FYSY160

C++ alkeet ja numeerisia menetelmiä

Vesa Apaja

5. joulukuuta 2012

Synopsis

- history of C++
- C++ web pages
- short introduction to C++
- how to use files
- basic ideas about classes
- what's a template?
- C++ template libraries
- C++ reference variables - why use references?
- **STL** containers, iterators, algorithms, for_each loop
- complex numbers
- function and operator overloading
- **Boost library**
- readable output of numeric data
- some linear algebra
- calling C or fortran routines from C++
- **GSL - Gnu Scientific Library** statistics, FFT, diff. equations, interpolation, Monte Carlo integration
- **Armadillo - matrix manipulations in Matlab style**
- new features: Lambda-functions/expressions
- suggestions: how to visualize my data

General

- 10 Lectures - in finnish from now on
  - lecturer Vesa Apaja (YN212, Nanoscience Center 2nd floor)
- 6 Demos, 2 groups, starting next week
  - held in Physics Dep. Computer class FL349
  - Markku Hyrkäs (YN252, Nanoscience Center 2nd floor)
- warm-up demos 1,2, and 3 solved during exercise
- Requirements: at least half of the exercises of the demos 4,5 and 6, plus one programming task - no exam.
  NEW: If you attend 60 percent of the lectures (that is, 6) you can drop one demo.
- You may use
  - a machine of your choice; you need a C++ compiler, GSL, armadillo and Boost libraries.
  - the Physics Department server `calc.phys.jyu.fi`, activate Unix usage in the web page salasana.jyu.fi and you can log in using your JYUNET login name and password.


FYS1
FL349
YA
YFL
Markku
YN252
4
NSC
Vesa
YN212
upstairs
YK
Syvyntie

JYUNET user id and password
from linux/Mac:

```
  ssh calc.phys.jyu.fi -l userid
edit the programs with, for example, emacs
  emacs koodi.cpp
compile and run (if no external libraries are needed)
  g++ code.cpp
  a.out
```

from Windows: (in the computer class)

```
- start Xming (or some other X-window support program for Windows)
- start the putty client to make a ssh connection
     in putty, set X11-forwarding on (allows opening of new windows)
     login to calc.phys.jyu.fi and continue working there
```

C++:n historiaa

- 70- ja 80-luku: fortran77:n käyttäjäkunta alkoi kyllästyä taulukoiden koon kiinteään määrittelyyn: `real taulukko(100)` ja kun tarvitaankin 101 alkiota on ohjelmaa muutettava :(
  - Pascal-kieli saa vauhtia Borlandin nopeasta kääntäjästä
  - Dennis Ritchie luo C-kielen dynaamisine tilanvarauksineen

Bjarne Stroustrup (Texas A&M)

- alkaen 1979 : Bjarne Stroustrup luo C++-kielen eräänlaisena "parempana C:nä" (nimi "C++" on vuodelta 1983)
- C++:lle ominaista:
  - muuttujien tiukka tyypitys vähentää ohjelmointivirheitä
  - olio-ohjelmointi (mahdollista, ei välttämätöntä)

  kootaan yhteenkuuluvaa tietoa "olioon", jota voi käsitellä kokonaisuutena
- 90-luvun alku : kääntäjävalmistajat tekevät omia ratkaisujaan kielen yksityiskohdista, koska standardia ei ole
- 1998 : C++ standardi valmistuu

  -laaja standardikirjasto
- 2011 : C++11 standardi (tunnettiin työnimellä C++0x)
  - helpompia tapoja tehdä tavallisia asioita

Tästä dokumentista

Koetan välttää osoittimien käyttöä ja suosin viittauksia. C-kieli ei tunne viittauksia, joten on pakko hiukan opetella osoittimien perusteita. Tavoitteena on opettaa kirjastojen käyttöä - välillä jopa aivan pikkujutuissa tehdäkseni koodista monikäyttöistä, turvallista ja siirrettävää. Vastuu jää kirjaston tekijälle...

Luokkien kirjoittamisen taidetta en käy läpi yksityiskohtaisesti vaan käytän enimmäkseen muiden kirjoittamia luokkia. Tarkoitus on lopulta laskea, ei koodata.

Asetan kaksi tavoitetta. Tämän esityksen perusteella :

(i) osaat kirjoittaa C++-ohjelman, joka ratkaisee numeerisen ongelman haluamaasi kirjastoa käyttäen.

(ii) osaat lukea C++ -ohjelmia ja saat suurimmaksi osaksi myös selvää niiden toiminnasta - ehkä jopa C-ohjelmat avautuvat.

C++ netissä

Muutamia ohjelmointisivuja:

```
cpp.mureakuha.com/cppohje/
www.ohjelmointiputka.net/oppaat.php
www.cplusplus.com
www.oonumerics.org/oon/    objektiorientoitunutta numeriikkaa
www.pragsoft.com/books/CppEssentials.pdf     (kirja, 2005)
www.greenteapress.com/thinkcpp/        (kirja,  1999)
```

Kieli kehittyy, älkää lukeko vanhoja kirjoja liian tarkkaan. Mitään varsinaista "hyvää" ulkoasua C++-ohjelmille ei kannata tyrkyttää, pääasia että **olet johdonmukainen ja pysyt valitsemassasi tyylissä**. Milloin käytät isoja alkukirjaimia tai et ja milloin alaviivaa on sinun päätöksesi. Mitä kannattaa panna otsikkotiedostoihin (*header files*) kannattaa harkita ja siitä kerron oman mielipiteeni. Jos olet kiinnostunut tyylikeskustelusta, suosittelen sivua

```
http://www.research.att.com/~bs/JSF-AV-rules.pdf
```

Muista kuitenkin, että numeriikan ohjelmointi on kaavojen ja algoritmien kääntämistä ohjelmiksi: numerot ovat pääosassa, eivät tekstit ja käyttäjävasteet. Pelin pelaaja istuu näppäimistö sylissä, numeerisen ohjelman käyttäjä istuu kahvilla.

(Todella) Lyhyt johdatus C++:n

Jaan erikseen listauksen kielen peruselementeistä, mutta tarkoitus on opetella niitä sitä mukaa kuin tarvetta uudelle piirteelle ilmaantuu.

Esim. 1: (ex1.cpp) Pääohjelma

```
#include <iostream>
using namespace std ;
int main()
{                           // ALL BEGINS
  int i,j;                  // integers i and j
  i=5;                      // set values to i and j
  j=20;
  int k=i+j;                // define k on the fly
  cout<<"k="<<k<<endl;      // output to screen
  return 0;                 // all fine
}                           // ALL ENDS
```

Selityksiä:
`main()` tarkoittaa, että pääohjelma main on funktio
`int main()` ja että se palauttaa kokonaisluvun (eli ulos tulee kokonaisluku)
`return 0;` kertoo, että funktio `main()` antaa ulos luvun 0.
Minne ”ulos”? Hmm, tietokoneen käyttöjärjestelmälle.

Mitä tarkoittaa #include < iostream >?

Lause #include< iostream > kertoo kääntäjälle, että ohjelmassa on jotain mikä on määritelty otsikossa (”headerissä”) iostream, joka on osa C++ standardikirjastoa. [1]. Omalla linux-koneellani on g++ kääntäjä versio 4.5.1, ja headerit ovat hakemistossa
/usr/include/c++/4.5.1.

Kaksi vaihtoehtoista syntaksia:

```
#include <iostream> // standardikirjaston osa
#include "myheader.h" // itse tehty
```

Erona näillä kahdella on hakupolku, eli mistä kaikkialta koneelta k.o. kirjastoa haetaan. Periaatteessa < ... > kun kyseessä on C++:n oma kirjasto, ”...” kun kyseessä on oma kirjasto.

Näillä headereillä pääsee alkuun:

- iostream: tulostusta
  cout, cin, cerr, clog - tässä c viittaa sanaan ”console”, joka on käytännössä ”ruutu”.
- iomanip: tulostuksen muotoilua
- cmath (C++) tai math.h (C tai C++) [2] : matem. funktiot
  sin, cos, log, sqrt, pow ...
  pow on hiukan omalaatuinen tapa kirjoittaa potenssi,
  x^2 on pow(x,2)
- complex: kompleksiluvut
- fstream: tiedostojen käsittelyä
  ofstream, ifstream, ...
- string : merkkijonot

[1] Hyvä paikka vilkaista mitä headereitä on olemassa ja mitä niissä on:
www.cplusplus.com/reference

[2] Jossain vaiheessa math.h voi poistua C++:sta.

Funktion voi kirjoittaa joko tyylillä

```
double f(double x){
    ...
}
```

tai

```
double f(double x)
{
    ...
}
```

tai jos se on lyhyt,

```
double f(double x) {...}
```

Esim. 2: (ex2.cpp) Funktio ennen pääohjelmaa

```
#include <iostream>
#include <cmath>
using namespace std ;
double f(double x){
  return sin(x);
}
int main()
{
  double x,y;
  x=2.333;
  y=f(x);
  cout<<"f("<<x<<")="<<y<<endl;
  return 0;
}
```

`f(2.333)=0.723316` Tässä funktio `f(x)` on määritelty **ennen pääohjelmaa**, joten sitä voi jo käyttää siellä.

Esim. 3: (ex3.cpp) Funktio on pääohjelman jälkeen

```
#include <iostream>
#include <cmath>
using namespace std ;
double f(double x);
int main()
{
  double x,y;
  x=2.333;
  y=f(x);
  cout<<"f("<<x<<")="<<y<<endl;
  return 0;
}
double f(double x)
{
  return sin(x);
}
```

Nyt funktio on **pääohjelman jälkeen**, kääntäjä haluaa kuitenkin tiedon sen kutsusta ennen kuin sitä voi käyttää pääohjelmassa. Rivi

```
double f(double x);
```

on **prototyyppi** eli **funktion esittely**; se tarvitaan myös jos funktio on **eri tiedostossa** kuin pääohjelma. Ilman sitä saadaan virhe `ex3.cpp:9: error: 'f' was not declared in this scope`

Mikä on ”scope”eli näkyvyysalue?

Näkyvyysalue kertoo missä koodin osissa otus on määritelty eli kuka saa tietää otuksen olemassaolosta[3]. Kaksi perusideaa, globaalit suureet jotka ”näkyvät”kaikkialle ja paikalliset suureet jotka ”näkyvät”vain oman näkyvyysalueensa sisällä, rajana aaltosulut.

```
double z; // I'm visible to anyone-mess me up at will
int main()
{
 double x; // I'm visible between {...}
 ...
}
double function f(double k)
{
        double x; // I'm not the same x as in main()!
        z=10;     // but I'm the one and only z
{
```

Huom: paikallinen silmukkamuuttuja näkyy vain silmukassa:

```
...
unsigned j = 5;
for(unsigned i=0;i<10;++i){if(i==j) break;}
cout<<"i=j when i="<<i<<endl;    // WON'T WORK
```

Kun ohjelma poistuu silmukasta, muuttuja `i` lakkaa olemasta. Siksi viimeinen rivi ei ole sallittu.

[3]Ei mikään uusi juttu, keksittiin jo 50-luvulla.

Yksinkertainen tiedostojen käyttö

Idea: luodaan tietovuo (*stream*), joko kirjoittamista (`ofstream`) tai lukemista varten (`ifstream`)

`o` on out, `i` on in ja `f` on file.

Streamiin tungetaan tavaraa <<-operaattorilla tai sieltä luetaan >>:lla.
Aiemmin käytetyt konsolille tulostava `cout` ja näppäimistöltä lukeva `cin` toimivat samalla periaatteella.

