eru þriðja tegundin af villum. Ef forritið þitt inniheldur rökvillur þá mun það samt

sem áður þýðast og keyra án þess að villuskilboð prentist út. Forritið mun hins vegar ekki gera það sem til stóð en þó það sem þú sagðir því að gera!

Vandamálið er sem sagt það að forritið sem þú skrifaðir er ekki forritið sem þú ætladir þér að skrifa. Merking forritsins er því röng. Það getur verið erfitt að finna rökvillur því það krefst þess að þú vinnir "aftur á bak" með því að skoða úttakið og reyna að finna út hvað geti verið að.

1.3.4 Tilraunakembing

Ein mikilvægasta færnin sem þú ættir að öðlast við að lesa og vinna með þessa bók er kembing. Þó kembing geti stundum verið pirrandi, þá getur hún jafnframt verið sá hluti forritunar sem er hvað mest krefjandi og áhugaverður.

Að sumu leyti er kembing eins og starf rannsóknarlögreglumanns. Vísbendingar eru gefnar og finna þarf út hvaða ferli og atburðir leiddu til niðurstöðunnar.

Kembing er einnig eins og tilraunavísindi. Þegar þig grunar hvað fór úrskeiðis þá breytir þú forritinu og keyrir það á ný. Ef tilgáta þín reynist rétt þá skilar forritið þitt réttri útkomu. Ef tilgátan er aftur á móti röng þá þarftu að finna nýja. Eða eins og Sherlock Holmes benti á: "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (úr *The Sign of Four* eftir A. Conan Doyle).

Sumir líta svo á að forritun og kembing sé einn og sami hluturinn, þ.e.a.s. að forritun sé það ferli að kemba forrit þangað til að það gerir það sem til stóð. Hugmyndin er sú að þú ættir alltaf að byrja með forrit sem virkar, þ.e. forrit sem gerir *eitthvað*, framkvæma síðan litlar breytingar á því, með kembingu ef þess er þörf, þannig að þú sért ávallt með forrit sem virkar.

Sem dæmi um þetta má nefna stýrikerfið Linux sem inniheldur þúsundir forritslína en byrjaði sem lítið forrit sem Linus Torvalds notaði til að kanna Intel 80386 örgjörvann. "One of Linus's earlier projects was a program that would switch between printing AAAA and BBBB. This later evolved to Linux" (úr *The Linux Users' Guide* Beta Version 1).

Í síðari köflum þessarar bókar mun ég koma með fleiri tillögur um kembingu og aðrar forritunaraðferðir.

1.4 Formleg og náttúruleg mál

Náttúruleg mál (e. natural languages) eru mál sem fólk talar, eins og enska, spænska, franska og íslenska. Þessi mál hafa ekki verið hönnuð af mönnum (þó svo að menn reyni oft að búa þeim einhverjar reglur) heldur hafa þau þróast á náttúrulegan hátt.

Formleg mál (e. formal languages) eru mál sem eru hönnuð af mönnum fyrir tiltekna notkun. Stærðfræðingar nota t.d. sérstakt formlegt mál sem hentar vel til að lýsa sambandi á milli talna og tákna. Efnafræðingar nota formlegt mál sem lýsir efnafræðilegri byggingu sameinda. Og síðast en ekki síst:

> **Forritunarmál eru formleg mál sem hafa verið hönnuð til að tjá vinnslu.**

Flest formleg mál hafa stífar málfræðireglur. $3 + 3 = 6$ er t.d. setningafræðilega rétt stærðfræðileg setning en ekki $3 = +6\$$. Sem annað dæmi má nefna að H_2O er rétt nafn á frumefni en ekki ${}_2Zz$.

Málfræðireglurnar eru af tvennum toga. Í fyrsta lagi eru reglur um hvað eru leyfilegir tókar (e. tokens) í viðkomandi máli. Tókar eru grunneiningar í öllum málum, t.d. orð, tölur og nöfn á frumefnum.

Vandamálið við `3=+6$` er að `$` er ekki leyfilegur tóki í stærðfræði. Á sama hátt er ${}_2Zz$ ekki leyfilegt því það er ekkert frumefni með skammstöfunina Zz.

Hin tegundin af málfræðireglum, málskipunarreglur, hefur að gera með hvernig tókar eru settir saman til að mynda setningar. T.d. er setningin `3=+6$` er ekki rétt sett saman því á eftir gildisveitingarvirkja (e. assignment operator) getur ekki komið virkinn plús.

Þegar þú lest setningu í þínu móðurmáli eða setningu í formlegu máli þá þarftu að finna út hvernig setningin er samansett af einstökum tókum (í náttúrulegu máli þá gerum við þetta reyndar ómeðvitað). Þetta ferli er kallað **þáttun** (e. parsing).

Þegar þú heyrir t.d. setninguna "The other shoe fell" þá áttar þú þig á því (með þáttun) að "the other shoe" er frumlagið (e. subject) og "fell" er sögnin (e. verb). Þegar þú hefur þáttað setninguna þá getur þú fundið út hver merking hennar er. Að því gefnu að þú vitir hvað "shoe" er og þú vitir hver merkingin "to fall" er þá skilur þú heildmerkingu setningarinnar.

Þrátt fyrir að formleg mál og náttúruleg mál eigi sér margar hliðstæður, t.d. í tókum, málskipan og merkingu, þá er jafnframt margt ólíkt með þessum tegundum mála:

margræðni (e. ambiguity): Margræðni er mjög algeng í náttúrlegum málum en við leysum margræðni með því að nýta okkur samhengið og aðrar upplýsingar. Formleg mál eru aftur á móti hönnuð til að vera laus við margræðni, þ.e. sérhver setning hefur nákvæmlega eina merkingu, án tillits til samhengis.

ofauki (e. redundancy): Til að koma í veg fyrir misskilning vegna margræðni þá er orðum oft ofaukið í setningum í náttúrulegum málum. Í formlegum málum eru hins vegar litlar sem engar málalengingar og setningar í þeim eru því gagnyrtar.

bókstafsmerking (e. literalness): Orðatiltæki og myndlíkingar eru algengar í náttúrulegum málum. Ef ég segi t.d. "Sjaldan er ein báran stök", þá er ég í rauninni ekki að tala um báru. Merking setningar í formlegu máli er hins vegar bókstafsleg, þ.e. nákvæmlega sú sem lesa má úr setningunni.

Þeir sem alast upp við náttúruleg mál (þ.e. við öll) eiga oft erfitt með að venjast formlegum málum. Að sumu leyti er munurinn á milli náttúrulegs og formlegs máls eins og munurinn á milli bundins og óbundins máls, en þó meiri:

Bundið mál: Orð eru notuð vegna þeirra hljóða sem þau mynda og einnig vegna merkingar þeirra. Í heild sinni hefur ljóðið einhver áhrif og veldur

tilfinningalegum viðbrögðum. Margræðni er ekki bara algeng heldur oft meðvituð.

Óbundið mál: Raunveruleg merking orða er mikilvæg og uppbygging setninga hefur mikið með merkingu að gera. Óbundið mál er yfirleitt auðveldara að greina en bundið mál, en er samt sem áður oft margrætt.

Forrit: Merking forrits er ekki margræð og hún er bókstafleg. Hægt er að skilja merkinguna með því að greina tóka og málskipan.

Hér fylgja nokkrar ráðleggingar við lestur forrits (eða annarra formlegra mála). Í fyrsta lagi mundu að formleg mál eru mun "samþjappaðri" en náttúrleg mál og því tekur lengri tíma að lesa þau. Í öðru lagi er uppbygging forrita mjög mikilvæg og því er yfirleitt ekki hentugt að lesa þau frá byrjun til enda, vinstri til hægri. Í staðinn skaltu læra að þátta forritið í huganum, bera kennsl á tókana og greina uppbygginguna. Mundu að lokum að smáatriðin skipta máli. Smáatriði, eins og stafsetningarvillur eða greinamerkjavillur sem hægt er að komast upp með í náttúrulegum málum, geta skipt miklu máli í formlegum málum.

1.5 Fyrsta forritið

Sú venja hefur skapast að fyrsta forritið sem fólk skrifar í nýju forritunarmáli er kallað "Hello, World" vegna þess að eina sem það gerir er að prenta út orðin "Hello, World". Svona lítur forritið út í forritunarmálinu C++:

```
#include <iostream>
using namespace std;

// main: generate some simple output

int main ()
{
  cout << "Hello, world." << endl;
  return 0;
}
```

Sumir dæma gæði forritunarmáls með því að skoða hversu einfalt er að skrifa "Hello, World" forrit í málinu. Ef við notum þennan mælikvarða þá kemur C++ ágætlega út. Samt sem áður inniheldur þetta einfalda forrit ýmsa eiginleika sem er erfitt að skýra út fyrir byrjendum í forritun. Til að byrja með munum við ekki skýra suma þeirra, eins og fyrstu tvær línurnar í þessu forriti.

Þriðja línan byrjar á `//` sem gefur til kynna að línan er **athugasemd** (e. comment). Athugasemd er texti (oftast skrifaður í náttúrulegu máli) sem hægt er að setja inn hvar sem er í forriti í þeim tilgangi að skýra út hvað forritið, eða hluti þess, gerir. Þegar þýðandinn rekst á `//` þá hunsar hann allan textann frá þeim stað og að enda línunnar.

Í fjórðu línunni getur þú, sem stendur, hunsað orðið (tókann) `int` en taktu eftir orðinu (tókanum) `main` sem er sérstakt nafn sem gefur til kynna staðinn þar sem keyrsla forritsins hefst. Þegar forritið hefur keyrslu þá mun það byrja með því að keyra fyrstu setninguna í `main`, heldur síðan áfram í réttri röð og hættir eftir framkvæmd síðustu setningarinnar.

Það eru engin takmörk á því hversu margar setningar `main` getur innihaldið en í dæminu að ofan er aðeins um eina setningu að ræða. Sú setning er úttakssetning, þ.e. setning sem skrifar tiltekin skilaboð út á skjáinn.

`cout` er sérstakur hlutur (e. object), innbyggður í C++, sem gerir þér kleift að senda tiltekið úttak á skjáinn (cout er úttaksstraumur). Táknið `«` er **virki** (e. operator), sem beitt er á `cout` og streng, og veldur því að strengurinn er skrifaður út.

`endl` er sérstakt tákn sem stendur fyrir enda á línu. Þegar þú sendir `endl` á `cout` þá færist bendillinn (e. cursor) í næstu línu á skjánum. Eftir það mun texti sem skrifaður er út birtast í þessari næstu línu.

Eins og gildir um allar setningar þá endar úttakssetningin með semikommu (`;`).

Það er nokkur önnur atriði sem vert er að hafa í huga varðandi málskipan þessa fyrsta forrits. Í fyrsta lagi að C++ notar slaufusviga (e. curly-braces) ({ og }) til að gefa til kynna að hlutir eigi saman. Í þessu tilviki er úttakssetningin inni í slaufusvigum sem þýðir að hún er hluti af skilgreiningunni á `main`. Einnig er vert að benda á að setninginn er inndregin (e. indented) sem sýnir á skýran hátt hvaða setningar eru hluti af skilgreiningunni.

Á þessum tímapunkti mæli ég með því að þú setjist fyrir framan tölvuna þína og þýðir og keyrir þetta forrit. Hvernig það er gert er háð því forritunarumhverfi[1] sem þú notar en ég mun héðan í frá í þessari bók gera ráð fyrir að þú kunnir að gera það.

Eins og ég nefndi áður þá tekur C++ þýðandinn hart á málskipunarvillum. Ef þú gerir einhverjar innsláttarvillur í forritinu þá mun forritið líklega ekki þýðast. T.d. ef þú slærð inn `oistream` í stað `iostream` þá færðu villuskilaboð eins og þessi:

```
hello.cpp:1: oistream.h: No such file or directory
```

Það eru verulegar upplýsingar fólgnar í þessum villuskilaboðum en þær eru á samþjöppuðu formi og það er ekki einfalt að túlka skilaboðin. Notandavænni þýðandi myndi sjálfsagt segja eitthvað á þessa leið:

> "Í línu 1 í frumkóðanum með nafninu hello.cpp reyndir þú að taka inn hausaskrá með nafninu oistream. Ég fann enga skrá með því nafni en fann aftur á móti skrá með nafninu iostream. Getur verið að þú hafir ætlað þér að taka þá skrá inn?"

Því miður eru fæstir þýðendur svona vingjarnlegir! Þýðandi er í raun ekki mjög "greindur" og í mörgum tilvikum eru villuskilaboðin sem þú færð í raun

[1] Í fyrsta forritunarnámskeiðinu í tölvunarfræði í HR er forritunarumhverfið Code::Blocks (`http://www.codeblocks.org/`) notað.

bara ábending um hvað geti verið að. Það mun taka tíma fyrir þig að læra að túlka villuskilaboðin á réttan hátt.

Þrátt fyrir þetta eru þýðendur mikilvægt tól til að læra málskipunarreglur tiltekins forritunarmáls. Gott er að byrja með forrit sem virkar (eins og hello.cpp), breyta því á ýmsan hátt og sjá hvað gerist. Ef þú færð villuskilaboð reyndu þá að muna hver þau eru og hvað olli þeim þannig að ef þú sérð þau aftur síðar meir þá veistu hvað þarf að gera til að lagfæra villuna.

1.6 Orðalisti

lausn verkefnis/vandamáls (e. problem-solving): Ferli sem felur í sér að skilgreina vandamál, finna lausn á því og setja lausnaraðferðina fram á skýran og nákvæman máta.

æðra forritunarmál (e. high-level language): Forritunarmál eins og C++ sem er hannað til að fólk geti auðveldlega lesið það og skrifað.

lágtæknimál (e. low-level language): Forritunarmál sem er hannað til að tölvur geti á auðveldan hátt keyrt forrit í málinu. Einnig kallað vélarmál (e. machine language) eða smalamál (e. assembly language).

færanleiki (e. portability): Einkenni forrits sem gerir það að verkum að hægt er að keyra það á fleiri en einni tegund véla.

formlegt mál (e. formal language): Öll mál sem fólk hefur búið til í sérstökum tilgangi, eins og t.d. mál sem standa fyrir stærðfræðilegar hugmyndir eða tölvuforrit, eru formleg mál. Öll forritunarmál eru formleg mál.

náttúrlegt mál (e. natural language): Öll mál sem töluð eru af mönnum og hafa þróast á náttúrulegan hátt.

túlka (e. interpret): Það að keyra forrit, sem skrifað er í tilteknu máli, með því að túlka það línu fyrir línu.

þýða (e. compile): Það að þýða forrit yfir í lágtæknimál í þeim tilgangi að geta keyrt það síðar, aftur og aftur.

frumkóði (e. source code): Forrit, sem er (yfirleitt) skrifað í æðra forritunarmáli, áður en það er þýtt.

markkóði (e. object code): Úttak þýðandans, þ.e. hið þýdda forrit.

keyrsluskrá (e. executable): Markkóði sem hægt er að keyra beint á tölvu.

forritunarvilla (e. bug): Villa í forriti.

málskipan (e. syntax): Uppbygging forrits.

merking (e. semantics): Merking forrits.

þátta (e. parse): Að greina forrit í einstakar setningafræðilegar einingar.

málskipunarvilla (e. syntax error): Villa í forriti sem veldur því að ekki er hægt að þátta forritið (og þar með ekki hægt að þýða það).

keyrsluvilla (e. run-time error): Villa í forrit sem kemur upp við keyrslu þess.

rökvilla (e. logical error): Villa í forriti sem veldur því að það gerir eitthvað annað en það sem forritarinn ætlaðist til.

kembing (e. debugging): Ferlið við að finna og fjarlægja allar tegundir af forritunarvillum.

Kafli 2

Breytur og tög

2.1 Meira úttak

Eins og bent var á í síðasta kafla þá getur þú haft eins margar setningar og þú vilt í `main`. T.d., til að skrifa út fleiri en eina línu:

```
#include <iostream>
using namespace std;
// main: generate some simple output

int main ()
{
  cout << "Hello, world." << endl;      // output one line
  cout << "How are you?" << endl;       // output another
  return 0;
}
```

Eins og hér sést þá er löglegt að setja inn athugasemdir í enda línu sem og í sér línu.

Þeir hlutar setninganna að ofan sem birtir eru innan gæsalappa eru kallaðir **strengir** (e. strings), vegna þess að þeir samanstanda af röð (streng) af bókstöfum. Reyndar vill svo til að strengir geta innhaldið hvaða samsetningu sem er af bókstöfum, tölum, greinarmerkjum og öðrum sérstökum táknum.

Oft er gagnlegt að sýna úttak úr mörgum úttakssetningum í einni og sömu línunni. Þetta er hægt að gera með því að sleppa fyrsta `endl`:

```
int main ()
{
  cout << "Goodbye, ";
  cout << "cruel world!" << endl;
  return 0
}
```

Í þessu tilviki birtist úttakið í einni línu sem `Goodbye, cruel world!`. Athugið að í forritinu er eitt bil (e. space) á milli stafarununnar "Goodbye," og seinni gæsalapparinnar. Þetta bil birtist í úttaki forritsins og bilið í forritinu sjálfu hefur því áhrif á virkni þess.

Bil sem koma fyrir utan gæsalappa í forriti hafa aftur á móti ekki áhrif á virkni forritsins. Ég gæti t.d. hafa skrifað:

```
int main ()
{
  cout<<"Goodbye, ";
  cout<<"cruel world!"<<endl;
  return 0;
}
```

Þetta forrit þýðist og keyrist á sama hátt og upphaflega forritið. Skil (e. break) í enda línu (e. newline) hafa heldur ekki áhrif á virkni forritsins. Ég gæti því einnig hafa skrifað:

```
int main(){cout<<"Goodbye, ";cout<<"cruel world!"<<endl;return 0;}
```

Þetta myndi líka virka en eins og þú hefur væntanlega tekið eftir þá hafa þessar breytingar gert það að verkum að forrritið hefur orðið ólæsilegra. Skil á milli lína og bil eru einmitt gagnleg fyrir okkur því lestur forritsins verður auðveldari og gerir okkur jafnframt auðveldara um vik við að finna málskipunarvillur.

2.2 Gildi

Gildi (e. value) (t.d. bókstafur eða tala) er eitt af þeim meginatriðum sem forrit vinnur með. Einu gildin sem við höfum unnið með hingað til eru strengjagildi, eins og `"Hello, world."`. Þú (og þýðandinn) þekkir strengjagildi frá öðrum gildum vegna þess að strengir eru runa af stöfum innan gæsalappa.

Önnur dæmi um gildi eru heiltölur (e. integers) og stafir (e. characters). Dæmi um heiltölu er `1` eða `17`. Þú getur skrifað út heiltölugildi á sama hátt og þú skrifar út gildi á strengjum:

```
  cout << 17 << endl;
```

Stafagildi er bókstafur, tölustafur eða greinarmerki innan einfaldra gæsalappa, t.d. `'a'` eða `'5'`. Þú getur skrifað út stafagildi á sama hátt:

```
  cout << '}' << endl;
```

Í þessu dæmi er slaufusvigi skrifaður út einn og sér í línu.

Það er einfalt á ruglast á hinum ýmsu tegundum af gildum eins og `"5"`, `'5'` og `5`, en ef þú skoðar greinarmerkin þá ætti að vera ljóst að fyrsta gildið er strengur, annað gildi er stafur og þriðja gildið er heiltala. Ástæðan fyrir mikilvægi þessarar aðgreiningar mun bráðum verða ljós.

2.3 Breytur

Einn öflugasti eiginleiki forritunarmáls er að sá að gera forriturum kleift að vinna með **breytur** (e. variables). Breyta er nafn sem stendur fyrir minnissvæði sem geymir gildi.

Ýmsar mismunandi tegundir af breytum eru til (á sama hátt og mismunandi tegundir af gildum). Þegar þú býrð til nýja breytu þá þarftu (í C++) að skilgreina af hvaða **tagi** (e. type) breytan er. Í C++ er stafur t.d. kallaður (skilgreindur sem) `char`. Eftirfarandi setning býr til nýja breytu með nafninu `fred` sem hefur tagið `char`.

```
char fred;
```

Ofangreind setning er kölluð **yfirlýsing** (e. declaration) því setningin lýsir yfir breytu af tilteknu tagi.

Tag breytu ákvarðar hvers konar gildi hún getur geymt. Breyta af taginu `char` getur geymt stafi og það ætti ekki að koma á óvart að `int` breyta geti geymt heiltölur. Í C++ eru nokkur tög sem geta geymt strengjagildi en við skoðum það seinna í kafla 7.

Málskipanin sem notuð er til að lýsa yfir heiltölubreytu er:

```
int bob;
```

Hér er `bob` eitthvað handahófskennt nafn á heiltölubreytunni. Almennt séð er góð regla að gefa breytum nöfn sem gefa til kynna fyrir hvað þær standa. Ef þú sæir t.d. eftirfarandi yfirlýsingar:

```
char firstLetter;
char lastLetter;
int hour, minute;
```

þá gætir þú væntanlega getið þér til um hvaða gildi stendur til að geyma í þessum breytum. Þetta dæmi sýnir líka málskipanina sem notuð er til þess að lýsa yfir mörgum breytum af sama tagi: `hour` og `minute` eru bæði heiltölur með tagið `int`.

2.4 Gildisveiting

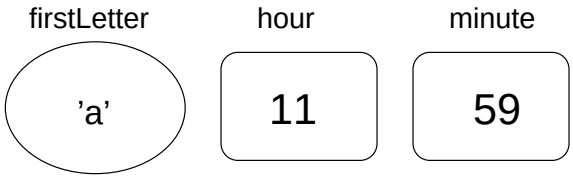
Þegar við höfum lýst yfir breytum þá viljum við gjarnan geyma einhver gildi í þeim. Það gerum við með svokölluðum **gildisveitingum** (e. assignments):

```
firstLetter = 'a';   // give firstLetter the value 'a'
hour = 11;           // assign the value 11 to hour
minute = 59;         // set minute to 59
```

Þetta dæmi sýnir þrjár gildisveitingar og athugasemdirnar sýna hvernig hægt er að tala um gildisveitingarsetningar á mismunandi hátt. Það má vera að orðaforðinn sé ruglingslegur en hugmyndin er einföld:

- Þegar þú lýsir yfir breytu þá býrð þú til minnissvæði sem geymir gildi breytunnar.
- Í gildisveitingarsetningu er breytu gefið gildi.

Algeng leið til að tákna breytu á pappír er að teikna kassa með nafni breytunnar fyrir utan kassann og gildi breytunnar innan í kassanum. Þessi tegund af mynd er kölluð **stöðurit** (e. state diagram) vegna þess að hún sýnir hver staða sérhverrar breytu er (hægt er að hugsa sér að breyta hafi "state of mind"). Eftirfarandi mynd sýnir hvaða áhrif gildisveitingarsetningarnar þrjár hafa:



Ég nota stundum mismunandi form til að gefa til kynna mismunandi tög breytna. Þessi form ættu að minna þig á að ein C++ reglan krefst þess að breyta hafa sama tag og gildið sem þú gefur henni. Þú getur t.d. ekki geymt streng í `int` breytu. C++ þýðandi samþykkir ekki eftirfarandi setningu:

```
int hour;
hour = "Hello.";       // WRONG !!
```

Þessi regla veldur stundum ruglingi því þar eru margar leiðir til að breyta gildi af einu tagi í annað og C++ breytir meira að segja tagi stundum sjálkrafa. Sem stendur skaltu muna að almenna reglan er sú að breytur og gildi þurfa að vera af sama tagi en við munum tala um sérstök tilfelli síðar.

Annað sem getur valdið misskilningi er að sumir strengir *líta út* eins og heiltölur en eru það ekki. Strengurinn `"123"`, sem samanstendur af stöfunum `1`, `2` og `3`, er t.d. ekki sami hluturinn og *talan* `123`. Eftirfarandi gildisveiting er ólögleg:

```
minute = "59";         // WRONG!
```

2.5 Breytur skrifaðar út

Þú getur skrifað út gildi breytu með því að nota sömu skipanir og við notuðum til að skrifa út "einföld" gildi:

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';
```

```
cout << "The current time is ";
cout << hour;
cout << colon;
cout << minute;
cout << endl;
```

Þetta forrit býr til tvær heiltölubreytur með nöfnunum `hour` og `minute`, og stafabreytu með nafninu `colon`. Forritið gefur þessum breytum viðeigandi gildi og notar síðan runu af úttakssetningum til að skrifa út eftirfarandi:

```
The current time is 11:59
```

Þegar við tölum um að "skrifa út" breytu þá eigum við við að skrifa út *gildi* breytunnar. Til að skrifa út *nafn* breytu þurfum við að setja nafnið inni í gæsalappir, t.d. `cout « "hour";`

Eins og við höfum áður séð þá er hægt að hafa fleiri en eitt gildi í einni og sömu úttakssetningunni. Það gerir forritið að ofan hnitmiðaðra (samþjappaðra) (e. concise):

```
int hour, minute;
char colon;

hour = 11;
minute = 59;
colon = ':';

cout << "The current time is " << hour << colon << minute << endl;
```

Í einni og sömu línunni skrifar þetta forrit út streng, tvær heiltölur, staf og sérstaka gildið `endl` – þetta er mjög áhrifamikið!

2.6 Lykilorð

Ég nefndi að ofan að þú gætir gefið breytum hvaða nafn sem þér dettur í hug en það er ekki alls kostar rétt. Það eru nefnilega nokkur orð í C++ sem eru frátekin vegna þess að þau eru notuð af þýðandanum til að þátta (e. parse) forritið þitt og ef þú notar þessi nöfn fyrir breytunöfn þá "ruglast" þýðandinn. Meðal þessara nafna, sem kölluð eru **lykilorð** (e. keywords), eru `int`, `char`, `void`, `endl` og mörg fleiri.

Heildarlistinn yfir lykilorð má finna í C++ staðlinum, sem er obinbera skilgreiningin á málinu samþykkt af "International Organization for Standardization (ISO)", þann 1. september 1998. Hægt er að hlaða niður afriti af staðlinum frá

`http://www.ansi.org/`

Í stað þess að læra listann af lykilorðum utan að þá legg ég til að þú nýtir þér eiginleikann sem boðið er upp á í mörgum þróunarumhverfum, þ.e. "code highlighting". Þetta merkir að þegar þú forritar þá birtast mismunandi hlutar forritsins í mismunandi litum. Til að mynda gætu lykilorð birst í bláum lit, strengir í rauðum og annar kóði í svörtum. Ef þú býrð til breytunafn sem birtist í bláum lit þá veistu að um lykilorð er að ræða!

2.7 Virkjar

Virkjar (e. operators) eru sérstök tákn sem standa fyrir einfalda útreikninga eins og samlagningu og margföldun. Flestir virkjar í C++ gera nákvæmlega það sem þú myndir búast við vegna þess að þeir eru þekkt stærðfræðitákn. T.d. er + notað til að leggja saman tvær heiltölur.

Hér má sjá dæmi um leyfilegar segðir (e. expressions) hvers merking ætti að vera augljós:

```
1+1        hour-1       hour*60 + minute     minute/60
```

Segðir geta bæði innhaldið breytunöfn og gildi. Í sérhverju tilviki er nafn breytu skipt út fyrir gildi hennar áður en útreikningurinn er framkvæmdur.

Samlagning, frádráttur og margföldun skila því sem búast má við en deiling gæti komið á óvart! Þetta forrit:

```
int hour, minute;
hour = 11;
minute = 59;
cout << "Number of minutes since midnight: ";
cout << hour*60 + minute << endl;
cout << "Fraction of the hour that has passed: ";
cout << minute/60 << endl;
```

skilar eftirfarandi úttaki:

```
Number of minutes since midnight: 719
Fraction of the hour that has passed: 0
```

Fyrri línan í úttakinu er það sem við mátti búast en seinni línan er skrýtin. Gildi breytunnar `minute` er 59 og 59 deilt með 60 er 0,98333 en ekki 0. Ástæðan fyrir þessu misræmi er sú að C++ framkvæmir það sem kallast **heiltöludeiling** (e. integer division).

Þegar báðir **þolendur** (e. operands) virkja eru heiltölur þá verður niðurstaðan líka heiltala. Samkvæmt skilgreiningu er niðurstaða heiltöludeilingar rúnuð *niður* jafnvel þó næsta heiltala sé mjög "nálægt".

Í þessu dæmi væri mögulegt að reikna út prósentur í stað almenns brots:

```
cout << "Percentage of the hour that has passed: ";
cout << minute*100/60 << endl;
```

Niðurstaðan er:

```
Percentage of the hour that has passed: 98
```

Í þessu tilviki er niðurstaðan líka rúnuð niður en nú er svarið aftur á móti um það bil rétt. Til að fá enn réttara svar getum við notað annað tag á breytu en heiltölutag, þ.e. kommutölutag (e. floating-point type). Kommutölubreytur geta geymt brot en við munum skoða kommutölur betur í næsta kafla.

2.8 Forgangur aðgerða

Þegar fleiri en einn virki er notaður í segð þá fer röð aðgerðanna eftir reglum um **forgang** (e. precedence). Nákvæm skýring á forgangsreglum getur verið flókin en við getum notað eftirfarandi reglur til að byrja með:

- Margföldun og deiling hafa hærri forgang en samlagning og frádráttur. M.ö.o. margföldun og deiling eru framkvæmdar áður en samlagning og frádráttur. Þannig er útkoman úr `2*3-1` 5 en ekki 4 og `2/3-1` skilar -1 en ekki 1 (útkoman úr heiltöludeilingunni `2/3` er 0).
- Ef virkjar hafa sama forgang þá eru þeim beitt frá vinstri til hægri. Í segðinni `minute*100/60` er margfölduninni beitt fyrst sem skilar `5900/60` og lokaniðurstaðan verður þá `98`. Ef virkjunum væri beitt frá hægri til vinstri þá yrði niðurstaðan `59*1` = `59` sem væri rangt.
- Ef þú vilt beita annarri forgangsröðun aðgerða en þeirri innbyggðu þá getur þú notað sviga. Segðir í svigum eru reiknaðar fyrst og því er niðurstaðan úr `2 * (3-1)` jafngild 4. Einnig er hægt að nota sviga til að gera segðir læsilegri eins og í `(minute * 100) / 60` þó svo að niðurstaðan sé sú sama án þess að nota sviga í þessu tilfelli.

2.9 Stafavirkjar

Það er athyglisvert að hægt er að beita sömu stærðfræðivirkjum á heiltölur og stafi. Dæmi:

```
char letter;
letter = 'a' + 1;
cout << letter << endl;
```

Þessi forritsbútur skrifar út stafinn `b`. Þrátt fyrir að það sé í raun setningafræðilega löglegt að margfalda stafi þá er nánast aldrei nauðsyn til að gera það.

Ég hef áður sagt að það sé eingöngu hægt að gefa heiltölubreytum heiltölugildi og stafabreytum stafagildi en það er ekki alveg rétt. Í ákveðnum tilfellum breytir C++ sjálfkrafa um tög. Eftirfarandi forritsbútur er til að mynda löglegur:

```
int number;
number = 'a';
cout << number << endl;
```

Útkoman er 97 en það er sú tala sem notuð er í C++ til að tákna stafinn `'a'`. Samt sem áður er almennt séð góð regla að meðhöndla stafi sem stafi og heiltölur sem heiltölur og aðeins að breyta einu tagi í annað ef nauðsyn krefur.

Sjálfvirk tagbreyting (e. type conversion) er dæmi um þekkt vandamál við hönnun forritunarmáls – annars vegar er áhersla á **formhyggju** (e. formalism) (skilyrði um að formlegt mál skuli hafa einfaldar reglur með fáum undantekningum) og hins vegar áhersla á **þægindi** (e. convenience) (skilyrði um að forritunarmál skuli vera þægilegt í notkun).

Oftar en ekki vinna þægindin sem er yfirleitt gott fyrir vana forritara sem er þá hlíft við strangri formhyggju. Aftur á móti er það stundum slæmt fyrir byrjendur í forritun sem finnast oft reglurnar vera flóknar og með fjölda undantekninga. Í þessari bók hef ég reynt að einfalda hlutina með því að leggja áherslu á reglurnar og sleppa að fjalla um margar undantekningar.

2.10 Samsetning

Hingað til höfum við skoðað einstakar einingar forritunarmáls – breytur, segðir og setningar – án þess að ræða hvernig þær eru settar saman.

Einn gagnlegasti eiginleiki forritunarmála er geta þeirra til að setja saman litlar forritseiningar og mynda þannig heildstætt forrit. Við vitum t.d. hvernig á að margfalda saman heiltölur og við vitum hvernig á að skrifa út gildi – svo vill reyndar til að við getum gert hvort tveggja á sama tíma:

```
cout << 17 * 3;
```

Reyndar ætti ég ekki að segja "á sama tíma" því í raun á margföldunin sér stað áður en niðurstaðan er skrifuð út en punkturinn er sá að að sérhver segð, sem inniheldur tölur, stafi og breytur, getur verið notuð í úttakssetningu. Við höfum þegar séð eitt dæmi um þetta:

```
cout << hour*60 + minute << endl;
```

Hvaða segð sem er getur einnig komið fyrir á hægri hlið gildisveitingarsetningar (e. assignment statement):

```
int percentage;
percentage = (minute * 100) / 60;
```

Þessi eiginleiki virðist kannski ekki mikilvægur núna en við munum sjá önnur dæmi þar sem samsetning gerir okkur kleift að tjá (e. express) útreikninga á "snyrtilegan" og hnitmiðaðan hátt.

VIÐVÖRUN: Það eru takmarkanir á því hvar hægt er að nota segðir. Þetta á sérstaklega við um vinstri hlið gildisveitingarsetningar sem verður að vera nafn á

breytu en ekki segð. Ástæðan er sú að vinstri hliðin stendur fyrir minnissvæði sem geyma mun gildi segðarinnar á hægri hlið. Aftur á móti standa segðir ekki fyrir minnissvæði heldur eingöngu gildi. Eftirfarandi er því ekki löglegt: `minute+1 = hour;`.

2.11 Orðalisti

breyta (e. variable): Nafn sem stendur fyrir minnissvæði sem geymir gildi. Allar breytur í C++ hafa tag sem ákvarðar hvers konar gildi breytan getur geymt.

gildi (e. value): Stafur, tala eða aðrir hlutir sem hægt er að geyma í breytu.

tag (e. type): Mengi af gildum. Tögin sem við höfum þegar séð eru t.d. heiltölur (`int`) og stafir (`char`).

lykilorð (e. keyword): Frátekið orð sem notað er af þýðandanum til að þátta forritið. Dæmi um lykilorð: `int`, `void` og `endl`.

setning (e. statement): Forritseining sem stendur fyrir tiltekna skipun eða aðgerð. Þær setningar sem við höfum séð hingað til eru yfirlýsingar, gildisveitingar og úttakssetningar.

yfirlýsing (e. declaration): Setning sem býr til nýja breytu og tilgreinir tag hennar.

gildisveiting (e. assignment): Setning sem gefur tiltekinni breytu gildi.

segð (e. expression): Samsetning breytna, virkja og gilda sem í heild sinni stendur fyrir eitt gildi (niðurstöðu). Segð hefur tag sem ákvarðast af þeim virkjum og þolendum sem koma fyrir í segðinni.

virki (e. operator): Sérstakt tákn sem stendur fyrir tiltekna aðgerð eins og samlagningu eða margföldun.

þolandi (operand): Eitt af þeim gildum sem virkja er beitt á.

forgangur (e. precedence): Segir til um í hvaða röð virkjum er beitt.

samsetning (e. composition): Sá eiginleiki að geta sett saman einfaldar segðir og setningar til að tjá flóknar aðgerðir á samþjappaðan hátt.

Kafli 3

Föll

3.1 Kommutölur

Í síðasta kafla áttum við í nokkrum vandræðum með tölur sem eru ekki heiltölur. Við leystum vandamálið að hluta til með því að reikna út prósentur í stað brots. Almennari lausn er hins vegar að nota kommutölur (e. floating-point) sem geta staðið fyrir bæði brot og heiltölur. Tvenns konar kommutölur eru í C++, `float` og `double`. Í þessari bók munum við eingöngu nota `double` tölur.

Þú getur búið til kommutölur og gefið þeim gildi með sömu málskipan og notuð er fyrir önnur tög. Dæmi:

```
double pi;
pi = 3.14159;
```

Það er einnig mögulegt að lýsa yfir breytu og gefa henni gildi á sama tíma:

```
int x = 1;
String empty = "";
double pi = 3.14159;
```

Reyndar vill svo til að þessi leið er mjög algeng. Samsett yfirlýsing (e. declaration) og gildisveiting (e. asssignment) er kölluð **frumstilling** (e. initialization).

Þó svo að kommutölur séu gagnlegar þá valda þær stundum ruglingi vegna þess að svo virðist sem heiltölur og kommutölur skarist. T.d., er gildið `1` heiltala, kommutala eða hvoru tveggja?

C++ gerir greinarmun á heiltölugildinu `1` og kommutölugildinu `1.0` þrátt fyrir að þau virðast standa fyrir sömu töluna. Ástæðan er sú að gildin tvö tilheyra mismunandi tögum og almennt gildir að ekki er leyfilegt að framkvæma gildisveitingu "á milli" taga. Þetta er t.d. ekki leyfilegt í C++

```
int x = 1.1;
```

vegna þess að breytan á vinstri hlið gildisveitingarinnar er af taginu `int` og gildið á hægri hlið er `double`. Það er aftur á móti auðvelt að gleyma þessari reglu, sérstaklega vegna þess að í sumum tilvikum breytir C++ þýðandi einu tagi í annað á sjálfvirkan hátt. T.d. ætti

```
double y = 1;
```

tæknilega séð ekki að vera leyfilegt en C++ þýðandi leyfir þetta með því að breyta sjálfur `int` (gildinu 1) í `double`. Þessi "linkind" af hálfu þýðandans getur verið þægileg fyrir forritarann en getur jafnframt leitt til vandræða. Dæmi:

```
double y = 1 / 3;
```

Hér gætir þú búist við því að breytan `y` fái gildið `0,333333`, sem er leyfilegt kommutölugildi, en í raun fær hún gildið `0.0`. Ástæðan er sú að segðin á hægri hlið gildisveitingarinnar er hlutfall tveggja heiltalna og því framkvæmir C++ *heiltöludeilingu*, hvers niðurstaða er heiltölugildið `0`. Þegar því gildi er breytt (af þýðandanum) í kommutölu þá er niðurstaðan `0,0`.

Ein leið til að leysa þetta vandamál er að gera gildi segðarinnar á hægri hlið að kommutölu:

```
double y = 1.0 / 3.0;
```

Þetta veldur því að `y` fær gildið `0,333333` eins og við var að búast.

Allar þær reikniaðgerðir sem við höfum séð hingað til – samlagning, frádráttur, margföldun og deiling – virka á kommutölum sem og á heiltölum. Það er hins vegar athyglisvert að undirliggjandi vélarmálsútreikningur er mismunandi eftir því hvort um kommutölur eða heiltölur er að ræða. Flestir örgjörvar hafa einmitt sérstakan búnað til að framkvæma aðgerðir á kommutölum.

3.2 Breyting á `double` í `int`

Eins og ég nefndi áður þá breytir C++ þýðandinn `int` í `double` á sjálfvirkan hátt ef þörf er á vegna þess að engar upplýsingar tapast í breytingunni. Á hinn bóginn þá krefst breyting á `double` í `int` afrúnunar (e. rounding off). C++ framkvæmir þá aðgerð ekki sjálfvirkt og því þarft þú, sem forritari, að vera meðvitaður um brotið sjálft (þ.e. sá hluti sem kemur á eftir kommunni) tapast.

Einfaldasta leiðin til að breyta kommutölu í heiltölu er að nota **tagmótun** (e. typecast). Tagmótun dregur nafn sitt af því að það gefur þér kost á því að "móta" gildi sem tilheyrir einu tagi yfir í annað tag.

Málskipan fyrir tagmótun er eins og málskipan fyrir fallakall. Dæmi:

```
double pi = 3.14159;
int x = int (pi);
```

`int` fallið skilar heiltölu þannig að `x` fær gildið 3. Að móta kommutölu í heiltölu hefur í för með sér að talan er rúnuð niður (e. rounded down) jafnvel þó svo að brotið sé 0.99999999.

Fyrir sérhvert tag í C++ er til samsvarandi fall sem tagmótar sitt viðfang í viðeigandi tag.

3.3 Stærðfræðiföll

Í stærðfræði hefur þú væntanlega séð föll eins og sin og log og hefur lært að reikna út gildi segða eins og $\sin(\pi/2)$ og $\log(1/x)$. Fyrst reiknar þú út gildi segðar innan sviga en það er kallað **viðfang** (e. argument) fallsins. T.d. er $\pi/2$ u.þ.b. 1,571 og $1/x$ er 0,1 (ef x hefur gildið 10).

Eftir þetta getur þú ákvarðar gildi fallsins sjálfs, annað hvort með því að fletta upp í töflu eða með því að framkvæma ýmsa útreikninga. sin af 1,571 er 1 og log af 0,1 er -1 (ef við gerum ráð fyrir því að log standi fyrir lógariþma með grunn 10).

Þetta ferli er hægt að endurtaka til að ákvarða gildi flóknari segða eins og $\log(1/\sin(\pi/2))$. Fyrst ákvörðum við gildi viðfangs innsta fallsins (þ.e. $(\pi/2)$), síðan reiknum við út gildi fallsins (þ.e. sin), o.s.frv.

C++ býður upp á mengi af innbyggðum (e. built-in) föllum sem inniheldur flestar þær stærðfræðilegu aðgerðir sem þú getur ímyndað þér. Kallað er á þessi stærðfræðiföll með því að nota málskipan sem er sambærileg við stærðfræðilega táknun:

```
double log = log (17.0);
double angle = 1.5;
double height = sin (angle);
```

Fyrsta setningin að ofan gefur breytunni `log` gildið lógariþmi af 17 (grunnur e). Það er einnig til fall `log10` sem reiknar út lógariþma miðað við grunn 10.

Þriðja setningin reiknar út sínus af gildinu sem geymt er í breytunni `angle`. C++ gerir ráð fyrir því að gildin sem notuð eru með sínus fallinu, og öðrum hornaföllum (`cos`, `tan`), séu í *radian*. Til að breyta gráðum í radian getur þú deilt með 360 og margfaldað með 2π.

Ef þú þekkir ekki gildið á π (með 15 aukastöfum!) þá getur þú reiknað það út með því að nota `acos` fallið. Arccosínus (eða andhverfa cosínus) af -1 er π vegna þess að cosínus af π er -1.