Esim. 4: (fileio.cpp) Kirjoittaminen tiedostoon

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  ofstream myfile("output");
  myfile << "Text to file output"<<endl;
  myfile.close();
  return 0;
}
```

Tässä oli streamin esittely ja avaus on tehty yhdellä `ofstream`-rivillä

```
ofstream myfile("output");
```

Tiedostoon `output` tallentuu teksti ”Text to file output”ja

```
myfile.close();
```

sulkee tiedoston.Tämä käyttää **ofstream**-luokan olion **myfile** menetelmää **close**. Saattaa auttaa, jos luet mielessäsi pisteen suomen genetiivinä, esim.

```
myfile.close(); // myfile'n close
```

Anglismeja tulvii, mutta minusta on kiusallista puhua kissoista kun esimerkeissä lukee cat. C++-kieli on englantia ja avainsanat ymmärtää parhaiten ajattelemalla englanniksi.

Luokat

Oletetaan että haluat ohjelmoida "matriisin". Siinä pitäisi olla reaalilukuja indeksoituna kuten `A(i,j)`. Tarvitset **luokan** (*class*), joko kirjaston jossa on valmis matriisi-luokka tai kirjoitat sellaisen itse.

Luokka on looginen yhdistelmä tietoja ja menetelmiä (siis funktioita) niiden käsittelyyn. Luokka on vasta abstrakti tyyppi, samaan tapaan kuin `int` kertoo millainen kokonaisluku on, mutta hiukan monipuolisempi. Koska luokalla on ominaisuuksia, niitä voidaan myös **periä**, eli jokin luokka voi periä toisen ominaisuudet ja menetelmät.

Hyvä C++-ohjelmoija laatii hyviä, järkeviä luokkia.

Tärkeimpiä luokkien tarjoamia etuja on ns. *data encapsulation*, eli lyhyesti dataa voidaan suojata tahattomilta muutoksilta tai hakkeroinnilta tekemällä siitä yksityistä (`private`) ja antamalla vain tarkkaan laadituille menetelmille oikeus koskea siihen. Etuna on myös se, että koodia parannellessa vain luokan menetelmiä on käytetty koodissa ja riittää kun parannellaan niitä.

Luokka tehdään näin:

```
class LuokanNimi{
   // tähän private eli yksityisasiat
   // esim. numerotietoa
public:
   // tähän public eli julkiset asiat ja menetelmät
   // kuten funktioita, joilla yksityisasioita voi muutella ja tutkia
};
```

Esim. 5: (lintu.cpp) Lintu-luokka

```
#include <iostream>
using namespace std;
class Lintu
{
  string vari;
public:
  string katsovari() const {return vari;};
  Lintu(string color) {vari=color;} // muodostin
  ~Lintu(){cout<<"Lintu pois\n";}   // hajotin
};

int main()
{
  Lintu harakka("mustavalkoinen");
  Lintu naakka("musta");
  cout<<"harakka on variltaan ";
  cout<<harakka.katsovari()<<endl;
  cout<<"naakka on variltaan ";
  cout<<naakka.katsovari()<<endl;
  Lintu mustarastas(naakka);
  cout<<"mustarastas on variltaan ";
  cout<<mustarastas.katsovari()<<endl;
}
```

`vari` on luokan `Lintu` ainoa ominaisuus.
`katsovari()` on menetelmä (metodi), jolla selvitetään linnun väri.

`Lintu`-luokan sisällä on funktio, jolla on sama nimi kuin luokalla:

```
Lintu()
```

on muodostin, jolla jokin lintu voidaan "muodostaa"ja samalla antaa sille väri (eli `vari`). Tätä muodostinta käytetään aina (ja vain) kun luotavan linnun `vari` annetaan suluissa.

Olio on luokan "ilmentymä", eli sellainen jolla on luokan määrittelemät ominaisuudet:

```
Lintu naakka("musta"); // lintu on luokka, naakka on olio
```

Tämän voi lukea hauskasti "naakka on yks Lintu ja se on musta". Kun tälle riville tullaan, kääntäjä hakee `Lintu`-luokasta muodostajaa, joka vastaa tarvetta, eli funktiota joka näyttää tältä:

```
 Lintu(string)
```

Sellainen on olemassa, joten mustan naakka-Linnun luominen onnistuu. Sen sijaan yritys tehdä väritön Lintu

```
Lintu haikara;
```

ei onnistu, koska luokassa `Lintu` ei ole muodostajaa joka olisi pelkästään

```
Lintu()
```

Tämä on muuten ns. *oletusmuodostin* (funktio ilman argumentteja) - se vain varaa oliolle tilaa muistista, eikä tee mitään muuta. Jos mitään muuta muodostinta ei ole määritelty, niin kääntäjä tekee itse tällaisen oletusmuodostimen - jokin muodostinhan luokalla on oltava että sitä voisi mitenkään käyttää. C++-kielessä on määritelty, että jos *sinä teet* luokalle muodostimen, niin kääntäjän tekemä oletusmuodostin katoaa!

Lintu-luokan tapauksessa oletusmuodostimen suokin kadota: jos loisi oletus-Linnun, ei sille enää voisi asettaa väriä. Muuttuja `vari` on yksityinen: tietoon `haikara.vari` (eli "haikaran väri") ei voi koskea ilman menetelmää.

Linnuilta puuttuu menetelmä, jolla ne voisi värittää luomisen jälkeen.

Linnun voi luoda myös kopioimalla, kuten rivillä

```
Lintu mustarastas(naakka);
```

Nyt mustarastaalle kopioituu naakan väri.

Jos kiinnostaa (tai jos Tuomas Veturi kiinnostaa) katso lisää luokan menetelmistä lisälapulta **C++-rautaisannos**.

Esimerkki käytti kömpelöä muodostinta, parempi olisi käyttää *alustuslistaa* (*initialization list*).

Mikä on Template?

Template on **malli** (tai mallipohja), yleiskäyttöinen kertomus siitä mitä malliin syötetylle tiedolle tehdään. Template on yksi C++:n vahvimmista piirteistä, ehkä jopa parasta mitä kielellä on tarjota. Eikä ohjelmoijan tarvitse edes itse osata kirjoittaa niitä. Useimmat C++-kirjastot on toteutettu template-tekniikalla, jolloin saavutetaan **erinomainen suorituskyky ja yleisyys**.

template on abstrakti ohjelma, jonka toiminta riippuu siitä, millaista tavaraa siihen syötetään.

Esim. `swap` on usein malli, joka vaihtaa kaksi "otusta".

[3] Yksi kummallisimmista piirteistä on ns. **Template metaprogramming** - päätelmiä ja laskuja voi tehdä jo kääntäjällä!

C++ Template-kirjastoja

Standard Template Library
Boost C++ Libraries
STLSoft C++ Libraries
Electronic Arts Standard Template Library
Adobe Source Libraries
Intel Threading Building Blocks
C++ Templated Image Processing Library
Database Template Library
Windows Template Library
Armadillo C++ Library (linear algebra)
Eigen Library (linear algebra)
Matrix Template Library
Loki (C++)
Native Template Library
PoCo C++ Libraries
CGAL
Blitz++
Fastflow
View Template Library
STXXL : Standard Template Library for Extra Large Data Sets

C++ Standardikirjastojen hierarkia

C++ Standard Library:

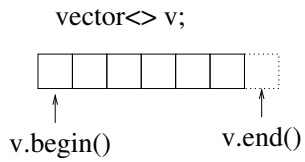
- C++ Standard Template Library
- C Standard Library

Siten C++ ohjelma voi sisältää C-kielen piirteitä. C++ koodin voi jopa kirjoittaa niin, ettei siinä ole pätkääkään C++:aa (ehkä vain kääntäjä on C++:lle, kuten g++)

STL: Standard Template Library

Tärkein template-kirjasto, tulee C++:n mukana

- kontit (containers) ovat olioiden kokoelmia
  `vector, stack, deque, list; map`
- iteraattorit (iterators) helpottavat kontin elementtien läpikäyntiä
  `begin(), end()` [4]



- algoritmit (algorithms) prosessoivat elementtien kokoelmia
  `sort, find, min_element, max_element, reverse`

[4] Osoittavat kontin alkuun ja *lopun ohi* ← helppokäyttöistä silmukoissa!

Esim. 6: (stl_swap.cpp) Vaihdetaan kaksi lukua käyttäen STL:n `swap` menetelmää

```
#include <iostream>
int main()
{
  using namespace std;
  double a = 5;
  double b = 10;
  cout <<"before "<< a<<" "<<b<<endl;
  swap(a,b);
  cout <<"after  "<< a<<" "<<b<<endl;
  return 0;
}
```

```
before 5 10
after  10 5
```

C++ viitemuuttujat (*reference variables*)

Tietoa voi lähettää funktioihin kolmella tavalla:

- Arvon (*value*) lähettäminen funktioon

```
void dothis(int);  // esitellään funktio
...
c=15;
dothis(c); // käsittelee numeroa 15
```

- (C++) viitteen (*reference*) lähettäminen funktioon

```
void dothat(int &);  // esitellään funktio
...
c=15;
dothat(c); // käsittelee muuttujaa c
```

- Osoittimen (*pointer*) lähettäminen funktioon

```
void doodd(int *);  // esitellään funktio
...
c=15;
doodd(&c);  // käsittelee muuttujan c osoitetta
```

Huomaatko: funktiokutsun perusteella ei voi sanoa meneekö sinne arvo vai viite! Funktion esittely paljastaa kumpi.

Miksi viite on turvallisempi kuin osoitin?

Kuvittele hetki, että tietokoneen muisti on laatikosto.
Osoitin `int *c` tarkoittaa "kokonaisluku c on ylimmässä laatikossa"tai itse asiassa vain "ylin laatikko, ota sieltä"!.

- Ylimmässä laatikossa voi olla väärä luku tai jokin muu kuin kokonaisluku
- ylin laatikko voi olla tyhjä
- koko laatikkoa ei ole olemassakaan

Ohjelmoija saa tehdä kaiken tämän ilman että kääntäjä huomaa mitään outoa. Tuloksena on ajon aikana *Segmentation fault* tai pahempaa.

C++ viite muuttujaan `c`, `int& c`, on kuin naru, jonka päässä on kokonaisluku `c`. Erehtyä ei voi, narun päässä on aina `c`, koska narua ei voi irrottaa ja sitoa kiinni mihinkään muuhun. C++ viitettä ei voi irrottaa muuttujastaan.

Hiukan huteria käytännön esimerkkejä:

Esimerkki arvon lähettämisestä (C tai C++):
Kirjoita lapulle numerot 5 ja 10, näytä kaverillesi lappua ja sano että hänen pitää kirjoittaa ne *omalle* lapulleen päinvastaisessa järjestyksessä.
Mutta: sinun lapullasi on edelleen 5 ja 10, ei 10 ja 5 .

Esimerkki viitteen lähettämisestä (vain C++):
Kirjoita lapulle numerot 5 ja 10, ojenna lappu kaverillesi mutta älä päästä siitä irti. Sano että hänen pitää vaihtaa lapulla numerot päinvastaisessa järjestykseen ja ota lappu takaisin.

Esimerkki osoittimen lähettämisestä (C tai C++):
Kirjoita lapulle numerot 5 ja 10, kerro kaverillesi että lappu on laatikossa ja sano että hänen pitää vaihtaa lapulla numerot päinvastaisessa järjestykseen ja panna se takaisin laatikkoon.

Pari huomiota:

1) Jos kavereita on useita ja/tai lapulla on 10000 lukua, niiden kopiointi toiselle lapulle kestää kauan ja voi kuluttaa paljon paperia.

Suosi viitteen lähettämistä, se on nopeaa ja taloudellista.

2) Jotta `swap` toimisi ei sinne voi lähettää käsittelyyn vain lukujen 5 ja 10 arvoja eli niiden *kopioita*.
Kaksi toimivaa tapaa:

- C++-tyyli: `swap` käsittelee *viitteitä*; funktion kutsu: `swap(a,b);`

```
void swap(int& a,int& b){
  int c;
  c=a; a=b; b=c;
}
```

- C-tyyli: `swap` käsittelee *osoittimia* funktion kutsu: `swap(&a,&b);`

```
void swap(int* a,int* b){
  int c;
  c=*a; *a=*b; *b=c;
}
```

Jos lähetät funktion argumentin arvona, suojaat sitä muuttamiselta koska siitä tehdään kopio. Tämä ei silti ole paras tapa, ei varsinkaan jos suojeltava data on iso taulukko, jonka pelkkä kopioiminen on kamalan iso työ ja muisti loppuu. Lähetä silloin datasi viitteenä, mutta anna sille `const`-määre: data on suojassa.

Edellä annettu viittauksella toteutettu `swap` on jo aika hyvä, mutta entä jos haluamme vaihtaa kaksi reaalilukua tai kaksi muun tyyppistä muuttujaa keskenään? Tee malli (*template*). Itse asiassa se on jo tehty, STL:n `swap(a,b)` näyttää periaatteessa tältä (samalla ensimmäinen esimerkki template-koodista):

Esim. 7: (template.cpp) STL:n swap-malli

```
template <class T> void swap ( T& a, T& b )
{
  T c(a); a=b; b=c;
}
```

Tässä on paljon uutta, näytän tätä vain että tunnistatte template-koodin kun näette sellaisen.
`template` tämä on malli
`class T` tarkoittaa, että "T"on luokka (näihin tullaan kohta),
`void swap` tarkoittaa että swap ei palauta mitään arvoa nimessään.
`T c(a)` tee luokan T oliosta a kopio c, joka on myös luokan T olio

Tämä template toimii vain jos pari asiaa on kunnossa:
1) `T c(a)` luo c:n käyttäen mallinaan a:ta
   tarvitaan *kopionmuodostin* (copy constructor)
   tämän kääntäjä tekee itse, jos sinä et tee.
2) `a=b` asettaa a:n arvoksi b:n arvon
   tarvitaan määritelmä `=`-operaatiolle (assignment)

Siksi ohjekirja sanoo: *The type must be copy constructible and support assignment operations.*

STL:n vector kontti (säiliö, *vector container*)

Paras opetella käyttämään STL:n vector kontteja.
Sinun ei enää koskaan tarvitse ohjelmoida suoraan dynaamisesti varattuja taulukoita.

Esim. 8: (stl_vector.cpp) Tehdään tyhjä kontti ja työnnetään siihen muutamia elementtejä.

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main()
{
  vector<double> x;
  for(unsigned i=0;i<6;i++){
    x.push_back(pow(i,2)); // push i^2 to vector x
    cout<<"i="<<i<<" x="<<x[i];
    cout<<" size of x="<<x.size()<<endl;
  }
  cout<<"final     size ="<<x.size()<<"\n";
  cout<<"final capacity ="<<x.capacity()<<"\n";

  return 0;
}
```

```
final     size =6              taulukon koko kasvoi automaattisesti
final capacity =8              tilaa varattiin hiukan yli tarpeen
```

STL:n vector kontti osa 2

Otimme käyttöön kontin `vector`, joten voisimme samantien käyttää myös `iteraattoreita`.

Esim. 9: Tulostetaan vector-kontin elementit käyttäen iteraattoria

```
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main()
{
  vector<double> x;
  vector<double>:: iterator it;  // iterator
  // push i^2
  for(unsigned i=0;i<6;i++) x.push_back(pow(i,2));
  // use iterator for output
  for(it=x.begin() ; it!=x.end() ; ++it){
    cout<<*it<<' ' ;
  }
  cout<<endl;
  return 0;
}
```

```
0 1 4 9 16 25
```

Muita vektoreiden operaatioita:

```
vector<double> z(y); // create y as a copy of z
vector<double> big;  // empty vector big
big.reserve(100);    // reserve space for 100 entries
big.capacity();      // tells how much space is reserved
big.resize(20);      // resize big to size 20
big.size();          // tells the size of big
```

Joskus kannattaa varata tilaa etukäteen "varastoon", jotta myöhempi vektorin pituuden kasvattaminen ei tuhlaa aikaa tilanvarauksiin.

Kätevä tapa täyttää vektori jollakin numeroarvolla on std::fill,

```
  std::vector<double> y(100);
  std::fill(y.begin(), y.end(), 1.0); // fill y with 1's
```

Nyt kontin y kaikki elementit sisältävät luvun 1.0.
C++11 antaa uuden tavan, käyttäen uutta `range-based for`- silmukkaa: (käännös: `g++ -std=c++0x ...`)

```
  std::vector<double> y(100);
  for( auto & elem:y) {elem=1.0}; // notice the & : use reference to elements!
```

Tässä on kaksi uutuutta: `auto` kertoo "käytä y-kontin elementtien tyyppiä"s ja silmukka käy kaikki `yss`:n alkiot läpi. Ole varovainen: jos jätät `&s`-merkin pois, niin arvo ei muutu!

```
  for( auto elem:y) {elem=1.0};  // this does nothing at all!
  for( auto elem:y) {cout<< elem<<" ";}; // this works (but doesn't try to change y)
```

Suosi C++11 standardin `auto` *tyyppiä; anna kääntäjän päätellä tyyppi.*

[4] Algebraa vektorin pituudella? `x.resize(0)` joten `x.size()` on 0, joten `x.size()-1` on 18446744073709551615, eikä -1...hmm, unsigned! Etumerkittömään kokonaislukuun ei voi tallentaa negatiivista lukua.

STL stream-iteraattorit

Liian hankalaa aloittelijalle; vilkaise mutta älä huolestu

Stream eli tietovuo on olio, joka edustaa I/O kanavaa. Stream-iteraattori käy läpi lukee tai kirjoittaa streamiin.

Esim. 10: (stl_streamiter1.cpp) Luetaan vektoriin merkkijonoja

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
  vector<string> s;
  // input from standard input (cin)
  copy(istream_iterator<string>(cin), //from
       istream_iterator<string>(),    //end
       back_inserter(s));             //to
  // output to standard output (cout)
  copy(s.begin(),s.end(),                    // from
       ostream_iterator<string>(cout," ")); // to
  cout<<endl;
  return 0;
}
```

[4]Muista: istream = sisään tuleva tietovuo, ostream=ulos menevä tietovuo.

STL algoritmit

Paljon hyödyllisiä menetelmiä, hakuja, lajitteluja yms. Kun tiedät mitä haluat tehdä, käy esim. sivulla

www.cplusplus.com/reference/algorithm

selaamassa olisiko sopivaa algoritmia jo STL:ssä.

Esimerkkejä:
Jos `v` ja `w` ovat vector-kontteja, ja `it` kontin iteraattori, niin

```
    it = max_element(v.begin(),v.end())
```
`it` osoittaa suurimpaan elementtiin, eli suurin elementti on `*it`.

```
    sort(v.begin(),v.end())
```
lajittelee kontin

```
    swap(v,w)
```
vaihtaa kontit `v` ja `w`.

Seuraavan sivun esimerkissä haetaan numeerisesta datasta pienin, suurin, tietty arvo ja lopuksi lajitellaan data. Funktio `vector_out()` on pikku apufunktio, joka tulostaa `double`-tyyppisen `vector`:in. Se käyttää edellisen sivun **stream-iteraattoria**. Toimintaperiaate on, että kopioidaan `vector`-olio `cout`-olioon ja väliin tulee välilyönti.

Esim. 11: stl_algo.cpp: STL algoritmit: minimi, maksimi, elementin haku

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void vector_out(vector<double> v)
{
  copy(v.begin(),v.end(), // from
       ostream_iterator<double>(cout," ")); // to
  cout<<endl;
}

int main()
{
  vector<double> v;
  vector<double>::iterator it;

  v.push_back(1.4);
  v.push_back(1.6);
  v.push_back(0.2);
  v.push_back(1.8);
  v.push_back(0.1);
  v.push_back(1.5);
  cout<<"original vector"<<endl;
  vector_out(v);

  it = min_element(v.begin(),v.end());
  cout<<"minimum element = "<<*it<<endl;
```

```
  it = max_element(v.begin(),v.end());
  cout<<"maximum element = "<<*it<<endl;

  //find element with some value
  it = find(v.begin(),v.end(),0.2);
  if(it==v.end())
    cout<<"failed to find value"<<endl;
  else {
    cout<<"found value "<<*it<<endl;
    // reverse some elements
    cout<<"reverse starting from "<<*it<<endl;
    reverse(it,v.end());
    vector_out(v);
  }
  cout<<"sorting..."<<endl;
  sort(v.begin(),v.end());
  vector_out(v);
  return 0;
}
```

Iteraattorit tekevät koodista hiukan rumaa, mutta ne voi usein haudata näkymättömiin. Varsinkin jos haluat ulos vain sen mihin iteraattori osoittaa (yllä toistui `*it`) .

Esim. 12: (stl_easyusemin.cpp) STL algoritmit: siisti minimin haku

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

double min_element(const vector<double> & v){
  // no explicit iterator! We need only *(iterator)
  return *(min_element(v.begin(),v.end()));
}
// calling routine is clean and simple:
int main()
{
  vector<double> v;
  v.push_back(1.7); v.push_back(1.3);
  v.push_back(2.8); v.push_back(4.1);
  cout<<"minimum element = "<<min_element(v)<<endl;
  return 0;
}
```

```
minimum element = 1.3
```

Varoitus: harkitse huolella haluatko tehdä STL:n algoritmeista spesialisoituja versioita. Tämäkin omatekoinen `min_element()` palauttaa numeron, ei iteraattoria kuten STL-versio.

Monet STL algoritmit ovat todella huippukäteviä ja tehokkaita. Kuten nyt tämäkin:
(tällä sivulla pelkkiä apufunktioita)

Esim. 13: (stl_algo_permutations.cpp) : STL permutaatioalgoritmi

```
// Finding permutations
// using stl algorithm next_permutation
//
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void vector_out(vector<int> v)
{
  copy(v.begin(),v.end(), // from
       ostream_iterator<int>(cout," ")); // to
  cout<<endl;
}

int factorial (const int n){
  int fact = 1;
  if (n <= 1)
    return 1;
  else
    fact  = n * factorial (n - 1); // recursion
  return fact;
}
```

```
int main () {
  const int N=5;
  vector<int> state;
  for (int i=0;i<N;++i) state.push_back(i);
  sort (state.begin(),state.end());
  int i = 1;
  do {
    cout<<"state # "<<i<<" is  ";
    vector_out(state);
    ++i;
  } while (next_permutation (state.begin(),state.end()));
  cout<<"N! = "<< factorial(N)<<endl;
  return 0;
}
```

```
state # 1 is  0 1 2 3 4
state # 2 is  0 1 2 4 3
state # 3 is  0 1 3 2 4
...
```

Mahdollisuuksia on $N!$, kertoma lasketaan tarkistuksen vuoksi rekursiivisessa funktiossa `factorial()`. Rekursiivinen funktio kutsuu itseään. Funktiokutsu

```
next_permutation (state.begin(),state.end())
```

tekee kaiken, muu on rekvisiittaa. Se hakee seuraavan permutaation kontin `state` alkoille ja palauttaa arvon `true`, kunnes se ei enää löydä uutta ja funktio saa arvon `false`. Silmukka

```
do {
 ...
 } while(next_permutation (state.begin(),state.end())
```

tekee jotain niin kauan kuin testi on `while(true)` ja loppuu kun tulee `while(false)`.

Permutaatioalgoritmia voi käyttää vaikkapa etsimään kaikki tilat joissa on tietty määrä fermioneja. Kuhunkin spintilaan voi panna 0 tai 1 fermionia. Esimerkiksi jos spintiloja on 4 ja fermioneja 2 saadaan tilat (0011),(0101),(0110),(1001),(1010),(1100) eli 6 monihiukkastilaa.

Esim. 14: (stl_algo_permutations2.cpp) : fermionitilat