```
double pi = acos(-1.0);
double degrees = 90;
double angle = degrees * 2 * pi / 360.0;
```

Áður en þú getur notað eitthvað af stærðfræðiföllunum þarftu að taka inn (e. include) sérstaka **hausaskrá** (e. header file) í forritið þitt. Hausaskrá inniheldur upplýsingar um föll, sem eru skilgreind annars staðar en í þínu eigin forriti, og sem þýðandinn þarf á að halda. Í "Hello, world!" forritinu tókum við t.d. inn haus nefndan `iostream` með því að nota **include** setningu:

```
#include <iostream>
using namespace std;
```

`iostream` inniheldur upplýsingar um inntaks- og úttaksstrauma (e. I/O streams), þ.m.t. um úttaksstrauminn `cout`. C++ notar öflugan eiginleika sem

kallaður er **nafnasvið** (e. namespaces) sem gerir þér t.d. kleift að útfæra `cout` á þinn eigin máta. Í flestum tilvikum notum við reyndar hina stöðluðu útfærslu sem skilgreind er í nafnasviðinu std. Við látum þýðandann vita af þessu með línunni

```
using namespace std;
```

Þumalputtareglan er sú að þú átt að skrifa `using namespace std;` í hvert sinn sem þú ætlar að nota `iostream`.

Á sambærilegan hátt inniheldur `cmath` hausaskráin upplýsingar um stærðfræðiföll. Þú getur tekið þá skrá inn, ásamt `iostream`, í upphafi forritsins þíns:

```
#include <cmath>
```

Hausaskrár sem byrja á 'c' gefa til kynna að þær hafi upphaflega verið búnar til fyrir forritunarmálið **C**.

3.4 Samsetning

Föll í C++ geta verið samsett á sama hátt og stærðfræðiföll. Þetta merkir að þú getur notað eina segð sem hluta af annarri. Þú getur t.d. notað hvaða segð sem er sem viðfang í fall:

```
double x = cos (angle + pi/2);
```

Í þessari setningu er deilt í gildið `pi` með tveimur og gildinu á breytunni `angle` bætt við niðurstöðuna. Summan er síðan send sem viðfang í fallið `cos`.

Þú getur einnig sent niðurstöðuna úr einu falli sem viðfang í annað fall:

```
double x = exp (log (10.0));
```

Hér er tekinn lógariþmi (með grunn e) af 10 og niðurstaðan (nefnum hana t) síðan send inn í exp fallið sem reiknar e í veldinu t. Breytan `x` fær að lokum gildið úr heildarniðurstöðunni sem ég vona að þú vitir hver er!

3.5 Nýjum föllum bætt við

Hingað til höfum við eingöngu notað föll sem eru innbyggð í C++ en við getum einnig bætt við nýjum föllum. Reyndar vill svo til að við höfum þegar bætt við einu nýju falli: `main`. Fallið `main` er sérstakt að því leyti til að það gefur til kynna hvar keyrsla forritsins á að byrja en málskipan fyrir `main` er sú sama og fyrir hvaða aðra fallaskilgreiningu sem er:

```
void NAFN ( LISTI AF VIÐFÖNGUM ) {
  SETNINGAR
}
```

Þú getur gefið fallinu þínu hvaða nafn sem er með þeirri undantekningu að þú getur hvorki kallað það `main` né notað annað C++ lykilorð. Listinn af viðföngum skilgreinir hvaða upplýsingar (ef nokkrar) þarf að gefa fallinu þegar það er notað (þegar **kallað** (e. call)) er á það.

`main` tekur engin viðföng eins og sjá má með tómum svigum `()` í skilgreiningunni á fallinu. Fyrstu tvö föllin sem við munum skrifa taka heldur engin viðföng – málskipanin lítur þá svona út:

```
void newLine () {
  cout << endl;
}
```

Þessu falli hefur verið gefið nafnið `newLine`. Það inniheldur aðeins eina setningu, þ.e. skrifar út stafinn `endl` sem stendur fyrir nýja línu.

Úr `main` getum við kallað á þetta nýja fall með því að nota málskipan sem er svipuð þeirri sem við notum þegar við köllum á innbyggð föll:

```
int main ()
{
  cout << "First Line." << endl;
  newLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

Úttakið úr forritinu er:

```
First line.

Second line.
```

Taktu eftir auka (tómu) línunni á milli línanna tveggja. En hvað ef við þyrftum á meira bili að halda á milli línanna? Við gætum kallað á þetta sama fall nokkrum sinnum:

```
int main ()
{
  cout << "First Line." << endl;
  newLine ();
  newLine ();
  newLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

Annar möguleiki væri sá að skrifa nýtt fall, t.d. með nafninu `threeLine`, sem skrifar út þrjár nýjar línur:

```
void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

Hér eru nokkur atriði sem vert er að gefa gaum:

- Hægt er að kalla á sama fallið aftur og aftur. Reyndar vill svo til að það er einmitt algengt og gagnlegt.
- Hægt er að láta eitt fall kalla á annað. Í forritinu að ofan kallar `main` á `threeLine` og `threeLine` kallar á `newLine`. Þetta er einnig algengt og gagnlegt.
- Í `threeLine` skrifaði ég þrjár setningar í einni og sömu línunni sem er setningafræðilega rétt (mundu að bil og tómar línur breyta ekki merkingu forrits í C++). Á hinn bóginn bendi ég á að það er yfirleitt betra að hafa eina setningu í hverri línu því þannig verður forritið læsilegra. Í þessari bók brýt ég stundum þessa reglu til að spara pláss.

Á þessum tímapunkti er kannski ekki ljóst hvað ávinnst með því að búa til öll þessi föll. Fyrir því eru margar ástæður en forritið að ofan sýnir fram á tvær þeirra:

1. Með því að búa til nýtt fall þá getur þú gefið safni setninga nafn. Föll geta einfaldað forrit með því að hylja flókna reikninga/aðgerðir "á bak við" eina skipun og með því að nota orð sem eru okkur töm í stað "skrýtinna" tákna. Hvort er læsilegra, `newLine` eða `cout << endl`?
2. Það að búa til fall getur eytt endurteknum kóða og þar með gert forrit styttra. Einföld leið til að prenta t.d. níu nýjar línur í röð væri að kalla á `threeLine` þrisvar sinnum. Hvernig myndir þú prenta út 27 nýjar línur?

3.6 Skilgreiningar og notkun

Ef við tökum saman alla kóðabútana úr kaflanum hér á undan þá lítur forritið í heild sinni svona út:

```
#include <iostream>
using namespace std;

void newLine ()
{
  cout << endl;
}

void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

Þetta forrit inniheldur þrjár fallaskilgreiningar: `newLine`, `threeLine` og `main`.

Skilgreiningin á `main` inniheldur setningu sem kallar á `threeLine`. Á sama hátt kallar `threeLine` þrisvar sinnum á `newLine`.

Taktu eftir að skilgreiningin á sérhverju falli kemur á undan þeim stað þar sem fallið er notað (þar sem kallað er á það). C++ krefst þess að skilgreiningin á falli komi á undan fyrstu noktun þess. Þú ættir að prófa að þýða þetta forrit með föllunum í annarri röð til að sjá hvers konar villumeldingar þú færð.

3.7 Forrit með mörgum föllum

Þegar þú skoðar forrit sem inniheldur nokkur föll þá er ruglandi að lesa forritið frá "toppi til táar" því það endurspeglar ekki **keyrsluröð** (e. order of execution) forritsins.

Keyrslan hefst alltaf í fyrstu setningunni í `main`, burtséð frá því hvar `main` er staðsett í forritinu (það er reyndar oft neðst í kóðanum). Setningar eru keyrðar, ein í einu, í röð þangað til komið er að fallakalli. Fallaköll eru eins og krókur í flæði keyrslunnar. Í stað þess að fara í næstu setningu þá er stokkið í fyrstu línu fallsins sem kallað er á, allar setningar fallsins keyrðar og síðan stokkið til baka og þráðurinn tekinn upp þar sem frá var horfið.

Þetta hljómar svo sem einfalt en mundu að eitt fall getur kallað á annað. Þannig að þegar við eru í miðri keyrslu á `main` gætum við þurft að stökkva burt og keyra setningarnar í `threeLine`. En meðan setningarnar í `threeLine` eru keyrðar þá eru við "truflið" (e. interrupted) þrisvar sinnum til að keyra `newLine`.

Sem betur fer þurfum við sem forritarar ekki að hafa áhyggjur af þessum stökkvum í föll því C++ þýðandinn býr til kóða sem sér um þetta fyrir okkur. Þegar `newLine` hættir heldur forritið áfram á réttum stað í `threeLine` og kemst að lokum til baka í `main` þar sem keyrslunni lýkur.

Boðskapurinn er sem sagt sá að þegar þú lest forritið þá skaltu ekki lesa það frá toppi til táar heldur fylgja keyrsluflæðinu (e. flow of execution).

3.8 Leppar og viðföng

Sum af þeim innbyggðu föllum sem við höfum skoðað hafa **leppa** (e. formal parameters) en þeir geyma þau gildi sem við látum fall hafa til að það geti gert það sem það á að gera. Ef þú þarft t.d. að finna sínus af einhverri tölu þá þarftu að gefa til kynna hver talan er. Þ.e. `sin` tekur `double` gildi sem viðfang.

Sum föll hafa fleiri en einn lepp. Dæmi um það er fallið `pow` sem tekur tvö `double` gildi, grunninn og veldisvísinn.

Í sérhverju þessara tilfella þarf bæði að taka fram hversu margir lepparnir eru og af hvaða tagi þeir eru. Það ætti því ekki að koma á óvart að þegar þú skrifar fallaskilgreiningu þá inniheldur listinn yfir leppana einnig tag þeirra: Dæmi:

```
void printTwice (char phil) {
  cout << phil << phil << endl;
}
```

Þetta fall er með lepp með nafninu `phil` sem er af taginu `char`. Gildið sem kemur inn, hvert svo sem það er (á þessum tímapunkti vitum við það ekki), er prentað tvisvar og ný lína á eftir. Ég notaði hér nafnið `phil` til að gefa til kynna að þú getur notað hvaða nafn sem er á leppum en almennt séð þá ættir þú að velja eitthvað meira lýsandi nafn en `phil`.

Til að kalla á þetta fall verðum við að gefa því `char` gildi. Við gætum t.d. skrifað `main` fallið svona:

```
int main () {
  printTwice ('a');
  return 0;
}
```

`char` gildið í kallinu á `printTwice` er kallað **viðfang** (e. actual parameter/argument). Talað er um að senda viðfangið (e. to pass the argument) til fallsins. Í þessu tilviki `'a'` sent sem viðfang í `printTwice` sem prentar gildið út tvisvar.

Ef við hefðum breytu af taginu `char` þá gætum við notað hana sem viðfang í staðinn:

```
int main () {
  char argument = 'b';
  printTwice (argument);
```

```
    return 0;
}
```

Hér er eitt mikilvægt atriðið: Nafnið á breytunni sem við sendum sem viðfang hefur ekkert að gera með nafnið á leppnum (`phil`). Ég endurtek:

> **Nafnið á breytunni sem við sendum sem viðfang hefur ekkert að gera með nafnið á leppnum.**

Nöfnin geta verið þau sömu en þau geta líka verið mismunandi. Það er mikilvægt að gera sér grein fyrir því að leppurinn og viðfangið eru ekki sami hluturinn þó svo að þau hafi sama gildið (í þessu tilviki stafinn `'b'`).

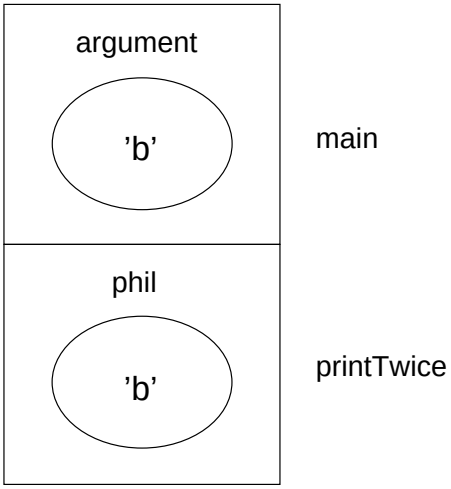
Gildið sem sent er sem viðfang verður að hafa sama tagið og leppurinn í fallinu sem kallað er á. Þessi regla er mikilvæg en getur verið ruglandi því C++ breytir stundum einu tagi í annað á sjálfvirkan hátt. Á þessum tímapunkti skaltu muna þessa almennu reglu en við munum ræða undantekningar frá henni síðar.

3.9 Leppar og breytur eru staðværar

Gildissvið (e. scope) leppa og breytna er aðeins innan í eigin föllum. Innan marka `main` er ekki neinn hlutur með nafninu `phil` aðgengilegur. Þýðandinn mun kvarta ef þú reynir að nota það nafn innan í `main`. Á sama hátt er enginn hlutur með nafninu `argument` aðgengilegur inni í `printTwice`.

Breytur eins og þessar eru sagðar vera **staðværar** (e. local). Það getur verið gott að teikna **staflarit** (e. stack diagram) til að gera sér grein fyrir leppum og staðværum breytum. Staflarit sýna (eins og stöðurit) gildi á sérhverri breytu en breyturnar eru innan í stærri boxum sem gefa til kynna hvaða föllum þær tilheyra.

Staflarit fyrir `printTwice` lítur svona út:



Í hvert skipti sem kallað er á fall til nýtt **tilvik** af upplýsingum um fallið (kallað **kvaðningafærsla** (e. activation record)). Sérhvert tilvik af fallinu inniheldur leppana og staðværu breyturnar. Í myndinni er tilvik af fallinu sýnt sem box með nafni fallsins að utanverðu en að innanverðu eru breytur og leppar.

Í dæminu hefur `main` eina staðværa breytu, `argument`, og enga leppa. `printTwice` hefur engar staðværar breytur en einn lepp með nafninu `phil`.

3.10 Föll með marga leppa

Málskipanin sem notuð er til að lýsa yfir og kalla á föll með mörgum leppum veldur of villum við þýðingu. Í fyrsta lagi þarf að muna að það þarf að lýsa yfir tagi á sérhverjum lepp. Dæmi:

```
void printTime (int hour, int minute) {
  cout << hour;
  cout << ":";
  cout << minute;
}
```

Það er freistandi að skrifa `(int hour, minute)` en sá ritháttur er aðeins löglegur fyrir yfirlýsingar á breytum en ekki á leppum!

Annað sem getur valdið ruglingi er að þú þarft ekki að lýsa yfir tagi á viðföngunum. Eftirfarandi er rangt:

```
int hour = 11;
int minute = 59;
printTime (int hour, int minute);   // WRONG!
```

Ástæðan er sú að þýðandinn getur séð hvert tagið á viðföngunum, breytunum `hour` og `minute`, er með því að fletta upp í skilgreiningunni á þeim. Það er sem sagt óþarfi og óleyfilegt að taka fram tagið á viðföngunum. Rétta málskipanin er `printTime (hour, minute)`.

3.11 Föll sem skila gildi

Þú ættir að hafa tekið eftir því að sum af þeim föllum sem við höfum notað, t.d. stærðfræðiföll, skila af sér gildi. Önnur föll, eins og `newLine`, framkvæma aðgerð en skila ekki gildi. Þetta vekur upp nokkrar spurningar:

- Hvað gerist ef þú kallar á fall og þú gerir ekkert við niðurstöðuna (þ.e. þú setur niðurstöðuna hvorki inn í breytu né notar hana sem hluta af stærri segð)?
- Hvað gerist ef þú nota fall sem ekki skilar gildi sem hluta af segð, eins og `newLine() + 7`?

- Getum við skrifað föll sem skila gildum eða sitjum við uppi með að skrifa föll eins og `newLine` og `printTwice`?

Svarið við þriðju spurningunni er "já", þ.e. við getum skrifað föll sem skila gildum og við munum einmitt gera það í næstu köflum. Ég mun láta þér eftir að svara fyrstu tveimur spurningunum með prófunum. Það er góð leið að spyrja þýðandann í sérhvert sinn sem þig vantar svar varðandi það hvort tiltekið atriði er leyfilegt eður ei í C++.

3.12 Orðalisti

kommutala (e. floating-point): Tag breytu (eða gildi) sem getur geymt brot sem og heiltölur. Það eru nokkur kommutölutög í C++ en við munum nota `double` í þessari bók.

frumstilling (e. initialization): Setning sem lýsir yfir breytu og gefur henni gildi á sama tíma.

fall (e. function): Röð setninga sem bera tiltekið nafn og framkvæma tiltekna(r) aðgerð(ir). Föll taka 0 eða fleiri viðföng og geta skilað af sér gildi.

leppur (e. parameter): Geymir upplýsingar sem gefnar eru upp þegar kallað er á fall. Leppar eru eins og breytur í þeim skilningi að þeir innihelda bæði gildi og eru af tilteknu tagi.

viðfang (e. argument): Gildi sem gefið er upp þegar kallað er á fall. Gildið verður að vera af sama tagi og viðkomandi leppur.

fallakall (e. function call): Veldur því að fall er keyrt.

Kafli 4

Skilyrði og endurkvæmni

4.1 Modulus virkinn

Modulus virkinn tekur tvær heiltölur sem þolendur og skilar *afganginum* sem myndast þegar fyrri þolandanum er deilt með þeim síðari. Í C++ er modulus virkinn táknaður með prósentumerkinu `%`. Málskipanin er nákvæmlega sú sama og fyrir aðra virkja:

```
int quotient = 7 / 3;
int remainder = 7 % 3;
```

Fyrsti virkinn að ofan, heiltöludeiling, skilar 2. Seinni virkinn skilar 1. Sem sagt, 7 deilt með 3 er 2 en afgangurinn er 1.

Það kann að koma á óvart hversu gagnlegur modulus virkinn er. Þú getur t.d. athugað hvort ein tala sé deilanleg með annarri: ef `x % y` er núll, þá er `x` deilanleg með `y`.

Þú getur líka notað modulus virkjann til að draga út tölustaf(i) sem koma fyrir lengst til hægri í tölu. T.d. skilar `x % 10` tölustafnum lengst til hægri í `x`. Á sama hátt skilar `x % 100` þeim tveimur tölustöfum sem eru lengst til hægri.

4.2 Skilyrt keyrsla

Til að skrifa gagnleg forrit þurfum við nánast alltaf að geta tékkað á ákveðnum skilyrðum og breytt flæði forritsins í samræmi við það.

Skilyrðissetningar (e. conditional statements) gera okkur þetta kleift. Einfaldasta form þeirra er `if` setning:

```
if (x > 0) {
  cout << "x is positive" << endl;
}
```

Segðin innan sviga er kallað skilyrðið. Ef það er satt (e. true) þá er setningin innan slaufusviganna keyrð. Ef skilyrðið er ekki satt (e. false) þá gerist ekkert.

Skilyrðið sjálft getur innihaldið sérhvern af eftirfarandi `skilyrðisvirkjum` (e. comparison operators):

```
x == y               // x equals y
x != y               // x is not equal to y
x > y                // x is greater than y
x < y                // x is less than y
x >= y               // x is greater than or equal to y
x <= y               // x is less than or equal to y
```

Þú þekkir vafalaust þessar aðgerðir en málskipanin sem C++ notar er aðeins frábrugðin stærðfræðitáknum eins og $=$, $\neq$ and $\leq$. Algeng forritunarvilla er að nota einfalt `=` í stað tvöfalds `==`. Þú þarft að muna að `=` er gildisveitingarvirki (e. assignment operator) en `==` er samanburðarvirki (e. comparison operator). Einnig er vert að benda á að hvorki `=<` né `=>` er til í C++.

Segðirnar sitt hvoru megin við samanburðarvirkja þurfa að vera af sama tagi. Þú getur eingöngu borið saman `int` og `int` annars vegar og `double` og `double` hins vegar. Á þessum tímapunkti getur þú ekki borið saman tvo strengi (`string`). Það er reyndar hægt og við munum fjalla um það síðar í bókinni.

4.3 Varaleið

Önnur tegund af skilyrtri keyrslu er varaleið (e. alternative execution). Í varaleið eru tveir möguleikar og skilyrðissegðin ákvarðar hvor leiðin er farin (keyrð). Málskipanin lítur svona út:

```
if (x%2 == 0) {
  cout << "x is even" << endl;
} else {
  cout << "x is odd" << endl;
}
```

Ef afgangurinn, sem fæst með því að deila `x` með 2, er núll þá vitum við að `x` er slétt tala og kóðinn skrifar út skilaboð þess efnis. Ef skilyrðið `(x%2 == 0)` er ósatt þá keyrast setningarnar í `else` hlutanum. Þar sem skilyrðið er annað hvort satt eða ósatt (true eða false) þá mun önnur hvor leiðin verða keyrð, en ekki báðar.

Ef þú telur að þú gætir oft þurft að tékka á því hvort tiltekin tala er slétt tala eða oddatala þá væri gott að "pakka" (e. "wrap") þessum kóða inn í fall, t.d.:

```
void printParity (int x) {
  if (x%2 == 0) {
    cout << "x is even" << endl;
```

```
  } else {
    cout << "x is odd" << endl;
  }
}
```

Þar með áttu fall með nafninu `printParity` sem skrifar út viðeigandi skilaboð fyrir hvaða heiltölu sem send er inn sem viðfang. Úr `main` gætir þú kallað á þetta fall á eftirfarandi hátt:

```
  printParity (17);
```

Mundu að þegar þú *kallar* á fall þá þarftu ekki að tilgreina tagið á viðföngunum því C++ þýðandinn finnur sjálfur út af hvað tagi þau eru. Ekki freistast til að skrifa:

```
int number = 17;
printParity (int number);          // WRONG!!!
```

4.4 Skilyrðiskeðjur

Stundum þarftu að tékka á mörgum tengdum skilyrðum og velja eina af mörgum mögulegum aðgerðum. Ein leið til að gera þetta er að tengja saman röð af `if` og `else` og mynda þannig **skilyrðiskeðjur** (e. chained conditional):

```
if (x > 0) {
  cout << "x is positive" << endl;
} else if (x < 0) {
  cout << "x is negative" << endl;
} else {
  cout << "x is zero" << endl;
}
```

Þessar keðjur geta verið eins langar og þú vilt hafa þær en eftir því sem þær eru lengri því ólæsilegri verða þær! Ein leið til að gera þær læsilegri er að nota inndrátt (e. indentation) í kóðanum, eins og gert er í dæminu að ofan. Ef þú sérð til þess að allar setningarnar og slaufusvigarnir séu með sama inndrátt þá er ólíklegra að þú gerir einhverjar málskipunarvillur og ef þú gerir villur þá finnur þú þær fljótar.

4.5 Hreiðruð skilyrði

Í stað þess að tengja skilyrði saman þá getur þú einnig hreiðrað (e. nest) eitt skilyrði inni í öðru. Við gætum hafa skrifað dæmið að ofan á þennan hátt:

```
if (x == 0) {
  cout << "x is zero" << endl;
```

```
  } else {
    if (x > 0) {
      cout << "x is positive" << endl;
    } else {
      cout << "x is negative" << endl;
    }
  }
```

Þessi kóði samanstendur af ytra skilyrði (`x == 0`) sem inniheldur tvær mögulegar kvíslir (e. branches). Fyrsta kvíslin er einfaldlega úttakssetning en seinni kvíslin samanstendur af annarri `if` setningu sem sjálf hefur tvær kvíslir. Þessar tvær kvíslir eru báðar úttakssetnignar en gætu þess vegna líka verið aðrar skilyrðissetningar.

Taktu eftir því að inndrátturinn hjálpar við að sýna uppbyggingu kóðans en samt sem áður geta hreiðruð skilyrði verið ólæsileg. Almennt séð ættir þú að forðast að skrifa hreiðruð skilyrði ef þú kemst hjá því.

Á hinn bóginn má segja að þar sem þessi tegund af **hreiðruðum strúktúr** er algeng, og við munum sjá hana aftur, þá er gott fyrir þig að venjast þessu strax.

4.6 `Return` setningin

`Return` setning gerir þér kleift að hætt keyrslu falls áður en komið er að lokum þess. Ein ástæða fyrir notkun return setningar er þegar villuskilyrði (e. error condition) reynist satt:

```
#include <cmath>

void printLogarithm (double x) {
  if (x <= 0.0) {
    cout << "Positive numbers only, please." << endl;
    return;
  }

  double result = log (x);
  cout << "The log of x is " << result;
}
```

Í þessu dæmi er skilgreining á fallinu `printLogarithm` sem hefur `double` nefnt `x` sem lepp. Fallið byrjar á því að athuga hvort `x` er minna eða jafnt og núll og ef satt reynist þá skrifar það úr villuskilaboð og notar síðan `return` til að hætta keyrslu fallsins. Keyrsluflæðið (e. flow of execution) færist þá sjálfkrafa til þess sem kallaði á fallið en þær setningar sem eftir standa í fallinu `printLogarithm` eru ekki framkvæmdar.

Taktu eftir að ég notaði kommutölugildi á hægri hlið skilyrðisins vegna þess að það er breyta af kommutölutagi á vinstri hliðinni.

4.7 Endurkvæmni

Ég nefndi í síðasta kafla að eitt fall getur kallað á annað og við höfum séð nokkur dæmi um það. Ég nefndi hins vegar ekki að fall getur einnig kallað á sjálft sig! Það er kannski ekki augljóst af hverju það getur verið gott en svo vill reyndar til að þessi möguleiki er einn af þeim mest töfrandi og áhugaverðustu sem forrit getur gert.

Skoðaðu t.d. eftirfarandi fall:

```
void countdown (int n) {
  if (n == 0) {
    cout << "Blastoff!" << endl;
  } else {
    cout << n << endl;
    countdown (n-1);
  }
}
```

Nafnið á þessu falli er `countdown` og það tekur heiltölu sem viðfang. Ef viðfangið er núll þá skrifar það út orðið "Blastoff." Annars skrifar það út gildi leppsins og kallar á fallið `countdown` – sjálft sig – og sendir gildið `n-1` sem viðfang.

Hvað gerist ef við köllum á þetta fall á eftirfarandi hátt:

```
#include <iostream>

void countdown (int n) {
  if (n == 0) {
    cout << "Blastoff!" << endl;
  } else {
    cout << n << endl;
    countdown (n-1);
  }
}

int main ()
{
  countdown (3);
  return 0;
}
```

Keyrslan á `countdown` byrjar með `n=3` og þar sem `n` er ekki núll þá skrifar það út gildið 3 og kallar síðan á sjálft sig ...

> Keyrslan á `countdown` byrjar með `n=2` og þar sem `n` er ekki núll þá skrifar það út gildið 2 og kallar síðan á sjálft sig ...
>
> > Keyrslan á `countdown` byrjar með `n=1` og þar sem `n` er ekki núll þá skrifar það út gildið 1 og kallar síðan á sjálft sig ...

Keyrslan á `countdown` byrjar með `n=0` og þar sem `n` er núll þá skrifar það út orðið "Blastoff!" og hættir síðan keyrslu.

countdown sem fékk `n=1` sem viðfang hættir keyrslu.

countdown sem fékk `n=2` sem viðfang hættir keyrslu.

countdown sem fékk `n=3` sem viðfang hættir keyrslu.

Og þá erum við aftur komin í `main` (þvílíkt ferðalag!). Endanlegt úttak lítur því svona út:

```
3
2
1
Blastoff!
```

Til að taka annað dæmi um endurkvæmni skulum við líta aftur á föllin `newLine` og `threeLine`.

```
void newLine () {
  cout << endl;
}

void threeLine () {
  newLine ();  newLine ();  newLine ();
}
```

Þrátt fyrir að þessi föll virki þá gæti ég ekki notað þau ef ég myndi vilja skrifa út 2 nýjar (auðar) línur eða 106 nýjar línur. Betri útfærsla væri:

```
void nLines (int n) {
  if (n > 0) {
    cout << endl;
    nLines (n-1);
  }
}
```

Þetta fall er sambærilegt við `countdown`. Svo lengi sem `n` er stærra en núll þá skrifar það út eina nýja línu og kallar svo aftur á sjálft sig til að skrifa út `n-1` nýjar línur. Fjöldi nýrra lína sem fallið skrifar út er sem sagt `1 + (n-1)` sem er auðvitað = `n`.

Það ferli þegar fall kallar á sjálft sig er kallað **endurkvæmni** (e. recursion) og þess konar fall er sagt vera **endurkvæmt** (e. recursive).

4.8 Óendanleg endurkvæmni

Þú hefur væntanlega tekið eftir því í dæmunum í síðasta kafla að þegar kallað var á fall endurkvæmt þá minnkaði viðfangið um einn í sérhvert sinn og varð að lokum að núlli. Þegar leppurinn fær að lokum gildið 0 þá hættir fallið keyrslu *án þess að framkvæma fleiri endurkvæm köll*. Þetta tilfelli – þegar fall hættir án þess að framkvæma endurkvæmt kall – er kallað **grunnþrep** (e. base case).

Ef endurkvæmnin kemst aldrei í grunnþrepið þá mun viðkomandi fall halda áfram að framkvæma endurkvæm köll aftur og aftur og fallið mun ekki hættta keyrslu! Þetta er þekkt sem **óendanleg endurkvæmni** (e. infinite recursion) og er almennt séð ekki talið vera góð latína!

Í flestum stýrikerfum mun forrit, sem inniheldur óendanlega endurkvæmni, ekki keyra út í hið óendanlega. Að lokum mun eitthvað gefa eftir með villutilkynningu. Þetta er fyrsta dæmið sem þú sérð um svokallaða **keyrsluvillu** (e. run-time error) þ.e. villu sem kemur ekki í ljós fyrr en forrit er keyrt.

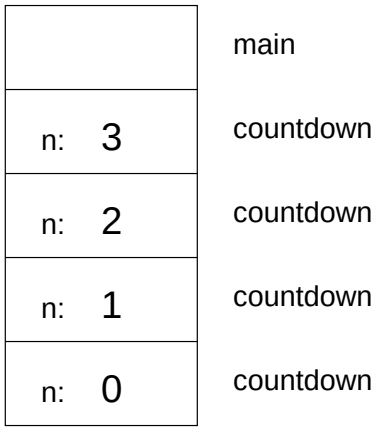
Þú ættir að prófa að skrifa forrit sem inniheldur óendanlega endurkvæmni til að sjá hvað gerist þegar þú keyrir það.

4.9 Staflarit fyrir endurkvæm föll

Í kaflanum á undan notuðum við staflarit til að sýna stöðu forrits á meðan á keyrslu falls átti sér stað. Við getum einnig notað staflarit til að sýna stöðu endurkvæms falls.

Mundu að í hvert sinn sem kallað er á fall þá býr það til nýtt tilvik af kvaðningafærslu sem inniheldur staðværar breytur og leppa.

Þessi mynd sýnir staflarit fyrir countdown með `n = 3`:



Hér er eitt tilvik af `main` og fjögur tilvik af `countdown`, sérhvert þeirra er með mismunandi gildi á leppnum `n`. Á botni staflans er `countdown` með `n=0` en það er einmitt grunnþrepið. Það tilvik framkvæmir ekki endurkvæmt kall og því eru ekki fleiri tilvik af `countdown` á staflanum.

Tilvikið af `main` er tómt vegna þess að `main` hefur hvorki leppa né staðværar breytur. Þú ættir að prófa að teikna staflarit fyrir `nLines` með leppinn `n=4`.

4.10 Orðalisti

modulus: Virki, sem beitt er á tvær heiltölur, og skilar afganginum þegar einni tölu er deilt með annarri. Í C++ er þessi virki táknaður með prósentumerkinu (`%`).

skilyrðissetningar (e. conditional): Blokk af setningum hvers keyrsla er háð útkomu úr tilteknu skilyrði.

tenging skilyrða (e. chaining): Leið til að tengja saman í röð nokkrar skilyrðissetningar.

hreiðrun (e. nesting): Að setja nokkrar skilyrðissetningar innan í eina eða fleiri kvíslir á annarri skilyrðissetningu.

endurkvæmni (e. recursion): Það ferli að kalla á sama fall og nú er þegar verið að keyra.

óendanleg endurkvæmni (e. infinite recursion): Gerist þegar fall, sem kallar á sjálft sig, kemst aldrei í grunnþrepið. Að lokum mun óendanleg endurkvæmni hafa keyrsluvillu í för með sér.

Kafli 5

Föll sem skila gildi

5.1 Skilagildi

Sum af þeim föllum sem við höfum notað, eins og t.d. stærðfræðiföllin, skila af sér einhverju gildi. Tilgangurinn með því að kalla á þess konar fall er að búa til nýtt gildi sem er síðan notað til að gefa breytu gildi eða notað sem hluti af segð. Dæmi:

```
double e = exp (1.0);
double height = radius * sin (angle);
```

Hins vegar vill svo til að öll þau föll sem við höfum sjálf skrifað hingað til eru **void** föll, þ.e. föll sem skila ekki neinu gildi. Kall í void fall er yfirleitt gert án nokkurrar gildisveitingar (því ekkert gildi kemur til baka úr fallinu):

```
nLines (3);
countdown (n-1);
```

Í þessum kafla munum við skrifa föll sem skila af sér gildum. Það mætti segja að þessi föll beri ávöxt! Fyrsta dæmið er `area`, sem tekur `double` sem viðfang, og skilar flatarmáli hrings með gefinn radíus:

```
double area (double radius) {
  double pi = acos (-1.0);
  double area = pi * radius * radius;
  return area;
}
```

Það fyrsta sem þú ættir að taka eftir er að byrjun fallaskilgreiningarinnar er öðruvísi. Í stað `void`, sem gefur til kynna void fall, þá sjáum við hér `double` sem gefur til kynna að skilagildið úr þessu falli sé af taginu `double`.

Taktu líka eftir að síðasta línan í fallinu er önnur útgáfa af `return` setningu, return setning sem inniheldur skilagildi. Þessi setning þýðir "hætta strax keyrslu

þessa falls og nota eftirfarandi segð sem skilagildið." Segðin, sem kemur á eftir lykilorðinu return, getur verið eins flókin og verða vill þannig að við gætum hafa skrifað fallið á samþjappaðri hátt:

```
double area (double radius) {
  return acos(-1.0) * radius * radius;
}
```

Á hinn bóginn má segja að **tímabundnar** (e. temporary) breytur eins og `area` geri kembingu (e. debugging) oft auðveldari. Í báðum tilvikum þarf tag segðarinnar í return setningunni að passa við skilatag fallsins. M.ö.o., þegar þú skilgreinir að skilagildið sé af taginu `double` þá "lofar" þú því að fallið muni að endingu skila `double`. Þýðandinn mun kvarta ef þú reynir að skila engri segð eða segð af röngu tagi.

Stundum getur verið hentug að hafa margar return setningar í falli – eina fyrir sérhverja kvísl í skilyrðissetningu:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else {
    return x;
  }
}
```

Aðeins ein af þessum return setningum mun verða keyrð þar sem return setningarnar eru í mismunandi kvíslum. Þrátt fyrir að það sé leyfilegt að hafa fleiri en eina `return` setningu í falli þá skaltu muna að um leið og ein þeirra er keyrð þá hættir fallið keyrslu og mun ekki keyra þær setningar sem á eftir koma.

Kóði sem kemur á eftir `return` setningu, eða á stað sem flæðið mun ekki komast í, er kallaður **dauður kóði** (e. dead code). Sumir þýðendur gefa einmitt aðvaranir ef hluti kóða er "dauður".

Ef þú setur return setningu innan í skilyrðissetningu þá þarftu að sjá til þess að *sérhver möguleg leið* gegnum forritið lendi að lokum á return setningu. Dæmi:

```
double absoluteValue (double x) {
  if (x < 0) {
    return -x;
  } else if (x > 0) {
    return x;
  }                            // WRONG!!
}
```

Þetta forrit er ekki rétt vegna þess að ef `x` er 0 þá er hvorugt skilyrðanna satt og fallið mun þá hætta keyrslu án þess að framkvæma return setningu. Því miður þá grípa ekki allir C++ þýðendur þessa villu. Því má vera að forritið þýðist og

keyrist en þegar `x==0` þá getur skilagildið í raun verið hvað sem er og líklega mismunandi eftir ólíkum keyrsluumhverfum.

Núna ert þú líklega orðin(n) hundleið(ur) á þýðandavillum en eftir því sem reynslan eykst þá áttar þú þig á því að það eina sem er verra en að fá þýðandavillu er að fá *ekki* þýðandavillu þegar forritið er ekki rétt!

Hér er dæmi um eitthvað sem gæti gerst: Þú prófar `absoluteValue` með ýmsum mismunandi gildum á `x` og það virðist virka rétt. Þú lætur síðan einhvern annan fá forritið og viðkomandi prófar það í öðru umhverfi. Á einhvern dularfullan hátt skilar það ekki réttu gildi og eftir nokkra daga kembingu kemstu að því að útfærslan á `absoluteValue` er ekki rétt. Bara ef þýðandinn hefði aðvarað þig!

Framvegis skaltu ekki álasa þýðandanum ef hann bendir á villu í forritinu þínu. Þú skalt frekar þakka honum fyrir að finna villu og spara þér nokkra daga vinnu við aflúsun. Sumir þýðendur hafa valkost sem segir þeim að vera sérstaklega "strangur" og greina frá öllum villum sem þeir finna. Þú ættir alltaf að velja þennan valkost í þínum þýðanda.

Sem innskot þá bendi ég á að það er fall í math safninu sem heitir `fabs`. Það reiknar tölugildið á `double` – á réttan hátt.

5.2 Þróunarferli

Á þessum tímapunkti ættir þú að geta skoðað C++ föll í heild sinni og sagt til um hvað þau gera. Aftur á móti er þér kannski ekki ljóst hvernig eigi að skrifa þau. Ég mun núna stinga upp á einni aðferð við þróun forrits sem ég kalla **stigvaxandi þróun** (e. incremental development).

Gefum okkur t.d. að þú þurfir að skrifa fall sem reiknar fjarlægðina á milli tveggja punkta sem gefnir eru með hnitunum (x_1, y_1) og (x_2, y_2). Hefðbundna skilgreiningin á fjarlægð á milli tveggja punkta er:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{5.1}$$

Fyrsta skrefið er að íhuga hvernig `distance` fall ætti að líta út í C++. M.ö.o., hvert er inntakið (leppar/viðföng) og hvert er úttakið (skilagildið).

Í þessu tilviki eru punktarnir tveir auðvitað viðföngin inn í fallið og það er eðlilegt að nota fjóra leppa af taginu `double`. Skilagildið er fjarlægð sem er líka eðlilegt að beri tagið `double`.

Nú getum við því skrifað drög að fallinu:

```
double distance (double x1, double y1, double x2, double y2) {
  return 0.0;
}
```

`Return` setningin er þarna að forminu til svo að fallið þýðist og keyri þrátt fyrir að setningin skili ekki réttu gildi. Á þessu stigi gerir fallið í raun ekkert gagnlegt en það er þess virði að reyna að þýða það strax svo við getum fundið málskipunarvillur áður en fallið verður flóknara.

Til að prófa nýja fallið verðum við að kalla á það með einhverjum tilraunagildum (viðföngum). Einhversstaðar í `main` myndi ég bæta við:

```
  double dist = distance (1.0, 2.0, 4.0, 6.0);
  cout << dist << endl;
```

Ég valdi gildin þannig að lárétta fjarlægðin er 3 og lóðrétta fjarlægðin er 4. Þá ætti niðurstaðan að vera 5 (langhliðin í 3-4-5 þríhyrningi). Það er gagnlegt að vita rétta svarið þegar maður prófar fall!