```
// Finding fermion basis states
// using stl algorithm next_permutation
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
#include <map>

using namespace std;

void vector_out(vector<int> v)
{
  copy(v.begin(),v.end(), // from
       ostream_iterator<int>(cout," ")); // to
  cout<<endl;
}

int main () {
  const int N=6; // single-fermion states
  const int Nfermions=3; // number of fermions

  cout<<"All "<<Nfermions<<"-fermion states for N="<<N<<endl;
  vector<int> state;
  for (int i=0;i<N;++i) {
    if(i<Nfermions) {
      state.push_back(1);
```

```
    }
    else {
      state.push_back(0);
    }
  }
  sort (state.begin(),state.end());
  int i = 1;
  vector<int> fermistate(N);
  do {
    cout<<"state # "<<i<<" is  ";
    vector_out(state);
    ++i;
  } while (next_permutation (state.begin(),state.end()));
  cout<<"found "<<i-1<<" fermion states\n";
  return 0;
}
```

```
All 3-fermion states for N=6
state # 1 is  0 0 0 1 1 1
state # 2 is  0 0 1 0 1 1
state # 3 is  0 0 1 1 0 1
...
state # 18 is  1 1 0 0 1 0
state # 19 is  1 1 0 1 0 0
state # 20 is  1 1 1 0 0 0
found 20 fermion states
```

Elektroneilla voi spin ylös ($\uparrow$) tai alas ($\downarrow$), tilat voisivat olla koodattu spin-tilojen miehityksinä vaikkapa ylös-alas pareina :

110010 tarkoittaa $\uparrow\downarrow\ 00\ \uparrow\ 0$,

eli 1. yksihiukkastilassa on sekä spin-ylös että spin-alas elektroni, toinen on tyhjä, kolmannessa on spin-ylös elektroni.

Kaksi tapaa laskea kertoma rekursiivisesti

Liian hankalaa aloittelijalle; vilkaise mutta älä huolestu

Käytä tuloksen ja välitulosten talletukseen riittävää kokonaislukutyyppiä (`int, long int` tai jokin muu). Edellä oli perusversio:

```
int factorial ( const int n ) {
  int fact = 1 ;
  if ( n <= 1 )
    return 1 ;
  else
    fact = n*factorial ( n - 1 ) ; // recursion
  return fact ;
}
```

C++11 osaa tehdä asioita jo käännösaikana (siis hiukan "Template metaprogramming-tyylillä, mutta helpommin luettavassa muodossas). Tässä esimerkki siitä miten koodin käyttämä 13! on laskettuna jo ennen kuin ajo alkaa!

```
long int constexpr factorial (int n)
{
    return n > 0 ? n * factorial( n - 1 ) : 1; // one return statement is allowed
}

int main()
{
   constexpr long int fact13=factorial(13);
  ...
}
```

Määrittelyn `constexpr` ansiosta kääntäjä laskee kertoman jo käännösaikana ja fact13 on käännetyssä koodissa jo luku 6227020800. Paluulauseessa on hauska yhden rivin testi: "jos n on suurempi kuin nolla, palauta n*factorial(n-1), jos ei, palauta yksi". Funktio kutsuu siis itseään kunnes argumentti `n` saa arvon 1. Joissain tilanteissa `constexpr` voi tehdä ohjelmastasi hyvin paljon nopeamman.

STL algoritmit2: statistiikkaa

Kootaan dataa vector-konttiin ja lasketaan datalle keskiarvo ja keskihajonta funktiossa `get_stats()`.
Toimintaperiaate:

STL:n accumulate() algoritmi laskee elementtien summan (ellei toisin määrätä). vector-kontin menetelmä .size() kertoo elementtien lukumäärän. Keskihajontaa varten lasketaan elementtient poikkeama keskiarvosta käyttäen STL:n transform-algoritmia. bind2nd:n käyttöä ei käydä tarkemmin tällä kurssilla, todetaan vain, että sen avulla neljänteen argumenttiin saadaan oikeanlainen operaatio [5]

Rivit

```
#ifndef STL_STATISTICS_HPP
#define STL_STATISTICS_HPP
...
#endif
```

varmistavat sen, ettei tätä koodia ladata monta kertaa. Jos käytät sitä, ota ohjelman alkuun esittelyt,

```
# include "stl_statistics.hpp"
```

mutta laajassa ohjelmistossa tämä rivi voi olla jo muissakin tiedostoissa. Jotta `stl_statistics.hpp` tiedoston funktioiden esittelyjä ei ladattaisi moneen kertaan, siinä on testi "onko määritelty `STL_STATISTICS_HPP`, jollei, niin määrittele se". Kääntäjä siis muistaa onko määritelty: jos ei, lataa koodi, jos on, älä vaivaudu.

Miksi tiedosto on nimeltään **.hpp**, eikä **.cpp**? Se on tapa kertoa, että tämä on eräänlainen apupaketti jota ei erikseen käännetä, otsikko eli **header**. C-kielessä nämä ovat **.h**-loppuisia. **C++ ohjelmissa yleisin tapa on laittaa otsikkotiedostoihin vain funktioiden esittelyt**, sellaisia otsikoita eli headereitä ei esim. javassa ole olemassakaan - ne ovatkin olemassa vain kääntäjää varten.

[5] transform-algoritmiin voi syöttää vain operaation, joka tekee jotain joko yhdelle tai kahdelle elementille. Pulmana on miten syöttää myös keskiarvo: sen tekee bind2nd.

Esim. 15: (stl_statistics.hpp) STL statistiikkaa

```
#ifndef STL_STATISTICS_HPP
#define STL_STATISTICS_HPP
#include <vector>
void  get_stats(const std::vector<double> &, double & , double & );

#endif
```

Esim. 16: (stl_statistics.cpp) Varsinainen `get_stats` funktio STL:n avulla toteutettuna

```
// Computes average and standard deviation
// for data stored in std::vector
#include <vector>
#include <algorithm>
#include <numeric>
#include <functional>
#include <cmath>

void get_stats(const std::vector<double> & x, double & average, double & sigma )
{
  // average = sum_i=1^N x_i /N
  //
  //              sum_i=1^N (x_i-<x>)^2          (x-<x>).(x-<x>)
  // sigma = sqrt( -------------------- ) = sqrt(---------------)
  //                     N-1                          N-1
  using namespace std;
  int N = x.size();
  average  = accumulate(x.begin(),x.end(),0.0)/N;   // sum up values
  vector<double> xx(x); // xx = x-<x>; subtract average from all elements
  transform(x.begin(),x.end(),xx.begin(),bind2nd(minus<double>(),average));
  sigma = sqrt(inner_product(xx.begin(),xx.end(),xx.begin(),0.0)/(N-1));
}
```

esim(std11_statistics.cpp) Funktio `get_stats` toteutettuna C++11 standardin avulla

```
// C++11 version
// Computes average and standard deviation
// for data stored in std::vector
#include <vector>
#include <cmath>

void get_stats(const std::vector<double> & x, double & average, double & sigma )
{
  // average = sum_i=1^N x_i /N
  //
  //              sum_i=1^N (x_i-<x>)^2
  // sigma = sqrt( -------------------- )
  //                     N-1
  int N = x.size();
  average=0; sigma = 0;
  for (auto x_i: x) {average += x_i;}
  average /= N;
  for (auto x_i: x) {sigma += pow(x_i-average,2);}
  sigma = sqrt(sigma/(N-1));
}
```

Koodi on paljon helpommin luettavaa kuin STL:n avulla toteutettu, koodin ja kaavojen välillä on parempi vastaavuus. Lisäksi *range-based for*-silmukka on nätti ja turvallinen käyttää.

Kompleksiluvut: standardiluokka `complex`

Tavalliset operaattorit (kertominen (*), jakaminen (/), yhteenlasku (+), reaaliosan otto (real()) jne. ovat valmiina.

Esim. 17: (complex_ex.cpp) Perusoperaatioita kompleksiluvuilla

```
#include <iostream>
#include <complex>
int main()
{
  using namespace std;
  complex<double> c1,c2;
  c1 = complex<double>(1.5,2.2);
  c2 = complex<double>(1.0,3.3);
  cout<<"c1="<<c1<<endl;
  cout<<"c2="<<c2<<endl;
  // real(c2) or c2.real()
  cout<<"real(c2)="<<real(c2)<<endl;
  cout<<"imag(c2)="<<imag(c2)<<endl;
  cout<<"c1+c2="<<c1+c2<<endl;
  cout<<"c1*c2="<<c1*c2<<endl;
  cout<<"conj(c1)="<<conj(c1)<<endl;
  cout<<"c1/c2="<<c1/c2<<endl;
  return 0;
}
```

Funktioiden lisämääritys (*function overloading*)

C-kieli ei tunne funktioiden lisämääritystä. Vain matemaattiset funktiot (kuten sin, exp...) on lisämääritelty, eli voit ottaa sinin reaaliluvusta tai kompleksiluvusta samalla sin()-funktiolla. C++-kielen tekijöillä on eri käsitys siitä mikä on hyväksi.

Oli miten oli, usein on hyödyllistä käyttää **samannimistä funktiota tekemään hiukan eri toimintoa**. Pääasia että kääntäjä pystyy erottamaan mitä funktion muunnosta milloinkin halutaan käyttää; Tähän riittää erot argumenttien tyypeissä ja määrissä. Samalla voidaan tehdä

- vapaaehtoisia (*optional*) argumentteja, eli sellaisia joita ei välttämättä tarvitse antaa. Samannimisiä funktioita on siis useita:
  `f(pakollinen)` *vs.* `f(pakollinen, vapaaehtoinen)`

- argumentteja, joille on asetettu oletusarvo: jos funktion kutsussa ei aseteta arvoa, käytetään oletusta.
  `f(pakollinen)` *vs.* `f(pakollinen, oletettu)`
  Huom: muuttujaa `oletettu` todella käytetään funktiossa ja sillä on aina arvo. Edellinen funktiokutsu käyttää oletusarvoa, jälkimmäinen sitä mikä arvo argumentissa on.

[5] Katso Vesa Lappalaisen oliosanasto: users.jyu.fi/~vesal/kurssit/winohj/oliosana.htm

Esim. 18: (function_overload.cpp) Funktion argumentit a ja b ovat vapaaehtoisia

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

double integ(void) {
  cout << "no args to integ, integrating from 0 to 1"<<endl;
  return (1.0); // just test
}

double integ(double & a,double & b) {
  cout << "two args to integ, integrating from "<<a<<" to "<<b<<endl;
  return (2.0);// just test
}


int main(){
  double a=5,b=10;
  integ();
  integ(a,b);
  return 0;
}
```

```
no args to integ, integrating from 0 to 1
two args to integ, integrating from 5 to 10
```

Esim. 19: (function_overload2.cpp) Funktion toinen argumentti on vapaaehtoinen, oletusarvo 1.0

```
#include <iostream>
using namespace std;
double fun(double a, double b= 1.0); //IMPORTANT LINE
int main(){
  double a=5,b=10;
  fun(a);
  fun(a,b);
  return 0;
}
double fun(double a,double b) {
  if(b==1) {
    cout <<"a="<< a<<" default case b=1"<<endl;
  }
  else {
    cout <<"a="<< a<<" not default case, b="<<b<<endl;
  }
  return (1.0);// just test
}
```

```
a=5 default case b=1
a=5 not default case, b=10
```

Huomaa miten oletusarvo annetaan vain funktion (“ensi”)esittelyssä!

Operaattorien lisämääritys

Liian hankalaa aloittelijalle; vilkaise mutta älä huolestu

Tätä aihetta ei käsitellä tällä kurssilla tarpeeksi käyttöä varten. Alla oleva materiaali kattaa vain irrallisia perusteita - ja on pahasti keskeneräinen.

Kompleksilukujen yhteenlasku on hiukan erilainen kuin reaalilukujen, silti sama operaattori (+) tekee molemmat. Tämä on saatu aikaan lisämäärittämällä `complex`-luokan +-operaattori, eli operaattori tekee eri toimituksen riippuen siitä millaisia tietotyyppejä käsitellään.

Operaattorin lisämääritys saattaa oleellisesti parantaa koodin luettavuutta.

Esimerkiksi ilman lisämääritystä kompleksilukujen `c1` ja `c2` yhteenlasku pitäisi tehdä jollain funktiolla tyyliin

```
c3=laskeYhteen(c1,c2);
```

On paljon siistimpää lisämäärittää +-operaattori siten, että yhteenlasku on koodissa muodossa

```
c3=c1 + c2;
```

Kääntäjä toki muuntaa +-merkin funktiokutsuksi, mutta se on näkymättömissä eikä sotke ohjelman luettavuutta.

Muistisääntöjä ja rajoituksia:

- Ajattele operaattoreita funktioina, joilla on joko yksi tai kaksi argumenttia (ns. unaariset ja binääriset operaattorit). Jos tarvitset operaatiossasi kahta argumenttia sinun on lisämääritettävä jokin *olemassaoleva* binäärinen operaattori.
- Uusia operaattoreita ei saa keksiä.
  ("`miunhienooperaattori`" ei käy)
  Osan wanhoista voi lisämäärittää:

```
+  -  *  /   %   ^  &  |  ~   !
=  <  >  += -= *= /= %= ^= &=
|= << >> <<= >>= == != <= >= &&
|| ++ -- ,   -> [] () new delete
```

- Operaattorien suoritusjärjestys säilyy ($*$ ennen $+$:aa)

Matemaattisen kaavan x^y ja funktiokutsun `pow(x,y)` ulkonäkö on niin erilainen, että tulee kiusaus lisämäärittää jokin operaattori laskemaan potenssia. Vältä kiusausta ja käytä kiltisti `pow()`-funktiota.

Esim. 20: (overload1.cpp) Oma complex-luokka ja +-operaattorin lisämääritys

```
// DO NOT USE!
#include <iostream>
class Complex {
  double re, im;
public:
  Complex() { re = 0.0; im = 0.0; }
  Complex(double r, double i) { re = r; im = i; }
  double real() const {return re ;}
  double imag() const {return im ;}
  Complex operator+(const Complex z) {
    Complex ctmp;
    ctmp.re  = re+z.re;
    ctmp.im  = im+z.im;
    return ctmp;
  }
};

int main() {
  Complex a = Complex( 1.2, 3.3 );
  Complex b = Complex( 1.4, 2.4 );
  Complex c ;
  c = a + b;
  std::cout<<c.real()<<"+"<<c.imag()<<" i \n";
  return 0;
}
```

Esimerkin luokka osaa laskea kaksi kompleksilukua yhteen, muttei laskea mitä on vaikkapa `c=5.0+b ;`, virheilmoitus on

```
overload2.cpp: In function 'int main()':
overload2.cpp:22:13: error: no match for 'operator+' in '5.0e+0 + b'
```

Miksei osaa? Koska operaatio `a+b` tarkoittaa "ota luokan `a` oliosta menetelmä `+` ja kutsu sitä argumentilla `b`. Kääntäjä yrittää siis jotain tämäntapaista : `c=a.add(b)`. Yhteenlasku `c=5.0+b ;` ei toimi, koska numerolla 5.0 ei ole +-nimistä menetelmää! Jos haluat tietää, niin tämä pulma hoidellaan ns. `friend` funktiolla, joka määrittelee puuttuvan +-operaation.

STL: for_each

STL:n algoritmi `for_each` suorittaa määritellyn operaation kaikille elementeille. Argumentteina annetaan alku, loppu ja mitä tehdään.

Esim. 21: (stl_foreach.cpp) STL for_each, yksinkertainen vector-kontin tulostus

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void doubleout(double y) { cout << " " << y; }
int main () {
  vector<double> x;
  x.push_back(1.1); x.push_back(2.2);  x.push_back(3.3);
  cout << "x vector: \n";
  for_each (x.begin(), x.end(), doubleout);
  cout<<endl;
  return 0;
}
```

Tässä suoritetaan funktio `doubleout()` kaikille elementeille. Tämä funktio voi olla mielivaltaisen monimutkainen, kunhan sen argumentti on `double`, eli `jotain(double x)` .

STL: for_each yksityiskohta

(Vain lisätietoa, voit jättää lukematta)

Miten `for_each` toimii? Oleellisesti näin (malli eli *template*):

```
template<class Iter, class Func>
Func for_each(Iter first, Iter last, Func f) {
  for ( ; first!=last; ++first ) f(*first);
  return f;
}
```

Huomaatko rivin `class Func`: kolmas argumentti voi siis olla kokonainen luokka - kunhan siinä on määritelty mitä `double`-argumenttinen funktio tekee!
Tämä voi jopa olla aivan oma funktiotyyppinsä, nimittäin **funktio-objekti**. Funktio-objekti on melko kehittynyttä C++:aa, mutta paljon käytetty numeriikassa. Periaatteena on määritellä luokan sisällä funktio, joka ei ole luokan menetelmä. Esim.

```
  double operator()(double & x){ x=cos(x); }
```

Tämä on se funktio, jonka `for_each` suorittaa joka elementille, eli tässä niistä jokainen korvataan kosinillaan.

Miksi käyttää funktio-objektia? Miksei vain tavallista funktiota? Koska

1) funktiota ei lähetetä osoittimena, kääntäjän on helppo tehdä funktiosta ns. *inline* funktio → nopeuttaa suoritusta

2) luokkaan voi lisätä tietoa, jotka toimivat kuten laskurit (staattiset muuttujat), mutta joihin pääsee käsiksi vaikkapa luokan menetelmillä. Esim. koodi voi laskea elementeistä kosinin ja samalla kaikkien kosinien summan - tämä tehdään ohjelmassa `stl_forarch_functor2.cpp`. [6]

[6] Siinä on yksi pikku mutka: `for_each` käyttää luokan *kopiota*, ei luokkaa itseään.

Esim. 22: (stl_foreach_functor.cpp) STL for_each käyttäen funktio-objektia

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
using namespace std;

class TakeCos{
public:
  void operator()(double& x){ x=cos(x); } // function object
};
void doubleout(double& x) { cout<< x<<" ";}
void vector_out(vector<double> & x){
  for_each (x.begin(), x.end(), doubleout);
  cout<<endl;
}
int main () {
  vector<double> x;
  x.push_back(1.1); x.push_back(2.2);  x.push_back(3.3);
  cout << "vector x      : ";
  vector_out(x);
  TakeCos Cos ; // create one instance of TakeCos, same time one function object
  for_each(x.begin(), x.end(), Cos);
  cout << "vector cos(x) : ";
  vector_out(x);
  return 0;
}
```

Funktio-objektiin voi lisätä monimutkaisen operaation, esimerkiksi edelliseen voi lisätä kosinien summauksen. Se on toteutettu esimerkissä stl_foreach_functor2.cpp.

STL: generate algoritmi

Kätevä tapa tuottaa eli generoida arvoja kontin alkioihin.

VAROITUS: generate ottaa kopion kolmannesta argumentista.

Esim. 23: (stl_generate.cpp) Täytetään kontti satunnaisluvuilla

```
#include<iostream>
#include <algorithm>
#include <vector>
using namespace std;
double double_random() {
  // poor random numbers 0...1
  return rand()*1.0/RAND_MAX; // avoid int/int !
}
int main() {
  vector<double> v(20);
  // fill vector with random numbers
  generate(v.begin(), v.end(), double_random);
  // output
  for(unsigned i=0; i<v.size(); ++i){
    cout<<i<<"  "<<v[i]<<endl;
  }
}
```

Miksi varoa `int/int`-jakolaskua, eli kokonaisluku jaettuna kokonaisluvulla? Koska kääntäjän mielestä kokonaisluku/kokonaisluku = kokonaisluku, eli $5/7 = 0$. Tätä laskuvirhettä kiertämään laitoin ohjelmaan rivin

```
  rand()*1.0/RAND_MAX
```

niin että rand()*1.0 on reaaliluku ja se jaettuna kokonaisluvulla on reaaliluku - voíla, oikea tulos!
Lue seuraava kappale, jotta tiedät miksi `generate` on hiukan vaarallinen satunnaisluvuille!

STL algoritmit - pidä varasi!

Edellisestä esimerkistä voisi ajatella, että `stl::generate` on erinomainen tapa täyttää kontti satunnaisluvuilla. Se onkin näppärä, mutta potentiaalisesti vaarallinen! Sen toiminta on tällainen:

```
template <class ForwardIterator, class Generator>
  void generate ( ForwardIterator first, ForwardIterator last, Generator gen )
{
  while (first != last)  *first++ = gen();
}
```

Katso 3. argumenttia, se on "`Generator gen`", missä `Generator` on luokka. Siinä ei lue '`Generator* gen`", joten `gen` ei ole osoitin, eikä siinä lue '`Generator & gen`", joten `gen` ei ole viite. Generaattori siis lähtee funktioon arvona, eli **kääntäjä tekee kopion** generaattorista `gen`! Kopiointiin se käyttää luokan `Generator` kopionmuodostinta. Nyt koetettiin tehdä satunnaislukuja, joten **satunnaislukugeneraattorista otetaan kopio**.

Jaa miksikö tämä saattaa olla vaarallista? Satunnaislukugeneraattori on vain ohjelma, joka tuottaa tietyn, melkein satunnaisen numerojonon, jono riippuu ainoastaan siemenluvusta (*seed*), joka annetaan vain kerran. Jos huonosti käy, sinulla on siis kopioinnin seurauksena *kaksi täysin identtistä* satunnaislukumyllyä, joten kun käännät kummankin kampea tulee ulos kaksi samaa lukusarjaa. Tämä saattaa pilata herkän satunnaisuuteen perustuvan simulaation.

Myös `for_each`-algoritmi tekee kopion 3. argumentista.

Omat datarakenteet

Pelkkien lukujen sijasta joutuu usein käsittelemään hiukan monimutkaisempia datarakenteita kuten lukupareja, lukujen ja konttien yhdistelmiä jne.

Esim. 24: (pairs_of_numbers.cpp) Luokka lukupareja varten

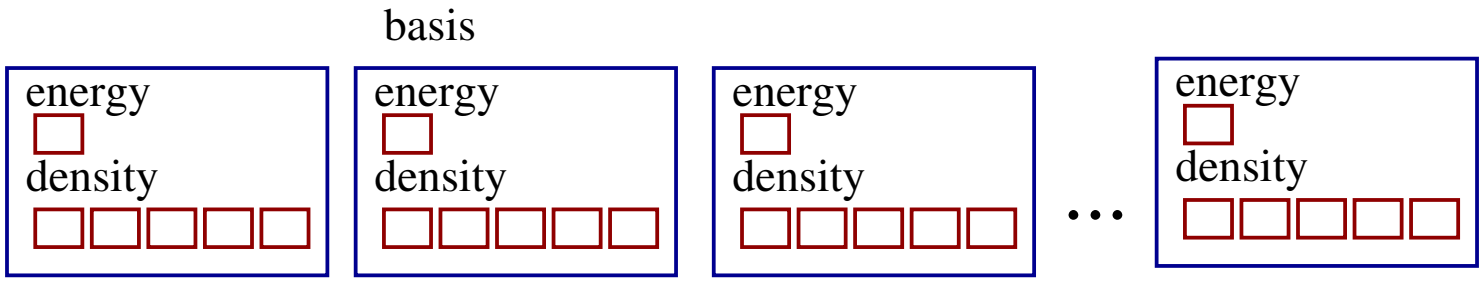
```
#include <iostream>
#include <cmath>
using namespace std;
// Class for pairs of numbers
class TwoDouble{
public:
  double x,y;
};
double dist(const TwoDouble &, const TwoDouble &) ;
int main()
{
  TwoDouble point1,point2;
  point1.x = 1.0;  // Class data member x can be accessed directly: It was public
  point1.y = 2.0;  point2.x = -2.0;    point2.y = 1.0;
  cout<<"distance="<<dist(point1,point2)<<endl;
  return 0;
}
// distance between two points (x,y)
double dist(const TwoDouble  & c1, const TwoDouble & c2) {
  return ( sqrt(pow(c1.x-c2.x,2) + pow(c1.y-c2.y,2) ) );
}
```

Koetapa muuttaa funktion nimeksi `distance`, kuten olisi mukavaa, ja saat palkaksi kamalasti virheilmoituksia! Käytössä on koko `std` nimiavaruus, ja sielläpä onkin jo funktio `distance`, mutta iteraattoreille. Jos haluat tietää miten voit käyttää `distance` nimeä, katso esimerkkiä `pairs_of_numbers2.cpp`.