Þegar við höfum athugað að málskipanin í fallaskilgreiningunni okkar er rétt getum við byrjað á því að bæta við setningum í fallið, einni í einu. Svo þýðum við og keyrum forritið eftir sérhverja (stigvaxandi) breytingu. Ef villa kemur upp þá vitum við nákvæmlega hvar hún er – í síðustu línunni sem við bættum við.

Næsta skref í útreikningnum okkar er að finna gildin á $x_2 - x_1$ og $y_2 - y_1$. Ég mun geyma þessi gildi (milliniðurstöður) í tímabundnum breytum sem ég nefni `dx` og `dy`.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  cout << "dx is " << dx << endl;
  cout << "dy is " << dy << endl;
  return 0.0;
}
```

Ég bætti við úttakssetningum sem skrifa út gildin á tímabundnu breytunum. Eins og ég nefndi að ofan þá veit ég þegar að þessi gildi ættu að vera 3,0 og 4,0.

Ég mun síðan fjarlægja þessar úttakssetningar þegar ég hef lokið við þróun fallsins. Kóði eins og þessi er einsskonar **stoðbúnaður** (e. scaffolding) vegna þess að hann hjálpar til við þróun fallsins en er ekki hluti af endanlegri útgáfu þess.

Stundum er reyndar gott að fjarlægja ekki stoðbúnaðinn algerlega heldur setja hann inn í athugasemdir (e. comments) ef ske kynni að við þyrftum á honum að halda síðar.

Næsta skref í þróun fallsins er þá að setja `dx` og `dy` í annað veldi og leggja niðurstöðurnar saman. Við gætum notað `pow` fallið en það er einfaldara og fljótvirkara að margfalda hvorn liðinn með sjálfum sér.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  cout << "dsquared is " << dsquared;
  return 0.0;
}
```

Hér myndi ég aftur þýða og keyra forritið og athuga milliniðurstöðuna (tímabundna gildið) sem ætti að vera 25,0.

Að lokum notum við `sqrt` fallið til að reikna og skila lokaniðurstöðunni.

```
double distance (double x1, double y1, double x2, double y2) {
  double dx = x2 - x1;
  double dy = y2 - y1;
  double dsquared = dx*dx + dy*dy;
  double result = sqrt (dsquared);
  return result;
}
```

Í `main` ættum við síðan að skrifa út og athuga gildið sem við fáum til baka úr kallinu á fallið distance.

Eftir því sem reynslan þín eykst þá muntu skrifa og kemba fleiri en eina línu í einu. Samt sem áður mun þetta stigvaxandi þróunarferli spara þér mikinn tíma við kembingu síðar meir.

Megin þættir í þessu ferli eru:

- Byrjaðu með forrit sem keyrir og gerðu litlar, stigvaxtandi breytingar. Ef einhver villa kemur upp þá veistu nákvæmlega hvar hún er.
- Notaðu tímabundnar breytur til að geyma milliniðurstöður þannig að þú getir skrifað þær út.
- Þegar forritið er tilbúið þá viltu kannski fjarlægja stoðbúnaðinn eða skipta mörgum setningum út fyrir samsettar segðir (e. compound expressions), en þó aðeins ef það gerir forritið þitt ekki ólæsilegra.

5.3 Samsetning

Þegar þú hefur skilgreint nýtt fall þá getur þú notað það sem hluta af segð og þú getur einnig skilgreint ný föll með því að nota þau föll sem eru þegar til. Hvað ef t.d. einhver gæfi þér tvo punkta, miðju hrings og punkt á hringnum, og spyrði um flatarmál hringsins?

Gefum okkur að miðjan sé geymd í breytunum `xc` og `yc` og punkturinn á hringnum í `xp` og `yp`. Fyrsta skrefið er þá að finna radíus hringsins, þ.e. fjarlægðina á milli punktanna tveggja. Við eigum einmitt fall, `distance`, sem gerir það!

```
  double radius = distance (xc, yc, xp, yp);
```

Næsta skref er síðan að finna flatarmál hrings með þennan radíus og skila því.

```
  double result = area (radius);
  return result;
```

Ef við setjum þennan kóða inn í sér fall þá fáum við:

```
double fred (double xc, double yc, double xp, double yp) {
  double radius = distance (xc, yc, xp, yp);
  double result = area (radius);
  return result;
}
```

Nafnið á þessu falli er `fred` sem er vissulega skrýtið. Í næsta kafla mun ég skýra út af hverju nafnið er undarlegt.

Tímabundnu breyturnar `radius` og `area` eru gagnlegar í þróuninni og í kembun en þegar við erum viss um að fallið virki þá getum skrifað það á samþjappaðri hátt með því að setja saman fallaköllin tvö:

```
double fred (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

5.4 Fjölbinding

Þú hefur kannski tekið eftir því, í kaflanum hér á undan, að föllin `fred` og `area` hafa sama hlutverk – að finna flatarmál hrings – en taka samt mismunandi viðföng. Við sendum radíus sem viðfang í `area` en tvo punkta sem viðföng í `fred`.

Ef tvö föll framkvæma sömu aðgerð þá er eðlilegt að gefa þeim sama nafn. M.ö.o, það væri eðlilegra ef fallið `fred` væri kallað `area`.

Í C++ er leyfilegt að hafa fleiri en eitt fall með sama nafninu svo framarlega sem sérhver útgáfa taki mismunandi viðföng. Þessi eiginleiki er kallaður **fjölbinding** (e. overloading). Við getum því endurskýrt `fred`:

```
double area (double xc, double yc, double xp, double yp) {
  return area (distance (xc, yc, xp, yp));
}
```

Nú lítur þetta út eins og endurkvæmt fall en svo er ekki. Það vill svo til að þessi útgáfa af `area` kallar á hina útgáfuna! Þegar þú kallar á fjölbundið fall þá veit C++ þýðandinn hvaða útgáfu verið er að kalla á með því að skoða viðföngin sem notuð eru. Ef þú skrifar

```
    double x = area (3.0);
```

þá mun C++ þýðandinn leita að falli með nafninu `area` sem tekur `double` sem viðfang og notar því fyrri úgáfuna. Ef þú skrifar

```
    double x = area (1.0, 2.0, 4.0, 6.0);
```

þá notar þýðandinn seinni útgáfuna af `area`.

Margar af innbyggðu C++ aðgerðunum eru fjölbundnar sem þýðir að það eru til mismunandir útgáfur sem taka mismunandi fjölda af viðföngunum eða að tag viðfanganna er mismunandi.

Fjölbinding er gagnlegur eiginleiki en skal samt sem áður nota með varúð. Það getur verið ruglandi ef þú kembir eina útgáfu af falli sem síðan kallar á aðra útgáfu þess.

Þetta minnir mig reyndar á eina grundvallarreglu varðandi kembingu: **Vertu viss um að útgáfan af forritinu sem þú ert að skoða sé útgáfan sem er í raun keyrandi!** Stundum lendir þú í því að gera breytingu eftir breytingu á forritinu þínu en samt sem áður sérðu alltaf sama úttak þegar þú keyrir það. Þetta er merki um það að af einhverjum ástæðum þá ertu ekki að keyra þá útgáfu af forritinu sem þú heldur að þú sért að keyra. Til að vera viss þá getur þú skotið inn úttakssetningu (það skiptir í raun ekki máli hvað þú skrifar út) og þannig gengið úr skugga um að rétta útgáfan af forritinu sé keyrandi.

5.5 Boole gildi

Tögin sem við höfum hingað til séð eru frekar stór. Það eru ansi margar heiltölur til í heiminum og enn fleiri kommutölur. Í samanburði er mengi stafa hins vegar frekar smátt. Það er reyndar eitt C++ tag sem er enn smærra. Það er kallað **boole** og einu leyfilegu gildi þess eru gildin `true` og `false`.

Í síðustu tveimur köflum höfum við notað boole gildi án þess að fjalla sérstaklega um það. Skilyrðið innan í `if` setningu er einmitt boole segð og niðurstaðan af beitingu samanburðarvirkja er boole gildi. Dæmi:

```
if (x == 5) {
  // do something
}
```

Samanburðarvirkinn `==` ber hér saman tvö heiltölugildi og skilar boole gildi.

Gildin `true` og `false` eru lykilorð í C++ og geta verið notuð hvar sem gert er ráð fyrir boole segð. Dæmi:

```
while (true) {
  // loop forever
}
```

Þetta er staðlað sniðmát fyrir lykkju sem ætti að keyra endalaust (eða þangað til að komið er að `return` eða `break` setningu).

5.6 Boole breytur

Fyrir sérhvert tag á gildi er samsvarandi tag á breytu. Í C++ er boole tagið kallað **bool**. Breytur af taginu bool eru skilgreindar á sama hátt og breytur af öðru tagi:

```
bool fred;
fred = true;
bool testResult = false;
```

Fyrsta línan er einföld yfirlýsing á breytu. Önnur línan er gildisveiting og þriðja línan er samsetning á yfirlýsingu og gildisveitingu, þ.e. frumstilling.

Eins og ég nefndi að ofan þá er niðurstaðan úr samanburði alltaf booole gildi þannig að hægt er að geyma hana í `bool` breytu

```
bool evenFlag = (n%2 == 0);     // true if n is even
bool positiveFlag = (x > 0);    // true if x is positive
```

og nota breytuna síðar í skilyrðissetningu

```
if (evenFlag) {
  cout << "n was even when I checked it" << endl;
}
```

Breyta sem notuð er á þennan hátt er oft kölluð **flagg** (e. flag) vegna þess að hún gefur til kynna ("flaggar") að eitthvað skilyrði sé til staðar eður ei.

5.7 Rökvirkjar

Í C++ eru þrír **rökvirkjar** (e. logical operators): AND, OR og NOT, sem eru táknaðir með `&&`, `||` og `!`. Merking þessar virkja er svipuð og merking þeirra í náttúrulegu máli (ensku). T.d. er `x > 0 && x < 10` true (satt) aðeins ef `x` er stærra en 0 OG (AND) minna en 10.

`evenFlag || n%3 == 0` er true ef *annað* skilyrðanna er true, þ.e. ef `evenFlag` er true EÐA (OR) að talan er deilanleg með 3.

Að lokum hefur NOT virkinn þau áhrif að neita eða snúa við gildi boole segðar. Þannig er `!evenFlag` true ef `evenFlag` er false, þ.e. ef talan er oddatala.

Rökvirkjar gefa oft möguleika á því að einfalda hreiðraðar skilyrðissetningar. Hvernig myndir þú t.d. skrifa eftirfarandi kóða með því að nota eitt skilyrði sem samsett er úr rökvirkjum?

```
if (x > 0) {
  if (x < 10) {
    cout << "x is a positive single digit." << endl;
  }
}
```

5.8 Boole föll

Föll geta skilað `bool` gildum eins og hvaða öðru gildi. Það er oft hentugt þegar hylja þarf flókið skilyrði innan í falli: Dæmi:

```
bool isSingleDigit (int x)
{
  if (x >= 0 && x < 10) {
    return true;
```

```
  } else {
    return false;
  }
}
```

Nafnið á þessu falli er `isSingleDigit`. Algengt er að gefa boole föllum nöfn sem hljóma eins og já/nei spurningar. Skilagildið er `bool` sem þýðir að sérhver return setning þarf að hafa segð af taginu `bool` í för með sér.

Kóðinn sjálfur er einfaldur þrátt fyrir að vera aðeins lengri en nauðsynlegt er. Mundu að segðin `x >= 0 && x < 10` hefur tagið `bool` þannig að það er ekkert athugavert við það að skila henni beint og þar með sleppa `if` setningunni:

```
bool isSingleDigit (int x)
{
  return (x >= 0 && x < 10);
}
```

Í `main` getur þú kallað á þetta fall á venjulegan hátt:

```
  cout << isSingleDigit (2) << endl;
  bool bigFlag = !isSingleDigit (17);
```

Fyrsta línan skrifar út gildið `true` vegna þess að talan 2 samanstendur af einum tölustaf. Það vill reyndar svo til að þegar C++ skrifar út `bool` gildi þá eru orðin `true` og `false` ekki skrifuð út heldur frekar heiltölurnar `1` og `0`.

Í annarri línunni fær breytan `bigFlag` gildið `true` vegna þess að talan 17 er *ekki* (not) tala með einum tölustaf.

Algengasta notkun á `bool` föllum er innan í skilyrðissetningu:

```
  if (isSingleDigit (x)) {
    cout << "x is little" << endl;
  } else {
    cout << "x is big" << endl;
  }
```

5.9 Skilagildi úr `main`

Nú þegar við höfum kynnst föllum sem skila gildum þá get ég sagt þér frá litlu leyndarmáli. `main` á í raun ekki að vera `void` fall. Það á nefnilega að skila heiltölu:

```
int main ()
{
  return 0;
}
```

Yfirleitt er skilagildið úr `main` 0 sem gefur til kynna að forritið endaði eðlilega. Ef eitthvað fer úrskeiðis þá er algengt að skila -1 eða einhverju öðru gildi sem gefur til kynna að villa hafi komið upp.

Þá vaknar eðlilega spurningin: Hver tekur við skilagildinu þar sem við köllum aldrei sjálf á `main`? Það vill svo til að þegar keyrsluumhverfið keyrir C++ forrit þá byrjar það á því að kalla á `main` á sambærilegan hátt og kallað er á hvaða annað fall.

Það eru meira að segja ákveðin viðföng sem `main` fær frá keyrsluumhverfinu en við munum fjalla um það síðar.

5.10 Meiri endurkvæmni

Fram að þessu höfum við aðeins lært lítið hlutmengi af C++. Það er hins vegar athyglisvert að þetta hlutmengi er núna **fullkomið** (e. complete) forritunarmál í þeim skilningi að allt það sem hægt er að reikna (e. compute) eða framkvæma með tölvu er hægt að tjá (e. express) í þessu máli. Sérhvert forrit sem hefur verið skrifað eða mun verða skrifað er hægt að endurskrifa með því að nota þá eiginleika C++ sem við höfum skoðað fram að þessu (við þyrftum reyndar nokkrar skipanir í viðbót til að stjórna jaðartækjum eins og lyklaborði, mús, hörðum diski, o.s.frv., en það er allt og sumt).

Að sanna þessa staðhæfingu er reyndar ekki einfalt en það gerði Alan Turing, einn fyrsti tölvunarfræðingurinn[1], fyrstur manna. Í kjölfarið nefnist þessi staðhæfing "Turing thesis". Ef þú tekur námskeið í reiknanleika (e. Theory of Computation) þá færðu tækifæri til að sjá sönnunina.

Til að gefa þér hugmynd um hvað þú getur gert með þeim tólum sem við höfum lært hingað til þá ætla ég að útfæra nokkur stærðfræðiföll sem skilgreind eru endurkvæmin hátt. Endurkvæm skilgreining svipar til hringskilgreiningar í þeim skilningi að skilgreiningin inniheldur tilvísun í það sem verið er að skilgreina! Skilgreining sem er eingöngu hringskilgreining er yfirleitt ekki mjög gagnleg:

skeflulegur: lýsingarorð sem notað til að lýsa einhverju sem er skeflulegt.

Ef þú sæir þessa skilgreiningu í orðabók þá yrðir þú væntanlega pirraður. Á hinn bóginn, ef þú flettir upp skilgreiningu á stærðfræðifallinu **hrópmerkt** (e. factorial) þá sæir þú eitthvað þessu líkt:

$$0! = 1$$
$$n! = n \cdot (n-1)!$$

(Hrópmerkt er yfirleitt táknað með ! sem ekki má rugla saman við C++ rökvirkjann `!` sem merkir NOT.) Þessi skilgreining segir að 0 hrópmerkt er 1

[1] Sumir myndu nú reyndar segja að Alan Turing hefði verið stærðfræðingur en margir af fyrstu tölvunarfræðingunum voru það nú reyndar.

og að hrópmerkt af hvaða öðru gildi, n, er n margfaldað með $n-1$ hrópmerkt. 3! er þá 3 margfaldað með 2!, sem er 2 margfaldað með 1!, sem er 1 margfaldað með 0!. Ef við tökum þetta allt saman þá fáum við 3! = 3 * 2 * 1 * 1, sem er 6.

Ef þú getur skrifað endurkvæma skilgreiningu fyrir eitthvað vandamál þá getur þú yfirleitt skrifað C++ forrit sem leysir það. Fyrst þarf að ákveða hver viðföngin eru í fallið og hvert skilagildið er. Þú ættir að sjá að fallið factorial tekur heiltölu sem viðfang og skilar af sér heiltölu:

```
int factorial (int n)
{
}
```

Ef leppurinn hefur gildið 0 þá þurfum við einfaldlega að skila 1 (þetta er grunnþrepið):

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  }
}
```

Annars framkvæmum við endurkvæmt kall til að finna $n-1$ hrópmerkt og margföldum það gildi með n.

```
int factorial (int n)
{
  if (n == 0) {
    return 1;
  } else {
    int recurse = factorial (n-1);
    int result = n * recurse;
    return result;
  }
}
```

Ef við skoðum keyrsluflæðið fyrir þetta fall þá sjáum við að það svipar til fallsins `nLines` úr síðasta kafla. Hvað gerist ef við köllum á `factorial` með gildinu 3:

Þar sem 3 er ekki núll þá keyrist `else` kvíslin sem reiknar $n-1$ hrópmerkt ...

> Þar sem 2 er ekki núll þá keyrist `else` kvíslin sem reiknar $n-1$ hrópmerkt ...
>
> > Þar sem 1 er ekki núll þá keyrist `else` kvíslin sem reiknar $n-1$ hrópmerkt ...

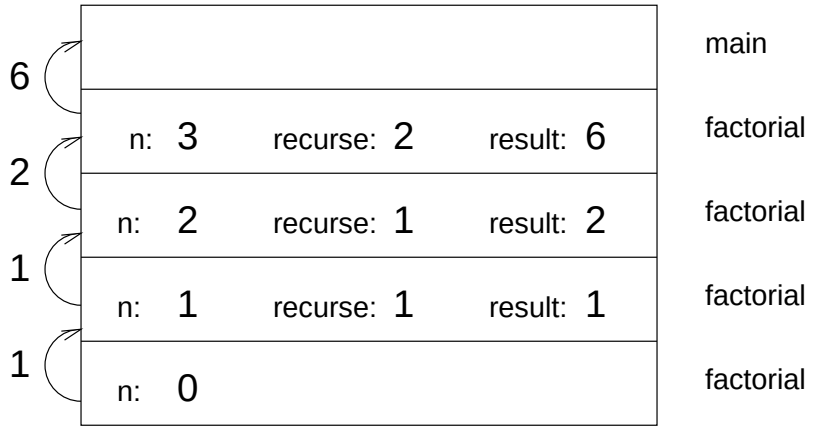
Þar sem 0 *er* núll þá keyrist fyrsta kvíslin og gildinu 1 er skilað án þess að fleiri endurkvæm köll séu framkvæmd.

Skilagildið 1 er margfaldað með `n` sem er 1 og niðurstöðunni skilað.

Skilagildið 1 er margfaldað með `n` sem er 2 og niðurstöðunni skilað.

Skilagildið 2 er margfaldað með `n` sem er 3 og niðurstaðan 6 er skilað til `main`, eða til þess sem kallaði á `factorial (3)`.

Hér sjáum við staflarit fyrir röð fallakallanna:



Við sjáum að skilagildin eru send upp staflann.

Taktu eftir að í síðasta tilvikinu á `factorial` (neðst á staflanum) eru staðværu breyturnar `recurse` og `result` ekki til því þegar `n=0` þá er kvíslin (blokkin) sem býr þau til ekki keyrð.

5.11 Að taka trúanlegt

Ein leið til að lesa forrit er að fylgja keyrsluflæði þess en eins og þú sást í kaflanum hér á undan þá getur það stundum verið eins og að finna leið út úr völundarhúsi. Önnur leið er það sem ég kalla "að taka trúanlegt" (e. leap of faith). Þegar þú kemur að fallakalli þá, í stað þess að fylgja flæðinu inn í fallið, *gerir þú ráð fyrir* að fallið virki rétt og skili réttu gildi.

Reyndar vill svo til að þú tekur virkni þegar trúanlega þegar þú notar innbyggð föll. Þegar þú kallar á `cos` eða `exp` fallið þá skoðar þú ekki útfærsluna á þeim. Þú hreinlega gerir ráð fyrir að þau virki vegna þess að fólkið sem skrifaði þessi innbyggðu forritunarsöfn eru góðir forritarar.

Sama gildir um þín eigin föll. Í kafla 5.8 skrifuðum við fallið `isSingleDigit` sem ákvarðar hvort tala er á milli 0 og 9. Þegar við höfðum fullvissað okkur um að þetta fall væri rétt – með prófunum og með því að skoða kóðann – þá gátum við notað fallið án þess að þurfa að skoða kóðann á ný.

Sama gildir um endurkvæm föll. Þegar komið er að endurkvæma kallinu þá ættir þú, í stað þess að fylgja keyrsluflæðinu, *að gera ráð fyrir* að endurkvæma kallið virki (skili réttu gildi) og síðan spyrja sjálfan þig: "Að því gefnu að ég geti reiknað $n-1$ hrópmerkt, get ég reiknað n hrópmerkt?" Í þessu tilviki er ljóst að það er hægt með því að margfalda með n.

Það er auðvitað dálítið skrýtið að gera ráð fyrir að fall virki rétt þegar maður hefur ekki einu sinni klárað að skrifa það en þetta er einmitt ástæðan fyrir því að ég segi "taktu það trúanlegt"!

5.12 Eitt dæmi í viðbót

Í síðasta dæmi notaði ég tímabundnar breytur í sérhverju skrefi til að gera kóðann auðveldari í kembingu en ég gæti hafa sparað nokkrar línur:

```
int factorial (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial (n-1);
  }
}
```

Héðan í frá mun ég nota samþjappaðri útgáfur af föllum en ég mæli með að þú notir "skilmerkilegri" (e. explicit) útgáfur á meðan þú ert að þróa fallið þitt. Þegar fallið er tilbúið og þú ert viss um að það virki rétt þá getur þú gert kóðann samþjappaðri.

Annað klassískt dæmi um stærðfræðifall sem skilgreint er á endurkvæman hátt er `fibonacci` – skilgreint á eftirfarandi hátt:

$$
\begin{aligned}
&fibonacci(0) = 1 \\
&fibonacci(1) = 1 \\
&fibonacci(n) = fibonacci(n-1) + fibonacci(n-2);
\end{aligned}
$$

Ef við þýðum þetta yfir á C++ þá fáum við

```
int fibonacci (int n) {
  if (n == 0 || n == 1) {
    return 1;
  } else {
    return fibonacci (n-1) + fibonacci (n-2);
  }
}
```

Ef þú reynir að fylgja keyrsluflæðinu í þessu falli, meira að segja fyrir lág gildi á `n`, þá mun rjúka úr höfðinu á þér! En ef við fylgjum "taktu það trúanlegt"

og gerum ráð fyrir að endurkvæmu köllin tvö (já, þú getur framkvæmt tvö endurkvæm köll!) virki rétt þá er ljóst að við fáum rétta niðurstöðu þegar við leggjum saman gildin úr köllunum tveimur.

5.13 Orðalisti

skilatag (e. return type): Tagið á gildinu sem fallið skilar.

skilagildi (e. return value): Gildið sem fallið skilar.

dauður kóði (e. dead code): Hluti forrits sem er aldrei keyrður, oft vegna þess að hann fylgir á eftir `return` setningu.

stoðbúnaður (e. scaffolding): Kóði sem er notaður í þróunarferlinu en er ekki hluti endanlegrar útgáfu.

void: Sérstakt skilatag sem gefur til kynna void fall, þ.e. fall sem skilar ekki neinu gildi.

fjölbinding (e. overloading): Það að hafa fleiri en eitt fall með sama nafni en með mismunandi leppa. C++ þekkir hvaða útgáfu falls er verið að kalla á með því að skoða viðföngin.

boole (e. boolean): Gildi eða breyta sem getur einungis haft tvær mögulegar stöður, *true* eða *false*. Í C++ eru boole gildi geymd í breytum af taginu `bool`.

flagg (e. flag): Breyta (af taginu `bool`) sem geymir útkomu úr tilteknu skilyrði.

samanburðarvirki (e. comparison operator): Virki sem ber saman tvö gildi og skilar boole gildi sem gefur til kynna samband á milli þolandanna.

rökvirki (e. logical operator): Virki sem sameinar boole gildi til að prófa samsett skilyrði.

Kafli 6

Ítrun

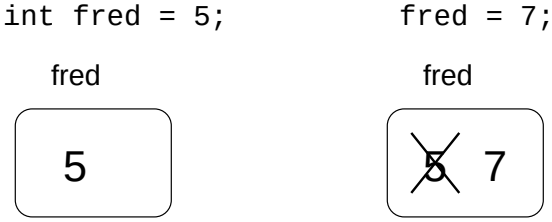
6.1 Fjölgildisveiting

Ég hef ekki nefnt það áður að í C++ er leyfilegt að gefa breytu gildi oftar en einu sinni. Tilgangur gildisveitingar nr. 2 er að skipta út gömlu gildi tiltekinnar breytu fyrir nýtt gildi.

```
int fred = 5;
cout << fred;
fred = 7;
cout << fred;
```

Úttakið úr þessu forriti er `57` vegna þess að í fyrra skiptið sem við skrifum út `fred` þá er gildi breytunnar 5 en í síðari skiptið er gildið 7.

Þessi tegund af **fjölgildisveitingu** (e. multiple assignment) er ástæðan fyrir því að ég lýsti breytum sem einskonar *gámi* (e. container) fyrir gildi. Þegar þú gefur breytu gildi þá breytir þú innihaldi gámsins eins og sést á eftirfarandi mynd:



Þegar um fjölgildisveitingu er að ræða þá er sérstaklega mikilvægt að gera greinarmun á gildisveitingu og samanburði. C++ notar `=` táknið fyrir gildisveitingu en það er freistandi að túlka setningu eins og `a = b` sem samanburð (jöfnuð). Svo er hins vegar ekki!

Í fyrsta lagi þá er samanburður víxlin (e. commutative) en gildisveiting er það ekki. Í stærðfræði ef t.d. $a = 7$ þá er $7 = a$. En í C++ þá er setningin `a =`

`7;` lögleg en `7 = a;` aftur á móti ekki. Af hverju ekki?

Jafnframt gildir í stærðfræði að setning um jöfnuð (e. statement of equality) er sönn án tillits til tímasetningar. Ef $a = b$ núna, þá verður a alltaf jafnt b. Í C++ getur gildisveiting gert tvær breytur jafnar en þær þurfa hins vegar ekki alltaf að vera jafnar eftir það!

```
int a = 5;
int b = a;      // a and b are now equal
a = 3;          // a and b are no longer equal
```

Þriðja línan breytir gildinu á `a` en breytir hins vegar ekki gildinu á `b` og því eru breyturnar tvær ekki lengur jafnar. Í mörgum forritunarmálum er annað tákn notað fyrir gildisveitingu, t.d. `<-` eða `:=`, til að koma í veg fyrir rugling.

Þrátt fyrir að fjölgildisveiting sé oft gagnleg þá skaltu nota hana með varúð. Ef gildi breytna eru stöðugt að breytast á mismunandi stöðum í forritunu þínu þá getur það orðið ólæsilegra og erfiðara að kemba.

6.2 Ítrun

Eitt af því sem tölvur eru oft notaðar í er að sjálfvirknivæða einhæf verkefni. Tölvur er, ólíkt fólki, góðar í því að endurtaka sama eða svipað verkefni án þess að gera villur.

Við höfum hingað til séð forrit sem beitir endurkvæmni til að framkvæma endurtekningar, t.d. `nLines` og `countdown`. Þessi tegund af endurtekningum er kölluð **ítrun** (e. iteration) og C++ hefur ýmsa innbyggða eiginleika sem gera okkur auðveldara að skrifa forrit sem notar ítranir.

Við ætlum nú að skoða einn af þessum eiginleikum: `while` setningu.

6.3 `While` setning

Með því að nota `while` setningu getum við endurskrifað `countdown` fallið:

```
int countdown (int n) {
  while (n > 0) {
    cout << n << endl;
    n = n-1;
  }
  cout << "Blastoff!" << endl;
  return 0;
}
```

Þú getur nánast lesið `while` setningu eins og um enskan texta væri að ræða. Merkingin hér er: "While `n` is greater than zero, continue displaying the value of `n` and then reducing the value of `n` by 1. When you get to zero, output the word 'Blastoff!"'

Við getum lýst keyrsluflæðinu í `while` setningu á formlegri hátt:

1. Gildið á skilyrðinu innan í svigunum er ákvarðað. Útkoman er annaðhvort `true` eða `false`.

2. Ef skilyrðið er ósatt (false) þá er strax hætt í `while` setningunni og haldið áfram keyrslu í næstu setningu á eftir.

3. Ef skilyrðið er satt (true) þá er sérhver setning í blokkinni sem afmarkast af slaufusvigunum keyrð og síðan farið til baka í skref 1.

Þessi tegund af flæði er kölluð **lykkja** (e. loop) vegna þess að þriðja skrefið stekkur til baka í fyrsta skrefið. Athugaðu að ef skilyrðið er ósatt (false) strax í upphafi þá eru setningarnar innan lykkjunnar aldrei keyrðar. Setningarnar innan í lykkju eru kallaðar **meginmál** (e. body) lykkjunnar.

Það er mikilvægt að meginmál lykkjunnar breyti gildi einnar eða fleiri breytna þannig að skilyrðið verði false á einhverjum tímapunkti og lykkjan hætti þá keyrslu. Að öðrum kosti mun lykkjan halda áfram sínum ítrunum (að eilífu!) og í því tilviki er um **óendanlega lykkju** (e. infinite loop) að ræða.

Í tilviki `countdown` getum við sannað að lykkjan muni klárast. Við vitum að gildið á `n` minnkar um 1 í hverri **ítrun** lykkjunnar þannig að `n` fær gildið 0 að lokum. Í öðrum tilfellum er kannski ekki svo auðvelt að segja til um þetta:

```
void sequence (int n) {
  while (n != 1) {
    cout << n << endl;
    if (n%2 == 0) {           // n is even
      n = n / 2;
    } else {                  // n is odd
      n = n*3 + 1;
    }
  }
}
```

Skilyrði fyrir áframhaldi þessarar lykkju er `n != 1` og hún heldur því áfram þangað til `n` er 1 (sem gerir skilyrðin false). Í sérhverri ítrun skrifar fallið út gildið á `n` og athugar síðan hvort það er slétt tala eða oddatala. Ef `n` er slétt þá er deilt í gildið með tveimur. Ef það er oddatala þá er gildinu skipt út fyrir $3n + 1$. Ef upphafsgildið (þ.e. gildi viðfangsins sem sent er í `sequence`) á `n` er t.d. 3 þá skrifast út eftirfarandi röð: 3, 10, 5, 16, 8, 4, 2, 1.

Þar sem `n` hækkar stundum eða lækkar þá er ekki auðvelt að sanna að `n` fái að lokum gildið 1 og þar með að lykkjan klárist að lokum. Fyrir tiltekin gildi á `n` getum við sannað að lykkjan klárist. Ef upphafsgildið á `n` er t.d. veldi af tveimur þá mun gildið á `n` vera slétt tala í sérhverri ítrun lykkjunnar þangað til það verður að lokum 1. Dæmið að ofan endar með þannig röð, þ.e. röðinni sem byrjar á 16.

Það er hins vegar athyglisvert vandamál að reyna að sanna að lykkjan hætti keyrslu fyrir *öll* gildi á `n`. Hingað til hefur engum tekist að sanna eða afsanna það!

6.4 Töflur

Eitt af því sem hentugt er að nota lykkur í er að búa til töflugögn. Fyrir tíma tölvunnar þurfti fólk að reikna lógaríþma, sínus og cósínus, og önnur stærðfræðiföll, í höndunum. Til að gera þetta auðveldara þá voru til bækur sem innhéldu langar töflur þar sem hægt var að fletta upp gildum fyrir ýmis föll. Það var tímafrekt og leiðinlegt að búa þessar töflur til og sumar þeirra innihéldu fullt af villum.

Ein fyrstu viðbrögðin við því þegar tölvur komu á sjónarsviðið voru: "Þetta er frábært! Við getum notað tölvurnar til að búa til töflur sem innihalda engar villur." Það reyndist vera (næstum því) rétt en skammsýnt. Brátt urðu nefnilega tölvur og reiknivélar svo útbreiddar að það var engin þörf á þessum töflum lengur.

Jæja, næstum því. Það vill reyndar svo til að fyrir tilteknar aðgerðir nota tölvur töflur til að nálga svörin og nota síðan útreikninga til að bæta nálganirnar. Í sumum tilvikum hafa verið villur í undirliggjandi töflum – sú þekktasta í töflunni sem fyrsta útgáfa af Intel Pentium örgjörvanum notaði til að framkvæma kommutöludeilingu.

Þrátt fyrir að "log tafla" sé ekki eins gagnleg og áður fyrr þá má nota hana sem gott dæmi um ítrun. Eftirfarandi forrit skrifar út röð gilda í vinstri dálki og samsvarandi lógaríþma í hægri dálki:

```
double x = 1.0;
while (x < 10.0) {
  cout << x << "\t" << log(x) << "\n";
  x = x + 1.0;
}
```

Stafarunan `\t` stendur fyrir **dálkastafinn** (e. tab character). Runan `\n` stendur fyrir **newline** (ný lína) stafinn. Báðar þessar runur er hægt að setja hvar sem er inn í streng en í þessu dæmi mynda runurnar heilan streng án nokkurra annara stafa.

Dálkastafurinn veldur því að bendillinn "stekkur" til hægri þangað til hann lendir á einum af **dálkastoppum** (e. tab stops) sem erum yfirleitt eftir hverja átta stafi.

Eins og við sjáum hér rétt á eftir þá er dálkastafurinn hentugur til að skrifa út texta sem passar í dálka.

Stafurinn **newline** virkar á nákvæmlega sama hátt og `endl`, þ.e. veldur því að bendillinn hoppar í næstu línu fyrir neðan. Ég nota yfirleitt `endl` ef newline stafurinn stendur einn og sér en annars nota ég `\n` ef hann er hluti strengs.

Úttak þessa forrits er:

```
1       0
2       0.693147
3       1.09861
4       1.38629
5       1.60944
```

```
6       1.79176
7       1.94591
8       2.07944
9       2.19722
```

Ef þér finnst þessi gildi vera undarleg, mundu þá að `log` fallið notar grunn e. Vegna þess hversu veldi af tveimur eru mikilvæg í tölvunarfræði þá viljum við oft reikna út lógariþma með grunn 2. Það getum við gert með því að nota eftirfarandi formúlu:

$$\log_2 x = \frac{log_e x}{log_e 2}$$

Með því að breyta úttakssetningunni í

```
cout << x << "\t" << log(x) / log(2.0) << endl;
```

þá fáum við:

```
1       0
2       1
3       1.58496
4       2
5       2.32193
6       2.58496
7       2.80735
8       3
9       3.16993
```

Hér sjáum við að 1, 2, 4 og 8 eru veldi af tveimur vegna þess að lógariþminn með grunn 2 er heil tala. Ef við vildum finna lógariþma af öðrum tölum af veldinu tveimur þá gætum við breytt forritinu á þennan hátt:

```
double x = 1.0;
while (x < 100.0) {
  cout << x << "\t" << log(x) / log(2.0) << endl;
  x = x * 2.0;
}
```

Í stað þess að bæta einhverju við `x` í sérhverri ítrun lykkjunnar, og fá þannig **jafnmunarunu** (e. arithmetic sequence), þá margföldum við `x` með einhverju og fáum út **kvótarunu** (e. geometric sequence). Niðurstaðan er:

```
1       0
2       1
4       2
8       3
16      4
32      5
64      6
```

Við notum dálkastafinn á milli dálka og því er staðsetning seinni dálksins ekki háð fjölda tölustafa í fyrri dálkinum.

Það má vera að log töflur séu ekki gagnlegar nú til dags en hins vegar er mikilvægt fyrir tölvunarfræðinga að þekkja veldi af tveimur! Prófaðu að breyta forritinu þannig að það skrifi út veldi af tveimur upp í 65536 (þ.e. 2^{16}). Prentaðu það út og mundu það!

6.5 Tvívíðar töflur

Í tvívíðri töflu (e. two-dimensional table) getur þú valið röð og dálk og lesið gildið þar sem röðin og dálkurinn mætast. Gott dæmi um tvívíða töflu er margföldunartafla. Segjum að þig langi til að prenta út margföldunartöflu fyrir gildin 1 til 6.

Þú gætir byrjað með því að skrifa einfalda lykkju sem prentar út margfeldi af 2, öll í sömu línu:

```
int i = 1;
while (i <= 6) {
  cout << 2*i << "   ";
  i = i + 1;
}
cout << endl;
```

Fyrsta línan frumstillir breytuna `i` en hún er hér notuð sem teljari eða **lykkjubreyta** (e. loop variable). Þegar lykkjan keyrir þá hækkar gildið á `i` úr 1 í 6 og lykkjan hættir keyrslu þegar `i` er 7. Í sérhverri ítrun lykkjunnar prentum við út gildið `2*i` ásamt þremur bilum. Við skrifum allt úttakið í einni og sömu línunni því við sleppum `endl` úr fyrstu úttakssetningunni.

Úttak þessa forrits er:

```
2   4   6   8   10   12
```

Næsta skref er að **hjúpa** (e. encapsulate) og **alhæfa** (e. generalize).

6.6 Hjúpun og alhæfing

Hjúpun (e. encapsulation) merkir yfirleitt að taka hluta af kóða og "pakka" (e. wrap) honum inn í fall og þannig nýta kostina sem almennt fylgja föllum. Við höfum séð tvö dæmi um hjúpun, þ.e. þegar við skrifuðum `printParity` í kafla 4.3 og `isSingleDigit` í kafla 5.8.

Alhæfing merkir að taka eitthvað einstakt tilfelli, eins og að prenta út margfeldi af tveimur, og gera það almennara, eins og að prenta út margfeldi af hvaða heiltölu sem er.

Hér er fall sem hjúpar lykkjuna að ofan og gerir hana almennari þannig að hún prenti út margeldi af `n`.

```
void printMultiples (int n)
{
  int i = 1;
  while (i <= 6) {
    cout << n*i << "   ";
    i = i + 1;
  }
  cout << endl;
}
```

Eina sem ég þurfti að gera til að hjúpa var að bæta fyrstu línunni við, þ.e. að tilgreina nafn fallsins, lepp og skilagildi. Til að alhæfa þurfti ég aðeins að skipta út gildinu 2 fyrir leppinn `n`.

Ef við köllum á þetta fall með viðfanginu 2 þá fáum við sama úttak og áður. Ef við köllum með viðfanginu 3 fæst úttakið:

```
3   6   9   12   15   18
```

og með viðfanginu 4 fæst úttakið:

```
4   8   12   16   20   24
```

Á þessum tímapunkti ættir þú að sjá hvernig við förum að því að prenta út margföldunartöflu: Við munum kalla endurtekið á `printMultiples` með mismunandi viðfögnum. Reyndar vill svo til að við munum nota aðra lykkjum til að ítra yfir raðirnar:

```
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i = i + 1;
  }
```

Taktu eftir hversu svipuð þessi lykkja er þeirri inni í fallinu `printMultiples`. Eina sem ég gerði var að skipta út úttakssetningunni fyrir fallakall.