Tehdään luokka, ja kootaan vector-konttiin tämän luokan olioita.
Esim. 25: (std11_vector_of_class_objects.cpp) std::vector jossa on luokan olioita

```
// g++ -std=c++0x std11_vector_of_class_objects.cpp
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
class WaveFunction{
public:
  double energy;
  vector<double> density;
};
int main()
{
  vector<WaveFunction> basis; // a vector of WaveFunctions
  WaveFunction wf;
  for (int i=0;i<10;++i){ // make a 10 wavefunction basis
    wf.energy = i*i;
    for (int j=0;j<5;++j) wf.density.push_back(sqrt(j)*i);
    basis.push_back(wf);
    wf.density.resize(0);// REMEMBER THIS; without reset wf.density keeps growing
  }
  // output just for testing
  for (auto wf: basis) {   // wf goes through elements of basis
    cout<<" energy = "<<wf.energy<<endl;
    cout<<"density = ";
    for (auto den: wf.density) cout<<den<<" "; // den goes through a density in wf
    cout<<endl;
  }
  return 0;
}
```

Kuten huomaat, voit täyttää vector-kontin millä hyvänsä tietotyyppillä. Tässä kontin alkiot ovat itse määriteltyä tyyppiä WaveFunction (siniset laatikot), ja kunkin sisällä on vielä luku (energy) ja vector-kontti (density).



Nämä kaksi ovat täsmälleen sama asia:

```
class WaveFunction{
 public:          // class: all is private by  default
  double energy;
  vector<double> density;
};
```

```
struct WaveFunction{
 double energy;  // struct: all is public by default
 vector<double> density;
};
```

C++11 satunnaisjakaumien generointia

(ennen tämä tehtiin Boost-kirjaston avulla)

Satunnaislukugeneraattori on ohjelma, joka laskee annettua siemenlukua (vaikkapa 23525176471263) käyttäen satunnaiselta vaikuttava pitkän lukusarjan. Se on kuin mylly: siemenluku sisään ja jauhetaan se satunnaisjauhoksi. Aina samalla tavalla. Jos haluat eri jauhot, heitä sisään eri siemen.

Vaiheet:

1) Ota mukaan otsikot

```
#include <random>
#include <functional>  // if you use std::function
```

2) Valitse satunnaislukugeneraattorityyppi (miten hieno sanahirviö!)

```
std::mt19937 gener; // Mersenne twister
```

Tässä `gener` on oma nimitykseni generaattorille; vain tämä esiintyy muualla koodissa ja voit vaihtaa toiseen generaattoriin (`linear_congruential_engine` tai `subtract_with_carry_engine`) muuttamalla yhtä riviä,

```
std::linear_congruential_engine gener;
```

3) Valitse jakauma
tasainen jakauma reaalilukuja on `uniform_real_distribution` (parametreina välin alku ja loppu) ja normaalijakauma `normal_distribution` (parametreina keskiarvo ja varianssi).
Tasainen jakauma `unif_dist`, joka on U[0,1), tehdään näin:

```
std::uniform_real_distribution<double> unif_dist(0,1);
```

ja normaalijakauma näin (oma nimeni luokan oliolle on `normal_dist`:

```
std::normal_distribution<double> normal_dist(0,1);
```

4) Alusta generaattori siemenluvulla **vain kerran**
(u luvun lopussa tarkoittaa unsigned)

```
gener.seed(4835267u); // same sequence every time you run the code
```

tai systeemin kellonajasta otettuna

```
gener.seed(static_cast<uint_fast32_t> (std::time(0)); // at least 32 bits
// different sequence every time you run the code (if time(0) changed  )
```

5) JOKO: Sido generaattori ja jakauma yhteen:

```
auto normal_random = std::bind(normal_dist, gener); // do this once
// here "auto" is actually static function<double()>
```

ja käytä yhdistelmää näin:

```
double random = normal_random();
```

TAI: tee satunnaisluku suoraan jakaumasta ja generaattorista näin:

```
double random = normal_dist(gener);
```

Haluttu satunnaisluku on tässä `random`.

Seuraavassa esimerkissä on joukko helppokäyttöisiä apuohjelmia, joilla voi alustaa generaattorin ja tuottaa tasan jakautuneita (`unirand()`), normaalijakautuneita (`gaussrand` ja `gaussrand2()`) sekä eksponenttijakautuneita (`exprand()`) satunnaislukuja.
Käyttö on helppoa, kaikki sotku on haudattu pois näkyvistä:

```
#include "std11_random.hpp"
...
double random = gaussrand()
```

Esim. 26: (std11_random.cpp) C++11 satunnaislukujen generointia

```
// std::mt19937 random number generator
// uses std::function and std::bind
#include <iostream>
#include <random>
#include <ctime>
#include <fstream>
#include <functional>
using namespace std;
std::mt19937 gener; // define generator
void initrng(void){
  static bool first = true;
  if(first){
    auto seed = static_cast<uint_fast32_t> (std::time(0));
    cout<<" seed = "<<seed<<endl;
    gener.seed(seed);
    first = false;
  }
}
double unirand(void){
  static bool first = true;
  static function<double()> rnd ;
  if(first){
    initrng();
    uniform_real_distribution<double> unif_dist(0,1);
    rnd = bind(unif_dist, gener);
    first = false;
  }
  return rnd();
}
```

```
double gaussrand(void){
  static bool first = true;
  static function<double()> rnd ;
  if(first) {
    initrng();
    normal_distribution<double> norm_dist(0,1);
    rnd = bind(norm_dist, gener);
    first = false;
  }
  return rnd();
}
double gaussrand2(void){ // warning: does not initialize generator
  static normal_distribution<double> norm_dist(0,1);
  return (norm_dist(gener));
}
double exprand(void){
  static bool first = true;
  static function<double()> rnd ;
  if(first) {
    initrng();
    exponential_distribution<double> expo;
    rnd = bind(expo, gener);
    first = false;
  }
  return rnd();
}
```

Tulostuksen muotoilua

Numeerisen datan luettavuus vaatii tulostuksilta hyvää asemointia.
Esim: on aika hankalaa lukea jos sarakkeet menevät näin sekaisin:

```
x y z
1.542234 12.4234 0.1213
13.0 4.234 1.00
```

Toinen muistettava seikka on, että laskun tuloksen tarkkuutta saa rajoittaa menetelmän huonous, ei tiedostoon tallennettujen desimaalien määrä. `std::cout`-olioon voi syöttää mm. kentän leveyden (10) ja esitystavan (fixed) ja desimaalien määrän (6):

```
cout<<fixed<<setprecision(6);  // 6 decimals
cout<<setw(10)<<"x"<<setw(10)<<"y"<<setw(10)<<"z"<<endl; //field width is 10
cout<<setw(10)<<x<<setw(10)<<y<<setw(10)<<z<<endl;
```

Sama muotoilu toimii myös tiedostoon kirjoitukseen:

```
ofstream output("data.out");
output<<fixed<<setprecision(6);
output<<setw(10)<<x<<setw(10)<<y<<setw(10)<<z<<"\n";
```

```
data.out:
  1.542234 12.423400  0.121300
 13.000000  4.234000  1.000000
```

Esimerkki `output_formatting.cpp` antaa lisää esimerkkejä. Osa asetuksista jää päälle, osa unohtuu heti! Pitäydyn C++-esimerkeissä `stream`-olioissa, enkä käytä C-kielen `printf`-funktiota. [7]

[7] Miksi `stream`? (i) *type safety*, stream joko toimii tietotyypille oikein tai kääntäjä huomaa ettei se toimi. (ii) stream on muokattavampi, sama toimii sekä ruudulle että tiedostoon.

Lineaarialgebraa - mitä kirjastoa käyttää?

Vakiovastaus on LAPACK ja BLAS, jotka on kirjoitettu fortranilla ja saatavilla kokoelmasta www.netlib.org. Prosessorivalmistajilla on tarjolla optimoidut versiot (ACML,MKL) ja automaattisesti optimoidut ATLAS-versiot. C-kieliset versiot (CLAPACK, CBLAS) on luotu f2c-kääntäjällä, mikä hiukan helpottaa käyttöä C++-koodissa[8]. GSL-kirjasto tarjoaa todella laajan C-numeriikkapaketin.

C++-ohjelmoijan unelmapaketti on kirjoitettu C++:lla. Tällainen oli LAPACK++, mutta projekti nuukahti v. 2000. Tilalle tuli TNT (Template Numerical Toolkit), joka sekin näytti heikolta, mutta näyttää olevan elossa (2010). LAPACK++ yritettiin herättää ja nyt odotellaan pysyykö se elossa. Varmimpia veikkauksia on Boost, mutta se on enimmäkseen muuta kuin numeriikkaa.

Koska optimoidut LAPACK ja BLAS ovat niin yleisiä, on joko

- opittava itse kutsumaan fortran-aliohjelmia C++:sta (tarkkaa puuhaa)
- käytettävä muiden tekemää ns. *wrapperia*, joka on C++-koodi joka tarvittaessa kutsuu fortran-aliohjelmia tai niiden C-kielisiä versioita.

Muutamia projekteja, jotka tarjoavat korkealuokkaisia wrappereitä:

- **Armadillo** (australialainen)
- **IT++**(ruotsalainen; viimeisin version vuodelta 2010)
- **NEWMAT** (Robert B. Davies, Uusiseelanti; näemmä pysähtyi vuoteen 2008).

Armadillo kehuu olevansa näistä nopein [9]. Syntaksi on suoraviivainen, lähellä Matlab/Octave syntaksia. Se tuntee nyt myös harvat matriisit (*sparse matrices*, vain pieni osa matriisielementeistä on nollasta poikkeavia).

[8] ... ja on yhtä arveluttavaa kuin rahan valokopion käyttö ostamiseen.

[9] Versio 3.4.4 on päivätty 2.11.2012.