Úttakið úr þessu forriti er

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
```

sem er (örlítið bjöguð) margföldunartafla. Ef þessi bjögun angrar þig þá ættir þú að prófa að skipta út bilum fyrir dálkastafinn og athuga hvernig taflan lítur út eftir þá breytingu.

6.7 Föll

Í síðasta kafla nefndi ég "og þannig nýta kostina sem almennt fylgja föllum". Á þessum tímapunkti veltir þú kannski fyrir þér hvaða kostir þetta eru. Hér er nokkrar ástæður fyrir gagnsemi falla:

- Með því að gefa röð setninga tiltekið nafn þá gerir þú forritið þitt bæði læsilegra og auðveldara að kemba.
- Með því að brjóta langt forrit upp í föll þá skiptir þú því í mismunandi hluta, getur unnið í einstökum hlutum þess án tillits til annarra hluta, og síðan sett þessa hluta saman til að mynda eina heild.
- Föll styðja bæði við endurkvæmni og ítrun.
- Vel hönnuð föll eru oft gagnleg fyrir mörg forrit. Þegar þú hefur skrifað fall þá getur þú oft endurnýtt það í öðrum forritum.

6.8 Meira um hjúpun

Í þeim tilgangi að sýna hjúpun aftur þá ætla ég núna að taka kóðann úr síðasta kafla og pakka honum inn í fallið `printMultTable`:

```
void printMultTable () {
  int i = 1;
  while (i <= 6) {
    printMultiples (i);
    i = i + 1;
  }
}
```

Þetta ferli sem ég hef sýnt þér er algengt þróunarferli í forritun. Þú þróar forrit smám saman með því að bæta setningu við `main` eða á einhverjum öðrum stað, og þegar forritið virkar þá dregur þú kóðann út og pakkar honum inn í fall.

Ástæðan fyrir því að þetta er gagnlegt er að stundum veistu ekki í upphafi hvernig best er að skipta forritinu þínu upp í einstök föll. Þessi aðferð gerir þér kleift að hanna um leið og þú skrifar forritið.

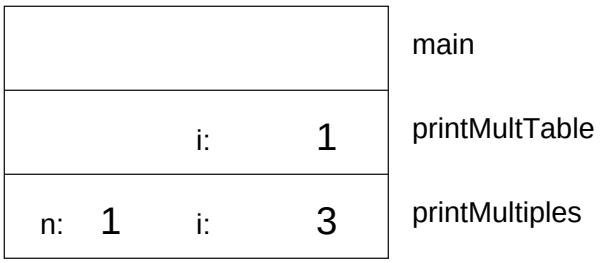
6.9 Staðværar breytur

Á þessum tímapunkti gætir þú velt því fyrir þér hvernig standi á því að hægt sé að nota sömu breytuna `i` í bæði `printMultiples` og `printMultTable`. Töluðum við ekki um að eingöngu er hægt að lýsa yfir breytu einu sinni? Og veldur það ekki vandræðum ef eitt fall breytir gildinu á breytunni?

Svarið við báðum þessum spurningum er "nei". Ástæðan er sú að `i` í `printMultiples` og `i` í `printMultTable` eru *ekki sama breytan*. Þær bera vissulega sama nafnið en minnissvæðið sem þær standa fyrir er ekki það sama og þess vegna hefur það ekki áhrif á aðra breytuna að breyta gildi hinnar.

Mundu að breyta sem er lýst yfir í falli er staðvær (e. local). Það er ekki hægt að nálgast gildi staðværrar breytu utan frá (þ.e. utan fallsins sem skilgreinir breytuna) og því getur þú haft margar breytur með sama nafni svo framarlega sem þær eru ekki skilgreindar í sama fallinu.

Staflaritið fyrir þetta forrit sýnir glögglega að breyturnar tvær með nafnið `i` standa ekki fyrir sama minnissvæði. Gildi þeirra getur verið mismunandi og breyting á annarri breytunni hefur engin áhrif á hina breytuna.



Taktu eftir að gildið á leppnum `n` í `printMultiples` er það sama og gildið á `i` í `printMultTable`. Á hinn bóginn hleypur gildið á `i` í `printMultiple` úr 1 upp í `n`. Á myndinni þá vill svo til að þetta gildi er 3 en í næstu ítrun lykkjunnar verður það 4.

Til að minnka líkur á ruglingi þá er yfirleitt góð forritunarvenja að nota mismunandi nöfn á breytum í mismunandi föllum. Aftur á móti eru líka ástæður fyrir því að nota sömu breytunöfn. Það er t.d. algengt að nota nöfnin `i`, `j` og `k` fyrir lykkjubreytur. Ef þú forðast að nota þau í einu falli vegna þess að þú notar þau í öðru falli þá verður forritið þitt líklega bara ólæsilegra.

6.10 Meira um alhæfingu

Tökum annað dæmi um alhæfingu. Gerum ráð fyrir að þú viljir skrifa forrit sem prentar út margföldunartöflu af hvaða stærð sem er, þ.e. ekki bara 6x6 töflu. Þú gætir bætt leppi við `printMultTable`:

```
void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i);
    i = i + 1;
  }
}
```

Hér hef ég skipt út gildinu 6 með leppnum `high`. Með því að kalla á `printMultTable` með viðfanginu 7 þá fæst:

```
1   2   3   4   5   6
2   4   6   8   10   12
3   6   9   12   15   18
4   8   12   16   20   24
5   10   15   20   25   30
6   12   18   24   30   36
7   14   21   28   35   42
```

Þetta er ágætt en gott væri ef taflan gæti sýnt jafnmargar raðir og dálka. Ég þarf því að bæta öðrum lepp við `printMultiples` sem tilgreinir þann fjölda dálka sem ég vil sjá.

Svona til að pirra þig þá ætla ég líka að kalla þennan lepp `high` og þar með sýna að mismunandi föll geta haft leppa (eins og staðværar breytur) með sama nafni:

```
void printMultiples (int n, int high) {
  int i = 1;
  while (i <= high) {
    cout << n*i << "   ";
    i = i + 1;
  }
  cout << endl;
}

void printMultTable (int high) {
  int i = 1;
  while (i <= high) {
    printMultiples (i, high);
    i = i + 1;
  }
}

int main() {
  printMultTable(7);
}
```

Taktu eftir því að þegar ég bætti nýjum leppi við þá þurfti ég að breyta fyrstu línunni í fallinu og einnig þeim stað þar sem kallað er á fallið í `printMultTable`. Eins og gera mátti ráð fyrir þá skrifar þetta forrit út 7x7 margföldunartöflu:

```
1   2   3   4   5   6   7
2   4   6   8   10   12   14
3   6   9   12   15   18   21
4   8   12   16   20   24   28
5   10   15   20   25   30   35
6   12   18   24   30   36   42
7   14   21   28   35   42   49
```

Þegar maður gerir fall almennara þá sér maður stundum að niðurstaðan hefur eiginleika sem ekki voru fyrirséðir. Þú tekur kannski eftir því núna að margföldunartaflan er samhverf (e. symmetric) því $ab = ba$ og því birtast öll gildi í töflunni tvisvar. Við gætum sparað okkur blekið (að því gefnu að úttakið væri sent á prentara) með því að skrifa bara út helming töflunnar. Við þurfum aðeins að breyta einni línu í `printMultTable` til að gera það. Breyttu

```
printMultiples (i, high);
```

í

```
printMultiples (i, i);
```

og þá færðu

```
1
2   4
3   6   9
4   8   12   16
5   10   15   20   25
6   12   18   24   30   36
7   14   21   28   35   42   49
```

Ég læt þig um að finna út hvernig þetta virkar!

6.11 Orðalisti

lykkja (e. loop): Setning sem er endurtekin á meðan að tiltekið skilyrði er satt.

óendanleg lykkja (e. infinite loop): Lykkja, hvers skilyrði er ávallt satt.

meginmál (e. body): Setningarnar innan í lykkju.

ítrun (e. iteration): Ein umferð í gegnum meginmál lykkju, þ.m.t. ákvörðun á gildi skilyrðisins.

dálkalykill (e. tab): Sérstakur stafur, skrifaður sem `\t` í C++, sem gerir það að verkum að bendillinn færist í næsta dálkastopp í núverandi línu.

hjúpa (e. encapsulate): Að setja tiltekna virkni inn í sérstaka einingu, t.d. fall, og einangra virknina (t.d. með því að nota staðværar breytur) frá öðrum hluta forritsins.

staðvær breyta (e. local variable): Breyta sem er skilgreind inni í falli og hvers líftimi er aðeins innan í fallinu. Ekki er hægt að nálgast staðværar breytur utan frá og þær hafa engin áhrif á virkni annarra falla.

alhæfa (e. generalize): Að breyta einhverju sem er óþarflega sérstakt (eins og fast gildi) yfir í eitthvað sem er almennara (eins og breytu eða lepp) Alhæfing gerir kóðann fjölhæfara og líklegri til að verða endurnýtanlegur.

þróunarferli (e. development process): Ferli til að þróa forrit. Í þessum kafla hef ég sýnt sérstaka aðferð sem byggir á því að skrifa kóða sem framkvæmir einfalda, sérstaka hluti og hef síðan beitt hjúpun og alhæfingu.

Kafli 7

Strengir

7.1 Geymsla fyrir strengi

Við höfum hingað til séð fimm tegundir af gildum – boole (e. booleans), stafi (e. characters), heiltölur (e. integers), kommutölur (e. floating-point numbers) og strengi (e. strings) – en aðeins fjórar tegundir af breytum – `bool`, `char`, `int` og `double`. Við höfum því ekki séð neina leið til að geyma streng í breytu og framkvæma aðgerðir á strengjum.

Það eru reyndar nokkrar tegundir af breytum í C++ sem geta geymt strengi. Einn þeirra er grunntag, hluti af C++ málinu, sem er stundum kallað "a native C-string." Málskipanin fyrir C-strengi er dálítið skrýtin og þar sem notkun C-strengja krefst þekkingar á atriðum sem við höfum ekki fjallað um þá munum við að mestu leyti sleppa C-strengjum.

Strengjatagið sem við munum nota er kallað `string` en það er einn af þeim klösum (e. classes) sem tilheyra C++ Standard Library.[1]

Því miður vill reyndar svo til að við getum ekki algerlega hunsað C-strengi. Á nokkrum stöðum í þessum kafla mun ég benda þér á vandamál sem hægt er að rekast á með því að nota `string` í stað C-strengja.

7.2 `string` breytur

Þú býrð til breytu af taginu `string` á hefðbundin hátt:

```
string first;
first = "Hello, ";
string second = "world.";
```

[1] Þú veltir því vafalaust fyrir þér hvað ég á við með **klasa**. Við eigum eftir að skoða nokkra kafla í viðbót áður en ég get gefið fullnægjandi skilgreiningu en að svo stöddu getum við sagt að klasi sé safn falla sem skilgreina aðgerðir sem hægt er að framkvæma á tilteknu tagi. `string` klasinn inniheldur öll föll sem hægt er að beita á tagið `string`.

Fyrsta línan lýsir yfir breytu af taginu `string` án þess að gefa henni gildi. Önnur línan gefur henni strengjagildið `"Hello."` Þriðja línan er bæði yfirlýsing (e. declaration) og gildisveiting (e. assignment), þ.e. frumstilling (e. initialization).

Þegar strengjagildi eins og `"Hello, "` eða `"world."` koma fyrir þá eru þau venjulega meðhöndluð sem C-strengir. Í þessu tilviki er þeim hins vegar breytt sjálfvirkt í `string` gildi vegna þess að við notum breytur af taginu `string` til að geyma gildin.

Við getum skrifað út strengi á hefðbundin máta:

```
cout << first << second << endl;
```

Þú þarft að setja hausaskrá (e. header file) fyrir `string` klasann inn í forritið þitt til að geta þýtt þennan kóða:

```
#include <string>
```

Áður en við höldum lengra þá skaltu slá ofangreint forrit inn og vera viss um að þú getir þýtt það og keyrt.

7.3 Útdráttur stafa úr streng

Strengir eru kallaðir "strengir" vegna þess er þeir eru settir saman af röð eða streng af stöfum. Fyrsta aðgerðin sem við ætlum að framkvæma á streng er að draga út einn af stöfunum. C++ notar hornklofa (e. square brackets) (`[` og `]`) fyrir þessa aðgerð:

```
string fruit = "banana";
char letter = fruit[1];
cout << letter << endl;
```

Segðin `fruit[1]` gefur til kynna að ég vilji fá staf nr. 1 í strengnum með nafninu `fruit`. Niðurstaðan er geymd í `char` breytu með nafninu `letter`. Þegar ég síðan skrifa út gildið á `letter` þá kemur niðurstaðan á óvart:

```
a
```

`a` er ekki fyrsti stafurinn í `"banana"`. Nema að þú sért tölvunarfræðingur! Af sérstökum ástæðum byrja tölvunarfræðingar alltaf að telja frá núlli. Núllti stafurinn í `"banana"` er `b`. Fyrsti stafurinn í er `a` og annar stafurinn er `n`.

Þannig að ef þú vilt fá núllta stafinn úr streng þá þarftu að setja núll inn í hornklofana:

```
char letter = fruit[0];
```

7.4 Lengd

Við getum notað `length` fallið til að finna lengd (fjölda stafa) strengs. Málskipanin sem notuð er til að kalla á þetta fall er dálítið frábrugðin þeirri sem við höfum séð hingað til:

```
int length;
length = fruit.length();
```

Til að lýsa þessu fallakalli getum við sagt að við séum að **kalla á** (e. invoking) þetta lengdarfall sem tilheyrir streng með nafninu `fruit`. Þetta hjómar kannski undarlega en við munum sjá mörg önnur dæmi um að kalla á fall sem tilheyrir hlut (e. object). Málskipanin fyrir þetta er kölluð "punktatáknun" (e. dot notation) vegna þess að punktur (e. dot) kemur á milli nafns hlutarins `fruit`, og nafns fallsins `length`.

`length` tekur engin viðföng eins og sjá má með tómu svigunum `()`. Skilagildið er heiltala, í þessu tilviki 6. Taktu eftir að það er leyfilegt að hafa breytu með sama nafn og fall.

Það gæti verið freistandi að reyna eftirfarandi til að draga út síðasta staf í streng:

```
int length = fruit.length();
char last = fruit[length];       // WRONG!!
```

Þetta gengur ekki vegna þess að það er enginn stafur nr. 6 í `"banana"`. Þar sem við byrjum að telja í 0 þá eru stafirnir 6 númeraðir frá 0 upp í 5. Við þurfum að draga 1 frá `length` til að draga út síðasta stafinn.

```
int length = fruit.length();
char last = fruit[length-1];
```

7.5 Að ferðast eftir

Þegar unnið er með strengi þá er algengt að byrja á byrjuninni, velja síðan næsta staf, framkvæma einhverja aðgerð á honum, og endurtaka þetta þangað til að komið er að enda strengsins. Þetta vinnslumynstur er kallað **að ferðast eftir** (e. traverse). Það er einfalt að kóða svona "rölt" með `while` setningu:

```
int index = 0;
while (index < fruit.length()) {
  char letter = fruit[index];
  cout << letter << endl;
  index = index + 1;
}
```

Þessi lykkja ferðast eftir strengnum og skrifar út sérhvern staf í sér línu. Taktu eftir því að skilyrðið er `index < fruit.length()` sem merkir að þegar `index` er jafnt lengd strengsins þá er skilyrðið ósatt og meginmál lykkjunnar er þá ekki keyrt. Síðasti stafurinn sem við drögum út er sá með vísinn (e. index) `fruit.length()-1`.

Nafn lykkjubreytunnar er `index`. **index** (eða vísir á íslensku) er breyta eða gildi sem notuð er til að tilgreina eitt stak í röðuðu mengi, í þessu tilviki mengi af stöfum í streng. Vísirinn tilgreinir hvaða stak maður hefur áhuga á. Mengið þarf að vera raðað þannig að sérhver stafur hafi vísi og að vísir standi fyrir einn tiltekinn staf.

Þú ættir núna, svona til að æfa þig, að skrifa fall sem tekur `string` sem viðfang og skrifar út stafi strengsins í öfugri röð, alla í einni og sömu línunni.

7.6 Keyrsluvilla

Í kafla 1.3.2 talaði ég um keyrsluvillur, þ.e. villur sem koma ekki fram fyrr en við keyrslu forrits.

Hingað til hefur þú líklega ekki séð margar keyrsluvillur vegna þess að við höfum ekki gert marga hluti sem gætu orsakað þær. Það breytist núna. Ef þú notar `[]` virkjann og gefur upp vísi sem er negatífur eða stærri en `length-1` þá færðu keyrsluvillu með villuskilaboðum eitthvað á þessa leið:

```
index out of range: 6, string: banana
```

Prófaðu þetta í þínu þróunarumhverfi til að sjá hvernig villuskilaboðin líta út.

7.7 `find` fallið

`string` klasinn býður upp á ýmis önnur föll sem þú getur beitt á strengi. `find` fallið er eins og andhverfan við `[]` virkjann. Í stað þess að gefa upp vísi og draga út stafinn sem tengist þeim vísi þá tekur `find` fallið staf sem viðfang og finnur vísinn þar sem viðkomandi stafur finnst.

```
string fruit = "banana";
int index = fruit.find('a');
```

Þetta dæmi finnur vísinn fyrir stafinn `'a'` í strengnum `fruit`. Í þessu tilviki birtist stafurinn þrisvar sinnum í strengnum þannig að það er ekki augljóst hvað `find` eigi að gera. Samkvæmt skjölun (e. documentation) um fallið þá skilar það vísinum á *fyrsta* tilvikinu, þannig að hér er niðurstaðan 1. Ef uppgefinn stafur finnst ekki í strengnum þá skilar `find` -1.

Önnur útgáfa af `find` er til sem tekur annan `string` (hlutstreng) sem viðfang og finnur vísinn sem samsvarar því hvar hlutstrengurinn byrjar í strengnum. Dæmi:

```
string fruit = "banana";
int index = fruit.find("nan");
```

Í þessu dæmi er niðurstaðan 2.

Þú ættir að muna frá kafla 5.4 að hægt er að vera með fleiri en eitt fall með sama nafni, svo framarlega sem þau taka mismunandi fjölda viðfanga eða mismunandi tög. Í þessu tilviki veit C++ þýðandinn hvora útgáfuna af `find` verið er að nota með því að skoða tög viðfanganna í kallinu.

7.8 Okkar eigin útgáfa af `find`

Ef við erum að leita að tilteknum staf í streng þá getur verið að við viljum ekki endilega byrja að leita frá upphafi strengsins. Ein leið til að gera `find` fallið almennara er að skrifa útgáfu af því sem tekur eitt auka viðfang – vísi sem stendur fyrir þann stað sem við viljum byrja leitina frá. Hér er útfærsla af þessu falli:

```
int find (string s, char c, int i)
{
  while (i<s.length()) {
    if (s[i] == c) return i;
    i = i + 1;
  }
  return -1;
}
```

Í stað þess að kalla á þetta fall með því að nota punktatáknun eins og við gerðum fyrir fyrstu útgáfuna af `find` þá þurfum við að senda `string` sem fyrsta viðfang. Hin viðföngin eru stafurinn sem við erum að leita að og vísirinn sem gefur til kynna hvar við byrjum leitina.

7.9 Talning

Eftirfarandi forrit telur fjölda tilvika af stafnum `'a'` í streng:

```
string fruit = "banana";
int length = fruit.length();
int count = 0;

int index = 0;
while (index < length) {
  if (fruit[index] == 'a') {
    count = count + 1;
  }
  index = index + 1;
```

```
  }
  cout << count << endl;
```

Þetta forrit er gott dæmi um notkun á **teljara** (e. counter). Breytan `count` er frumstillt með núlli og síðan hækkuð í sérhvert sinn sem við finnum stafinn `'a'`. (Á ensku er talað um **to increment**, þ.e. að hækka um einn, sem er andstæðan við **to decrement**, að lækka um einn.) Þegar við hættum í lykkjunni þá mun `count` innihalda niðurstöðuna: fjöldann af stafnum `'a'`.

Svona til að æfa þig þá skaltu núna hjúpa þennan kóða sem fall með nafninu `countLetters` og beita síðan alhæfingu (gera fallið almennt) þannig að það taki við streng og staf sem viðföngum.

7.10 Hækkunar- og lækkunarvirkjar

Það að bæta við einum (e. incrementing) og draga einn frá (e. decrementing) eru svo algengar aðgerðir að C++ býður upp á sérstaka virkja fyrir þær. `++` virkinn (sem kalla má "auki" á íslensku) bætir einum við núverandi gildi á `int`, `char` eða `double`, og `--` (sem kalla má "frádrag" á íslensku) dregur einn frá. Hvorugur virkjanna virkar á `string` og hvorugum þeirra *ætti* að beita á `bool`.

Tæknilega séð er löglegt að hækka breytu og nota hana í segð á sama tíma. Þú gætir t.d. séð svona kóða:

```
  cout << i++ << endl;
```

Þegar við horfum á þetta þá er ekki ljóst hvort hækkun breytunnar muni eiga sér stað fyrir eða eftir að gildi hennar er skrifað út. Ég mæli ekki með að þú notir `++` eða `--` á þennan hátt því þessi notkun getur valdið ruglingi. Reyndar ætla ég ekki að segja hver niðurstaðan er. Þú verður bara að prófa þetta ef þú iðar í skinninu að vita það!

Með því að nota hækkunarvirkjann getum við endurskrifað kóðann sem telur tilvik af staf:

```
  int index = 0;
  while (index < length) {
    if (fruit[index] == 'a') {
      count++;
    }
    index++;
  }
```

Algeng villa er að skrifa eitthvað á þessa leið:

```
  index = index++;              // WRONG!!
```

Því miður vill svo til að þetta er málfræðilega rétt og því mun þýðandinn ekki gera neinar athugasemdir. Niðurstaðan af þessari setningu er sú að gildið á `index` helst óbreytt. Það er oft erfitt að finna þessa villu.

Mundu að þú getur skrifað `index = index +1;` eða `index++;`. Þú ættir hins vegar ekki að blanda þessu tvennu saman.

7.11 Samskeyting strengja

Það er athyglisvert að hægt er að beita virkjanum + á strengi en þá framkvæmir virkinn **samskeytingu** (e. concatenation). Samskeyting merkir að skeyta saman tveimur (eða fleiri) strengjum. Dæmi:

```
string fruit = "banana";
string bakedGood = " nut bread";
string dessert = fruit + bakedGood;
cout << dessert << endl;
```

Úttakið úr þessu forriti er `banana nut bread`.

Því miður virkar + virkinn hins vegar ekki á "native" C-strengi og því er ekki hægt að skrifa

```
string dessert = "banana" + " nut bread";
```

vegna þess að báðir þolendur + virkjans eru hér C-strengir (`"banana"` og `" nut bread"`). Svo framarlega sem annar þolandanna er `string` þá mun C++ reyndar breyta hinum sjálfvirkt í `string`.

Það er líka mögulegt að skeyta stökum staf við byrjun eða enda á streng. Í eftirfarandi dæmi notum við samskeytingu og stafareikning (e. character arithmetic) til að skrifa út "abecedarian" röð.

"Abecedarian" vísar til raðar eða lista hvers stök eru í stafrófsröð. Í bók Robert McCloskey's *Make Way for Ducklings* eru t.d. nöfn andarunganna Jack, Kack, Lack, Mack, Nack, Ouack, Pack og Quack. Hér er lykkja sem skrifar út þessi nöfn í réttri röð:

```
string suffix = "ack";

char letter = 'J';
while (letter <= 'Q') {
  cout << letter + suffix << endl;
  letter++;
}
```

Úttak forritsins er:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Auðvitað er þetta ekki alveg rétt því "Ouack" og "Quack" skrifast ekki alveg rétt út. Þú ættir að breyta forritinu og leiðrétta þessa villu.

Hér þurfum við aftur að passa okkur að nota samskeytingu aðeins með `string` taginu en ekki með "native" C-strengjum. Því miður er segð eins og `letter + "ack"` málfræðilega rétt í C++ en útkoman er mjög skrýtin, a.m.k. í mínu þróunarumhverfi.

7.12 Strengir eru breytanlegir

Þú getur breytt einstökum stöfum í streng með því að nota `[]` virkjann á vinstri hlið gildisveitingar: Þetta dæmi,

```
string greeting = "Hello, world!";
greeting[0] = 'J';
cout << greeting << endl;
```

skrifar út `Jello, world!`.

7.13 Strengir eru samanburðarhæfir

Allir samanburðarvirkjarnir sem virka fyrir `int` og `double` virka líka fyrir `string`. Ef þú vilt t.d. vita hvort tveir strengir eru jafnir:

```
if (word == "banana") {
  cout << "Yes, we have no bananas!" << endl;
}
```

Hinir samanburðarvirkjarnir eru t.d. gagnlegir til að raða orðum í stafrófsröð:

```
if (word < "banana") {
  cout << "Your word, " << word << ", comes before banana." << endl;
} else if (word > "banana") {
  cout << "Your word, " << word << ", comes after banana." << endl;
} else {
  cout << "Yes, we have no bananas!" << endl;
}
```

Athugaðu þó að `string` klasinn meðhöndlar ekki hástafi og lágstafi á sama hátt og við gerum. Allir hástafir koma á undan öllum lágstöfum. Þess vegna,

```
Your word, Zebra, comes before banana.
```

Algeng leið til að bregðast við þessu vandamáli er að breyta strengjum yfir á staðlað form, t.d. yfir í lágstafi, áður en samanburður er framkvæmdur. Næsti kafli skýrir hvernig það er gert.

7.14 Flokkun stafa

Það er oft gagnlegt að skoða staf og athuga hvort hann er hástafur eða lágstafur eða hvort hann er bókstafur eða tölustafur. C++ fylgir safn falla (e. library functions) sem getur flokkað stafi á þennan hátt.

```
char letter = 'a';
if (isalpha(letter)) {
  cout << "The character " << letter << " is a letter." << endl;
}
```

Hér er eðlilegt að gera ráð fyrir því að skilagildið úr fallinu `isalpha` sé `bool`, en af skrýtnum ástæðum er skilagildið reyndar heiltala sem er 0 ef viðfangið er ekki bókstafur en einhver önnur tala ef viðfangið er bókstafur.

Þessi undarlegheit eru reyndar ekki eins óþægileg eins og kannski virðist því það er leyfilegt að nota svona heiltölu í skilyrði eins og sést í dæminu. Gildið 0 er meðhöndlað sem `false` en allar aðrar tölur eru meðhöndlaðar sem `true`.

Tæknilega séð ætti svona lagað ekki að vera leyfilegt – heiltölugildi eru frábrugðin boole gildum. Samt sem áður getur þessi C++ venja, að breyta sjálfvirkt á milli taga, stundum verið gagnleg.

Önnur föll sem flokka stafi eru t.d. `isdigit`, sem ber kennsl á tölustafina 0 til 9, og `isspace`, sem ber kennsl á allar tegundir af "hvítum" bilum (e. white spaces), þ.m.t. bil, dálkastaf og nýja línu. Einnig má nefna föllin `isupper` og `islower` sem gera greinarmun á hástöfum og lágstöfum.

Að lokum nefni ég tvö föll sem breyta stöfum úr eini formi í annað, `toupper` og `tolower`. Bæði þessi föll taka einn staf sem viðfang og skila (hugsanlega) breyttum staf.

```
char letter = 'a';
letter = toupper (letter);
cout << letter << endl;
```

Úttakið úr þessum kóða er `A`.

Til að æfa þig ættir þú að nota ofangreind föll til að skrifa föllin `stringToUpper` og `stringToLower` sem bæði taka einn `string` sem viðfang og breyta honum þannig að öllum stöfum er breytt í hástafi eða lágstafi. Skilagildi fallanna ætti að vera `void`.

7.15 Önnur strengjaföll

Í þessum kafla höfum við ekki talað um öll strengjaföllin. Við munum ræða tvö í viðbót, `c_str` í kafla 15.2 og `substr` í kafla 15.4.

7.16 Orðalisti

hlutur (e. object): Tilvik af tilteknum klasa. Tilvikinu fylgja mengi af föllum sem framkvæma aðgerðir á því. Hlutirnir sem við höfum notað hingað til eru `cout` og tilvik af `string`.

vísir (e. index): Breyta eða gildi sem notað er til að velja eitt stak í röðuðu mengi, t.d. einn staf úr streng.

ferðast eftir (e. traverse): Að ítra í gegnum öll stök mengis og framkvæma sömu aðgerð á sérhverju staki.

teljari (e. counter): Breyta sem er notuð til að telja eitthvað. Breytan er yfirleitt frumstillt með 0 og síðan hækkuð.

hækka (e. increment): Að hækka gildi breytu um einn. Hækkunarvirkinn í C++ er `++`. Þetta er einmitt ástæðan fyrir því að C++ er kallað C++, því markmiðið var að það væri einum betra en C!

lækka (e. decrement): Að lækka gildi breytu um einn. Lækkunarvirkinn í C++ er `--`.

skeyta saman (e. concatenate): Að sameina tvo þolendur.

Kafli 8

Strúktúrar

8.1 Samsett gildi

Flest af þeim gagnatögum sem við höfum unnið með hingað til standa fyrir eitt tiltekið gildi – heiltölu, kommutölu og boole gildi. Strengir eru öðruvísi að því leyti til að þeir eru settir saman úr smærri gildum, þ.e. stöfum. Strengir eru því dæmi um **samsett** gildi (e. compound type).

Við gætum þurft að meðhöndla samsett gildi sem einn tiltekinn hlut og við gætum þurft að komast í einstaka hluta samsetta gildisins.

Það er einnig gagnlegt fyrir þig að geta búið til þín eigin samsettu gildi. C++ býður upp á tvær leiðir til að gera það: **strúktúra** (e. structures) og **klasa** (e. classes). Við munum byrja á því að fjalla um strúktúra og förum síðan í klasa í kafla 14 (það er ekki mikill munur á milli þeirra).

8.2 `Point` hlutir

Við skulum skoða stærðfræðilegan punkt (eins og í hnitakerfi) sem dæmi um samsett gildi. Punktur er í raun tvær tölur (hnit) sem við meðhöndlum sem einn tiltekinn hlut. Í stærðfræði eru punktar oft skrifaðir innan sviga með kommu á milli hnitanna. T.d. gefur $(0, 0)$ til kynna upphafspunkt (hnitamiðju) og (x, y) stendur fyrir punkt sem er x einingar til hægri og y einingar upp miðað við upphafspunktinn.

Eðlileg leið til að tákna punkt í C++ er að nota tvær kommutölur, `double`. Spurningin er hins vegar hvernig hægt er að setja þessi tvö gildi saman í samsettan hlut eða strúktúr. Það er hægt með því að nota `struct` skilgreiningu:

```
struct Point {
  double x, y;
};
```

`struct` skilgreining er yfirleitt sett fram utan fallaskilgreininga, í upphafi forrits (á eftir `include` setningum).

Þessi skilgreining gefur til kynna að í strúktúrnum eru tvö stök (gildi), nefnd `x` og `y`. Þessi stök eru kölluð **tilvikabreytur** (e. instance variables), en ég mun skýra síðar hver ástæðan er fyrir því.

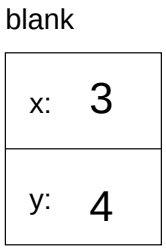
Það er algeng villa að gleyma semíkommunni í enda strúktúrskilgreiningar. Það virðist skrýtið að setja semíkommu á eftir slaufusviga en þú venst því fljótt.

Þegar þú hefur skilgreint nýjan strúktúr þá getur þú búið til breytur af því tagi:

```
Point blank;
blank.x = 3.0;
blank.y = 4.0;
```

Fyrsta línan er hefðbundin yfirlýsing á breytu: `blank` er af taginu `Point`. Næstu tvær línur frumstilla tilvikabreytur strúktúrsins. Hér er punktatáknun notuð á svipaðan hátt og þegar kallað er á fall sem tilheyrir tilteknum hlut, eins og í `fruit.length()`. Munurinn er auðvitað sá að fallakalli fylgir viðfangalisti, jafnvel þó hann sé tómur.

Niðurstaðan af þessum gildisveitingum sést í eftirfarandi stöðuriti:



Að venju birtist nafn breytunnar `blank` utan kassans en gildi hennar innan hans. Í þessu tilviki er gildið samsettur hlutur með tveimur tilvikabreytum.

8.3 Aðgangur að tilvikabreytum

Þú getur lesið gildi tilvikabreytu með því að nota sömu málskipan og við notuðum til að gefa henni gildi:

```
int x = blank.x;
```

Segðin `blank.x` merkir "farðu í hlutinn með nafninu `blank` og náðu í gildið á `x`." Í þessu tilviki gefum við staðværri breytu með nafninu `x` það gildi. Taktu eftir því að það er enginn "árekstur" á milli staðværu breytunnar `x` og tilvikabreytunnar `x`. Tilgangur punktatáknunar er einmitt sá að gefa til kynna hvaða breytu verið er að vísa í án þess að einhver margræðni sé til staðar.

Hægt er að nota punktatáknun sem hluta af hvaða C++ segð sem er, þannig að eftirfarandi er t.d. löglegt:

```
cout << blank.x << ", " << blank.y << endl;
double distance = blank.x * blank.x + blank.y * blank.y;
```

Fyrri línan skrifar út `3, 4` og seinni línan reiknar út gildið 25.

8.4 Aðgerðir á strúktúrum

Flestum af þeim aðgerðum sem við höfum notað á önnur tög, eins og stærðfræðivirkjarnir (e. mathematical operators) (`+`, `%`, o.s.frv.) og samanburðarvirkjarnir (e. comparison operators) (`==`, `>`, o.s.frv.) er ekki hægt að beita á strúktúra. Reyndar er hægt að breyta merkingu þessara virkja fyrir ný tög en við munum ekki fjalla um það í þessari bók.

Gildisveitingarvirkjanum (e. assignment operator) er aftur á móti hægt að beita á strúktúra. Hægt er að nota hann á tvo vegu: til að frumstilla tilvikabreytu strúktúrs eða til að afrita gildi tilvikabreytu úr einum strúktúr í annan. Frumstilling lítur svona út:

```
Point blank = { 3.0, 4.0 };
```

Tilvikabreytur breytunnar `blank` fá hér gildin úr slaufusviganum, í þeirri röð sem þau eru sett fram. Þannig að hér fær `blank.x` fyrsta gildið (3.0) og `blank.y` annað gildið (4.0).

Því miður er eingöngu hægt að nota þessa málskipan í frumstillingu en ekki í gildisveitingarsetningu. Eftirfarandi er því ekki löglegt:

```
Point blank;
blank = { 3.0, 4.0 };       // WRONG !!
```

Það er eðlilegt að velta því fyrir sér hver ástæðan er fyrir því að jafn eðlileg setning sé óleyfileg. Ég er ekki alveg viss en vandamálið gæti verið það að þýðandinn veit ekki hvert tagið á hægri hliðinni er. Ef þú bætir við tagmótun (e. typecast) þá er allt í fína lagi:

```
Point blank;
blank = (Point){ 3.0, 4.0 };
```

Það er jafnframt leyfilegt að gefa einum strúktúr gildi annars strúkturs. Dæmi:

```
Point p1 = { 3.0, 4.0 };
Point p2 = p1;
cout << p2.x << ", " <<  p2.y << endl;
```

Úttakið úr þessu forriti er `3, 4`.

8.5 Strúktúrar sem viðföng

Þú getur haft strúktúr sem lepp í falli, t.d.:

```
void printPoint (Point p) {
  cout << "(" << p.x << ", " << p.y << ")" << endl;
}
```

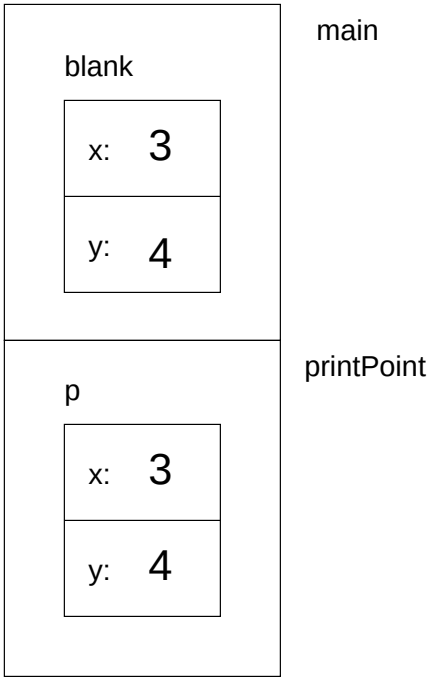
`printPoint` tekur punkt sem viðfang og skrifar gildi hans út á staðlaðan hátt. Ef þú kallar á fallið með `printPoint (blank)` þá skrifast út `(3, 4)`.

Tökum annað dæmi. Við getum endurskrifað `distance` fallið úr kafla 5.2 þannig að það taki tvo punkta sem viðföng í stað fjögurrra kommutalna:

```
double distance (Point p1, Point p2) {
  double dx = p2.x - p1.x;
  double dy = p2.y - p1.y;
  return sqrt (dx*dx + dy*dy);
}
```

8.6 Kallað með gildi

Það er mikilvægt að gera sér grein fyrir því að þegar strúktúr er sendur sem viðfang í fall þá er viðfangið (e. argument/actual parameter) og leppurinn (e. formal parameter) ekki sama breytan. Um er að ræða tvær breytur (önnur í þeim sem kallar (e. caller) og hin í þeim sem kallað er á (e. callee)) sem hafa sama gildið, a.m.k. í upphafi. Þegar við t.d. köllum á `printPoint` þá lítur stöðuritið svona út:



Ef `printPoint` breytir annarri (eða báðum) tilvikabreytum `p` þá mun það ekki hafa nein áhrif á `blank` (auðvitað er engin ástæða fyrir `printPoint` að breyta leppnum sínum).

Þessi tegund af **stikun færibreytna** (e. parameter passing) er kölluð "kall með gildi" (e. "pass by value") vegna þess að gildi (e. value) strúktúrsins (eða hvaða tags sem er) er sent til fallsins og afritað yfir í leppinn.

8.7 Kallað með tilvísun

Önnur aðferð við stikun færibreytna í C++ er "kall með tilvísun" (e. "pass by reference"). Þessi aðferð gerir þér kleift að senda strúktúr í fall og breyta gildum hans!

Þú gætir t.d. speglað punkti um 45-gráðu línuna með því að skipta á gildum hnitanna tveggja. Augljósasta leiðin (en ekki sú rétta) er að útfæra `reflect` fallið á þennan hátt:

```
void reflect (Point p)      // WRONG !!
{
  double temp = p.x;
  p.x = p.y;
  p.y = temp;
}
```

Þetta virkar ekki vegna þess að það að gera breytingar á leppnum í `reflect` hefur engin áhrif á þann sem kallaði.

Í staðinn verðum við að tilgreina að við ætlum að senda viðfangið með tilvísun (e. by reference). Það gerum við með því að bæta tákninu `&` við í skilgreiningu á leppunum:

```
void reflect (Point& p)
{
  double temp = p.x;
  p.x = p.y;
  p.y = temp;
}
```

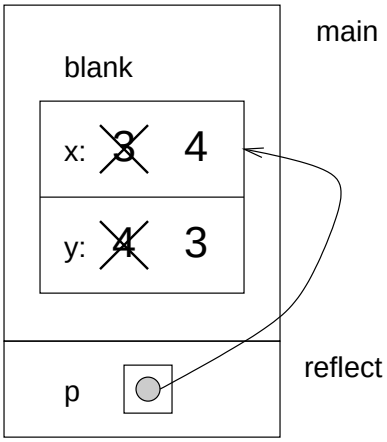
Núna getum við kallað á fallið á hefðbundin hátt:

```
  printPoint (blank);
  reflect (blank);
  printPoint (blank);
```

Úttak forritsins er eins og við gerum ráð fyrir:

```
(3, 4)
(4, 3)
```

Svona lítur stöðurit út fyrir þetta forrit:



Leppurinn `p` er tilvísun í strúktúr með nafnið `blank`. Hefðbundin táknun fyrir tilvísun er punktur með ör sem bendir á það sem tilvísunin vísar á.

Það mikilvæga sem hægt er að lesa úr þessu stöðuriti er að allar breytingar sem `reflect` gerir á `p` munu einnig hafa áhrif á `blank`.

Það að stika strúktúr með tilvísun (e. by reference) er sveigjanlegra heldur en stika hann með gildi (e. by value) vegna þess að sá sem kallað er á getur breytt strúktúrnum. Það er jafnframt hraðvirkara vegna þess að kerfið þarf ekki að afrita heilan strúktúr. Á hinn bóginn má segja að það sé ekki eins öruggt vegna þess að það er erfiðara að gera sér grein fyrir hvaða breytingar eru gerðar hvar. Samt sem áður eru strúktúrar yfirleitt stikaðir með tilvísun í C++ forritum og ég mun fylgja þeirri venju í þessari bók.

8.8 Rétthyrningar

Gerum nú ráð fyrir að við viljum búa til strúktúr sem stendur fyrir rétthyrning. Hvaða upplýsingar þurfum við að veita til þess að skilgreina rétthyrning? Til að einfalda málið skulum við gera ráð fyrir því að rétthyrningnum sé stillt upp lárétt eða lóðrétt en ekki miðað við tiltekið horn (gráður).

Það eru nokkrir möguleikar. Ég gæti tilgreint miðju rétthyrnings (tvö hnit) og stærð hans (breidd og hæð), eða tilgreint eitt af hornum hans ásamt stærðinni, eða tilgreint hnit tveggja andstæðra horna.

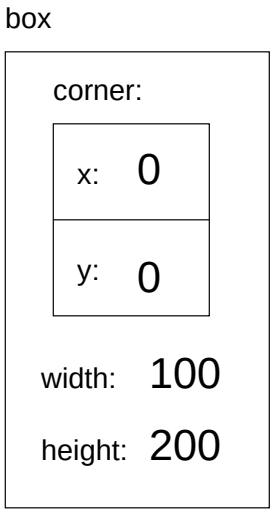
Algeng leið er að tilgreina efra vinstra horn rétthyrningsins ásamt stærðinni. Til að gera það í C++ þá skilgreinum við strúktúr sem inniheldur `Point` og tvær kommutölur.

```
struct Rectangle {
  Point corner;
  double width, height;
};
```

Taktu eftir að einn strúktúr getur innifalið annan og það er reyndar mjög algengt. Það þýðir auðvitað að til að búa til `Rectangle` þá þurfum við fyrst að búa til `Point`:

```
Point corner = { 0.0, 0.0 };
Rectangle box = { corner, 100.0, 200.0 };
```

Þessi kóði býr til nýjan `Rectangle` strúktúr og frumstillir tilvikabreytur hans. Myndin sýnir áhrif þess.



Við getum nálgast `width` og `height` á venjulegan máta:

```
box.width += 50.0;
cout << box.height << endl;
```

Til að nálgast tilvikabreytur `corner` getum við notað tímabundna breytu:

```
Point temp = box.corner;
double x = temp.x;
```

En við gætum líka sett þessar tvær setningar saman í eina:

```
double x = box.corner.x;
```

Það er eðlilegast að lesa þessa setningu frá hægri til vinstri: "Sæktu `x` úr `corner` úr `box` og láttu staðværu breytuna `x` fá það gildi."

Fyrst við erum að tala um samsetningu setninga þá bendi ég jafnframt á að þú getur reyndar búið til `Point` og `Rectangle` á sama tíma:

```
Rectangle box = { { 0.0, 0.0 }, 100.0, 200.0 };
```

Innri slaufusviginn tilgreinir hnit hornsins sem stendur þá fyrir fyrsta gildið af þremur sem þarf fyrir `Rectangle`. Þessi setning er dæmi um **hreiðraðan strúktúr** (e. nested structure).

8.9 Strúktúrar sem skilagildi

Þú getur skrifað föll sem skila strúktúrum. Eftirfarandi fall `findCenter` tekur `Rectangle` sem viðfang og skilar `Point` sem inniheldur hnit miðjunnar á `Rectangle`:

```
Point findCenter (Rectangle& box)
{
  double x = box.corner.x + box.width/2;
  double y = box.corner.y + box.height/2;
  Point result = {x, y};
  return result;
}
```

Til að kalla á þetta fall verðum við að senda box sem viðfang (taktu eftir að það er sent með tilvísun) og setjum skilagildið inn í `Point` breytu:

```
  Rectangle box = { {0.0, 0.0}, 100, 200 };
  Point center = findCenter (box);
  printPoint (center);
```

Úttakið úr þessu forriti er `(50, 100)`.

8.10 Að senda önnur tög með tilvísun

Það eru ekki eingöngu strúktúrar sem hægt er að senda með tilvísun í föll. Það sama gildir um öll önnur tög sem við höfum séð. Til að skipta á gildum tveggja heiltölubreytna getum við t.d. gert eftirfarandi:

```
void swap (int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

Við myndum kalla á þetta fall á hefðbundin hátt:

```
  int i = 7;
  int j = 9;
  swap (i, j);
  cout << i << j << endl;
```

Úttakið úr þessu forriti er `97`. Teiknaðu stöðurit fyrir þetta forrit til að fullvissa þig um að þetta sé rétt. Ef lepparnir `x` og `y` væru skilgreindir sem venjulegir leppar (þ.e. án `&`), þá myndi `swap` ekki virka rétt. Í því tilviki myndi það breyta `x` og `y` án þess að hafa nokkur áhrif á `i` og `j`.

Þegar fólk byrjar að senda hluti eins og heiltölur með tilvísun þá reynir það stundum að nota segðir sem tilvísunarviðföng. Dæmi:

```
int i = 7;
int j = 9;
swap (i, j+1);          // WRONG!!
```

Þetta er ekki leyfilegt vegna þess að segðin `j+1` er ekki breyta – hún stendur ekki fyrir minnishólf sem hægt er að vísa til. Það getur verið svolítið erfitt að átta sig á því hvers konar segðir er hægt að senda með tilvísun en það er ágætis þumalputtaregla að segja að tilvísunarviðföng þurfa að vera breytur.

8.11 Að sækja gögn af lyklaborði

Forritin sem við höfum skrifað hingað til eru nokkuð fyrirsjáanleg, þ.e. þau gera nákvæmlega það sama í hvert skipti sem þau eru keyrð. Í flestum tilvikum viljum við, aftur á móti, forrit sem lesa inntak frá notanda og bregðast við því.

Það eru margar leiðir til að sækja gögn, þ.m.t. lyklaborðsinntak, músarhreyfingar og músarsmellir, svo og framandi aðferðir eins og talgreining og sjónhimnuskönnun. Í þessari bók munum við aðeins sækja gögn af lyklaborði.

Í hausaskránni `iostream` er hluturinn `cin` skilgreindur en hann meðhöndlar inntak á svipaðan hátt og `cout` meðhöndlar úttak. Við getum gert eftirfarandi til að sækja heiltölugildi frá notanda:

```
int x;
cin >> x;
```

» virkinn veldur því að forritið stoppar keyrslu og bíður eftir að notandinn slái eitthvað inn af lyklaborði. Ef notandi slær inn tölustaf þá breytir forritið honum í heiltölugildi og geymir það í `x`.

Ef notandinn slær inn eitthvað annað en tölustaf þá mun keyrsluumhverfið ekki koma með neina villu. Í staðinn mun eitthvað merkingarlaust gildi verða sett inn í `x` og keyrslan heldur áfram.

Sem betur fer höfum við leið til að athuga hvort inntakssetning gekk upp. Við getum kallað á `good` fallið sem tileyrir `cin` til að athuga það sem kallað er **stream state**. `good` skilar `bool`: ef true, þá gekk síðasta inntakssetning upp. Ef ekki þá vitum við að einhver fyrri aðgerð virkaði ekki og jafnframt að næsta aðgerð mun heldur ekki virka.

Að sækja inntak frá notanda gæti því litið svona út:

```
#include <iostream>

using namespace std;

int main ()
{
  int x;

  // prompt the user for input
```

```
  cout << "Enter an integer: ";

  // get input
  cin >> x;

  // check and see if the input statement succeeded
  if (cin.good() == false) {
    cout << "That was not an integer." << endl;
    return -1;
  }

  // print the value we got from the user
  cout << x << endl;
  return 0;
}
```

Það er einnig hægt að nota `cin` til að lesa `string`:

```
  string name;

  cout << "What is your name? ";
  cin >> name;
  cout << name << endl;
```

Því miður les þessi setning aðeins fyrsta orðið í inntakinu en næstu orð verða lesin í næstu inntakssetningu. Þannig að ef þú keyrir þetta forrit og skrifar inn fullt nafn (með bili á milli fyrra og seinna nafns) þá mun það eingöngu skrifa út fyrra nafnið.

Vegna þessara vandamála (vanhæfni til að meðhöndla villur og skrýtin virkni) þá forðast ég að nota » virkjann nema að ég sé að lesa gögn sem ég veit að innihalda ekki neinar villur.

Í staðinn nota ég fall sem heitir `getline` og er skilgreint í hausaskránni `string`.

```
  string name;

  cout << "What is your name? ";
  getline (cin, name);
  cout << name << endl;
```

Fyrra viðfangið í `getline` er `cin` sem tilgreinir hvaðan inntakið er að koma. Seinna viðfangið er nafnið á strengnum sem á að geyma það sem lesið er.

`getline` les heila línu allt að Return (Enter) tákninu. Þetta fall er sem sagt t.d. gagnlegt að nota fyrir strengi sem innihalda bil.

Reyndar er `getline` gagnlegt til að lesa inntak af hvaða tagi sem er. Ef þú vildir t.d. að notandinn slæi inn heiltölu þá gætir þú lesið inntakið sem streng og síðan athugað hvort strengurinn stæði fyrir löglega heiltölu. Ef inntakið væri

löglegt þá myndir þú breyta því í heiltölugildi. Ef ekki þá gætir þú prentað út viðeigandi villuskilaboð og beðið notandann að reyna aftur.

Þú getur notað fallið `atoi` (sem skilgreint er í hausaskránni `cstdlib`) til að breyta streng í heiltölu. Við mun einmitt gera það í kafla 15.4.

8.12 Orðalisti

strúktúr (e. structure): Safn gagna sem eru sett saman til að mynda einn tiltekinn hlut.

tilvikabreyta (e. instance variable): Ein af þeim breytum sem tilheyra strúktúr.

tilvísun (e. reference): Gildi sem vísar í breytu eða strúktúr. Í stöðuriti er tilvísun táknuð með ör.

kall með gildi (e. call/pass by value): Aðferð við stikun færibreytna þar sem gildi viðfangsins (e. argument) er afritað inn í leppinn (e. formal parameter) en viðfangið og leppurinn standa fyrir mismunandi minnishólf.

kall með tilvísun (e. call/pass by reference): Aðferð við stikun færibreytna þar sem leppurinn er tilvísun í viðfangið. Breytingar á leppnum hafa því áhrif á viðfangið.

Kafli 9

Meira um strúktúra

9.1 Time

Í þeim tilgangi að sýna annað dæmi um strúktur munum við nú skilgreina tag með nafnið `Time` sem notað er til að halda utan um tíma innan dags. Tími samanstendur af klukkustund (hour), mínútu (minute) og sekúndu (second) þannig að þetta munu verða tilvikabreytur strúktúrsins.

Fyrsta skrefið er þá að ákveða hvert tag sérhverrar tilvikabreytu á að vera. Það virðist ljóst að `hour` og `minute` ættu að vera heiltölur. Til að hafa þetta áhugaverðara þá skulum við láta `second` vera `double` þannig að við getum haldið utan um brot af sekúndu.

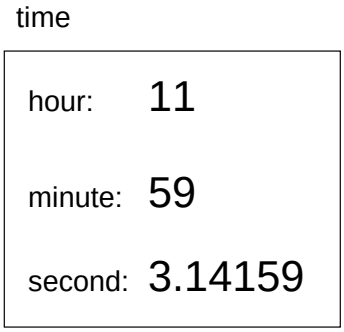
Svona lítur þá strúktúrinn okkar út:

```
struct Time {
  int hour, minute;
  double second;
};
```

Við getum þá búið til `Time` hlut á venjulegan hátt:

```
Time time = { 11, 59, 3.14159 };
```

Stöðuritið fyrir þennan hlut lítur svona út:



Orðið "tilvik" (e. instance) er stundum notað þegar við tölum um hluti (e. objects) vegna þess að sérhver hlutur er tilvik af einhverju tagi. Ástæðan fyrir því að tilvikabreytur bera það nafn er að sérhvert tilvik af einhverju tagi eiga sér afrit af breytum þess tags.

9.2 printTime

Þegar við skilgreinum nýtt tag þá er góð regla að skrifa fall sem skrifar út tilvikabreyturnar á læsilegan hátt. Dæmi:

```
void printTime (Time& t) {
  cout << t.hour << ":" << t.minute << ":" << t.second << endl;
}
```

Ef við sendum inn `time` sem viðfang í þetta fall þá er úttakið `11:59:3.14159`.

```
#include <iostream>

using namespace std;

struct Time {
  int hour, minute;
  double second;
};


void printTime (Time& t) {
  cout << t.hour << ":" << t.minute << ":" << t.second << endl;
  cout << "Time is " << t.hour << " hour " << t.minute << " minutes "
       << t.second << "  seconds  "<<endl;
}


int main ()
{
 Time time = { 11, 59, 3.14159 };
 printTime(time);

 return 0;
}
```

9.3 Föll fyrir hluti

Í næstu köflum mun ég sýna dæmi um nokkur möguleg skil (e. interfaces) fyrir föll sem vinna með hluti. Þú munt hafa val um nokkur möguleg skil fyrir tilteknar aðgerðir þannig að þú ættir að meta kosti og galla sérhvers möguleika:

hreint fall (e. pure function): Tekur hlut og/eða grunntag sem viðföng en breytir ekki hlutnum. Skilagildið er annaðhvort grunntag eða nýr hlutur sem búinn er til í fallinu.

breytir (e. modifier): Tekur hluti sem viðföng og breytir sumum eða öllum þeirra. Skilar oft void.

fyllir (e. fill-in function): Eitt af viðföngunum er "tómur" hlutur sem fallið fyllir inn í. Tæknilega séð er þetta þá í raun tegund af breyti.

9.4 Hrein föll

Fall er sagt vera hreint ef skilagildi þess er eingöngu háð viðföngunum og að það hafi ekki neinar aukaverkanir (e. side effects) eins og að breyta viðfangi eða skrifa eitthvað út. Það eina sem gerist þegar kallað er á hreint fall er að skilagildi þess kemur til baka.

Eitt dæmi um hreint fall er `after` sem ber saman tvo `Time` hluti og skilar `bool` sem gefur til kynna hvort fyrra viðfangið komi á eftir því seinna (tímalega séð):

```
bool after (Time& time1, Time& time2) {
  if (time1.hour > time2.hour) return true;
  if (time1.hour < time2.hour) return false;

  if (time1.minute > time2.minute) return true;
  if (time1.minute < time2.minute) return false;

  if (time1.second > time2.second) return true;
  return false;
}
```

Hvert er skilagildi þessa falls ef tímarnir tveir eru jafnir? Finnst þér það vera viðeigandi skilagildi fyrir þetta fall? Ef þú værir að skjala þetta fall, myndir þú nefna þetta tilfelli sérstaklega?

Annað dæmi er `addTime` sem reiknar summuna af tveimur tímasetningum. Ef tíminn er t.d. `9:14:30` og brauðgerðin þín tekur 3 klukkustundir og 35 mínútur þá gætir þú notað `addTime` til að finna út hvenær brauðið verður tilbúið.

Hér eru drög, sem eru reyndar ekki alveg rétt, af þessu falli:

```
Time addTime (Time& t1, Time& t2) {
  Time sum;
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;
  return sum;
}
```

Hér er dæmi um hvernig hægt er að nota þetta fall. Ef `currentTime` inniheldur núverandi tíma og `breadTime` inniheldur tímann sem tekur að baka brauðið þá getur þú notað `addTime` til að reikna út hvenær brauðið verður tilbúið.

```
Time currentTime = { 9, 14, 30.0 };
Time breadTime = { 3, 35, 0.0 };
Time doneTime = addTime (currentTime, breadTime);
printTime (doneTime);
```

Úttak þessa forrits er `12:49:30` sem er rétt. Á hinn bóginn eru tilvik þar sem niðurstaðan er ekki rétt. Getur þú fundið dæmi um þess konar tilvik?

Vandamálið er að þetta fall ræður ekki við þau tilvik þar sem summa sekúndna eða mínútna er stærri en 60. Þegar það gerirst þá þurfum við að "færa" auka sekúndur yfir í mínútudálkinn eða auka mínútur yfir í klukkustundadálkinn.

Hér er önnur, nú rétt, útgáfa af fallinu:

```
Time addTime (Time& t1, Time& t2) {
  Time sum;
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
  return sum;
}
```

Þó svo að þessi útgáfa sé rétt þá er hún orðin dálítið löng. Ég mun síðar leggja til aðra lausnaraðferð sem er mun styttri.

Kóðinn að ofan sýnir dæmi um tvo virkja sem við höfum ekki séð áður, `+=` og `-=`. Þessir virkjar eru notaðir sem samþjöppuð leið til að hækka eða lækka breytur. Setningin `sum.second -= 60.0;` er t.d. jafngild setningunni `sum.second = sum.second - 60;`

9.5 `const` leppar

Þú hefur væntanlega tekið eftir því að viðföngin í föllin `after` og `addTime` eru send með tilvísun (e. by reference). Þar sem um er að ræða hrein föll þá breyta

þau ekki viðföngunum og því hefði ég alveg eins getað sent þau sem gildi (e. by value).

Kosturinn við að senda viðföng sem gildi er að fallið sem kallað er á og sá sem kallar eru hjúpuð á viðeigandi hátt – breyting í öðrum þeirra getur ekki leitt til breytingar í hinum, nema í tengslum við skilagildið.

Á hinn bóginn er yfirleitt skilvirkara að senda viðföng með tilvísun vegna þess að þá þarf ekki að afrita nein gildi úr viðföngunum yfir í leppana. Svo vill líka til að C++ inniheldur eiginleika sem kallaður er `const` og gerir það að verkum að það er jafn öruggt að senda tilvísunarviðföng eins og gildisviðföng.

Ef þú skrifar fall og ætlar ekki að breyta viðfangi þá getur þú skilgreint leppinn sem **constant reference parameter** (fast tilvísunarviðfang). Málskipanin lítur svona út:

```
void printTime (const Time& time) ...
Time addTime (const Time& t1, const Time& t2) ...
```

Hér sýni ég aðeins fyrstu línuna í föllunum. Ef þú lætur þýðandann vita að þú ætlir ekki að breyta viðfangi í falli þá getur hann minnt þig á það! Ef þú reynir að breyta viðfangi þá mun þýðandinn kvarta.

9.6 Breytiföll

Auðvitað vill svo til stundum að þú vilt einmitt breyta viðfangi. Fall sem gerir það eru kallað breytir (e. modifier).

Skoðum fallið `increment` sem dæmi um breyti en það bætir tilteknum fjölda sekúndna við `Time` hlut. Gróf drög fyrir þetta fall gæti litið svona út:

```
void increment (Time& time, double secs) {
  time.second += secs;

  if (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  if (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

Fyrsta línan framkvæmir grunnaðgerð en það sem á eftir kemur sér um sérstök tilfelli, eins og við sáum áður.

Er þetta fall rétt? Hvað gerist ef leppurinn `secs` er mikið stærri en 60? Í því tilfelli er ekki nóg að draga 60 frá einu sinni – við þurfum að gera það þangað til `second` er lægra en 60.

Við getum gert það með því að skipta `if` setningum út fyrir `while` setningar:

```
void increment (Time& time, double secs) {
  time.second += secs;

  while (time.second >= 60.0) {
    time.second -= 60.0;
    time.minute += 1;
  }
  while (time.minute >= 60) {
    time.minute -= 60;
    time.hour += 1;
  }
}
```

Þessi lausn er rétt en ekki mjög skilvirk. Getur þú séð fyrir þér lausn sem þarfnast ekki ítrunar?

9.7 Fylliföll

Af og til sérð þú fall eins og `addTime` skrifað með því að nota önnur skil (e. interface), þ.e. önnur viðföng og annað skilagildi. Í stað þess að búa til nýjan hlut í sérhvert sinn sem kallað er á `addTime` þá gætum við krafist þess að sá sem kallar útvegi "tóman" hlut sem `addTime` getur sett upplýsingar inn í. Berðu saman eftirfarandi útgáfu og þá fyrri:

```
void addTimeFill (const Time& t1, const Time& t2, Time& sum) {
  sum.hour = t1.hour + t2.hour;
  sum.minute = t1.minute + t2.minute;
  sum.second = t1.second + t2.second;

  if (sum.second >= 60.0) {
    sum.second -= 60.0;
    sum.minute += 1;
  }
  if (sum.minute >= 60) {
    sum.minute -= 60;
    sum.hour += 1;
  }
}
```

Kosturinn við þessa útfærslu er að sá sem kallar hefur möguleika á að endurnýta sama hlutinn aftur og aftur til að framkvæma röð af samlagningum á `Time` hlutum. Þetta getur verið aðeins skilvirkara þó að þetta geti verið ruglandi og valdið lúmskum villum. Fyrir flest forritunarverkekfni getur verið gott að eyða meiri keyrslutíma til að koma í veg fyrir langan kembitíma síðar meir.

Athugaðu að hægt er að skilgreina fyrstu tvo leppana sem `const` en ekki þann þriðja.

9.8 Hvað er best?

Það sem hægt er að gera með breyti eða fylli er líka hægt að gera með hreinu falli. Það vill reyndar svo til að ákveðin tegund forritunarmála, **fallaforritunarmál**, leyfa eingöngu hrein föll. Sumir forritarar trúa því að fljótlegra sé að skrifa forrit sem nota eingöngu hrein föll og þau séu líka áreiðanlegri (innihaldi færri villur) en þau sem nota breytiföll. Aftur á móti geta breytiföll stundum verið hentug og tilvik koma upp þar sem fallaforrit eru ekki eins skilvirk.

Almennt séð mæli ég með því að þú skrifir hrein föll þar sem það liggur beint við og notir aðeins breytiföll ef það er augljós kostur. Þetta viðhorf mætti kalla fallaforritunarstíl (e. functional programming style).

9.9 Stigvaxandi þróun vs. áætlunargerð

Ég hef í þessum kafla sýnt dæmi um aðferð við forritunarþróun sem leiðir af sér **skjóta frumgerð með stigvaxandi endurbótum** (e. rapid prototyping with iterative improvement). Í sérhverju tilviki skrifaði ég drög (frumgerð) sem framkvæmdi grunnútreikninga, síðan prófaði ég þá á nokkrum tilvikum og leiðrétti villur þegar þær komu í ljós.

Þó svo að þessi aðferð geti verið markvirk (e. effective) þá getur kóðinn orðið óþarflega flókinn, vegna þess að hann meðhöndlar mörg mismunandi tilvik, og óáreiðanlegur, vegna þess að það getur verið erfitt að fullvissa sig um að maður hafi fundið allar villurnar.

Önnur aðferð er hönnun eða áætlunargerð sem felur í sér að smá innsýn í vandamálið getur gert forritunina mikið einfaldari. Í þessu tilviki er innsýnin sú að `Time` er í raun tala með þremur tölustöfum með grunn 60! Sekúndan (`second`) er "ones column", mínútan (`minute`) er "60's column", og klukkustundin (`hour`) er "3600's column".

Þegar við skrifuðum `addTime` og `increment` þá gerðum við í raun samlagningu með grunn 60, sem er einmitt ástæðan fyrir því að við þurftum að "færa" frá einum dálki yfir í annan.

Önnur leið til að leysa vandamálið er því að breyta `Time` hlutum yfir í `double` breytur og nýta sér það að tölvan veit þegar hvernig gera á útreikninga með `double`. Hér er fall sem breytir `Time` yfir í `double`:

```
double convertToSeconds (const Time& t) {
  int minutes = t.hour * 60 + t.minute;
  double seconds = minutes * 60 + t.second;
  return seconds;
}
```

Allt sem við þurfum þá til viðbótar er leið til að breyta `double` yfir í `Time` hlut:

```
Time makeTime (double secs) {
  Time time;
  time.hour = int (secs / 3600.0);
```

```
  secs -= time.hour * 3600.0;
  time.minute = int (secs / 60.0);
  secs -= time.minute * 60;
  time.second = secs;
  return time;
}
```

Þú þarft líklega að hugsa dálítið um þetta til að fullvissa þig um að aðferðin sem ég nota hér til að breyta úr einum grunn yfir í annan sé rétt. Að því gefnu að þú sért sannfærð(ur) þá getum við notað þessi föll til að endurskrifa `addTime`:

```
Time addTime (const Time& t1, const Time& t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return makeTime (seconds);
}
```

Þessi útgáfa er miklu styttri en upphaflega útgáfan og það er líka mun einfaldara að sýna að hún sé rétt. Þú ættir núna að æfa þig með því að endurskrifa `increment` á sama hátt.

9.10 Alhæfing

Á vissan hátt er erfiðara að breyta úr grunni 60 yfir í grunn 10, og til baka, en að meðhöndla tíma. Breyting á grunni er meira abstrakt, tilfinning okkar fyrir tíma er meiri.

Á hinn bóginn má segja að ef við meðhöndlum tíma sem tölur með grunn 60 og skrifum umbreytingaföllin (`convertToSeconds` og `makeTime`) þá endum við með forrit sem er styttra, læsilegra og einfaldara að kemba og jafnframt áreiðanlegra.

Það er jafnframt auðveldara að bæta eiginleikum við forritið síðar. Gefum okkur t.d. að við þurfum að geta dregið einn tíma frá öðrum til að finna tímann á milli þeirra. Bein leið væri að útfæra frádráttinn með "láni" (e. borrowing). Það væri hins vegar auðveldara að nota umbreytingaföllin og jafnframt líklegra til að vera rétt.

Það er kaldhæðnislegt að stundum verður vandamál auðveldara (færri sérstök tilvik og færri möguleikar á villum) ef það er gert erfiðara (almennara)!

9.11 Reiknirit

Þegar þú skrifar almenna lausn fyrir safn af vandamálum, í stað þess að skrifa sérstaka lausn fyrir eitt tiltekið vandamál, þá hefur þú útfært **reiknirit** (e. algorithm). Ég nefndi þetta orð í kafla 1 en skilgreindi það ekki nákvæmlega. Það er reyndar ekki auðvelt að skilgreina en hér mun ég reyna það með tveimur aðferðum.

Í fyrsta lagi skaltu hugsa um eitthvað sem er ekki reiknirit. Þegar þú lærðir t.d. að margfalda saman tölur með einum tölustaf þá lærðir þú væntanlega

margföldunartöfluna utan að. Í rauninni settir þú 100 mismunandi lausnir á minnið! Þess konar þekking er í raun ekki reikniritanleg.

En ef þú varst "löt/latur" þá svindlaðir þú líklega og lærðir nokkur trikk. Til að finna t.d. margfeldið af n and 9 getur þú skrifað $n - 1$ sem fyrri tölustafinn og $10 - n$ seinni stafinn. Þetta trikk er almenn lausn til að margfalda tölu (með einum tölustaf) með 9. Þetta er reiknirit!

Á sama hátt eru aðferðirnar sem þú lærðir til að leggja saman, með því færa yfir, og að draga frá, með því að taka að láni, líka reiknirit. Eitt sem einkennir reiknirit er að þau krefjast ekki neinnar sérstakar greindar til að framkvæma þau. Þau eru sjálfvirk ferli þar sem sérhvert skref er framkvæmt á eftir undanfarandi skrefi í samræmi við einfalt mengi af reglum.

Það er mín skoðun að það sé í raun vandræðalegt hversu löngum tíma við eyðum í skóla að keyra reiknirit sem þarfnast í raun engrar greindar.

Á hinn bóginn er ferlið við að hanna reiknirit mjög áhugavert og krefjandi og í raun hryggjarstykkið í því sem við köllum forritun.

Sumt af því sem manneskjan gerir á náttúrulegan hátt, án vandræða eða meðvitaðrar hugsunar, er hvað erfiðast að tjá með reikniriti. Það að greina og skilja náttúrulegt tungumál er gott dæmi. Við gerum það öll en hingað til hefur engum tekist að skýra út *hvernig* við gerum það, a.m.k. ekki í formi reiknirits.

Seinna í þessari bók færðu tækifæri til að skrifa einföld reiknirit fyrir ýmis vandamál. Ef þú ert í tölvunarfræðinámi þá muntu síðar taka námskeiðið Reiknirit (e. Data Structures) og kynnast mörgum áhugaverðum, snjöllum og gagnlegum reikniritum sem tölvunarfræðingar hafa þróað.

9.12 Orðalisti

tilvik (e.instance): Eitt dæmi úr tilteknum flokki. Kötturinn minn er tilvik úr flokknum "læður". Sérhver hlutur (e. object) er tilvik af einhverju tagi.

tilvikabreyta (e. instance variable): Ein af þeim breytum sem mynda strúktúr. Sérhvert tilvik af strúktúr á sér eigin afrit af tilvikabreytunum fyrir viðkomandi tag.

fast tilvísunarviðfang (e. constant reference parameter): Viðfang sem er sent sem tilvísun í fall en er samt sem áður ekki hægt að breyta í fallinu.

hreint fall (e. pure function): Fall, hvers skilagildi er eingöngu háð viðföngum þess, og sem hefur engar aukaverkanir en þær að skila gildi.

fallaforritunarstíll (e. functional programming style): Forritunarstíll sem leggur áherslu á að meginhluti falla séu hrein.

breytir (e. modifier): Fall sem breytir einu eða fleiri viðföngum og skilar yfirleitt `void`.

fyllir (e. fill-in function): Fall sem tekur "tóman" hlut sem viðfang og fyllir inn í tilvikabreytur hlutarins í stað þess að skila gildi.

reiknirit (e. algorithm): Eins konar uppskrift til að leysa tiltekna tegund af vandamálum með sjálfvirku ferli.

Kafli 10

Vektorar

Vektor (e. vector) er mengi gilda þar sem hvert þeirra er auðkennt með tölu sem kölluð er vísir (e. index). Strengur (`string`) er svipaður og vektor því hann samanstendur af mengi af stöfum sem hægt er að vísa í með tölu. Vektorar í C++ eru hentugir í notkun því þeir geta geymt ýmiss konar gögn, þ.m.t. grunntög eins og `int` og `double`, eða tög sem skilgreind eru af notanda, eins og `Point` og `Time`.

Tagið `vector` er skilgreint í C++ "Standard Template Library" (STL). Til að nota það þarf að taka inn (e. include) hausaskrána `vector` – hvernig það er gert er háð þínu forritunarumhverfi.

Þú getur búið til vektor á sama hátt og þú býrð til breytu af hvaða öðru tagi:

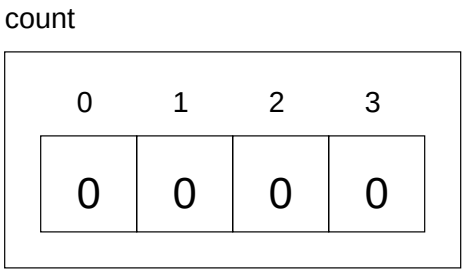
```
vector<int> count;
vector<double> doubleVector;
```

Tagið sem vektorinn mun geyma kemur fram í hornsvigunum (e. angle brackets) (< og >). Fyrri línan býr til vektor af heiltölum með nafnið `count` en seinni línan býr til vektor af `double`. Þrátt fyrir að þessar setningar séu löglegar þá eru þær ekki mjög gagnlegar vegna þess að þær búa til vektora sem hafa engin stök (stærð þeirra er núll). Það er því mun algengara að skilgreina stærð vektorsins innan sviga:

```
vector<int> count (4);
```

Málskipanin er hér dálítið skrýtin því hún lítur út eins og samsetning breytuyfirlýsingar og fallakalls. Það er reyndar nákvæmlega það sem hún er! Fallið sem við erum að kalla á er svokallaður smiður í `vector` klasanum. **Smiður** (e. constructor) er sérstakt fall sem býr til nýtt tilvik og frumstillir tilvikabreytur þess. Í þessu tilviki tekur smiðurinn eitt viðfang sem er stærð vektorsins.

Eftirfarandi mynd sýnir hvernig vektorar eru táknaðir á stöðuritum:



Stóru tölurnar innan í boxunum eru **stök** (e. elements) vektorsins. Litlu tölurnar fyrir utan boxin eru vísarnir (e. indices) sem notaðir eru til að einkenna sérhvert box. Stök vektors eru ekki frumstillt þegar nýr vektor er búinn til og því gætu þau gætu í raun innihaldið hvaða gildi sem er.

Það er til annar smiður fyrir `vector` sem tekur tvö viðföng; seinna viðfangið er "frumstillingargildi", þ.e. gildið sem sérhvert stak vektorsins fær í upphafi.

```
vector<int> count (4, 0);
```

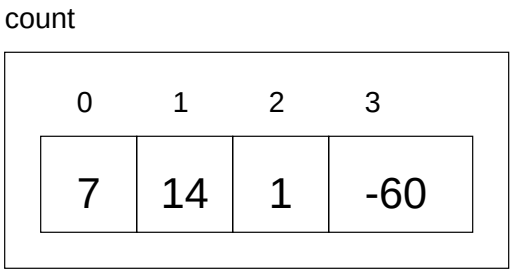
Þessi setning býr til vektor með fjórum stökum og frumstillir öll stökin með núlli.

10.1 Aðgangur að stökum

Virkjann `[]` er hægt að nota til að lesa og skrifa stök vektors á sambærilegan hátt og gert er með strengi. Vísarnir byrja í núlli, þannig að `count[0]` vísar til "núllta" staks vektorsins `count` og `count[1]` vísar til "fyrsta" staksins. Þú getur notað virkjann `[]` hvar sem er í segð:

```
count[0] = 7;
count[1] = count[0] * 2;
count[2]++;
count[3] -= 60;
```

Allar þessar setningar eru löglegar og áhrif þessa kóða á vektorinn er:



Ekkert stak er með vísinn 4 þar sem stök þessa vektors eru númeruð frá 0 to 3. Það er algeng villa að reyna að vísa "út fyrir" vektor og það veldur

keyrsluvillu. Í því tilfelli skrifar forritið út villuskilaboð eins og "Illegal vector index", og hættir síðan keyrslu.

Þú getur notað hvaða segð sem er sem vísi svo framarlega sem hún hefur tagið `int`. Ein algengasta leiðin til að vísa í stök vektors er að nota lykkjubreytu (e. loop variable). Dæmi:

```
int i = 0;
while (i < 4) {
  cout << count[i] << endl;
  i++;
}
```

Þessi `while` lykkja "hleypur" frá 0 í 4. Þegar lykkjubreytan `i` fær gildið 4 þá verður skilyrðið `false` og lykkjan hættir. Meginmál lykkjunnar er því eingöngu keyrt þegar `i` er 0, 1, 2 og 3.

Í sérhverri ítrun lykkjunnar notum við `i` sem vísi inn í vektorinn og skrifum út `i`-ta stakið. Þessi tegund af "vektorrölti" er mjög algeng enda vinna vektorar og lykkjur vel saman.

10.2 Afritun vektora

Það er til einn smiður í viðbót fyrir `vector` sem kallaður er "afritatökusmiður" (e. copy constructor) vegna þess að hann tekur einn `vector` sem viðfang og býr til nýjan vektor af sömu stærð og með sömu stök.

```
vector<int> copy (count);
```

Þrátt fyrir að þessi málskipan sé lögleg þá er hún sjaldan notuð fyrir vektorar því til er betri leið:

```
vector<int> copy = count;
```

Gildisveitingarvirkinn `=` virkar fyrir vektorar á þann hátt sem ætla má.

10.3 `for` lykkjur

Lykkjurnar sem við höfum skrifað hingað til eiga ýmislegt sameiginlegt. Allar byrja þær á því að frumstilla breytur. Þær innihalda jafnframt skilyrði sem er háð breytunni og meginmál lykkjunnar gerir eitthvað við breytuna, eins og að hækka hana.

Þessi tegund af lykkju er svo algeng að það er til annars konar lykkjusetning, kölluð `for`-setning, sem tjáir þetta á samþjappaðri hátt. Málskipanin lítur svona út:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
  BODY
}
```

Ofangreind setning er nákvæmlega jafngild þessu:

```
INITIALIZER;
while (CONDITION) {
  BODY
  INCREMENTOR
}
```

`for`-setningin er aftur á móti gagnyrtari og læsilegri því í henni eru allar lykkjusetningarnar á einum stað. Dæmi:

```
int i;
for (i = 0; i < 4; i++) {
  cout << count[i] << endl;
}
```

er jafngilt:

```
int i = 0;
while (i < 4) {
  cout << count[i] << endl;
  i++;
}
```

10.4 Stærð vektors

Það eru margvísleg föll sem hægt er að keyra fyrir `vector`. Eitt af þeim er þó sérstaklega gagnlegt: `size()`. Þetta fall skilar stærð vektors, þ.e. fjölda staka.

Það er góð regla að nota þetta gildi sem efri mörk (e. upper bound) lykkju frekar heldur en einhvern fasta. Þannig þarf ekki að breyta lykkjunni ef stærð vektorsins breytist.

```
int i;
for (i = 0; i < count.size(); i++) {
  cout << count[i] << endl;
}
```

Í síðasta skiptið sem meginmál lykkjunnar er keyrt er gildið á `i` jafnt `count.size() - 1`, sem er vísirinn á síðasta stakinu. Þegar `i` er jafnt `count.size()` mun skilyrðið verða `false` og meginmál lykkjunnar verður því ekki keyrt lengur. Það er gott því annars myndi keyrsluvilla koma upp!

Taktu eftir því að kallað er á `size()` fallið í hverri ítrun lykkjunnar. Það að kalla á fall aftur og aftur hefur áhrif á keyrslutímann þannig að í raun væri betra að geyma stærð vektorsins í einhverri breytu með því að kalla á `size()` áður en lykkjan hefst og nota síðan breytuna til að tékka á síðasta gildinu. Þú ættir að æfa þig með því að prófa að gera þessa breytingu.

10.5 Vektorföll

Einn besti eiginleiki vektors er geta hans til að stækka eða minnka ef þurfa þykir. Eftir að vektor hefur verið búinn til þá er hægt að stækka hann eða minnka hvar sem er í forritinu. Gerum t.d. ráð fyrir að við lesum inn tölur frá notanda inn í vektor þangað til notandi slær inn `-1` en þá skrifum við tölurnar út. Í svona tilviki vitum við ekki stærðina á vektornum fyrirfram. Við þurfum því að geta bætt nýjum stökum við enda vektorsins um leið og notandinn slær inn ný gildi. Við getum notað fallið `push_back()` í þessum tilgangi:

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
  vector<int> values;
  int c,i,len;
  cin>>c;

  while(c != -1) {
    values.push_back(c);
    cin >> c;
  }
  len=values.size();
  for(i = 0; i < len; i++) {
    cout << values[i] << endl;
  }
}
```

10.6 Slembitölur

Flestar tölvur gera það sama í hvert sinn sem þær eru keyrðar með sama forritinu og eru kallaðar **löggengar** (e. deterministic). Löggengni er yfirleitt kostur því við viljum jú að sömu útreikningar skili alltaf af sér sömu niðurstöðu. Í sumum tilvikum gætum við aftur á móti viljað að tölvur væru óútreiknanlegar. Tölvuleikir eru gott dæmi um þetta.

Það að gera forrit algerlega **brigðgengt** (e. nondeterministic) er ekki auðvelt en það eru til leiðir til að láta það líta svo út. Ein leið er að búa til hermislembitölur (e. pseudo random numbers) og nota þær til að stýra útkomunni úr forriti. Hermislembitölur eru ekki algerlega handahófskenndar í stærðfræðilegum skilningi en þær duga fyrir það sem við ætlum að gera.

Í hausaskránni `cstdlib` (sem inniheldur ýmis konar "standard library" föll) er fallið `random` skilgreint en það býr til hermislembitölur.

Skilagildið úr `random` er heiltala á milli 0 og `RAND_MAX` en `RAND_MAX` er stór tala (um það bil 2 milljarðar á minni tölvu) sem einnig er skilgreind í hausa-

skránni. Í sérhvert sinn sem þú kallar á `random` færðu nýja slembitölu. Til að sjá dæmi um þetta skaltu keyra eftirfarandi lykkju:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main ()
{
  for (int i = 0; i < 4; i++) {
    int x = random ();
    cout << x << endl;
  }
  return 0;
}
```

Ég fæ eftirfarandi úttak á minni vél:

```
1804289383
846930886
1681692777
1714636915
```

Þú færð væntanlega eitthvað annað, en þó svipað, á þinni tölvu.

Auðvitað viljum við ekki alltaf vinna með svona stórar heiltölur. Það er algengara að búa til heiltölur á milli 0 og einhvers efri marks. Einföld leið til að gera það er að nota "modulus" virkjann: Dæmi:

```
  int x = random ();
  int y = x % upperBound;
```

Þar sem `y` er afgangurinn sem fæst með því að deila `x` með `upperBound` þá liggja gildin fyrir `y` á bilinu 0 og `upperBound - 1` (að báðum meðtöldum). Hafðu í huga að `y` er aldrei jafnt og `upperBound`.

Það er einnig oft gagnlegt að búa til slembikommutölur. Algeng leið til að gera það er að deila með `RAND_MAX`. Dæmi:

```
  int x = random ();
  double y = double(x) / RAND_MAX;
```

Þessi kóði gefur `y` gildi slembikommutölu á milli 0,0 and 1,0 (að báðum meðtöldum). Þú ættir núna að íhuga hvernig hægt er að búa til slembikommutölu á ákveðnu bili, t.d. á milli 100,0 og 200,0.

10.7 Tölfræði

Tölurnar sem `random` fallið býr til eiga að dreifast jafnt. Það þýðir að sérhvert gildi á hinu valda bili ætti að vera jafn líklegt. Fjöldinn af sérhverju gildi ætti að vera nokkurn veginn sá sami að því gefnu að við látum `random` búa til nógu mörg gildi fyrir okkur.

Hér á eftir munum við skrifa forrit sem mynda röð af slembitölum og athugar hvort þessi staðhæfing sé rétt.

10.8 Vektor af slembitölum

Fyrsta skrefið er að búa til mikinn fjölda af slembitölum og geyma þær í vektor. Með "mikinn fjölda" á ég auðvitað við 20! Það er góð regla að byrja með lítinn fjölda (sem auðveldar kembun) og fjölga tölum síðar.

Eftirfarandi fall tekur eitt viðfang, stærð vektors. Það úthlutar minni fyrir nýjan vektor af heiltölum og fyllir hann með slembitölum á bilinu 0 og `upperBound-1`.

```
vector<int> randomVector (int n, int upperBound) {
  vector<int> vec (n);
  for (int i = 0; i<vec.size(); i++) {
    vec[i] = random () % upperBound;
  }
  return vec;
}
```

Skilagildið er `vector<int>`, þ.e. vektor af heiltölum. Til að prófa þetta fall er þægilegt að hafa fall sem skrifar út innihalds vektors:

```
void printVector (const vector<int>& vec) {
  for (int i = 0; i<vec.size(); i++) {
    cout << vec[i] << " ";
  }
}
```

Taktu eftir því að það er löglegt að senda `vector` með tilvísun. Það er einmitt mjög algengt því þá þarf ekki að afrita öll stök vektorsins (eins og gera þyrfti ef vektorinn væri sendur sem gildi). Við skilgreinum leppinn sem `const` þar sem `printVector` breytir ekki viðfangi sínu.