Esim. 27: (arma_matrix_multi.cpp) Armadillo: Matriisitulo (lähde: Armadillo www-sivu)

```
#include <iostream>
#include <armadillo>

using namespace std;
using namespace arma;

int main()
{
  mat A = randu<mat>(4,5); // mat is double
  mat B = randu<mat>(4,5);

  cout << "A*trans(B) =" << endl;
  cout << A*trans(B) << endl;

  return 0;
}
```

Tämä kutsuu piilossa CBLAS-rutiinia, koska käännösyritys

```
g++ arma_matrix_multi.cpp
```

tuottaa virheilmoituksen

```
undefined reference to `cblas_dgemm'.
```

(itse asiassa pitkän litanian virheilmoitusta, tämä on kuitenkin oleellisin pätkä)

Esim. 28: (arma_eigenvalues.cpp) Armadillo: Symmetrisen matriisin ominaisarvot

```
#include <iostream>
#include <iomanip>
#include <armadillo>
using namespace std;
using namespace arma;
int main(){
  mat A = randu<mat>(5,5);
  vec eigval;
  mat eigvec;
  A = A+trans(A);  cout<< "A= \n"<<A<<endl;;

  eig_sym(eigval, eigvec, A);

  vec x(eigval);
  for (unsigned i=0;i<A.n_rows;i++){
    cout<<i<<"th eigenvalue = "<<eigval(i)<<endl;
    x = eigvec.col(i); //x(j) =  eigvec(j,i);
    cout<<"                x = "<<trans(x);
    x = A*x/eigval(i);
    cout<<"check: Ax/lambda = "<<trans(x)<<endl;
  }
  return 0;
}
```

Varsinainen työ tehdään yhdellä rivillä:

```
eig_sym(eigval, eigvec, A);
```

Tulostus:

```
A=
   1.6804   0.5919   1.2605   1.7146   0.9279
   0.5919   0.6704   1.3971   0.9135   0.7969
   1.2605   1.3971   0.7296   1.2307   1.0895
   1.7146   0.9135   1.2307   0.2832   1.4111
   0.9279   0.7969   1.0895   1.4111   0.3134

0th eigenvalue = -1.30315
               x =   -0.3020  -0.1113   0.0491   0.8003  -0.5035
check: Ax/lambda =   -0.3020  -0.1113   0.0491   0.8003  -0.5035

1th eigenvalue = -0.803214
               x =    0.1275   0.5699  -0.7894   0.1882   0.0197
check: Ax/lambda =    0.1275   0.5699  -0.7894   0.1882   0.0197

2th eigenvalue = -0.281196
               x =    0.3701   0.3639   0.2307  -0.3048  -0.7645
check: Ax/lambda =    0.3701   0.3639   0.2307  -0.3048  -0.7645

...
```

C tai fortran osana C++ -ohjelmaa

Tässä periaatteet, jotta osaat ainakin lukea koodia.

Esim. 29: fortran aliohjelma `dgemm` laskee kahden matriisin tulon.

BLAS:

```
SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,
                 B,LDB,BETA,C,LDC)
DOUBLE PRECISION ALPHA,BETA
INTEGER K,LDA,LDB,LDC,M,N
CHARACTER TRANSA,TRANSB
DOUBLE PRECISION A(LDA,*),B(LDB,*),C(LDC,*)
```

Esittely C++-ohjelmassa:

```
extern "C"
{
    void dgemm_(char* transa, char* transb, int* m,
                int* n, int* k, double* alpha,
                double* a, int* lda, double* b,
                int* ldb, double* beta, double* c, int* ldc);
}
```

- `extern` "C"
  tarkoittaa "käännä C-tyylillä".
  C++-kääntäjälle pitää kertoa, että funktio tunnistetaan sen nimen (`dgemm_`) perusteella kuten C:ssä, ei argumenttien perusteella. C++ funktiot voivat olla lisämääritettyjä, siksi se ei nyt käy.
- Käännetty fortran-aliohjelma on objektikoodissa (eli tiedostossa `dgemm.o`) alaviivallisena, eli `dgemm()` on `dgemm_()`.
- fortran välittää argumentit "pass by reference", joten nyt kaikkien funktion argumenttien pitää olla osoittimia.
  `double* a` osoittaa kaksinkertaisen tarkkuuden taulukon alkuun (`d` nimessä `dgemm` on "double precision")

Vielä on yksi kiusallinen ongelma: matriisi on fortranissa (ja Matlabissa) taulukkona A(i,j), missä indeksi i (*row*) muuttuu nopeimmin eli *column major*. C ja C++ tallentaa sen *row major* järjestyksessä. Alla kuva matriisista muistissa:

Matriisi	fortran	C ja C++
$\begin{pmatrix} 1\,2\,3\,4 \\ 5\,6\,7\,8 \end{pmatrix}$	1526348	12345678

Kumpikin tapa on yhtä luonnollinen.

Jos taulukon indeksin koko on määritelty vakioksi näin,

```
const int n=100;
double a[n][n];
```

on edellä fortran-kutsun esittelyyn on myös laitettava avainsanat `const int* n` .

Muista myös, että N:n alkion taulukon `V` indeksit ovat

fortran : $1 \rightarrow$ N, eli `V(1),V(2)...,V(N)`

C++ : $0 \rightarrow$(N-1), eli `V(0),V(1)...,V(N-1)`

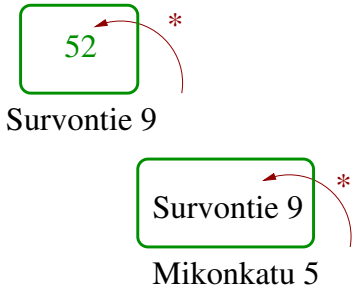
Lisää osoittimista (eli pointtereista)

Kaksi operaattoria:
`*`:llä otetaan esiin se mihin osoitin viittaa (*dereferencing*)
`&`:lla otetaan esiin otuksen osoitin

Miten luetaan merkintöjä, joissa on osoittimia:

- `int* p` tai `int *p` : "p osoittaa kokonaislukuun" tai "p on osoitin kokonaislukuun".
  `p` on siis osoitin eli pointteri ja `*p` on kokonaisluku.
- `int c; int *p; p=&c;` : "c on kokonaisluku; p on osoitin johonkin kokonaislukuun; p on osoitin kokonaislukuun c".
- `&c` : "osoitin c:hen" tai "c:n osoitin"
- `*p` : "se mihin p osoittaa"
- `*&c` :"se mihin c:n osoitin osoittaa" eli sehän on c itse. Operaatiot kumoavat toisensa (tämä jättää hiukan vaihteluvaraa - mitä on "c itse", jos c on luokka?)
- **Iteraattori ei ole osoitin!**
  Jos haluat kaivaa esiin osoittimen iteraattorista `iter`, tee näin:
  ```
  &*iter
  ```
  Huomaatko mikä tämän idea on: `*iter` on se kontin `alkio`, johon iteraattori osoittaa, joten *sillä* (alkiolla siis) on kelpo osoitin, se on `&alkio`, eli juuri `&*iter`.
- `int **d` : "d on osoitin osoittimeen, joka osoittaa kokonaislukuun". Voit ajatella että `d` on muistiosoite, johon on talletettu toinen muistiosoite, johon on talletettu kokonaisluku.

Älä tutki tätä kuvaa liian tarkasti :)



d=Mikonkatu 5 (osoite, ei turvallista olettaa kokonaisluvuksi)
*d=Survontie 9
**d=52 (kokonaisluku)

Taulukot (*arrays*)

Käytä mieluummin kontteja. Tämä on silti hyödyllistä yleistietoa jos luet muiden tekemiä ohjelmia.

Vaikka olenkin suosinut vector-kontteja, on monet C++ ohjelmat kirjoitettu käyttäen taulukoita. Taulukko (*array*) on "tyhmempi" kuin vector, sitä ei voi venytellä eikä se tiedä omaa kokoaan. Usein erehtyy käyttämään taulukon alkiota, jota ei ole: tästä seuraa ylivuoto ja ohjelman voi välillä käyttäytyä kummallisesti tai kaatua. Taulukko on matalan tason ohjelmointia ja vaatii ohjelmoijalta huolellisuutta.

Kiinteän kokoisia taulukoita (*array variables*) tehdään näin;

```
double array1[10]; // memory alloc. for elements array1[0],array1[1],...,array1[9]
int array_int[]={3,6,12}; // memory allocation and fill with values
```

tai matriisia vastaava kaksiulotteinen taulukko

```
double array2[5][3]; // 5x3 array
```

Edellä luodun taulukon elementtiin (2,1) viitataan

```
double x = array2[2][1]; // element array2(2,1), row = 2, column 1
```

ja muistissa elementit ovat peräkkäin, ns. *row major* järjestyksessä, eli 1. indeksi juoksee nopeimmin. Muistissä tämä on siis yhtenäinen pätkä

$$...\textit{muuta koodia}\ (0,0),(1,0),(2,0),(3,0),(4,0),(0,1),(1,1),(2,1)...(4,2)\ \textit{muuta koodia}...$$

Taulukon tilanvaraus toimii eri tavalla kuin vector-kontille, esimerkiksi

```
int n;
cin >>n;
double a_1[n]; //  MAY NOT WORK, size n is unknown to compiler
               // g++ compiles happily
vector<double> vec(n); // OK, vectors are more flexible
...
k = 15;
double a_3[k]; //  SUSPICIOUS, but size k is sort of known and g++ is happy
...
const int m=50;
double a_2[m]; //  Ok, size m is known
int k;
```

Yllä ensimmäinen on oppikirjaesimerkki väärästä tavasta.

Heap and Stack

Suomeksi keko ja pino; en juuri käytä näitä nimiä, koska virheilmoitukset eivät ole suomeksi. Kiinteäkokoiset taulukot menevät *stack*-muistiin, eli pinomuistiin; stackin rajallinen koko tulee joskus vastaan isoilla taulukoilla. Dynaamisesti varattavat taulukot menevät *heap*-muistiin, niitä tehdään näin:

```
int needsize = ...;
double *tab = new double[needsize] ; // reserve memory for double tab[]
... // use array tab[]
delete[] tab; // free memory, tab is no longer used
```

Jos unohdat hakasulut `delete`-operaatiosta tulet korruptoimaan *heap*-muistin,

```
 delete tab; // Bad things happen, usually program fails
```

Moniulotteisen dynaamisen taulukon luominen ja hävittäminen onkin sitten sotkuisempi juttu, se **ei ole** `delete[][] array2` . Voit joutua varaamaan taulukon joka rivin erikseen ja lopulta tuhoamaan ne erikseen. Tämä on hyvä paikka tehdä virhe joka turmelee koodisi, koska kääntäjä ei voi tarkistaa teetkö asiat oikein.

Hae jostain kirjastosta matrix-luokka, älä käytä dynaamisia moniulotteisia taulukoita.

Sinua on varoitettu.

Tavallisen muuttujan nimi on muuttujan arvo, mutta

Taulukon (*array*) nimi on osoitin taulukon alkuun

Esim. 30: (array_name_is_pointer.cpp) Tulostetaan taulukon nimi

```
#include <iostream>
using namespace std;
int main(){
  int i=123;
  int j[]={111,222};
  cout<< i<<endl;
  cout<< j<<endl;
  return 0 ;
}
```

123
0x7fff43dbde20

Ensimmäinen numero in muuttujan `i` sisältö, mutta jälkimmäinen on muistiosoite (0x etuliite kertoo kantaluvun 16) . Usein funktio haluaa argumenttina osoittimen, siispä taulukon nimi käy siihen tarkoitukseen. Osoite vaihtuu ajokerrasta toiseen, koska `j` tallentuu milloin minnekin.

Edellä tehtiin dynaaminen taulukko `tab` rivillä

```
double *tab = new double[needsize]
```

Taulukon nimi `tab` on osoitin taulukon alkuun, eli tämä rivi varaa tilaa `needsize`'lle double-tyyppiselle luvulle ja laittaa `tab`in osoittamaan varatun tilan alkuun.

Esim. 31: (array_name_is_pointer2.cpp) Käsitellään taulukon alkioita (ja pikku virhekin)

```
#include <iostream>
using namespace std;
int main(){
  int j[]={111,222};
  cout<< j[0]<<" "<<j[1]<<" "<<j[2]<<endl;
  //      ok          ok         bug!
  return 0 ;
}
```

111 222 4196096

Taulukon ensimmäinen alkio on `j[0]`, toinen `j[1]` ja - kolmas ! - `j[2]`. Tämä viimeisin on virhe, taulukossa ei ole alkiota 3. Siksi kolmas tulostuva numero on vain jotain, mitä sattui muistissa olemaan taulukon `j` jälkeen (tulkittuna kokonaisluvuksi). Tällaista väärää taulukon käsittelyä sanotaan **(taulukon) ylivuodoksi**

Debuggausoptiot päällä ohjelma lopettaa virheilmoitukseen ylivuotokohdassa ja kertoo mikä vuotaa. Harmi kyllä ylivuotojen metsästys hidastaa ohjelman suoritusta pahasti, siksi tuotantoajossa koodi käännetään optimoituna eikä ylivuotoja enää testata. Muista kääntää kehitysvaiheessa koodi kaikki mahdolliset varoitukset päällä ja koeaja se! Yksi hyödyllinen apuohjelma on **valgrind** :

```
g++ array_name_is_pointer2.cpp
valgrind a.out
==9283== Memcheck, a memory error detector
 ... paljon virheilmoituksia, koska koodissa on ylivuoto ...
```

Toimivasta koodista on helpompi saada nopea, kuin nopeasta koodista toimiva..