Eftirfarandi kóði býr til vektor og skrifar innihald hans út:

```
  int numValues = 20;
  int upperBound = 10;
  vector<int> vector = randomVector (numValues, upperBound);
  printVector (vector);
```

Á minni vél er úttakið

```
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6
```

sem virðist vera nokkuð handahófskennt. Þitt úttak er væntanlega öðruvísi.

Ef þessar tölur eru raunverulegar slembitölur þá megum við gera ráð fyrir því að sérhver tala komi jafn oft fyrir, þ.e. tvisvar sinnum. Það vill reyndar svo til að talan 6 kemur fyrir fimm sinnum og tölurnar 4 og 8 koma ekki fyrir.

En þýðir þetta þá að gildin fylgi ekki jafnri dreifingu? Það er erfitt að segja til um því þegar gildin eru svona fá þá eru afskaplega litlar líkur á því að fá nákvæmlega það sem við eigum von á. Þegar gildum fjölgar þá ætti útkoman, aftur á móti, að vera fyrirsjáanlegri.

Til að prófa þessa tilgátu munum við skrifa forrit sem telur hversu oft sérhvert gildi kemur fyrir og síðan athuga hvað gerist þegar við hækkum `numValues`.

10.9 Talning

Góð leið til að leysa vandamál eins og þetta er að skrifa einföld föll sem er auðvelt að skrifa og gera má ráð fyrir að verði gagnleg. Þá er hægt að skrifa heildarlausnina með því að nýta sér þessi einföldu föll. Þessi forritunaraðferð er stundum kölluð **neðansækin hönnun** (e. bottom-up deisgn). Auðvitað er það svo að það er ekki auðvelt að vita fyrirfram hvaða föll koma til með að verða gagnleg en eftir því sem reynslan eykst því auðveldara verður fyrir þig að átta þig á því.

Að auki er ekki alltaf augljóst hvers konar föll er auðvelt að skrifa en það er góð leið að leita að hlutvandamálum sem passa við eitthvað mynstur sem þú hefur séð áður.

Í kafla 7.9 skoðuðum við lykkju sem "ferðaðist um" í streng og taldi hversu oft tiltekinn stafur kom fyrir í strengnum. Þú getur litið á það forrit sem dæmi um mynstur sem kallast "ferðast um og telja". Einstakir hlutar þessa mynsturs eru:

- Mengi eða gámur (e. container) sem hægt er að ferðast um í, t.d. strengur eða vektor.
- Próf (e. test) sem hægt er að beita á sérhvert stak í gámnum.
- Teljari sem heldur utan um hversu mörg stök standast prófið.

Í þessu tilviki hef ég fall í huga sem ég kalla `howMany` sem telur fjölda staka í vektor sem eru jöfn tilteknu gildi. Viðföngin í fallið eru vektor og heiltalan sem leitað er að. Skilagildið er gildi sem segir til um hversu oft heiltalan fannst í vektornum.

```
int howMany (const vector<int>& vec, int value) {
  int count = 0;
  for (int i=0; i< vec.size(); i++) {
    if (vec[i] == value) count++;
  }
```

```
  return count;
}
```

10.10 Athugun á öðrum gildum

`howMany` telur aðeins fjölda tilvika á af tilteknu gildi í vektornum en við höfum áhuga á hversu oft sérhvert gildi kemur fyrir. Það getum við leyst með því að nota lykkju:

```
  int numValues = 20;
  int upperBound = 10;
  vector<int> vector = randomVector (numValues, upperBound);

  cout << "value\thowMany" << endl;

  for (int i = 0; i<upperBound; i++) {
    cout << i << '\t' << howMany (vector, i) << endl;
  }
```

Taktu eftir því að það er löglegt að skilgreina breytu innan í `for`-setningu. Þessi málskipan er stundum hentug en athugaðu að breyta sem skilgreind er innan í lykkju er eingöngu lifandi í lykkjunni. Þú færð villu frá þýðandanum ef þú reynir að vísa í `i` utan lykkjunnar.

Þessi kóði sendir lykkjubreytuna sem viðfang í `howMany` í þeim tilgangi að tékka á sérhverju gildi á milli 0 og 9 í réttri röð. Niðurstaðan er:

```
value   howMany
0       2
1       1
2       3
3       3
4       0
5       2
6       5
7       2
8       0
9       2
```

Það er erfitt að segja hvort tölurnar birtast í raun jafn oft. Ef við hækkum hins vegar `numValues` í 100.000 þá fæst:

```
value   howMany
0       10130
1       10072
2       9990
3       9842
```

```
4       10174
5       9930
6       10059
7       9954
8       9891
9       9958
```

Í sérhverju tilviki er fjöldi tilvika innan við 1% af væntu gildi (10.000) þannig að við getum ályktað sem svo að tölurnar dreifist líklega jafnt.

10.11 Súlurit

Það er oft gagnlegt að geyma gögn, eins og úr töflunni hér að ofan, og sækja þau síðar í stað þess að prenta þau bara út. Til þess þurfum við einhverja leið til að geyma 10 heiltölur. Við gætum búið til 10 heiltölubreytur með nöfn eins og `howManyOnes`, `howManyTwos`, o.s.frv. Þá þyrftum við hins vegar að slá mikið inn og það yrði erfitt fyrir okkur að breyta forritinu ef bilið (nú 0-9) breytist.

Miklu betri lausn er að nota vektor af stærðinni 10. Þannig getum við búið til minnið fyrir allar 10 heiltölurnar í einu og getum nálgast þær með því að nota vísa (e. indices) í stað 10 mismunandi breytunafna. Hér er dæmi um þetta:

```
int numValues = 100000;
int upperBound = 10;
vector<int> vector = randomVector (numValues, upperBound);
vector<int> histogram (upperBound);

for (int i = 0; i<upperBound; i++) {
  int count = howMany (vector, i);
  histogram[i] = count;
}
```

Ég kallaði vektorinn **histogram** (súlurit) því það er tölfræðilegt hugtak fyrir vektor af tölum sem heldur utan um fjölda tilvika af gildum á ákveðnu bili.

Taktu eftir að hér nota ég lykkjubreytuna á tvo mismunandi vegu. Í fyrsta lagi sem viðfang í `howMany` fallið til að tilgreina hvaða gildi ég hef áhuga á í hverri ítrun. Í öðru lagi er breytan notuð sem vísir inn í súluritið, þ.e. til að tilgreina hvar (í súluritinu) geyma eigi fjöldann.

10.12 Skilvirkari lausn

Þrátt fyrir að þessi kóði virki þá er hann ekki eins skilvirkur eins og hann gæti verið. Ferðast er um í öllum vektornum í hvert sinn sem kallað er á `howMany`. Í þessu dæmi er því ferðast um í vektornum 10 sinnum!

Það væri miklu betra að renna aðeins einu sinni (e. make a single pass) í gegnum vektorinn. Fyrir sérhvert gildi í vektornum gætum við fundið samsvarandi teljari og hækkað hann. Við getum, m.ö.o., notað gildið í vektornum sem vísi inn í súluritið. Svona myndi þetta líta út:

```
vector<int> histogram (upperBound, 0);

for (int i = 0; i<numValues; i++) {
  int index = vector[i];
  histogram[index]++;
}
```

Fyrsta línan frumstillir stök súluritsins með núlli. Það þýðir að þegar við notum virkjann ++ innan í lykkjunni þá vitum við að byrjað er í 0. Það er einmitt algeng villa að gleyma að frumstilla með núlli.

Þú ættir núna að hjúpa þennan kóða í fall sem kallað er `histogram`. Fallið tekur vektor og bil (í þessu tilviki 0-9) sem viðföng og skilar súluriti yfir gildin í vektornum.

10.13 Slembifræ

Ef þú hefur keyrt kóðann í þessum kafla nokkrum sinnum þá hefur þú kannski tekið eftir því að þú hefur fengið sömu "slembitölurnar" aftur og aftur. Það er ekki mjög handahófskennt!

Einn af eiginleikum hermislembitalna er að ef byrjað er á sama stað þá myndast alltaf sama talnaröðin. Byrjunarpunkturinn er kallað "fræ" (e. seed) og C++ notar allaf sama fræið í hvert sinn sem þú keyrir forritið.

Það getur oft verið gagnlegt að sjá sömu niðurstöðu (talnaröðina) aftur og aftur á meðan þú ert að kemba forrit. Þannig getur þú auðveldlega séð hvort breyting á forritinu þínu hefur í raun áhrif á úttak þess.

Ef þú vilt hins vegar nota mismunandi fræ fyrir myndun slembitalna þá getur þú notað `srand` fallið. Það tekur eitt viðfang sem er heiltala á milli 0 og `RAND_MAX`.

Í mörgum forritum, t.d. leikjum, er þörf á því að fá mismunandi slembitölurunur í hvert sinn sem forritið er keyrt. Algeng leið til að gera það er að nota fall eins og `gettimeofday` sem býr til eitthvað sem er tiltölulega ófyrrisjáanlegt og ekki auðvelt að endurtaka, eins og fjöldi millisekúndna síðan kallað var síðast, og nota það gildi sem fræ.

10.14 Orðalisti

vektor (e. vector): Safn af gildum sem öll hafa sama tag. Sérhvert gildi er einkennt með vísi (e. index).

stak (e. element): Eitt af gildunum í vektor. Virkinn `[]` velur stök í vektor.

vísir (e. index): Heiltölubreyta eða gildi sem notað er til að einkenna stak í vektor.

smiður (e. constructor): Sérstakt fall sem býr til nýtt tilvik og frumstillir tilvikabreytur þess.

löggengt forrit (e. deterministic program): Forrit sem gerir það sama í hvert skipti sem það er keyrt.

hermislembitöluruna (e. pseudo random sequence): Runa af tölum sem virðist vera handahófskennd en er í raun niðurstaðan af löggengum útreikningum.

fræ (e. seed): Gildi sem notað er til að frumstilla slembitölurunu. Ef sama fræ er notað þá ætti að fást sama slembitöluruna.

neðansækin hönnun (e. bottom-up design): Forritunaraðferð sem byggir á því að skrifa fyrst lítil, gagnleg föll, sem síðan eru nýtt í stærri heildarlausn.

súlurit (e. histogram): Vektor af heiltölugildum þar sem hver tala stendur fyrir fjölda gilda á ákveðnu bili.

Kafli 11

Meðlimaföll

11.1 Hlutir og föll

C++ er hlutbundið forritunarmál sem merkir að það hefur eiginleika sem styðja við hlutbundna forritun.

Það er ekki auðvelt að skilgreina hlutbundna forritun en við höfum þó þegar séð suma eiginleika hennar:

1. Forrit samanstanda af safni af strúktúrskilgreiningum og fallaskilgreiningum þar sem flest föllin vinna með tiltekna tegundur af strúktúrum (hlutum).

2. Sérhver strúktúrskilgreining stendur fyrir einhvern hlut eða hugtak í raunveruleikanum og föllin sem framkvæma aðgerðir á viðkomandi strúktúr líkja eftir því hvernig raunverulegir hlutir eiga samskipti.

`Time` strúktúrinn, sem við skilgreindum í kafla 9, samsvarar því hvernig fólk skráir tíma innan dagsins og aðgerðirnar sem við skilgreindum samsvara því hvað fólk gerir við hugtakið tími.

Á sambærilegan hátt má segja að `Point` og `Rectangle` hlutirnir samsvari stærðfræðilegu hugtökunum hnit og rétthyrningur.

Hingað til höfum við ekki nýtt okkur þá eiginleika C++ sem styðja við hlutbundna forritun. Strangt til tekið eru þessir eiginleikar reyndar ekki nauðsynlegir. Að mestu leyti gera þeir forriturum kleift að nota aðra málskipan til að framkvæma eitthvað sem við höfum þegar gert áður en í mörgum tilvikum er þessi nýja málskipan samanþjappaðri (e. more concise) og sýnir á nákvæmari hátt högun forritsins.

Í `Time` forritinu er t.d. ekkert augljóst samband á milli strúktúrskilgreiningarinnar og fallanna sem á eftir koma. Við nánari athugun er hins vegar ljóst að sérhvert fallanna tekur a.m.k. einn `Time` hlut sem viðfang og það leiðir okkur að hugtakinu **meðlimafall** (e. member function). Meðlimaföll eru ólík öðrum föllum sem við höfum skrifað á tvennan hátt:

1. Þegar við köllum á meðlimafall þá tengjum við kallið við tiltekinn hlut (e. object). Stundum er sagt að kall á meðlimafall hafi í för með sér aðgerð á hlut (e. operation on an object) eða verið sé að senda boð til hlutar (e. sending a message to an object).

2. *Yfirlýsing* (e. declaration) meðlimafallsins á sér stað innan `struct` skilgreiningar í þeim tilgangi að gera sambandið á milli strúktúrsins og fallsins skýrt (e. explicit).

Í því sem á eftir kemur munum við taka föllin úr kafla 9 og umbreyta þeim í meðlimaföll. Athugaðu að þessi umbreyting er algerlega vélræn, m.ö.o. hægt er að framkvæma hana með því að fylgja ákveðinni forskrift.

Eins og ég nefndi að ofan þá getur allt það sem hægt er að gera með meðlimafalli líka verið gert með falli sem stendur eitt og sér (e. nonmember function / free-standing function). Oft eru hins vegar kostir/ókostir sem fylgja notkun annars í stað hins. Ef þú getur á auðveldan hátt umbreytt öðru forminu yfir í hitt þá munt þú geta valið betri leiðina í því forriti sem þú vinnur að sérhverju sinni.

11.2 `print`

Í kafla 9 skilgreindum við strúktúrinn `Time` og skrifuðum fallið `printTime`:

```
struct Time {
  int hour, minute;
  double second;
};

void printTime (const Time& time) {
  cout << time.hour << ":" << time.minute << ":" << time.second << endl;
}
```

Til að kalla á þetta fall þurftum við að senda `Time` hlut sem viðfang.

```
  Time currentTime = { 9, 14, 30.0 };
  printTime (currentTime);
```

Fyrsta skrefið í áttina að gera `printTime` að meðlimafalli er að breyta nafninu á fallinu úr `printTime` í `Time::print`. Virkinn `::` kemur á milli nafns strúktúrsins og nafns fallsins. Til saman gefa þessi nöfn til kynna að um sé að ræða fallið `print` sem tilheyrir `Time` strúktúr.

Næsta skref felst í því að fjarlægja leppinn. Í stað þess að senda hlut inn sem viðfang þá vekjum við fallið upp með tilteknum hlut (e. invoke the function on an object).

Niðurstaðan er sú að inni í fallinu er ekki lengur leppur með nafnið `time`. Í staðinn er um að ræða **núverandi hlut** (e. current object) sem er hluturinn

sem fallið er keyrt á. Hægt er að vísa í núverandi hlut með því að nota C++ lykilorðið `this`.

Það flækir aðeins málið að `this` er í raun **bendir** á strúktúr frekar en strúktúrinn sjálfur. Bendir er svipaður og tilvísun en ég vil þó ekki fjalla nákvæmlega um bendanotkun enn sem komið er. Eina bendaaðgerðin sem við þurfum á að halda núna er `*` virkinn (e. dereference operator) sem breytir bendi á strúktúr yfir í strúktur. Í eftirfarandi falli notum við þennan virkja til að gefa staðværu (e. local) breytunnni `time` gildið á `this`:

```
void Time::print () {
  Time time = *this;
  cout << time.hour << ":" << time.minute << ":" << time.second << endl;
}
```

Fyrstu tvær línurnar í þessu falli breyttust þó nokkuð þegar við umbreyttum fallinu yfir í meðlimafall en taktu eftir því að úttakssetningin sjálf breyttist ekki neitt.

Þessa nýju útgáfu af `print` keyrum við síðan á `Time` hlut (taktu eftir punktatáknuninni):

```
  Time currentTime = { 9, 14, 30.0 };
  currentTime.print ();
```

Síðasta skrefið í umbreytingarferlinu er að við þurfum að lýsa yfir fallinu inni í strúktúrskilgreiningunni:

```
struct Time {
  int hour, minute;
  double second;

  void print ();
};
```

Yfirlýsing falls lítur eins út og fyrsta línan í skilgreiningu fallsins nema að því leyti að yfirlýsingin er með semíkommu í endann og ekki þarf að tiltaka nafn strúktúrsins sem fallið tilheyrir (í þessu tilfelli Time). Yfirlýsingin lýsir **skilum** (e. interface) fallsins, þ.e. fjölda og tagi leppa og tagi skilagildisins.

Þegar þú lýsir yfir falli þá "lofar" þú þýðandanum að þú munir, á einhverjum öðrum stað í forritinu, setja skilgreiningu fallsins fram. Skilgreiningin sjálf er **útfærsla** (e. implementation) fallsins því hún inniheldur nákvæmar upplýsingar um hvernig fallið virkar. Þýðandinn mun kvarta ef þú sleppir skilgreiningunni eða setur fram skilgreiningu sem inniheldur önnur skil en þú lofaðir.

11.3 Dulinn aðgangur að breytum

Nýja útgáfan okkar af `Time::print` er reyndar flóknari en hún þarf að vera. Við þurfum í raun ekki að búa til staðværa breytu í þeim tilgangi að vísa í tilvikabreytur núverandi hlutar.

Ef fallið vísar í `hour`, `minute` eða `second`, án þess að nota punktatáknun, þá veit C++ þýðandinn að um er að ræða tilvísun í núverandi hlut. Við gætum því skrifað fallið á þennan hátt:

```
void Time::print ()
{
  cout << hour << ":" << minute << ":" << second << endl;
}
```

Þessi tegund af aðgangi að breytum er kallaður "dulinn" (e. implicit) vegna þess að nafn hlutarins er ekki ljóst (e. explicit). Þetta er ein ástæða þess að meðlimaföll eru oft samþjappaðri en föll sem eru ekki meðlimaföll.

11.4 Annað dæmi

Nú skulum við umbreyta `increment` í meðlimafall. Við munum breyta einu viðfanginu í dulda viðfangið `this`. Síðan getum við gert allan aðgang að einstökum breytum dulinn.

```
void Time::increment (double secs) {
  second += secs;

  while (second >= 60.0) {
    second -= 60.0;
    minute += 1;
  }
  while (minute >= 60) {
    minute -= 60.0;
    hour += 1;
  }
}
```

Athugaðu að þessi útfærsla fallsins er ekki sú skilvirkasta. Þú ættir núna að skrifa skilvirkari útfærslu ef þú gerðir það ekki þegar þú last kafla 9.

Til að lýsa fallinu yfir þurfum við aðeins að afrita fyrstu línuna úr skilgreiningu þess (og sleppa tilvísuninni í Time strúktúrinn):

```
struct Time {
  int hour, minute;
  double second;

  void print ();
  void increment (double secs);
};
```

Nú getum við kallað á fallið með því að vekja það upp með tilteknum `Time` hlut:

```
  Time currentTime = { 9, 14, 30.0 };
  currentTime.increment (500.0);
  currentTime.print ();
```

Útttakið úr þessu forriti er `9:22:50`.

11.5 Enn eitt dæmi

Upphaflega útgáfan af `convertToSeconds` leit svona út:

```
double convertToSeconds (const Time& time) {
  int minutes = time.hour * 60 + time.minute;
  double seconds = minutes * 60 + time.second;
  return seconds;
}
```

Það er einfalt að umbreyta þessu í meðlimafall:

```
double Time::convertToSeconds () const {
  int minutes = hour * 60 + minute;
  double seconds = minutes * 60 + second;
  return seconds;
}
```

Hér er athyglisvert að dulda viðfanginu er lýst yfir sem `const` vegna þess að við breytum því ekki inni í fallinu. Það er reyndar ekki augljóst hvar setja á upplýsingar um viðfang sem er ekki til! Eins og sést í þessu dæmi felst lausnin í því að setja þetta á eftir leppalistanum (sem er reyndar tómur í þessu tilviki).

Fallið `print` í fyrri kafla hefði líka á að lýsa dulda viðfanginu yfir sem `const`.

11.6 Flóknara dæmi

Þrátt fyrri að ferlið við að umbreyta föllum í meðlimaföll sé vélrænt þá geta aðstæður verið sérkennilegar. Fallið `after`, t.d., vinnur á tveimur `Time` hlutum (ekki bara einum) og við getum ekki gert þá báða að duldum viðföngum. Í staðinn köllum við á fallið í gegnum annan þeirra en sendum hinn sem viðfang.

Inni í fallinu getum við notað dulinn aðgang að breytum hlutarins sem notaður var til að vekja fallið upp en fáum síðan aðgang að breytum hins hlutarins með því að nota punktatáknun.

```
bool Time::after (const Time& time2) const {
  if (hour > time2.hour) return true;
  if (hour < time2.hour) return false;

  if (minute > time2.minute) return true;
  if (minute < time2.minute) return false;
```

```
  if (second > time2.second) return true;
  return false;
}
```

Svona köllum við þá á þetta fall:

```
  if (doneTime.after (currentTime)) {
    cout << "The bread will be done after it starts." << endl;
  }
```

Það er nánast hægt að lesa þennan kóða eins og ensku: "If the done-time is after the current-time, then..."

11.7 Smiðir

Í kafla 9 skrifuðum við fallið `makeTime`:

```
Time makeTime (double secs) {
  Time time;
  time.hour = int (secs / 3600.0);
  secs -= time.hour * 3600.0;
  time.minute = int (secs / 60.0);
  secs -= time.minute * 60.0;
  time.second = secs;
  return time;
}
```

Fyrir sérhvert nýtt tag (e. type) þurfum við að geta búið til nýja hluti af því tagi. Það vill einmitt reyndar svo til að föll eins og `makeTime` eru svo algeng að sérstök málskipan er notuð fyrir þau. Þessi föll eru kölluð **smiðir** (e. constructors) og málskipanin er þessi:

```
Time::Time (double secs) {
  hour = int (secs / 3600.0);
  secs -= hour * 3600.0;
  minute = int (secs / 60.0);
  secs -= minute * 60.0;
  second = secs;
}
```

Hér eru tvö atriði sem vert er að minnast á. Í fyrsta lagi ber smiðurinn sama nafn og strúktúrinn sjálfur og hann hefur ekkert skilagildi. Viðföngin hafa hins vegar ekkert breyst.

Í öðru lagi þurfum við ekki að búa til nýjan Time hlut og við þurfum ekki að skila neinu. Bæði þessi skref eru gerð fyrir okkur af þýðandanum. Við getum vísað í nýja hlutinn – þann sem við erum að smíða – með því að nota lykilorðið

`this`, eða á dulinn hátt eins og gert er hér að ofan. Þegar við notum nöfnin `hour`, `minute` og `second` þá veit þýðandinn að við erum að vísa í meðlimabreytur (tilvikabreytur) nýja hlutarins.

Til að vekja (kalla á) smiðinn upp þá notum við sérstaka málskipan sem er einhvers konar millivegur á milli breytuyfirlýsingar og fallakalls:

```
Time time (seconds);
```

Þessi setning lýsir yfir að breytan `time` hafi tagið `Time` og vekur smiðinn upp, sem við vorum að skrifa, með því að senda `seconds` sem viðfang. Kerfið úthlutar minni fyrir nýja hlutinn og smiðurinn upphafsstillir meðlimabreyturnar. Niðurstaðan er gildi sem breytan `time` fær.

11.8 Að upphafsstilla eða smíða?

Í fyrri kafla lýstum við yfir og upphafsstilltum `Time` strúktúr með því að nota slaufusviga (e. squiggly-braces):

```
Time currentTime = { 9, 14, 30.0 };
Time breadTime = { 3, 35, 0.0 };
```

Með því að nota smiði getum við nú notað aðra aðferð við yfirlýsingu og upphafsstillingu:

```
Time time (seconds);
```

Þessar tvær aðferðir standa fyrir mismunandi forritunarstíl og reyndar mismunandi tíma í sögu C++. Það er kannski ástæðan fyrir því að C++ þýðandinn krefst þess að þú notar aðra aðferðina, en ekki báðar, í sama forritinu.

Ef þú skilgreinir smið fyrir strúktúr þá verður þú að nota smiðinn til að upphafsstilla öll ný tilvik af viðkomandi tagi. Hin málskipanin, þ.e. að nota slaufusviga við upphafsstillingu, er þá ekki leyfð.

Sem betur fer er leyfilegt að fjölbinda (e. overload) smiði og sama hátt og hægt er að fjölbinda föll. M.ö.o, það geta verið fleiri en einn smiður með sama nafn svo framarlega sem smiðirnir taki mismunandi viðföng. Þegar við upphafsstillum nýjan hlut mun þýðandinn reyna að finna þann smið sem tekur viðeigandi/rétt viðföng.

Það er t.d. algengt að hafa einn smið sem tekur eitt viðfang fyrir sérhverja meðlimabreytu og gefur meðlimabreytunum gildi viðfanganna:

```
Time::Time (int h, int m, double s)
{
  hour = h;  minute = m;  second = s;
}
```

Við notum sömu skrýtnu málskipanina og áður til að vekja þennan smið upp nema að nú verða viðföngin að vera tvær heiltölur og ein `double`:

```
Time currentTime (9, 14, 30.0);
```

11.9 Eitt dæmi að lokum

Síðasta dæmið sem við skoðum er `addTime`:

```
Time addTime2 (const Time& t1, const Time& t2) {
  double seconds = convertToSeconds (t1) + convertToSeconds (t2);
  return makeTime (seconds);
}
```

Við þurfum að gera nokkrar breytingar á þessu falli, þ.m.t.:

1. Breyta nafninu á fallinu úr `addTime` í `Time::add`.
2. Fjarlægja fyrsta viðfangið og gera það að duldu `const` viðfangi.
3. Skipta út kalli í `makeTime` fyrir vakningu á smið.

Hér er niðurstaðan:

```
Time Time::add (const Time& t2) const {
  double seconds = convertToSeconds () + t2.convertToSeconds ();
  Time time (seconds);
  return time;
}
```

Þegar við köllum á `convertToSeconds` í fyrsta sinn þá virðist vanta hlutinn sem vekja á fallið upp með! Inni í meðlimafalli (eins og Time::add) gerir þýðandinn ráð fyrir því að við viljum vekja föll með núverandi hlut (þ.e. `this`). Þess vegna á fyrsta kallið í `convertToSeconds` við `this` en í seinna kallinu er `convertToSeconds` vakið upp með `t2`.

Í næstu línu fallsins er smiður vakinn upp sem tekur eitt `double` sem viðfang. Í síðustu línunni er síðan nýja hlutnum (time) skilað til baka úr fallinu.

11.10 Hausaskrár

Það kann að virðast undarlegt að lýsa yfir föllum inni í strúktúrskilgreiningu og síðan að skilgreina (útfæra) föllin síðar. Í sérhvert sinn sem þú breytir skilum (e. interface) falls þarftu þá að breyta þeim á tveimur stöðum, jafnvel þó um sé að ræða smávægilega breytingu eins og að lýsa einum lepp yfir sem `const`.

Fyrir þessu er sérstök ástæða. Hægt er að setja strúktúrskilgreiningu og fallaútfærslu í tvær mismunandi skrár: **hausaskrá** (e. header file), sem geymir strúktúrskilgreininguna, og útfærsluskrá (e. implementation file) sem geymir útfærslu fallanna.

Hausaskrár hafa venjulega sama nafn og útfærsluskrár en bera viðskeytið `.h` í stað `.cpp`. Fyrir dæmið sem við höfum verið að skoða ber hausaskráin nafnið `Time.h` og hún geymir eftirfarandi skilgreiningu:

```
struct Time {
  // instance variables
  int hour, minute;
  double second;

  // constructors
  Time (int hour, int min, double secs);
  Time (double secs);

  // modifiers
  void increment (double secs);

  // functions
  void print () const;
  bool after (const Time& time2) const;
  Time add (const Time& t2) const;
  double convertToSeconds () const;
};
```

Skráin `Time.cpp` geymir síðan skilgreininguna á meðlimaföllunum (til þess að spara pláss sleppi ég "líkama" (e. body) sérhvers falls):

```
#include <iostream>
using namespace std;
#include "Time.h"

Time::Time (int h, int m, double s)  ...

Time::Time (double secs) ...

void Time::increment (double secs) ...

void Time::print () const ...

bool Time::after (const Time& time2) const ...

Time Time::add (const Time& t2) const ...

double Time::convertToSeconds () const ...
```

Í þessu tilviki birtist skilgreiningin á föllum í `Time.cpp` í sömu röð og yfirlýsingin í `Time.h` þó svo að það sé reyndar ekki nauðsynlegt.

Á hinn bóginn er nauðsynlegt að taka inn hausaskrána í .cpp skrána með `include` setningu. Þannig getur þýðandinn borið saman yfirlýsingu og skilgreiningu og látið vita um villur ef einhverjar eru.

Skráin `main.cpp` geymir síðan fallið `main` ásamt ýmsum öðrum föllum sem við þurfum á að halda og eru ekki meðlimaföll í `Time` (í þessu tilviki eru það

reyndar ekki nein föll):

```
#include <iostream>
using namespace std;
#include "Time.h"

int main ()
{
  Time currentTime (9, 14, 30.0);
  currentTime.increment (500.0);
  currentTime.print ();

  Time breadTime (3, 35, 0.0);
  Time doneTime = currentTime.add (breadTime);
  doneTime.print ();

  if (doneTime.after (currentTime)) {
    cout << "The bread will be done after it starts." << endl;
  }
  return 0;
}
```

Taktu eftir því að `main.cpp` þarf líka að taka inn hausaskrána `Time.h` vegna þess að main fallið notar tilvik af `Time` hlut.

Það er kannski ekki augljóst af hverju það er gagnlegt að skipta svona litlu forriti upp í þrjár einingar (`Time.h`, `Time.cpp` og `main.cpp`). Reyndar er það svo að kostirnir koma helst fram þegar við vinnum með stærri forrit:

Endurnýting (e. reuse): Þegar þú hefur skrifað einingu eins og `Time` þá gætir þú viljað nýta hana í fleiri en einu forriti. Með því að skilja að skilgreininguna á `Time` frá `main.cpp` gerir þú á auðveldan hátt kleift að taka inn (e. include) `Time` strúktúrinn í annarri forritsskrá.

Stjórna tengslum (e. manage interactions): Eftir því sem kerfi stækka því flóknari verða tengsl á milli einstakra eininga þess og það verður fljótt erfitt að stjórna þeim. Þess vegna er oft gagnlegt að lágmarka þessi tengsl með því að skilja að einingar eins og `Time.cpp` frá þeim forritum sem nota einingarnar.

Aðskilin þýðing (e. separate compilation): Hægt er að þýða aðskildar skrár sérstaklega og tengja (e. link) þær seinna þannig að úr verði keyrandi forrit. Hvernig þetta er gert er háð því forritunarumhverfi sem þú notar. Aðskilin þýðing getur sparað mikinn tíma þegar kerfi stækka því þá þarf yfirleitt aðeins að þýða nokkrar skrár í sérhvert sinn en ekki allar skrárnar sem kerfið notar.

Þegar um er að ræða lítil forrit, eins og þau sem sýnd eru í þessari bók, þá eru engir sérstakir kostir þess samfara að skipta forritinu upp í aðskildar skrár. Það er hins vegar mjög gott fyrir þig að vita af þessum möguleika, sérstaklega vegna þess að nú ætti að vera skýrt hvaða tilgangi fyrstu setningarnar, sem birtust í fyrsta forritinu sem við skrifuðum, þjóna:

```
#include <iostream>
using namespace std;
```

`iostream` er hausaskráin sem inniheldur yfirlýsingar á `cin` og `cout` og föllunum sem þessir straumar nota. Þegar þú þýðir þitt forrit þarftu á upplýsingum að halda sem geymdar eru í þessari hausaskrá.

Útfærslan á þessum föllum er hins vegar geymd í forritunarsafni (e. library) sem kallað er "Standard Library" og sú útfærsla er tengd (e. linked) við forritið þitt á sjálfvirkan hátt. Athugaðu að það er engin þörf á því að endurþýða þetta forritunarsafn í sérhvert sinn sem þú þýðir forritið þitt. Þetta forritunarsafn er einmitt ekki að breytast og því er engin ástæða til að endurþýða það.

11.11 Orðalisti

meðlimafall (e. member function): Fall sem vinnur með hlut (e. object) sem sendur er inn sem "dulið" (e. implicit) viðfang með nafninu `this`.

"ekki-meðlimafall" (e. nonmember function): Fall sem tilheyrir ekki tilteknum strúktúr. Líka kallað "free-standing" fall.

vekja upp (e. invoke): Að kalla á meðlimafall í gegnum tiltekinn hlut í þeim tilgangi að senda hlutinn sem dulið viðfang.

núverandi hlutur (e. current object): Hluturinn sem tilheyrir því meðlimafalli sem vakið var upp. Inni í meðlimafallinu er hægt að vísa í hlutinn á dulinn máta eða með því að nota lykilorðið `this`.

this: Lykilorð sem vísar til núverandi hlutar. `this` er bendir (sem gerir notkun þess erfiða vegna þess að við fjöllum ekki um benda í þessari bók).

skil (e. interface): Lýsing á því hvernig fall er notað, þ.m.t. fjöldi og tag viðfanga og tag skilagildis.

fallayfirlýsing (e. function declaration): Setning sem lýsir yfir skilum falls án þess að sýna útfærsluna. Yfirlýsing á meðlimaföllum kemur fram í strúktúrskilgreiningu.

útfærsla (e. implementation): "Líkami" (e. body) falls, þ.e. kóðinn sem sýnir hvernig fallið virkar.

smiður (e. constructor): Sérstakt fall sem býr til tilvik (e. instance) af tilteknum hlut (e. object) og upphafsstillir meðlimabreytur hans.

Kafli 12

Vektorar af hlutum

12.1 Samsetning

Hingað til höfum við séð nokkur dæmi um samsetningu (e. composition), þ.e. þegar einstakir eiginleikar forritunarmálsins eru settir saman á ýmsan máta. Eitt af fyrstu dæmunum sem við sáum var að nota fallakall sem hluta af segð (e. expression). Annað dæmi er hreiðruð skipan setninga: hægt er að setja `if` setningu inn í `while` lykkju eða inn í aðra `if` setningu, o.s.frv.

Nú þegar við höfum séð þessi samsetningarmynstur, og lært um vektorara og hluti, þá ætti ekki að koma á óvart að hægt er að búa til vektora af hlutum. Reyndar vill svo til að það er líka hægt að búa til hluti sem innihalda vektora (sem meðlimabreytur) vektora sem innihalda vektora, hluti sem innihalda aðra hluti, o.s.frv.

Í næstu tveimur köflum munum við skoða nokkur dæmi um svona samsetningar með því að nota `Card` hluti sem sýnidæmi.

12.2 `Card` hlutir

Ef þú þekkir ekki handspil (e. playing cards) þá er nú góður tímapunktur að sækja spilastokk (e. deck) því annars er hætt við því að efni þessa kafla fari fyrir ofan garð og neðan. Það eru 52 spil í spilastokki og sérhvert spil tilheyrir einum af fjórum litum (e. suits) og 13 gildum (e. ranks). Litirnir eru (í lækkandi röð í bridds): Spaðar (e. Spades), hjörtu (e. Hearts), tíglar (e. Diamonds) og lauf (e. Clubs). Gildin eru ás (e. Ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, gosi (e. Jack), drottning (e. Queen) og kóngur (e. King). Það fer eftir því hvaða spil þú ert að spila hvort ásinn er hærri en kóngur eða lægri en tvistur.

Það er nokkuð augljóst hverjar meðlimabreyturnar eiga að vera ef við viljum skilgreina nýjan hlut sem stendur fyrir tiltekið spil: `rank` og `suit`. Það er hins vegar ekki jafn augljóst hvert tag meðlimabreytnanna ætti að vera. Einn möguleiki er að nota `string` sem þá myndi innihalda t.d. `"Spade"` fyrir lit og `"Queen"` fyrir gildi. Eitt vandamálið við þá útfærslu er að það yrði ekki einfalt

að bera saman tvö spil, þ.e. að finna út hvort þeirra er með hærri lit eða hærra gildi.

Annar möguleiki er sá að nota heiltölur til að **kóta** (e. encode) gildin og litina. Það sem tölvunarfræðingar eiga við með "að kóta" er t.d. að skilgreina vörpun á milli talnarunu og hlutanna sem sérhvert tala raðarinnar stendur fyrir. Dæmi:

Spades	$\mapsto$	3
Hearts	$\mapsto$	2
Diamonds	$\mapsto$	1
Clubs	$\mapsto$	0

Táknið $\mapsto$ er stærðfræðileg tákn fyrir "varpast í". Það má augljóslega lesa úr þessari vörpun að litirnir varpast í heiltölur í ákveðinni röð. Þannig getum við borið saman liti með því að bera saman heiltölur. Vörpunin fyrir gildi eru nokkuð augljós: sérhvert spil með talnagildi varpast yfir í viðkomandi heiltölu en fyrir mannspil notum við eftirfarandi vörpun:

Jack	$\mapsto$	11
Queen	$\mapsto$	12
King	$\mapsto$	13

Ástæðan fyrir því að ég nota stærðfræðilega táknun fyrir þessar varpanir er sú að þær eru ekki hluti af C++ forritunarmálinu. Varpanirnar eru hluti af hönnun forritsins en þær koma ekki beint fyrir í forritskóðanum. Strúktúrskilgreiningin fyrir `Card` tagið lítur svona út:

```
struct Card
{
  int suit, rank;

  Card ();
  Card (int s, int r);
};

Card::Card () {
  suit = 0;  rank = 0;
}

Card::Card (int s, int r) {
  suit = s;  rank = r;
}
```

Það eru tveir smiðir (e. constructors) fyrir `Card`. Þú sérð að um er að ræða smiði vegna þess að þeir hafa ekkert skilagildi og nafn þeirra er það sama og nafn strúktúrsins. Fyrri smiðurinn tekur engin viðföng og upphafsstillir meðlimabreyturnar í raun með gagnlausum gildum (0 í laufi!).

Seinni smiðurinn er gagnlegri. Hann tekur tvö viðföng, lit og gildi spilsins.

Eftirfarandi kóði býr til hlut með nafninu `threeOfClubs` sem stendur fyrir laufaþrist:

```
Card threeOfClubs (0, 3);
```

Fyrsta viðfangið, `0`, stendur fyrir litinn lauf og seinna viðfangið fyrir gildið 3.

12.3 `printCard` fallið

Fyrsta skrefið í því að búa til nýtt tag felst yfirleitt í því að lýsa yfir meðlimabreytum og skrifa smiði. Annað skrefið felst oft í því að skrifa út viðkomandi hlut á læsilegan máta (e. human-readable).

Í tilviki `Card` hlutanna þá merkir "læsilegur máti" að við verðum að varpa innri framsetningu litar og gildis í orð. Eðlileg leið til að gera það er að nota vektor af strengjum (`string`). Þú getur búið til vektor af strengjum á sama hátt og þú býrð til vektor af hvaða öðru tagi sem er:

```
vector<string> suits (4);
```

Við getum notað röð gildisveitingarsetninga (e. assignment statements) til að upphafsstilla stök vektors:

```
suits[0] = "Clubs";
suits[1] = "Diamonds";
suits[2] = "Hearts";
suits[3] = "Spades";
```

Stöðurit fyrir þennan vektor lítur þá svona út:

Við getum búið til sambærilegan vektor til að afkóta (e. decode) gildin. Þá getum við valið viðeigandi stök með því að nota `suit` og `rank` sem vísa (e. indices). Að lokum getum við síðan skrifað fallið `print` sem skrifar út spilið:

```
void Card::print () const
{
  vector<string> suits (4);
  suits[0] = "Clubs";
  suits[1] = "Diamonds";
  suits[2] = "Hearts";
  suits[3] = "Spades";

  vector<string> ranks (14);
  ranks[1] = "Ace";
  ranks[2] = "2";
  ranks[3] = "3";
  ranks[4] = "4";
  ranks[5] = "5";
  ranks[6] = "6";
  ranks[7] = "7";
  ranks[8] = "8";
  ranks[9] = "9";
  ranks[10] = "10";
  ranks[11] = "Jack";
  ranks[12] = "Queen";
  ranks[13] = "King";

  cout << ranks[rank] << " of " << suits[suit] << endl;
}
```

Segðin `suits[suit]` merkir að "nota meðlimabreytuna `suit` í núverandi hlut sem vísi inn í vektorinn `suits` og velja viðeigandi streng."

Vegna þess að `print` er meðlimafall í `Card` þá getur það vísað í meðlimbreytur núverandi hlutar á dulinn hátt (án þess að nota punktatáknun og "this" til að tilgreina hlutinn). Úttakið úr þessum kóða

```
  Card card (1, 11);
  card.print ();
```

er `Jack of Diamonds`.

Þú tekur kannski eftir því að við notum ekki núllta stakið í `ranks` vektornum. Ástæðan er sú að leyfileg gildi eru aðeins þau á bilinu 1–13. Með því að skilja fyrsta stakið (núllta stakið) eftir ónotað í vektornum þá fáum við vörpun sem varpar 2 í "2", 3 í "3", o.s.frv. Notandi verður ekkert var við þessa vörpun (eða innri framsetningu) sem við notum í forritinu því allt inntak og úttak er sett fram á læsilegan hátt. Á hinn bóginn er það oft þægilegt fyrir forritarann ef vörpun er sett fram á máta sem auðvelt er að muna.

12.4 Fallið `equals`

Tvö spil eru jöfn ef þau eru með sama gildi og bera sama lit. Því miður getum við ekki notað == virkjann til að bera saman tvö spil því hann virkar ekki fyrir tög sem notandinn skilgreinir (e. user-defined types) eins og `Card`. Við verðum því að skrifa fall sem ber saman tvö spil. Köllum það `equals`. Það er reyndar hægt að fjölbinda (e. overload) == virkjann en við munu ekki fjalla um þann möguleika í þessari bók.

Það er ljóst að skilagildið úr `equals` ætti að vera bool gildi sem gefur til kynna hvort tvö spil eru jöfn eður ei. Það er jafnframt ljóst að fallið þarf að taka tvo `Card` hluti sem viðföng. Nú stöndum við frammi fyrir eftirfarandi vali: Á `equals` að vera meðlimafall (e. member function) eða fall sem stendur eitt og sér (e. free-standing function)?

Sem meðlimafall lítur `equals` svona út:

```
bool Card::equals (const Card& c2) const
{
  return (rank == c2.rank && suit == c2.suit);
}
```

Til að nota fallið vekjum við það upp með einu spili og sendum hitt spilið sem viðfang:

```
  Card card1 (1, 11);
  Card card2 (1, 11);

  if (card1.equals(card2)) {
    cout << "Yup, that's the same card." << endl;
  }
```

Mér finnst þessi aðferð við vakningu (e. invocation) alltaf líta dálítið undarlega út þegar um er að ræða fall eins og `equals` þar sem viðföngin tvö eru samhverf (e. symmetric). Það sem ég á við með samhverfum viðföngum er að það skiptir ekki máli hvort ég spyr "Is A equal to B?" eða "Is B equal to A?". Í þessu tilviki finnst mér því eðlilegra að skrifa `equals` sem "ekki-meðlimafall" (e. nonmember function).

```
bool equals (const Card& c1, const Card& c2)
{
  return (c1.rank == c2.rank && c1.suit == c2.suit);
}
```

Þegar við köllum á þessa útgáfu fallsins þá birtast viðföngin hlið við hlið á máta sem mér finnst eðlilegri.

```
  if (equals (card1, card2)) {
    cout << "Yup, that's the same card." << endl;
  }
```

Þetta er samt auðvitað bara spurning um smekk. Punkturinn hér er sá að þú ættir að geta skrifað hvort sem er meðlimafall eða "ekki-meðlimafall" þannig að þú getir ákveðið þau skil (e. interface) sem eru hentugust því verkefni sem liggur fyrir hverju sinni.

12.5 Fallið `isGreater`

Fyrir grunntög eins og `int` og `double` eru til innbyggðir samanburðarvirkjar sem bera saman gildi og ákvarða hvort eitt gildi er stærra eða minna en annað. Þessir virkjar (< og > og fleiri) virka hins vegar ekki fyrir tög sem skilgreind eru af notanda. Við þurfum því, á sama hátt og við gerðum í tilviki == virkjans, að skrifa samanburðarfall sem líkir eftir > virkjanum. Við munum síðar nota þetta fall til að raða spilum spilastokks.

Sum mengi eru fullröðuð (e. totally ordered) sem merkir að hægt er að bera saman hvaða tvö stök sem er og segja til um hvort er stærra. Heiltölur og kommutölur eru t.d. fullraðaðar. Sum mengi eru hins vegar ekki röðuð sem merkir að það er engin nærtæk leið til að segja til um hvort eitt stak sé stærra en annað. Ávextir eru t.d. ekki raðaðir sem er einmitt ástæðan fyrir því að við getum ekki borið saman epli og appelsínur! Annað dæmi um mengi sem er ekki raðað er `bool` tagið – við getum ekki sagt að `true` sé stærra en `false`.

Mengi handspila er hlutraðað (e. partially ordered), þ.e. stundum er hægt að bera saman spil og stundum ekki. Ég veit t.d. að laufaþristur er hærri en laufatvistur vegna þess að sá fyrrnefndi er með hærra gildi en sá síðarnefndi og að tígulþristur er hærri en laufaþristur vegna þess að sá fyrrnefndi er með hærri lit. En hvort er laufaþristur eða tígultvistur betra spil? Annað er með hærra gildi en hitt en með hærri lit.

Til að tvö spil séu samanburðarhæf verðum við að ákveða hvort litur eða gildi er mikilvægara Ég ætla að segja að litur sé mikilvægari og ástæðan er sú að þegar þú kaupir nýjan spilastokk þá er hann raðaður eftir litum, þ.e. öll laufin saman, síðan tíglarnir, o.s.frv.

Nú þegar þetta er ákveðið þá getum við skrifað fallið `isGreater`. Aftur ætti að vera augljóst að viðföngin eru tveir `Card` hlutir og að skilagildið er `bool`. Við þurfum á ný að velja á milli meðlimafalls og "ekki-meðlimafalls". Í þetta skiptið eru viðföngin ekki samhverf. Það skipir máli hvort við viljum vita "Is A greater than B?" eða "Is B greater than A?". Þess vegna finnst mér eðlilegra að útfæra `isGreater` sem meðlimafall:

```
bool Card::isGreater (const Card& c2) const
{
  // first check the suits
  if (suit > c2.suit) return true;
  if (suit < c2.suit) return false;

  // if the suits are equal, check the ranks
  if (rank > c2.rank) return true;
```

```
  if (rank < c2.rank) return false;

  // if the ranks are also equal, return false
  return false;
}
```

Þegar við vekjum fallið upp þá er augljóst út frá málskipaninni hvora af hinum tveimur spurningum að ofan við erum að setja fram:

```
  Card card1 (2, 11);
  Card card2 (1, 11);

  if (card1.isGreater (card2)) {
    card1.print ();
    cout << "is greater than" << endl;
    card2.print ();
  }
```

Það er nánast hægt að lesa þetta eins og ensku: "If card1 isGreater card2 ..." Úttak forritsins er:

```
Jack of Hearts
is greater than
Jack of Diamonds
```

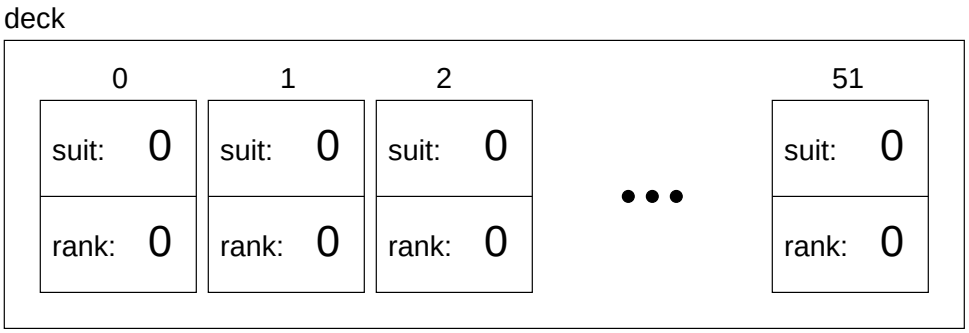
Samkvæmt `isGreater` eru ásar lægri en tvistar. Þú ættir núna að laga fallið þannig að ásar séu hærri en kóngar eins og þeir eru í flestum spilum.

12.6 Vektor af spilum

Ástæðan fyrir því að ég valdi spil (`Cards`) sem hluti í þessum kafla er að það er augljóst notkun fyrir vektor af spilum, þ.e. spilastokkur. Hér er forritskóði sem býr til nýjan stokk af 52 spilum:

```
  vector<Card> deck (52);
```

Og hér er stöðurit fyrir þennan hlut:



Punktarnir þrír standa fyrir spilin 48 sem ég sleppti að teikna. Taktu eftir að við höfum ekki upphafsstillt meðlimabreytur sérhvers spils. Í sumum keyrsluumhverfum munu þær verða upphafsstilltar sjálfvirkt með núllum (eins og sýnt er á myndinni) en annars staðar gætu þær fengið hvaða gildi sem er.

Ein leið til að upphafsstilla meðlimabreyturnar er að senda `Card` hlut sem annað viðfang í smiðinn:

```
Card aceOfSpades (3, 1);
vector<Card> deck (52, aceOfSpades);
```

Þessi forritskóði býr til stokk me 52 eins spilum, svona eins og gæti verið notaður í töfrabragði! Það er auðvitað eðlilegra að búa til stokk með 52 mismunandi spilum í. Við getum gert það með því að nota hreiðraða (e. nested) lykkju.

Ytri lykkjan "telur upp" (e. enumerates) litina, frá 0 til 3. Fyrir hvern lit telur innri lykkjan upp gildin, frá 1 til 13. Þar sem ytri lykkjan keyrir fjórum sinnum og innri lykkjan 13 sinnum þá keyrir meginmál (e. body) lykkjunnar samtals 52 sinnum (13*4).

```
int i = 0;
for (int suit = 0; suit <= 3; suit++) {
  for (int rank = 1; rank <= 13; rank++) {
    deck[i].suit = suit;
    deck[i].rank = rank;
    i++;
  }
}
```

Ég notaði breytuna `i` til að halda utan um í hvaða sæti í spilastokknum næsta spil ætti að fara.

Taktu eftir því að við getum sett saman málskipanina til að velja stak úr fylki (með `[]` virkjanum) og málskipanina sem velur meðlimabreytu úr hlut (punktatáknið). Segðin `deck[i].suit` merkir þá "the suit of the ith card in the deck".

Til að æfa þig ættir þú nú að hjúpa ofangreindan kóða í fall sem kallast `buildDeck`. Fallið tekur engin viðföng og skilar vektor af `Card` sem hefur verið upphafsstilltur með 52 spilum.

12.7 Fallið `printDeck`

Þegar unnið er með vektor þá er þægilegt að eiga fall sem skrifar út innihald vektorsins. Við höfum nokkrum sinnum séð mynstur til að "ferðast um" í vektor þannig að eftirfarandi fall ætti að vera kunnuglegt:

```
void printDeck (const vector<Card>& deck) {
  for (unsigned int i = 0; i < deck.size(); i++) {
    deck[i].print ();
```

```
  }
}
```

Það ætti heldur ekki að koma á óvart að við getum sett saman málskipanina til að nálgast stak í vektor og málskipanina til að vekja upp fall.

Þar sem `deck` hefur tagið `vector<Card>` þá hefur stak í `deck` tagið `Card`. Þess vegna er leyfilegt að vekja upp fallið `print` með `deck[i]`.

12.8 Leit

Næsta fall sem mig langar til að skrifa er `find` sem leitar í vektor af spilum (`Card`s) og athugar hvort tiltekið spil finnst þar. Það er kannski ekki augljóst af hverju þetta fall er gagnlegt en það gefur mér þó tilefni til að sýna tvær leiðir til að leita að hlutum: `runuleit` (e. linear search) og `tvíundarleit` (e. binary/bisection search).

Runuleit er einfaldari í útfærslu en tvíundarleit. Runuleitin okkar snýst um að ferðast um í spilastokknum, spil fyrir spil, og bera saman sérhvert spil við það spil sem við leitum að. Ef við finnum spilið þá skilum við vísinum sem segir til um hvar spilið fannst. Ef spilið finnst ekki í spilastokknum þá skilum við -1.

```
int find (const Card& card, const vector<Card>& deck) {
  for (unsigned int i = 0; i < deck.size(); i++) {
    if (equals (deck[i], card))
      return i;
  }
  return -1;
}
```

Lykkjan hér er nákvæmlega sú sama og í lykkjunni í `printDeck`. Það vill einmitt svo til að þegar ég skrifaði þetta fall þá afritaði ég lykkjuna sem sparaði mér forritun og það að þurfa að aflúsa tvisvar.

Inni í lykkjunni berum við saman sérhvert stak í spilastokknum (deck) við `card`. Fallið hættir keyrslu um leið og það finnur leitarspilið sem þýðir þá að ekki þarf að ferðast um í öllum stokknum ef spilið finnst. Ef lykkjan hættir keyrslu án þess að spilið finnist þá vitum við að spilið er ekki í stokknum og skilum `-1`.

Ég skrifaði eftirfarandi kóða til að prófa fallið:

```
  vector<Card> deck = buildDeck ();

  int index = card.find (deck[17]);
  cout << "I found the card at index = " << index << endl;
```

Úttakið er:

```
I found the card at index = 17
```

12.9 Tvíundarleit

Ef spilastokkurinn er ekki raðaður þá er engin önnur leitaraðferð hraðvirkari en runuleit. Við verðum að skoða sérhvert spil því annars getum við ekki verið viss um að spilið sem við leitum að sé ekki í stokknum.

Þegar við hins vegar flettum upp orði í orðabók þá beitum við ekki runuleit. Ástæðan er sú að orðin eru í stafrófsröð í orðabókinni. Það er líklegt að við uppflettingu í orðabók notir þú algrím sem er svipað og tvíundarleit:

1. Byrja einhvers staðar í miðjunni.
2. Velja orð á síðunni og bera það saman við leitarorðið.
3. Ef leitarorðið finnst þá er leit lokið.
4. Ef leitarorðið kemur á eftir orðinu á síðunni þá er farið á einhverja síðu aftar í bókinni og farið til baka í skref 2.
5. Ef leitarorðið kemur á undan orðinu á síðunni þá er farið á einhverja síðu framar í bókinni og farið til baka í skref 2.

Ef sú staða kemur upp að leitarorðið ætti að vera á milli tveggja samliggjandi orða á síðunni þá getur þú ályktað sem svo að leitarorðið sé ekki í orðabókinni. Hinn möguleikinn er sá að leitarorðið sé á öðrum stað í orðabókinni en það er í mótsögn við forsenduna um að orðin í orðabókinni séu í stafrófsröð.

Ef við vitum að spilastokkurinn er raðaður þá getum við skrifað aðra útgáfu af `find` sem er miklu skilvirkari (hraðvirkari). Besta leiðin til að útfæra tvíundarleit er að beita endurkvæmni (e. recursion). Ástæðan er sú að tvíundarleit er í eðli sínu endurkvæm.

"Trikkið" er að skrifa fall, sem við getum kallað `findBisect`, sem tekur tvo vísa sem viðföng, `low` og `high`, sem gefa til kynna bil úr vektornum sem leita skal í.

1. Til að leita í vektornum er vísir valinn á milli `low` og `high` og kallaður `mid`. Spilið í `mid` í vektornum er síðan borið saman við leitarspilið.
2. Ef spilið fannst þá er leit hætt.
3. Ef spilið í `mid` er hærra en leitarspilið þá er leitað í bilinu `low` til `mid-1`.
4. Ef spilið í `mid` er lægra en leitarspilið þá er leitað í bilinu `mid+1` til `high`.

Skref 3 og 4 eru grunsamlega lík endurkvæmri kvaðningu! Svona mætti útfæra þetta í C++:

```
int findBisect (const Card& card, const vector<Card>& deck,
                int low, int high) {
  int mid = (high + low) / 2;
```

```
  // if we found the card, return its index
  if (equals (deck[mid], card)) return mid;

  // otherwise, compare the card to the middle card
  if (deck[mid].isGreater (card)) {
    // search the first half of the deck
    return findBisect (card, deck, low, mid-1);
  } else {
    // search the second half of the deck
    return findBisect (card, deck, mid+1, high);
  }
}
```

Þó svo að þessi forritskóði innihaldi kjarnann fyrir tvíundarleit þá vantar samt mikilvægan hlut. Eins og þetta er núna skrifað þá mun koma upp óendanleg endurkvæmni ef leitarspilið er ekki í stokknum. Við þurfum því leið til að mæta þessu skilyrði og meðhöndla það á réttan hátt (með því að skila `-1`).

Einfaldasta leiðin til að uppgötva að leitarspilið er ekki í stokknum er þegar *engin* spil eru í stokknu, þ.e. þegar `high` er minna `low`. Það eru auðvitað reyndar spil í stokknum en það sem ég á við er að það eru engin spil á því bili sem `low` og `high` gefa til kynna. Þetta er grunnþrepið í endurkvæmninni.

Ef við bætum við þessu skilyrði þá virkar fallið rétt:

```
int findBisect (const Card& card, const vector<Card>& deck,
                int low, int high) {

  cout << low << ", " << high << endl;

  if (high < low) return -1;

  int mid = (high + low) / 2;

  if (equals (deck[mid], card)) return mid;

  if (deck[mid].isGreater (card)) {
    return findBisect (card, deck, low, mid-1);
  } else {
    return findBisect (card, deck, mid+1, high);
  }
}
```

Ég bætti líka við úttakssetningu í byrjun fallsins þannig að ég geti séð röð endurkvæmu kallanna og fullvissað mig um að grunnþrepið verði að lokum virkt. Ég prófaði eftirfarandi kóða:

```
  cout << findBisect (deck, deck[23], 0, 51));
```

Úttakið varð þetta:

```
0, 51
0, 24
13, 24
19, 24
22, 24
I found the card at index = 23
```

Síðan bjó ég til spil sem er ekki í stokknum (tígull með gildið 15) og reyndi að finna það. Þá varð útkoman þessi:

```
0, 51
0, 24
13, 24
13, 17
13, 14
13, 12
I found the card at index = -1
```

Þessar prófanir mínar sanna ekki að forritið sé rétt. Reyndar er það svo að engar prófanir sem þessar (hversu viðamiklar sem þær eru) geta sannað að forrit sé rétt. Ef þú aftur á móti skoðar nokkur tilfelli og skoðar kóðann vel þá ættir þú að geta sannfært sjálfan þig um að forritið sé rétt.

Fjöldi endurkvæmra kalla er tiltölulega lítill, yfirleitt 6 eða 7. Það þýðir að við þurftum aðeins að kalla á `equals` og `isGreater` 6 eða 7 sinnum, samanborið við allt að 52 sinnum í tilfelli runuleitarinnar. Almennt séð er tvíundarleit því miklu skilvirkari en runuleit, sérstaklega þegar um er að ræða stóra vektora.

Tvær algengar villur við útfærslu á endurkvæmum föllum eru þær að gleyma að setja inn grunnþrep (e. base case) og að skrifa endurkvæmu köllinn þannig að grunnþrepið verði aldrei virkt. Önnur þessara villna veldur óendanlegri endurkvæmni sem hefur að lokum í för með sér keyrsluvillu.

12.10 Stokkar og hlutstokkar

Skoðum nánar skilin fyrir `findBisect`:

```
int findBisect (const Card& card, const vector<Card>& deck,
int low, int high) {
```

Hér gæti verið eðlilegra að meðhöndla viðföngin þrú, `deck`, `low` og `high`, sem eitt viðfang sem tilgreinir **hlutstokk** (e. subdeck).

Þetta er nokkuð algengt og ég kalla svona viðfang stundum **hugrænt viðfang**. Það sem ég á við með "hugrænt" (e. abstract) er eitthvað sem er í raun ekki hluti af forritunartextanum, heldur eitthvað sem lýsir hlutverki forritsins.

Þegar þú kallar t.d. á fall og sendir inn vektor og bilið `low` til `high` þá er ekkert sem kemur í veg fyrir að fallið reyni að nálgast stök úr vektornum sem

eru ekki innan hans (e. out of bounds). Þannig að í raun er ekki verið að senda inn í fallið hlutstokk heldur allan stokkinn í heild sinni. Svo framarlega sem fallið "hagar sér" rétt þá er eðlilegt að líta á viðföngin, á hugrænan hátt, sem hlutstokk.

Vel má vera að þú hafir tekið eftir öðru dæmi um svona útdrátt (e. abstraction) í kaflanum 9.3 þar sem vísaði í "tóman" hlut. Ástæðan fyrir því að ég setti "tómur" í gæsalappir var sú að gefa til kynna að þetta væri ekki alls kostar rétt. Allar breytur hafa gildi á sérhverjum tíma. Þegar þú býrð þær til þá fá þær einhver sjálfgefin gildi. Þannig að í raun er ekkert til sem heitir "tómur" hlutur.

Ef forrit sér hins vegar til þess að upphafsgildi breytu er aldrei lesið áður en breytunni er gefið gildi þá skiptir upphafsgildið í raun ekki máli. Á hugrænan hátt má því líta svo á að umræddur hlutur sé "tómur".

Þessi hugsanagangur, þ.e. þegar forrit hefur einhverja merkingu sem ekki er beint hægt að lesa úr forritstextanum, er mjög mikilvægur fyrir tölvunarfræðinga. Stundum er orðið "hugrænn" notað svo oft og í svo mörgu mismunandi samhengi að það getur verið erfitt að skilja. Útdráttur er samt sem áður grunnhugtak í tölvunarfræði (og reynar í mörgum öðrum greinum).

Almennari skilgreining á útdrætti er þessi: "Ferlið við að líkja eftir flóknu kerfi með einfaldri lýsingu í þeim tilgangi að fela ónauðsynleg smáatriði en á sama tíma grípa mikilvæga virkni".

12.11 Orðalisti

kóta (e. encode): Að setja fram eitt mengi af gildum með því að nota annað mengi af gildi og vörpun á milli þessara mengja.

hugrænt viðfang (e. abstract parameter): Mengi af viðföngum sem til saman hafa hlutverk eins viðfangs.

Kafli 13

Hlutir með vektorum

13.1 Upptalningartög

Í kaflanum hér á undan ræddi ég um vörpun á milli raunverulegra gilda, eins og gilda eða lita tiltekinna spila, og innri framsetningar, eins og heiltalna og strengja. Þó svo að við höfum annars vegar búið til vörpun á milli gilda og heiltalna og hins vegar á milli lita og heiltalna þá benti ég á að vörpunin sjálf kemur ekki beint fram í forritskóðanum.

Reyndar vill svo til að C++ hefur eiginleika sem kallast **upptalningartag** (e. enumerated type) sem gerir kleift (1) að gera vörpun sem hluta af forritskóða og (2) að skilgreina mengi af gildum sem vörpunin samanstendur af. Hér er t.d. skilgreiningin á upptalningartagi fyrir `Suit` og `Rank`:

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES };

enum Rank { ACE=1, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE,
TEN, JACK, QUEEN, KING };
```

Sjálfgefin vörpun er sú að fyrsta gildið í upptalningartagi varpast í heiltöluna 0, annað gildið í 1, o.s.frv. Í `Suit` taginu stendur því gildið `CLUBS` fyrir heiltöluna 0, `DIAMONDS` fyrir 1, o.s.frv.

Skilgreiningin á `Rank` yfirskrifar sjálfgefnu vörpunina og tilgreinir að `ACE` standi fyrir heiltöluna 1. Þar með varpast `TWO` í 2, `THREE` í 3, o.s.frv.

Um leið og við höfum skilgreint þessi tög getum við notað þau hvar sem er. Meðlimabreytunum `rank` og `suit` getur t.d. verið lýst yfir með tögunum `Rank` og `Suit`:

```
struct Card
{
  Rank rank;
  Suit suit;
```

```
  Card (Suit s, Rank r);
};
```

Taktu eftir að tag leppanna í smiðnum hefur líka breyst. Til að "smíða" card hlut getum við þá notað gildi úr upptalningartögunum sem viðföng:

```
  Card card (DIAMONDS, JACK);
```

Það er hefð fyrir því að gildi úr upptalningartögum séu nöfn með stórum stöfum. Þessi yfirlýsing á card hlut er miklu læsilegri en sú sem við notuðum áður með heiltölum:

```
  Card card (1, 11);
```

Við getum notað gildi upptalningartaga sem vísi inn í vektora vegna þess að við vitum að þessi gildi standa fyrir heiltölur. Af þessum sökum mun gamla `print` fallið okkar virka á nokkurra breytinga. Við þurfum hins vegar að gera breytingar á `buildDeck` fallinu:

```
  int index = 0;
  for (Suit suit = CLUBS; suit <= SPADES; suit = Suit(suit+1)) {
    for (Rank rank = ACE; rank <= KING; rank = Rank(rank+1)) {
      deck[index].suit = suit;
      deck[index].rank = rank;
      index++;
    }
  }
```

Með því að nota upptalningartögin er kóðinn læsilegri en það er ein flækja. Strangt til tekið getum við ekki beitt útreikningum á upptalningartög þannig að `suit++` er ekki löglegt. Hins vegar breytir C++ upptalningargildinu suit í segðinni `suit+1` í heiltölu. Þar með getum við tekið niðurstöðuna og notað tagmótun (e. typecast) til að "kasta" gildinu til baka í upptalningartag:

```
  suit = Suit(suit+1);
  rank = Rank(rank+1);
```

Það er reyndar til betri leið til að gera þetta með því að yfirskrifa ++ virkjann fyrir upptalningartögin en við fjöllum ekki um það í þessari bók.

13.2 `switch` setning

Það er erfitt að ræða upptalningartög án þess að nefna `switch` setningar því oft haldast þær í hendur. `switch` setning er önnur leið til að setja fram keðjur af if-skilyrðissetningum og oft reyndar bæði læsilegri og skilvirkari. Hún lítur svona út:

```
switch (symbol) {
case '+':
  perform_addition ();
  break;
case '*':
  perform_multiplication ();
  break;
default:
  cout << "I only know how to perform addition and multiplication" << endl;
  break;
}
```

Þessi `switch` setning er jafngild eftirfarandi keðju af if-setningum:

```
if (symbol == '+') {
  perform_addition ();
} else if (symbol == '*') {
  perform_multiplication ();
} else {
  cout << "I only know how to perform addition and multiplication" << endl;
}
```

`break` setningarnar eru nauðsynlegar í sérhverri "grein" í `switch` setningu vegna þess að án `break` setningar þá "fellur" keyrsluflæðið í næsta tilvik (e. case). Án `break` setninga myndi táknið `+` verða þess valdandi að forritið myndi framkvæma samlagningu (e. addition), síðan margföldun (e. multiplication) og að lokum prenta út villuskilaboðin. Af og til er þessi eiginleiki reyndar gagnlegur en í flestum tilvikum koma upp villur í keyrslu þegar `break` setningar gleymast.

`switch` setningar virka fyrir heiltölur, stafi og upptalningatög. Til að breyta `Suit` í samsvarandi streng þá getum við t.d. gert eftirfarandi:

```
switch (suit) {
case CLUBS:     return "Clubs";
case DIAMONDS:  return "Diamonds";
case HEARTS:    return "Hearts";
case SPADES:    return "Spades";
default:        return "Not a valid suit";
}
```

Í þessu dæmi þurfum við ekki á `break` setningu að halda vegna þess að `return` setningar valda því að keyrsluflæðið fer til baka til þess sem kallaði í stað þess að "falla" í næsta tilvik.

Almennt séð er góður forritunarstíll að hafa sjálfgefið (`default`) tilvik í sérhverri `switch` setningu til að meðhöndla villur eða óvænt gildi.

13.3 Stokkar

Í köflunum hér á undan unnum við með vektor af hlutum en ég nefndi það líka að það væri hægt að búa til hluti sem innihalda vektor sem meðlimabreytu. Í þessum kafla ætla ég að búa til nýjan hlut, `Deck`, sem inniheldur vektor af `Card`.

Strúktúrskilgreiningin lítur svona út:

```
struct Deck {
  vector<Card> cards;

  Deck (int n);
};

Deck::Deck (int size)
{
  vector<Card> temp (size);
  cards = temp;
}
```

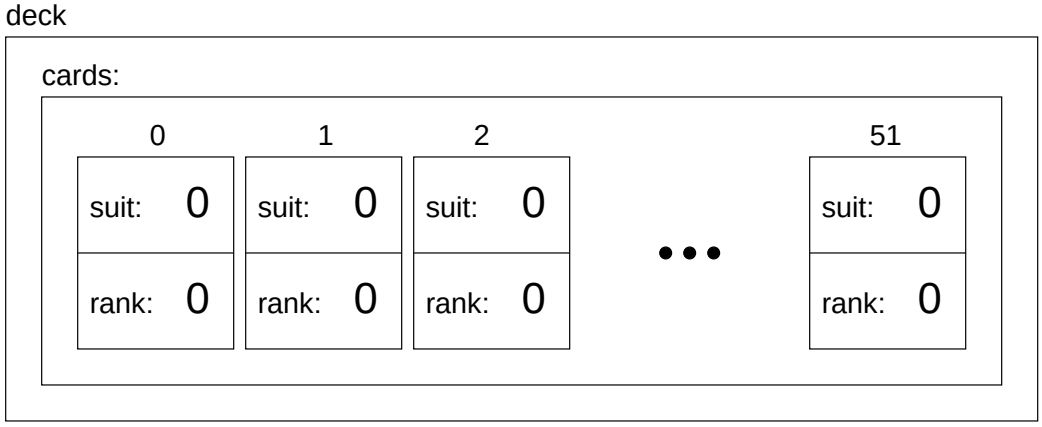
Nafnið á meðlimabreytunni er `cards` sem hjálpar okkur að gera greinarmun á `Deck` (stokknum) sjálfum og vektor af `Card` sem stokkurinn inniheldur.

Sem stendur er aðeins um einn smið að ræða. Sá nýtir sér staðværa breytu með nafninu `temp` sem er upphafsstillt með því að vekja upp smiðinn í `vector` klasanum (stærðin, size, er send sem viðfang). Síðan afritar smiðurinn `temp` yfir í meðlimabreytuna `cards`.

Nú getum við þá búið til stokk af spilum á eftirfarandi hátt:

```
  Deck deck (52);
```

Stöðurit fyrir `Deck` hlut lítur svona út:



Hluturinn með nafnið `deck` á sér eina meðlimabreytu með nafnið `cards` sem er vektor af `Card` hlutum. Til að nálgast spil í stokknum þurfum við að setja

saman málskipanina fyrir að nálgast meðlimabreytu og málskipanina fyrir að velja stak úr vektor. Segðin `deck.cards[i]` er t.d. i-ta spilið í stokknum og `deck.cards[i].suit` er litur þess.

Eftirfarandi lykkja

```
for (int i = 0; i<52; i++) {
  deck.cards[i].print();
}
```

sýnir hvernig er hægt að ferðast um í stokknum og prenta út sérhvert spil.

13.4 Annar smiður

Það getur verið gagnlegt að upphafsstilla spilin í `Deck` hlutnum. Við gætum notað fallið `buildDeck` úr fyrri kafla (með smávægilegum breytingum) en það er líklega eðlilegra að skrifa annan smið í `Deck`.

```
Deck::Deck ()
{
  vector<Card> temp (52);
  cards = temp;

  int i = 0;
  for (Suit suit = CLUBS; suit <= SPADES; suit = Suit(suit+1)) {
    for (Rank rank = ACE; rank <= KING; rank = Rank(rank+1)) {
      cards[i].suit = suit;
      cards[i].rank = rank;
      i++;
    }
  }
}
```

Taktu eftir því hversu líkt þetta fall er `buildDeck` – við þurftum aðeins að breyta málskipaninni til að gera þetta að smið. Nú getum við búið til hefðbundinn 52-spila stokk með einfaldri yfirlýsingu: `Deck deck;`

13.5 Meðlimaföll í `Deck`

Nú þegar við höfum sérstakan `Deck` hlut þá er eðlilegt að gera öll föll, sem hafa með stokk að gera, að meðlimaföllum í `Deck`. Einn augljós kandídat er fallið `printDeck` (kafli 12.7). Svona lítur það út eftir að því hefur verið breytt í meðlimafall í `Deck`:

```
void Deck::print () const {
  for (int i = 0; i < cards.length(); i++) {
    cards[i].print ();
```

```
  }
}
```

Að venju getum við vísað í meðlimabreytu núverandi hlutar (t.d. `cards`) án þess að nota punktatáknun (e. dot notation).

Það er hins vegar ekki augljóst fyrir sum önnur föll hvort þau ættu að vera meðlimaföll í `Card`, í `Deck` eða föll sem standa eitt og sér (e. nonmember function) og taka `Card` og `Deck` hluti sem viðföng. Útgáfan af `find` úr fyrri kafla tekur t.d. `Card` og `Deck` hluti sem viðföng en við gætum gert fallið að meðlimafalli annars hvors hlutarins. Til að æfa þig skaltu núna endurskrifa `find` sem meðlimafall í `Deck` sem tekur `Card` hlut sem viðfang.

Það að skrifa `find` sem meðlimafall í `Card` er aftur á móti dálítið snúið. Hér er mín útgáfa:

```
int Card::find (const Deck& deck) const {
  for (int i = 0; i < deck.cards.length(); i++) {
    if (equals (deck.cards[i], *this)) return i;
  }
  return -1;
}
```

Fyrsta "vandamálið" er að við verðum að nota lykilorðið `this` til að vísa í það spil sem fallið er vakið upp með.

Annað vandamálið er það að C++ gerir ekki auðvelt að skrifa strúktúrskilgreiningar sem vísa hvor á aðra. Vandamálið er að þegar þýðandinn les fyrri skilgreininguna þá hefur það ekki séð þá seinni.

Ein lausn er að lýsa `Deck` yfir (e. declare) á undan `Card` og síðan skilgreina (e. define) `Deck` eftir það:

```
// declare that Deck is a structure, without defining it
struct Deck;

// that way we can refer to it in the definition of Card
struct Card
{
  int suit, rank;

  Card ();
  Card (int s, int r);

  void print () const;
  bool isGreater (const Card& c2) const;
  int find (const Deck& deck) const;
};

// and then later we provide the definition of Deck
struct Deck {
```

```
  vector<Card> cards;

  Deck ();
  Deck (int n);
  void print () const;
  int find (const Card& card) const;
};
```

13.6 Að stokka

Í flestum spilum þarf að vera hægt að stokka spilin, þ.e. setja þau í handahófskennda röð í stokknum. Við sáum í kafla 10.6 hvernig hægt er að búa til slembitölur en það er ekki augljóst hvernig hægt er að nota þær til að stokka spil.

Ein leið er sú að líkja eftir því hvernig við sjálf stokkum, t.d. að skipta stokknum í tvennt og setja hann svo saman á ný með því að velja eitt spil í einu frá hvorum helmingnum. Þetta er kallað fullkomin stokkun (e. perfect shuffle). Það getur tekið okkur um 7 ítranir þangað til spilin eru vel stokkuð með þessari aðferð. Forrit hefur hins vegar þann pirrandi eiginleika að stokka fullkomlega í sérhvert sinn sem er í raun ekki mjög handahófskennt! Það vill reyndar svo til að eftir 8 fullkomnar stokkanir þá er stokkurinn í sömu röð og þegar byrjað var. Þú getur séð umfjöllun um þessa fullyrðingu á `http://www.wiskit.com/marilyn/craig.html` eða leitað á vefnum með leitarorðunum "perfect shuffle."

Betra stokkunaralgrím er að ferðast um í stokknum, spil fyrir spil, og skipta á núverandi spili og öðru spili (sem valið er af handahófi) í sérhverri ítrun.

Hér má sjá drög af því hvernig þetta algrím virkar. Til að setja fram forritið nota ég sambland af C++ setningum og ensku – þetta er stundum kallað called **sauðakóði** (e. pseudocode):

```
for (int i=0; i<cards.length(); i++) {
  // choose a random number between i and cards.length()
  // swap the ith card and the randomly-chosen card
}
```

Kosturinn við að nota sauðakóða er oft sá að hann gefur skýrt til kynna hvaða föll eru nauðsynleg. Í þessu tilviki þurfum við fall eins og `randomInt` sem velur heiltölu, á handahófskenndan hátt, á milli viðfanganna `low` og `high`, og fall eins og `swapCards` sem tekur tvo vísa og skiptir á spilunum í viðkomandi sætum.

Þú ættir að geta fundið út hvernig skrifa á `randomInt` með því að kíkja til baka í kafla 10.6. Þú þarft þó að passa þig á því að búa ekki til vísa sem eru út fyrir bilið (e. out of range).

Þú ættir líka að geta skrifað `swapCards` sjálf(ur).

13.7 Röðun

Nú þegar við getum stokkað spilin þurfum við líka að geta raðað þeim í rétta röð. Það vill reyndar svo til að algrímið fyrir röðun er mjög svipað stokkunar-algríminu.

Við munum aftur ferðast um í spilastokknum og skipta á núverandi spili og öðru spili í sérhverri ítrun. Eini munurinn er sá að í stað þess að velja annað spil af handahófi þá munum við finna lægsta spilið sem eftir er í stokknum. Það sem ég á við með "eftir er í stokknum" eru spil sem eru jöfn eða hærri heldur er núverandi vísir `i`.

```
for (int i=0; i<cards.length(); i++) {
  // find the lowest card at or to the right of i
  // swap the ith card and the lowest card
}
```

Sauðakóðinn hjálpar hér aftur við hönnunina á **hjálparföllum**. Í þessu tilviki getum við aftur notað `swapCards` fallið og við þurfum því einungis eitt nýtt fall, `findLowestCard`, sem tekur vektor af spilum og vísi sem gefur til kynna hvar byrja á að leita í vektornum.

Þetta ferli að nota sauðakóða til að finna út hvaða hjálparföll eru nauðsynleg eru stundum kallað **ofansækin hönnun** (e. top-down design), sem er andstaðan við **neðansækna hönnun** (e. bottom-up design) sem ég ræddi um í kafla 10.9.

Þú ættir núna að spreyta þig á útfærslunni á þessum sauðakóða.

13.8 Hlutstokkur

Hvernig ættum við að tákna hönd eða annað hlutmengi af spilastokki (hlut-stokk)? Ein einföld leið er að búa til `Deck` hlut sem hefur færri en 52 spil.

Við gætum skrifað fall, `subdeck`, sem tekur vektor af spilum og vísa (sem tákna bil) og skilar nýjum vektor af spilum sem innheldur viðkomandi hlutmengi úr stokknum:

```
Deck Deck::subdeck (int low, int high) const {
  Deck sub (high-low+1);

  for (int i = 0; i<sub.cards.length(); i++) {
    sub.cards[i] = cards[low+i];
  }
  return sub;
}
```

Til að búa til staðværu breytuna `subdeck` notum við hér smiðinn í `Deck` sem tekur stærðina á nýja stokknum (hlutstokknum) sem viðfang og upphafsstillir ekki spilin í stokknum. Spilin eru í raun upphafsstillt þegar þau eru afrituð úr upphaflega stokknum.

Stærðin á hlutstokknum er `high-low+1` vegna þess að spilin í bæði `low` og `high` eru innifalin. Svona útreikningur getur verið ruglandi og hefur oft í för með sér "off-by-one" villur. Það er oft gott að teikna mynd til að koma í veg fyrir þessar villur.

13.9 Að stokka og gefa

Í kafla 13.6 skrifaði ég sauðakóða fyrir stokkunaralgrímið. Ef við gerum nú ráð fyrir því að eiga fallið `shuffle`, sem stokkar viðfang sitt, þá getum við búið til og stokkað spilastokk:

```
Deck deck;              // create a standard 52-card deck
deck.shuffle ();        // shuffle it
```

Síðan getum við gefið spil með því að nota `subdeck`:

```
Deck hand1 = deck.subdeck (0, 4);
Deck hand2 = deck.subdeck (5, 9);
Deck pack = deck.subdeck (10, 51);
```

Þessi kóði gefur fyrstu 5 spilin til einnar handar, næstu 5 spil til annarrar handar og afgangurinn fer í bunkann.

Hvarflaði að þér að við ættum að gefa eitt spil í einu til sérhverrar handar eins og algengt er í raunverulegum spilum? Ég hugsaði um það en áttaði mig svo á því að það væri ekki nauðsynlegt í forriti. Það að gefa eitt spil í einu er yfirleitt gert til að leiðrétta ófullkomna stokkun og gera það að verkum að erfiðara sé fyrir þann sem gefur að svindla. Í tilviki tölvu þarf ekki að hugsa um þessi atriði.

13.10 Mergesort

Í kafla 13.7 sáum við einfalt röðunaralgrím sem vill svo til að er ekki mjög skilvirkt. Til að raða n hlutum (spilum) þarf algrímið að ferðast um í vektornum n sinnum (for-lykkjan) og sérhver umferð (finna lægsta spil) tekur tíma sem er í réttu hlutfalli við n. Heildartíminn sem algrímið tekur er því í réttu hlutfalli við n^2.

Í þessum kafla mun ég setja fram mun skilvirkara röðunaralgrím sem kallast **mergesort**. Tíminn sem mergesort tekur að raða n hlutum er í réttu hlutfalli við $n \log n$. Þetta virðist ekki vera mikil bæting en eftir því sem n stækkar því meiri munur verður á n^2 og $n \log n$. Prófaðu nokkur gildi fyrir n og sjáðu muninn.

Grundvallarhugmyndin á bak við mergesort er eftirfarandi: Ef þú hefur tvo hlutstokka, sem hvor um sig er þegar raðaður, þá er einfalt (og skilvirkt) að setja þá saman (e. merge) í einn raðaðan stokk. Prófaðu þetta með raunverulegum spilastokki:

1. Búðu til tvo hlutstokka með um 10 spilum í hvorum fyrir sig og raðaðu þeim þannig að þegar spilin snúa upp þá er lægsta spilið efst. Settu báða stokkana fyrir framan þig (þannig að spilin snúi upp).

2. Berðu saman efstu spilin í hvorum stokki og veldu það lægra. Snúðu því við og bættu því við nýja samsetta stokkinn (sem upphaflega er tómur).

3. Endurtaktu skref tvö þangað til annar stokkurinn er tómur. Bættu þá restinni af spilunum við samsetta stokkinn.

Niðurstaðan ætti að vera einn raðaður stokkur. Svona lítur þetta út í sauðakóða:

```
Deck merge (const Deck& d1, const Deck& d2) {
  // create a new deck big enough for all the cards
  Deck result (d1.cards.length() + d2.cards.length());

  // use the index i to keep track of where we are in
  // the first deck, and the index j for the second deck
  int i = 0;
  int j = 0;

  // the index k traverses the result deck
  for (int k = 0; k<result.cards.length(); k++) {

    // if d1 is empty, d2 wins; if d2 is empty, d1 wins;
    // otherwise, compare the two cards

    // add the winner to the new deck
  }
  return result;
}
```

Ég ákvað að gera `merge` að falli sem stendur eitt og sér (e. nonmember function) vegna þess að viðföngin tvö eru samhverf (e. symmetric).

Besta leiðin til að prófa `merge` er að búa til stokk, nota subdeck til að búa til tvær (litlar) hendur og nota síðan röðunaralgrímið úr síðasta kafla til að raða höndunum tveimur. Eftir það getur þú sent hendurnar tvær (hlutstokkana) inn í `merge` til að sjá hvort það virkar.

Ef þú færð þetta til að virka ættir þú að prófa eftirfarandi útfærslu af `mergeSort`:

```
Deck Deck::mergeSort () const {
  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using sort
  // merge the two halves and return the result
}
```

Taktu eftir því að núverandi hlutur er skilgreindur sem `const` vegna þess að `mergeSort` breytir honum ekki. Í staðinn býr fallið til nýjan `Deck` hlut og skilar honum.

Fjörið byrjar fyrst núna ef þetta gengur hjá þér! Það sem er töfrandi við mergesort er að fallið er endurkvæmt. Af hverju ættir þú að kalla á gömlu, óskilvirku útgáfunum af `sort` þegar þú ætlar að raða hlutstokkunum? Af hverju ekki að kalla á `mergeSort` sem þú ert einmitt að skrifa?

Það er ekki einungis góð hugmynd heldur er það *nauðsynlegt* til að algrímið í heild sinni verði skilvirkara eins og ég lofaði. Til að fá það til að virka verðum við þó að bæta við grunnþrepi þannig að ekki verði um óendanlega endurkvæmni að ræða. Einfalt grunnþrep er hlutstokkur með 0 eða einu spili. Ef `mergesort` fær svona lítinn hlutstokk þá getur fallið einfaldlega skilað stokknum óbreyttum til baka því hann er þá sannarlega þegar raðaður.

Endurkvæma útgáfan af `mergesort` lítur einhvern veginn svona út:

```
Deck Deck::mergeSort (Deck deck) const {
  // if the deck is 0 or 1 cards, return it

  // find the midpoint of the deck
  // divide the deck into two subdecks
  // sort the subdecks using mergesort
  // merge the two halves and return the result
}
```

Að venju eru tvær leiðir til að hugsa um endurkvæm föll: Hægt er fylgja keyrsluflæði þess í heild sinni eða "taka það trúanlegt" (e. make the leap of faith). Ég setti þetta dæmi fram vísvitandi til að hvetja þig til að "taka það trúanlegt".

Þegar þú notaðir `sort` til að raða hlutstokkum þá fannst þér ekki ástæða til að fylgja keyrsluflæði þess í heild sinni, ekki satt? Þú gerðir einfaldlega ráð fyrir því að `sort` fallið virkaði vegna þess að þú hafðir áður aflúsað það. Eina sem þú þurftir að breyta til að gera `mergeSort` endurkvæmt var að skipta út einu röðunaralgrími fyrir annað. Það er engin ástæða til að líta á nýja forritið á annan hátt.

Reyndar þurftir þú að hugsa um að ná grunnþrepinu réttu og að það myndi á einhverjum tímapunkti reyna á grunnþrepið en að öðru leyti ætti ekki að vera neitt sérstakt vandamál að skrifa endurkvæmu lausnina. Gangi þér vel!

13.11 Orðalisti

sauðakóði (e. pseudocode): Leið til að hanna forrit með því að skrifa gróf drög í blöndu af náttúrulegu tungumáli og C++.

hjálparfall (e. helper function): Oftast lítið fall, sem eitt og sér gerir ekki neitt sérlega gagnlegt, en hjálpar öðru gagnlegra falli.

mergesort: Algrím til að raða safni af gildum. Mergesort er skilvirkara en hið einfalda röðunaralgrím í fyrri kafla, sérstaklega fyrir stór söfn.

Kafli 14

Classes and invariants

14.1 Private data and classes

I have used the word "encapsulation" in this book to refer to the process of wrapping up a sequence of instructions in a function, in order to separate the function's interface (how to use it) from its implementation (how it does what it does).

This kind of encapsulation might be called "functional encapsulation," to distinguish it from "data encapsulation," which is the topic of this chapter. Data encapsulation is based on the idea that each structure definition should provide a set of functions that apply to the structure, and prevent unrestricted access to the internal representation.

One use of data encapsulation is to hide implementation details from users or programmers that don't need to know them.

For example, there are many possible representations for a `Card`, including two integers, two strings and two enumerated types. The programmer who writes the `Card` member functions needs to know which implementation to use, but someone using the `Card` structure should not have to know anything about its internal structure.

As another example, we have been using `apstring` and `apvector` objects without ever discussing their implementations. There are many possibilities, but as "clients" of these libraries, we don't need to know.

In C++, the most common way to enforce data encapsulation is to prevent client programs from accessing the instance variables of an object. The keyword `private` is used to protect parts of a structure definition. For example, we could have written the `Card` definition:

```
struct Card
{
private:
  int suit, rank;
```

```
public:
  Card ();
  Card (int s, int r);

  int getRank () const { return rank; }
  int getSuit () const { return suit; }
  void setRank (int r) { rank = r; }
  void setSuit (int s) { suit = s; }
};
```

There are two sections of this definition, a private part and a public part. The functions are public, which means that they can be invoked by client programs. The instance variables are private, which means that they can be read and written only by `Card` member functions.

It is still possible for client programs to read and write the instance variables using the **accessor functions** (the ones beginning with `get` and `set`). On the other hand, it is now easy to control which operations clients can perform on which instance variables. For example, it might be a good idea to make cards "read only" so that after they are constructed, they cannot be changed. To do that, all we have to do is remove the `set` functions.

Another advantage of using accessor functions is that we can change the internal representations of cards without having to change any client programs.

14.2 What is a class?

In most object-oriented programming languages, a **class** is a user-defined type that includes a set of functions. As we have seen, structures in C++ meet the general definition of a class.

But there is another feature in C++ that also meets this definition; confusingly, it is called a `class`. In C++, a class is just a structure whose instance variables are private by default. For example, I could have written the `Card` definition:

```
class Card
{
  int suit, rank;

public:
  Card ();
  Card (int s, int r);

  int getRank () const { return rank; }
  int getSuit () const { return suit; }
  int setRank (int r) { rank = r; }
  int setSuit (int s) { suit = s; }
};
```

I replaced the word `struct` with the word `class` and removed the `private:` label. This result of the two definitions is exactly the same.

In fact, anything that can be written as a `struct` can also be written as a `class`, just by adding or removing labels. There is no real reason to choose one over the other, except that as a stylistic choice, most C++ programmers use `class`.

Also, it is common to refer to all user-defined types in C++ as "classes," regardless of whether they are defined as a `struct` or a `class`.

14.3 Complex numbers

As a running example for the rest of this chapter we will consider a class definition for complex numbers. Complex numbers are useful for many branches of mathematics and engineering, and many computations are performed using complex arithmetic. A complex number is the sum of a real part and an imaginary part, and is usually written in the form $x + yi$, where x is the real part, y is the imaginary part, and i represents the square root of -1.

The following is a class definition for a user-defined type called `Complex`:

```
class Complex
{
  double real, imag;

public:
  Complex () { }
  Complex (double r, double i) { real = r;  imag = i; }
};
```

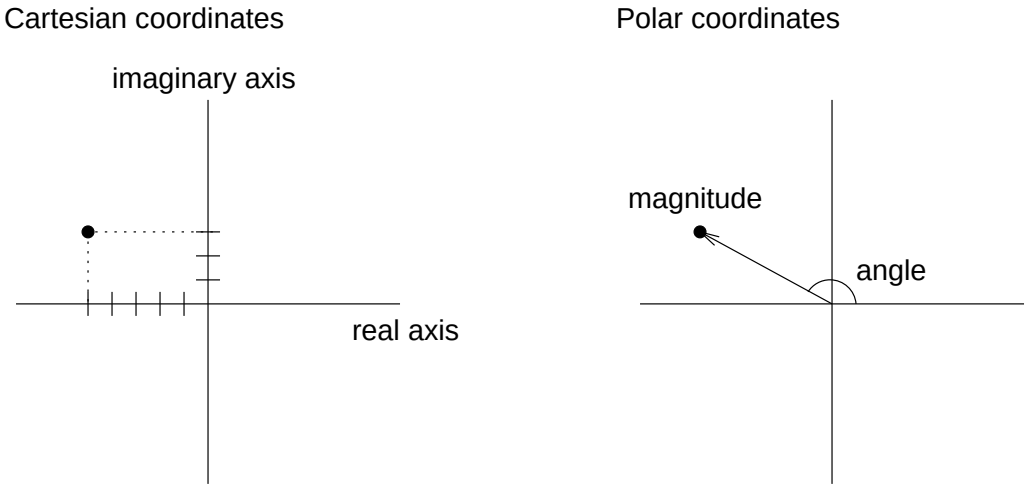
Because this is a `class` definition, the instance variables `real` and `imag` are private, and we have to include the label `public:` to allow client code to invoke the constructors.

As usual, there are two constructors: one takes no parameters and does nothing; the other takes two parameters and uses them to initialize the instance variables.

So far there is no real advantage to making the instance variables private. Let's make things a little more complicated; then the point might be clearer.

There is another common representation for complex numbers that is sometimes called "polar form" because it is based on polar coordinates. Instead of specifying the real part and the imaginary part of a point in the complex plane, polar coordinates specify the direction (or angle) of the point relative to the origin, and the distance (or magnitude) of the point.

The following figure shows the two coordinate systems graphically.



Complex numbers in polar coordinates are written $re^{i\theta}$, where r is the magnitude (radius), and θ is the angle in radians.

Fortunately, it is easy to convert from one form to another. To go from Cartesian to polar,

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan(y/x) \end{aligned}$$

To go from polar to Cartesian,

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}$$

So which representation should we use? Well, the whole reason there are multiple representations is that some operations are easier to perform in Cartesian coordinates (like addition), and others are easier in polar coordinates (like multiplication). One option is that we can write a class definition that uses *both* representations, and that converts between them automatically, as needed.

```
class Complex
{
  double real, imag;
  double mag, theta;
  bool cartesian, polar;

public:
  Complex () { cartesian = false;  polar = false; }

  Complex (double r, double i)
```

```
  {
    real = r;  imag = i;
    cartesian = true;  polar = false;
  }
};
```

There are now six instance variables, which means that this representation will take up more space than either of the others, but we will see that it is very versatile.

Four of the instance variables are self-explanatory. They contain the real part, the imaginary part, the angle and the magnitude of the complex number. The other two variables, `cartesian` and `polar` are flags that indicate whether the corresponding values are currently valid.

For example, the do-nothing constructor sets both flags to false to indicate that this object does not contain a valid complex number (yet), in either representation.

The second constructor uses the parameters to initialize the real and imaginary parts, but it does not calculate the magnitude or angle. Setting the `polar` flag to false warns other functions not to access `mag` or `theta` until they have been set.

Now it should be clearer why we need to keep the instance variables private. If client programs were allowed unrestricted access, it would be easy for them to make errors by reading uninitialized values. In the next few sections, we will develop accessor functions that will make those kinds of mistakes impossible.

14.4 Accessor functions

By convention, accessor functions have names that begin with `get` and end with the name of the instance variable they fetch. The return type, naturally, is the type of the corresponding instance variable.

In this case, the accessor functions give us an opportunity to make sure that the value of the variable is valid before we return it. Here's what `getReal` looks like:

```
double Complex::getReal ()
{
  if (cartesian == false) calculateCartesian ();
  return real;
}
```

If the `cartesian` flag is true then `real` contains valid data, and we can just return it. Otherwise, we have to call `calculateCartesian` to convert from polar coordinates to Cartesian coordinates:

```
void Complex::calculateCartesian ()
{
```

```
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

Assuming that the polar coordinates are valid, we can calculate the Cartesian coordinates using the formulas from the previous section. Then we set the `cartesian` flag, indicating that `real` and `imag` now contain valid data.

As an exercise, write a corresponding function called `calculatePolar` and then write `getMag` and `getTheta`. One unusual thing about these accessor functions is that they are not `const`, because invoking them might modify the instance variables.

14.5 Output

As usual when we define a new class, we want to be able to output objects in a human-readable form. For `Complex` objects, we could use two functions:

```
void Complex::printCartesian ()
{
  cout << getReal() << " + " << getImag() << "i" << endl;
}

void Complex::printPolar ()
{
  cout << getMag() << " e^ " << getTheta() << "i" << endl;
}
```

The nice thing here is that we can output any `Complex` object in either format without having to worry about the representation. Since the output functions use the accessor functions, the program will compute automatically any values that are needed.

The following code creates a `Complex` object using the second constructor. Initially, it is in Cartesian format only. When we invoke `printCartesian` it accesses `real` and `imag` without having to do any conversions.

```
  Complex c1 (2.0, 3.0);

  c1.printCartesian();
  c1.printPolar();
```

When we invoke `printPolar`, and `printPolar` invokes `getMag`, the program is forced to convert to polar coordinates and store the results in the instance variables. The good news is that we only have to do the conversion once. When `printPolar` invokes `getTheta`, it will see that the polar coordinates are valid and return `theta` immediately.

The output of this code is:

```
2 + 3i
3.60555 e^ 0.982794i
```

14.6 A function on Complex numbers

A natural operation we might want to perform on complex numbers is addition. If the numbers are in Cartesian coordinates, addition is easy: you just add the real parts together and the imaginary parts together. If the numbers are in polar coordinates, it is easiest to convert them to Cartesian coordinates and then add them.

Again, it is easy to deal with these cases if we use the accessor functions:

```
Complex add (Complex& a, Complex& b)
{
  double real = a.getReal() + b.getReal();
  double imag = a.getImag() + b.getImag();
  Complex sum (real, imag);
  return sum;
}
```

Notice that the arguments to `add` are not `const` because they might be modified when we invoke the accessors. To invoke this function, we would pass both operands as arguments:

```
  Complex c1 (2.0, 3.0);
  Complex c2 (3.0, 4.0);

  Complex sum = add (c1, c2);
  sum.printCartesian();
```

The output of this program is

```
5 + 7i
```

14.7 Another function on Complex numbers

Another operation we might want is multiplication. Unlike addition, multiplication is easy if the numbers are in polar coordinates and hard if they are in Cartesian coordinates (well, a little harder, anyway).

In polar coordinates, we can just multiply the magnitudes and add the angles. As usual, we can use the accessor functions without worrying about the representation of the objects.

```
Complex mult (Complex& a, Complex& b)
{
  double mag = a.getMag() * b.getMag()
```

```
  double theta = a.getTheta() + b.getTheta();
  Complex product;
  product.setPolar (mag, theta);
  return product;
}
```

A small problem we encounter here is that we have no constructor that accepts polar coordinates. It would be nice to write one, but remember that we can only overload a function (even a constructor) if the different versions take different parameters. In this case, we would like a second constructor that also takes two `doubles`, and we can't have that.

An alternative it to provide an accessor function that *sets* the instance variables. In order to do that properly, though, we have to make sure that when `mag` and `theta` are set, we also set the `polar` flag. At the same time, we have to make sure that the `cartesian` flag is unset. That's because if we change the polar coordinates, the cartesian coordinates are no longer valid.

```
void Complex::setPolar (double m, double t)
{
  mag = m;  theta = t;
  cartesian = false;  polar = true;
}
```

As an exercise, write the corresponding function named `setCartesian`.

To test the `mult` function, we can try something like:

```
  Complex c1 (2.0, 3.0);
  Complex c2 (3.0, 4.0);

  Complex product = mult (c1, c2);
  product.printCartesian();
```

The output of this program is

```
-6 + 17i
```

There is a lot of conversion going on in this program behind the scenes. When we call `mult`, both arguments get converted to polar coordinates. The result is also in polar format, so when we invoke `printCartesian` it has to get converted back. Really, it's amazing that we get the right answer!

14.8 Invariants

There are several conditions we expect to be true for a proper `Complex` object. For example, if the `cartesian` flag is set then we expect `real` and `imag` to contain valid data. Similarly, if `polar` is set, we expect `mag` and `theta` to be valid. Finally, if both flags are set then we expect the other four variables to

be consistent; that is, they should be specifying the same point in two different formats.

These kinds of conditions are called `invariants`, for the obvious reason that they do not vary—they are always supposed to be true. One of the ways to write good quality code that contains few bugs is to figure out what invariants are appropriate for your classes, and write code that makes it impossible to violate them.

One of the primary things that data encapsulation is good for is helping to enforce invariants. The first step is to prevent unrestricted access to the instance variables by making them private. Then the only way to modify the object is through accessor functions and modifiers. If we examine all the accessors and modifiers, and we can show that every one of them maintains the invariants, then we can prove that it is impossible for an invariant to be violated.

Looking at the `Complex` class, we can list the functions that make assignments to one or more instance variables:

```
the second constructor
calculateCartesian
calculatePolar
setCartesian
setPolar
```

In each case, it is straightforward to show that the function maintains each of the invariants I listed. We have to be a little careful, though. Notice that I said "maintain" the invariant. What that means is "If the invariant is true when the function is called, it will still be true when the function is complete."

That definition allows two loopholes. First, there may be some point in the middle of the function when the invariant is not true. That's ok, and in some cases unavoidable. As long as the invariant is restored by the end of the function, all is well.

The other loophole is that we only have to maintain the invariant if it was true at the beginning of the function. Otherwise, all bets are off. If the invariant was violated somewhere else in the program, usually the best we can do is detect the error, output an error message, and exit.

14.9 Preconditions

Often when you write a function you make implicit assumptions about the parameters you receive. If those assumptions turn out to be true, then everything is fine; if not, your program might crash.

To make your programs more robust, it is a good idea to think about your assumptions explicitly, document them as part of the program, and maybe write code that checks them.

For example, let's take another look at `calculateCartesian`. Is there an assumption we make about the current object? Yes, we assume that the `polar`

flag is set and that `mag` and `theta` contain valid data. If that is not true, then this function will produce meaningless results.

One option is to add a comment to the function that warns programmers about the **precondition**.

```
void Complex::calculateCartesian ()
// precondition: the current object contains valid polar coordinates
and the polar flag is set
// postcondition: the current object will contain valid Cartesian
coordinates and valid polar coordinates, and both the cartesian
flag and the polar flag will be set
{
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

At the same time, I also commented on the **postconditions**, the things we know will be true when the function completes.

These comments are useful for people reading your programs, but it is an even better idea to add code that *checks* the preconditions, so that we can print an appropriate error message:

```
void Complex::calculateCartesian ()
{
  if (polar == false) {
    cout <<
    "calculateCartesian failed because the polar representation is invalid"
 << endl;
    exit (1);
  }
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
}
```

The `exit` function causes the program to quit immediately. The return value is an error code that tells the system (or whoever executed the program) that something went wrong.

This kind of error-checking is so common that C++ provides a built-in function to check preconditions and print error messages. If you include the `assert.h` header file, you get a function called `assert` that takes a boolean value (or a conditional expression) as an argument. As long as the argument is true, `assert` does nothing. If the argument is false, assert prints an error message and quits. Here's how to use it:

```
void Complex::calculateCartesian ()
```

```
{
  assert (polar);
  real = mag * cos (theta);
  imag = mag * sin (theta);
  cartesian = true;
  assert (polar && cartesian);
}
```

The first `assert` statement checks the precondition (actually just part of it); the second `assert` statement checks the postcondition.

In my development environment, I get the following message when I violate an assertion:

```
Complex.cpp:63: void Complex::calculatePolar(): Assertion 'cartesian' failed.
Abort
```

There is a lot of information here to help me track down the error, including the file name and line number of the assertion that failed, the function name and the contents of the assert statement.

14.10 Private functions

In some cases, there are member functions that are used internally by a class, but that should not be invoked by client programs. For example, `calculatePolar` and `calculateCartesian` are used by the accessor functions, but there is probably no reason clients should call them directly (although it would not do any harm). If we wanted to protect these functions, we could declare them `private` the same way we do with instance variables. In that case the complete class definition for `Complex` would look like:

```
class Complex
{
private:
  double real, imag;
  double mag, theta;
  bool cartesian, polar;

  void calculateCartesian ();
  void calculatePolar ();

public:
  Complex () { cartesian = false;  polar = false; }

  Complex (double r, double i)
  {
    real = r;  imag = i;
```

```
    cartesian = true;  polar = false;
  }

  void printCartesian ();
  void printPolar ();

  double getReal ();
  double getImag ();
  double getMag ();
  double getTheta ();

  void setCartesian (double r, double i);
  void setPolar (double m, double t);
};
```

The `private` label at the beginning is not necessary, but it is a useful reminder.

14.11 Glossary

class: In general use, a class is a user-defined type with member functions. In C++, a class is a structure with private instance variables.

accessor function: A function that provides access (read or write) to a private instance variable.

invariant: A condition, usually pertaining to an object, that should be true at all times in client code, and that should be maintained by all member functions.

precondition: A condition that is assumed to be true at the beginning of a function. If the precondition is not true, the function may not work. It is often a good idea for functions to check their preconditions, if possible.

postcondition: A condition that is true at the end of a function.

Kafli 15

File Input/Output and `apmatrixes`

In this chapter we will develop a program that reads and writes files, parses input, and demonstrates the `apmatrix` class. We will also implement a data structure called `Set` that expands automatically as you add elements.

Aside from demonstrating all these features, the real purpose of the program is to generate a two-dimensional table of the distances between cities in the United States. The output is a table that looks like this:

```
Atlanta 0
Chicago 700     0
Boston  1100    1000    0
Dallas  800     900     1750    0
Denver  1450    1000    2000    800     0
Detroit 750     300     800     1150    1300    0
Orlando 400     1150    1300    1100    1900    1200    0
Phoenix 1850    1750    2650    1000    800     2000    2100    0
Seattle 2650    2000    3000    2150    1350    2300    3100    1450    0
        Atlanta Chicago Boston  Dallas  Denver  Detroit Orlando Phoenix Seattle
```

The diagonal elements are all zero because that is the distance from a city to itself. Also, because the distance from A to B is the same as the distance from B to A, there is no need to print the top half of the matrix.

15.1 Streams

To get input from a file or send output to a file, you have to create an `ifstream` object (for input files) or an `ofstream` object (for output files). These objects are defined in the header file `fstream`, which you have to include.

A **stream** is an abstract object that represents the flow of data from a source like the keyboard or a file to a destination like the screen or a file.

We have already worked with two streams: `cin`, which has type `istream`, and `cout`, which has type `ostream`. `cin` represents the flow of data from the keyboard to the program. Each time the program uses the » operator or the `getline` function, it removes a piece of data from the input stream.

Similarly, when the program uses the « operator on an `ostream`, it adds a datum to the outgoing stream.

15.2 File input

To get data from a file, we have to create a stream that flows from the file into the program. We can do that using the `ifstream` constructor.

```
ifstream infile ("file-name");
```

The argument for this constructor is a string that contains the name of the file you want to open. The result is an object named `infile` that supports all the same operations as `cin`, including » and `getline`.

```
int x;
apstring line;

infile >> x;               // get a single integer and store in x
getline (infile, line);    // get a whole line and store in line
```

If we know ahead of time how much data is in a file, it is straightforward to write a loop that reads the entire file and then stops. More often, though, we want to read the entire file, but don't know how big it is.

There are member functions for `ifstreams` that check the status of the input stream; they are called `good`, `eof`, `fail` and `bad`. We will use `good` to make sure the file was opened successfully and `eof` to detect the "end of file."

Whenever you get data from an input stream, you don't know whether the attempt succeeded until you check. If the return value from `eof` is `true` then we have reached the end of the file and we know that the last attempt failed. Here is a program that reads lines from a file and displays them on the screen:

```
apstring fileName = ...;
ifstream infile (fileName.c_str());

if (infile.good() == false) {
  cout << "Unable to open the file named " << fileName;
  exit (1);
}

while (true) {
  getline (infile, line);
  if (infile.eof()) break;
```

```
    cout << line << endl;
  }
```

The function `c_str` converts an `apstring` to a native C string. Because the `ifstream` constructor expects a C string as an argument, we have to convert the `apstring`.

Immediately after opening the file, we invoke the `good` function. The return value is `false` if the system could not open the file, most likely because it does not exist, or you do not have permission to read it.

The statement `while(true)` is an idiom for an infinite loop. Usually there will be a `break` statement somewhere in the loop so that the program does not really run forever (although some programs do). In this case, the `break` statement allows us to exit the loop as soon as we detect the end of file.

It is important to exit the loop between the input statement and the output statement, so that when `getline` fails at the end of the file, we do not output the invalid data in `line`.

15.3 File output

Sending output to a file is similar. For example, we could modify the previous program to copy lines from one file to another.

```
ifstream infile ("input-file");
ofstream outfile ("output-file");

if (infile.good() == false || outfile.good() == false) {
  cout << "Unable to open one of the files." << endl;
  exit (1);
}

while (true) {
  getline (infile, line);
  if (infile.eof()) break;
  outfile << line << endl;
}
```

15.4 Parsing input

In Section 1.4 I defined "parsing" as the process of analyzing the structure of a sentence in a natural language or a statement in a formal language. For example, the compiler has to parse your program before it can translate it into machine language.

In addition, when you read input from a file or from the keyboard you often have to parse it in order to extract the information you want and detect errors.

For example, I have a file called `distances` that contains information about the distances between major cities in the United States. I got this information from a randomly-chosen web page

`http://www.jaring.my/usiskl/usa/distance.html`

so it may be wildly inaccurate, but that doesn't matter. The format of the file looks like this:

```
"Atlanta"       "Chicago"       700
"Atlanta"       "Boston"        1,100
"Atlanta"       "Chicago"       700
"Atlanta"       "Dallas"        800
"Atlanta"       "Denver"        1,450
"Atlanta"       "Detroit"       750
"Atlanta"       "Orlando"       400
```

Each line of the file contains the names of two cities in quotation marks and the distance between them in miles. The quotation marks are useful because they make it easy to deal with names that have more than one word, like "San Francisco."

By searching for the quotation marks in a line of input, we can find the beginning and end of each city name. Searching for special characters like quotation marks can be a little awkward, though, because the quotation mark is a special character in C++, used to identify string values.

If we want to find the first appearance of a quotation mark, we have to write something like:

```
  int index = line.find ('\"');
```

The argument here looks like a mess, but it represents a single character, a double quotation mark. The outermost single-quotes indicate that this is a character value, as usual. The backslash (\) indicates that we want to treat the next character literally. The sequence \" represents a quotation mark; the sequence \' represents a single-quote. Interestingly, the sequence \\ represents a single backslash. The first backslash indicates that we should take the second backslash seriously.

Parsing input lines consists of finding the beginning and end of each city name and using the `substr` function to extract the cities and distance. `substr` is an `apstring` member function; it takes two arguments, the starting index of the substring and the length.

```
void processLine (const apstring& line)
{
  // the character we are looking for is a quotation mark
  char quote = '\"';
```

```
  // store the indices of the quotation marks in a vector
  apvector<int> quoteIndex (4);

  // find the first quotation mark using the built-in find
  quoteIndex[0] = line.find (quote);

  // find the other quotation marks using the find from Chapter 7
  for (int i=1; i<4; i++) {
    quoteIndex[i] = find (line, quote, quoteIndex[i-1]+1);
  }

  // break the line up into substrings
  int len1 = quoteIndex[1] - quoteIndex[0] - 1;
  apstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  apstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  apstring distString = line.substr (quoteIndex[3]+1, len3);

  // output the extracted information
  cout << city1 << "\t" << city2 << "\t" << distString << endl;
}
```

Of course, just displaying the extracted information is not exactly what we want, but it is a good starting place.

15.5 Parsing numbers

The next task is to convert the numbers in the file from strings to integers. When people write large numbers, they often use commas to group the digits, as in 1,750. Most of the time when computers write large numbers, they don't include commas, and the built-in functions for reading numbers usually can't handle them. That makes the conversion a little more difficult, but it also provides an opportunity to write a comma-stripping function, so that's ok. Once we get rid of the commas, we can use the library function `atoi` to convert to integer. `atoi` is defined in the header file `cstdlib`.

To get rid of the commas, one option is to traverse the string and check whether each character is a digit. If so, we add it to the result string. At the end of the loop, the result string contains all the digits from the original string, in order.

```
int convertToInt (const apstring& s)
{
  apstring digitString = "";

  for (int i=0; i<s.length(); i++) {
```

```
    if (isdigit (s[i])) {
      digitString += s[i];
    }
  }
  return atoi (digitString.c_str());
}
```

The variable `digitString` is an example of an **accumulator**. It is similar to the counter we saw in Section 7.9, except that instead of getting incremented, it gets accumulates one new character at a time, using string concatentation.

The expression

```
digitString += s[i];
```

is equivalent to

```
digitString = digitString + s[i];
```

Both statements add a single character onto the end of the existing string.

Since `atoi` takes a C string as a parameter, we have to convert `digitString` to a C string before passing it as an argument.

15.6 The `Set` data structure

A data structure is a container for grouping a collection of data into a single object. We have seen some examples already, including `apstring`s, which are collections of characters, and `apvector`s which are collections on any type.

An ordered set is a collection of items with two defining properties:

Ordering: The elements of the set have indices associated with them. We can use these indices to identify elements of the set.

Uniqueness: No element appears in the set more than once. If you try to add an element to a set, and it already exists, there is no effect.

In addition, our implementation of an ordered set will have the following property:

Arbitrary size: As we add elements to the set, it expands to make room for new elements.

Both `apstring`s and `apvector`s have an ordering; every element has an index we can use to identify it. Both none of the data structures we have seen so far have the properties of uniqueness or arbitrary size.

To achieve uniqueness, we have to write an `add` function that searches the set to see if it already exists. To make the set expand as elements are added, we can take advantage of the `resize` function on `apvector`s.

Here is the beginning of a class definition for a `Set`.

```
class Set {
private:
  apvector<apstring> elements;
  int numElements;

public:
  Set (int n);

  int getNumElements () const;
  apstring getElement (int i) const;
  int find (const apstring& s) const;
  int add (const apstring& s);
};

Set::Set (int n)
{
  apvector<apstring> temp (n);
  elements = temp;
  numElements = 0;
}
```

The instance variables are an `apvector` of strings and an integer that keeps track of how many elements there are in the set. Keep in mind that the number of elements in the set, `numElements`, is not the same thing as the size of the `apvector`. Usually it will be smaller.

The `Set` constructor takes a single parameter, which is the initial size of the `apvector`. The initial number of elements is always zero.

`getNumElements` and `getElement` are accessor functions for the instance variables, which are private. `numElements` is a read-only variable, so we provide a `get` function but not a `set` function.

```
int Set::getNumElements () const
{
  return numElements;
}
```

Why do we have to prevent client programs from changing `getNumElements`? What are the invariants for this type, and how could a client program break an invariant. As we look at the rest of the `Set` member function, see if you can convince yourself that they all maintain the invariants.

When we use the `[]` operator to access the `apvector`, it checks to make sure the index is greater than or equal to zero and less than the length of the `apvector`. To access the elements of a set, though, we need to check a stronger condition. The index has to be less than the number of elements, which might be smaller than the length of the `apvector`.

```
apstring Set::getElement (int i) const
```

```
{
  if (i < numElements) {
    return elements[i];
  } else {
    cout << "Set index out of range." << endl;
    exit (1);
  }
}
```

If `getElement` gets an index that is out of range, it prints an error message (not the most useful message, I admit), and exits.

The interesting functions are `find` and `add`. By now, the pattern for traversing and searching should be old hat:

```
int Set::find (const apstring& s) const
{
  for (int i=0; i<numElements; i++) {
    if (elements[i] == s) return i;
  }
  return -1;
}
```

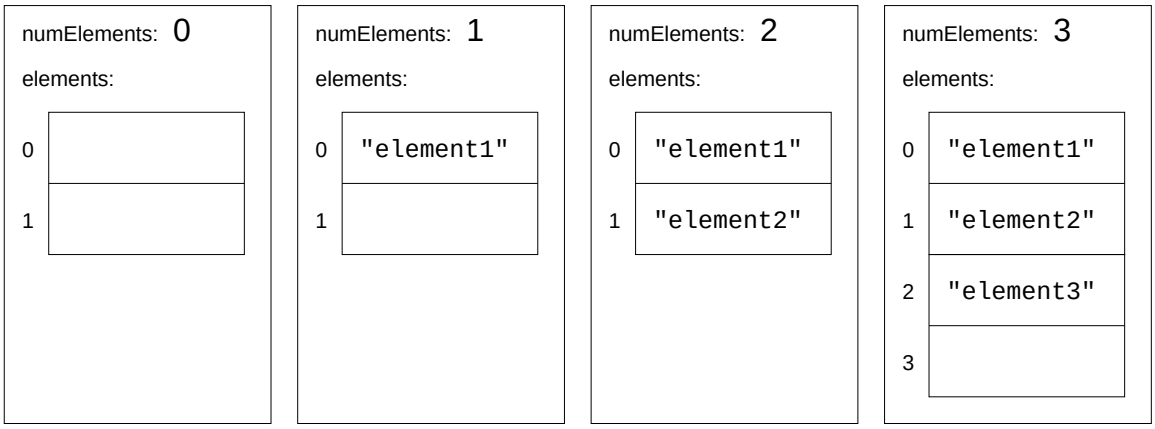
So that leaves us with `add`. Often the return type for something like `add` would be void, but in this case it might be useful to make it return the index of the element.

```
int Set::add (const apstring& s)
{
  // if the element is already in the set, return its index
  int index = find (s);
  if (index != -1) return index;

  // if the apvector is full, double its size
  if (numElements == elements.length()) {
    elements.resize (elements.length() * 2);
  }

  // add the new elements and return its index
  index = numElements;
  elements[index] = s;
  numElements++;
  return index;
}
```

The tricky thing here is that `numElements` is used in two ways. It is the number of elements in the set, of course, but it is also the index of the next element to be added.

It takes a minute to convince yourself that that works, but consider this: when the number of elements is zero, the index of the next element is 0. When the number of elements is equal to the length of the `apvector`, that means that the vector is full, and we have to allocate more space (using `resize`) before we can add the new element.

Here is a state diagram showing a `Set` object that initially contains space for 2 elements.



Now we can use the `Set` class to keep track of the cities we find in the file. In `main` we create the `Set` with an initial size of 2:

```
Set cities (2);
```

Then in `processLine` we add both cities to the `Set` and store the index that gets returned.

```
int index1 = cities.add (city1);
int index2 = cities.add (city2);
```

I modified `processLine` to take the `cities` object as a second parameter.

15.7 apmatrix

An `apmatrix` is similar to an `apvector` except it is two-dimensional. Instead of a length, it has two dimensions, called `numrows` and `numcols`, for "number of rows" and "number of columns."

Each element in the matrix is indentified by two indices; one specifies the row number, the other the column number.

To create a matrix, there are four constructors:

```
apmatrix<char> m1;
apmatrix<int> m2 (3, 4);
apmatrix<double> m3 (rows, cols, 0.0);
apmatrix<double> m4 (m3);
```

The first is a do-nothing constructor that makes a matrix with both dimensions 0. The second takes two integers, which are the initial number of rows and columns, in that order. The third is the same as the second, except that it takes an additional parameter that is used to initialized the elements of the matrix. The fourth is a copy constructor that takes another `apmatrix` as a parameter.

Just as with `apvectors`, we can make `apmatrixes` with any type of elements (including `apvectors`, and even `apmatrixes`).

To access the elements of a matrix, we use the `[]` operator to specify the row and column:

```
m2[0][0] = 1;
m3[1][2] = 10.0 * m2[0][0];
```

If we try to access an element that is out of range, the program prints an error message and quits.

The `numrows` and `numcols` functions get the number of rows and columns. Remember that the row indices run from 0 to `numrows() -1` and the column indices run from 0 to `numcols() -1`.

The usual way to traverse a matrix is with a nested loop. This loop sets each element of the matrix to the sum of its two indices:

```
for (int row=0; row < m2.numrows(); row++) {
  for (int col=0; col < m2.numcols(); col++) {
    m2[row][col] = row + col;
  }
}
```

This loop prints each row of the matrix with tabs between the elements and newlines between the rows:

```
for (int row=0; row < m2.numrows(); row++) {
  for (int col=0; col < m2.numcols(); col++) {
    cout << m2[row][col] << "\t";
  }
  cout << endl;
}
```

15.8 A distance matrix

Finally, we are ready to put the data from the file into a matrix. Specifically, the matrix will have one row and one column for each city.

We'll create the matrix in `main`, with plenty of space to spare:

```
apmatrix<int> distances (50, 50, 0);
```

Inside `processLine`, we add new information to the matrix by getting the indices of the two cities from the `Set` and using them as matrix indices:

```
int dist = convertToInt (distString);
int index1 = cities.add (city1);
int index2 = cities.add (city2);

distances[index1][index2] = distance;
distances[index2][index1] = distance;
```

Finally, in `main` we can print the information in a human-readable form:

```
for (int i=0; i<cities.getNumElements(); i++) {
  cout << cities.getElement(i) << "\t";

  for (int j=0; j<=i; j++) {
    cout << distances[i][j] << "\t";
  }
  cout << endl;
}

cout << "\t";
for (int i=0; i<cities.getNumElements(); i++) {
  cout << cities.getElement(i) << "\t";
}
cout << endl;
```

This code produces the output shown at the beginning of the chapter. The original data is available from this book's web page.

15.9 A proper distance matrix

Although this code works, it is not as well organized as it should be. Now that we have written a prototype, we are in a good position to evaluate the design and improve it.

What are some of the problems with the existing code?

1. We did not know ahead of time how big to make the distance matrix, so we chose an arbitrary large number (50) and made it a fixed size. It would be better to allow the distance matrix to expand in the same way a `Set` does. The `apmatrix` class has a function called `resize` that makes this possible.

2. The data in the distance matrix is not well-encapsulated. We have to pass the set of city names and the matrix itself as arguments to `processLine`, which is awkward. Also, use of the distance matrix is error prone because we have not provided accessor functions that perform error-checking. It

might be a good idea to take the `Set` of city names and the `apmatrix` of distances, and combine them into a single object called a `DistMatrix`.

Here is a draft of what the header for a `DistMatrix` might look like:

```
class DistMatrix {
private:
  Set cities;
  apmatrix<int> distances;

public:
  DistMatrix (int rows);

  void add (const apstring& city1, const apstring& city2, int dist);
  int distance (int i, int j) const;
  int distance (const apstring& city1, const apstring& city2) const;
  apstring cityName (int i) const;
  int numCities () const;
  void print ();
};
```

Using this interface simplifies `main`:

```
#include <iostream>
#include <fstream>
using namespace std;


int main ()
{
  apstring line;
  ifstream infile ("distances");
  DistMatrix distances (2);

  while (true) {
    getline (infile, line);
    if (infile.eof()) break;
    processLine (line, distances);
  }

  distances.print ();
  return 0;
}
```

It also simplifies `processLine`:

```
void processLine (const apstring& line, DistMatrix& distances)
```

```
{
  char quote = '\"';
  apvector<int> quoteIndex (4);
  quoteIndex[0] = line.find (quote);
  for (int i=1; i<4; i++) {
    quoteIndex[i] = find (line, quote, quoteIndex[i-1]+1);
  }

  // break the line up into substrings
  int len1 = quoteIndex[1] - quoteIndex[0] - 1;
  apstring city1 = line.substr (quoteIndex[0]+1, len1);
  int len2 = quoteIndex[3] - quoteIndex[2] - 1;
  apstring city2 = line.substr (quoteIndex[2]+1, len2);
  int len3 = line.length() - quoteIndex[2] - 1;
  apstring distString = line.substr (quoteIndex[3]+1, len3);
  int distance = convertToInt (distString);

  // add the new datum to the distances matrix
  distances.add (city1, city2, distance);
}
```

I will leave it as an exercise to you to implement the member functions of `DistMatrix`.

15.10 Glossary

ordered set: A data structure in which every element appears only once and every element has an index that identifies it.

stream: A data structure that represents a "flow" or sequence of data items from one place to another. In C++ streams are used for input and output.

accumulator: A variable used inside a loop to accumulate a result, often by getting something added or concatenated during each iteration.

Atriðisorðaskrá