Esim. 32: (array_to_function.cpp) Taulukon lähettäminen funktioon

```
#include <iostream>
using namespace std;
void f(int d[],const int sized){ // C++ style, empty [] tells compiler d is array
  // d is passed by reference!
  // You are dealing with the original array, not with it's copy.
  for (int i=0;i<sized;++i) cout<<i<<" "<<d[i]<<endl;
}
int main(){
  int j[]={2,4,5};
  f(j,3);
  return 0 ;
}
```

Esim. 33: (array_to_function2.cpp) Taulukon lähettäminen funktioon, C-tyylillä

```
#include <iostream>
using namespace std;
void f(int *d,const int sized){ // C-style, think of d as a pointer
  for (int i=0;i<sized;++i) cout<<i<<" "<<d[i]<<endl;
}
int main(){
  int j[]={2,4,5};
  f(j,3);
  return 0 ;
}
```

Ainoa ero esimerkeissä oli funktion argumenttien tyypeissä, samantekevää kumpaa tyyliä käytät.

Osoitin funktioon tarvitaan, jos funktioon pitää lähettää tieto mitä toista funktiota sen pitää käyttää. Demotehtävänä ollut integrointi on hyvä esimerkki: ei ole taloudellista kirjoittaa integrointifunktiota, joka toimii vain yhdelle integrandille.

Esim. 34: Kirjastofunktioiden lähettäminen funktioon laske()

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

double laske(double (*f)(double)){
  double x=5.5;
  return f(x);
}
int main(){
  cout << laske(sinh) <<endl;;
  cout << laske(cos) <<endl;;
  cout << laske(acos) <<endl;;
  return 0;
}
```

```
122.344
0.70867
nan      ‘‘nan’’ tai ‘‘NaN’’ tarkoittaa NotaNumber.
         Matikka on hakoteillä, acos(5.5) ei ole määritelty
```

Myös omat funktiot voi lähettää argumentteina samalla tavalla.
Esim. 35: Itse tehdyn funktion lähettäminen funktioon laske()

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

double oma(double x){
  return pow(x,3)+sin(x)+cos(x);
}
double laske(double (*f)(double)){
  double x=5.5;
  return f(x);
}
int main(){
  cout << laske(oma) <<endl;;
  return 0;
}
```

```
166.378
```

Virhetilanteiden hallinta: `throw` ja `catch`

C++ standardikirjastossa on virheille luokka `std:exception`, jossa on mm. osa `runtime_error`. Jos haluat kehitellä oman virhekäsittelyn, kannattaa *periä* tämä luokka tyyliin

```
class MyException : public std::exception{
 ...
}
```

ja lisätä oma ominaisuus.
En useimmiten mene näin pitkälle, vaan käytän vain seuraavaa yksinkertaista mekanismia:

```
  try{
    my_function();
  }
  catch (char const* e) {
    cerr << e << endl;
    return 1;
  }
```

ja funktiossa on rivi

```
  void my_function(void){
  {
        ...
    if(test) throw "test failed";
    ...
  }
```

Virheen sattuessa (`test` on `true`) heitetään teksti "test failed", poistutaan funktiosta ja napataan virhe `catch`:illä [10]
Tässä virheilmoitus tulostettiin `cout`:n kaltaiseen vuohon `cerr`, joka on erikoistunut virhetilanteisiin.

[10] Funktio `my_function()` siis heittää ilmaan virheen ja toivoo, että jokin virheenhallinta nappaa sen myöhemmin ja tekee asialle jotain. Tämä voi olla parempi kuin lopettaa ohjelma kuin seinään.

GSL: Gnu Scientific Library

wikipedia

Kirjasto on ilmainen ja yleisesti saatavilla. GSL on kirjoitettu C:llä, mutta sitä voi mainiosti käyttää C++:sta.

> *The library header files automatically define functions to have extern "C"linkage when included in C++ programs. This allows the functions to be called directly from C++.*

Et siis tarvitse `extern ''C''` rivejä! Tämä helpottaa suuresti GSL-funktioiden kutsumista C++-ohjelmasta.

Esim. 36: (gsl_bessel.c) Besselin funktion $J0$ arvon laskeminen (C-kielinen)

käännös: g++ gsl_bessel.c -lgsl -lgslcblas tai g++ gsl_bessel.c ‘gsl-config –libs‘ (kokeilepa komentoa gsl-config –libs)

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
int main(void) {
  double x = 5.0;
  double y = gsl_sf_bessel_J0(x);
  printf("J0(%g) = %.18e\n", x, y);
  return 0;
}
```

Esim. 37: (gsl_bessel.cpp) Sama C++:lla) :

```
#include <iostream>
#include <iomanip>
#include <gsl/gsl_sf_bessel.h>
using namespace std;
int main(void) {
  double x = 5.0;
  double y = gsl_sf_bessel_J0(x);
  cout<<setprecision(18);
  std::cout<<"J0("<<x<<") = "<<fixed<<y<<endl;
  return 0;
}
//output J0(5) = -0.177596771314338264
```

GSL: statistiikkaa

`double gsl_stats_mean` (*const double* `data[]`, *size_t* `stride`, *size_t* `n`) [Function]

This function returns the arithmetic mean of *data*, a dataset of length n with stride *stride*. The arithmetic mean, or *sample mean*, is denoted by $\hat{\mu}$ and defined as,

$$\hat{\mu} = \frac{1}{N} \sum x_i$$

where x_i are the elements of the dataset *data*. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N.

`double gsl_stats_variance` (*const double* `data[]`, *size_t* `stride`, *size_t* `n`) [Function]

This function returns the estimated, or *sample*, variance of *data*, a dataset of length n with stride *stride*. The estimated variance is denoted by $\hat{\sigma}^2$ and is defined by,

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2$$

where x_i are the elements of the dataset *data*. Note that the normalization factor of $1/(N-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/N$.

This function computes the mean via a call to `gsl_stats_mean`. If you have already computed the mean then you can pass it directly to `gsl_stats_variance_m`.

Esim. 38: (gsl_statistics.cpp) Aritmeettinen keskiarvo ja keskipoikkeama ensin tavallisella taulukolla, sitten std::vector-luokan vektorilla

```
#include <iostream>
#include <vector>
#include <gsl/gsl_statistics.h>
using namespace std;
int main(void) {
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6};
  double mean, variance;
  mean     = gsl_stats_mean(data, 1, 5);
  variance = gsl_stats_variance(data, 1, 5);
  cout<<"     mean = "<<mean<<endl;
  cout<<" variance = "<<variance<<endl;
  // same with a vector container
  vector<double> v;
  v.assign(data,data+5); // reuse data
  mean     = gsl_stats_mean(&v[0], 1, 5);
  variance = gsl_stats_variance(&v[0], 1, 5);
  cout<<"     mean = "<<mean<<endl;
  cout<<" variance = "<<variance<<endl;
  }
```

Kuten huomaat, `std::vector`-kontin lähettäminen osoittimena on hiukan ruma syntaksiltaan. Alkuosoite `&v[0]` ja pituus 5 ovat *riittävä* tieto: vektorikontin alkiot ovat peräkkäin muistissa.

GSL: nopea Fourier-muunnos (FFT)

FFT:tä on tarkoitus kutsua yksinkertaisesti

```
fft(data,suunta)
```

missä data on kompleksista ja suunta on 1 Fourier-muunnokselle ja -1 käänteiselle muunnokselle. Huomaa, ettei mihinkään tiettyyn kirjastoon viitata tässä kutsussa. Päätös käyttää GSL:n Fourier-muunnosta tehdään include-lauseella, joka lataa alempana olevan itse kirjoitetun header-tiedoston.

Yksi asia on päätettävä: missä muodossa `data` esitetään? GSL käyttää C:n taulukkoa ja sitä me emme missään tapauksessa halua käyttää C++ ohjelmissa. Vaihtoehtoja ovat mm.

1.) `std::vector` -kontti, nyt siis kompleksiluvuille eli
`std::vector<complex<double> >`
Etuna on, että saamme paljon menetelmiä käyttöön, mutta `std::vector` on numeeriseen työhön hankala.
Miten kerrot kontin elementit luvulla 55?

```
transform(v.begin(),v.end(),v.begin(),bind2nd(multiplies<int>(),55));
```

tai (onneksi on C++11 standardi)

```
for( auto & x: v) {x*=55;}
```

2.) `std::valarray` -luokka, nyt
`std::valarray<complex<double> >`
`std::valarray` on suunniteltu numeeriselle datalle ja monet matemaattiset operaatiot sujuvat helposti, toisin kuin `std::vector`-kontille. Mutta: `std::valarray` ei tunne iteraattoreita, joten `std::vector`-kontille tehtyä koodia joutuu muokkaamaan paljon jotta se sopisi `std::valarray`-luokalle.
Miten kerrot luokan elementit luvulla 55?

```
 v *= 55; // or : v=v*55, but *= is better
```

Näin sen pitäisikin olla! Armadillo vektori osaa sen myös:

```
  v *= 55; // or : v=v*55, but *= is better
```

En osaa päättää, joten kirjoitan FFT-headerin mallina (template), joka toimii vector- ja valarray-kontille ja Armadillon vektorille.

Esim. 39: (gsl_fft.hpp) GSL:n FFT header-tiedostona (gsl_fft.hpp)

```
#ifndef GSL_FFT_HPP
#define  GSL_FFT_HPP
// Use data: either std::vector or std::valarray or armadillo vector
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

template <class T>
int fft(T& data, int direction){
  int status;
  const size_t stride=1;
  size_t n;
#ifdef ARMA
  n=data.n_elem; // Armadillo data has no member size, use nelem
#else
  n = data.size();
#endif
  double* pdata = reinterpret_cast<double*> (&data[0]);
  if(direction>0){
    status = gsl_fft_complex_radix2_forward(pdata, stride, n);
  }
  else {
    status = gsl_fft_complex_radix2_backward(pdata, stride, n);
  }
  if(status!=GSL_SUCCESS) return 1;
  return 0;
}
#endif
```

Myös funktion toteutus on samassa otsikkotiedostossa: mallien (template) erottaminen otsikkoon ja varsinaiseen koodiin tekee siitä kääntäjälle vaikeaa.

C++11 valittaa jos yritän ottaa osoitteen compleksiluvun reaaliosasta näin:

```
double* pdata = &data[0].real(); // may not compile
```

Kääntäjä voi tulkita, että otus `data[0].real()` on ns. *rvalue*, eli jotain, millä ei ole kunnon osoitetta (väliaikainen osoite voi olla). Siksi sitä ei voi myöskään ottaa talteen & -operaatiolla. Kierrän tämän ottamalla osoitteen otuksesta `data[0]` (sillä on varmasti osoite) ja muokkaan siitä osoittimen `double`-tyyppiiseen muuttujaan (`reinterpret_cast`):

```
double* pdata = reinterpret_cast<double*> (&data[0]);
```

Ja sitten huomaan, ettei Armadillon vektorilla ole menetelmää `size()`, enkä suurin surminkaan halua sotkea joka `fft`-kutsuun kolmatta argumenttia sitä varten. Siksi sovin, että Armadilloa käyttäessäni määrittelen suureen `ARMA`, ja annan preprosessorin hoitaa työn.

Alla oleva esimerkki on C-kielisenä GSL:n manuaalissa.

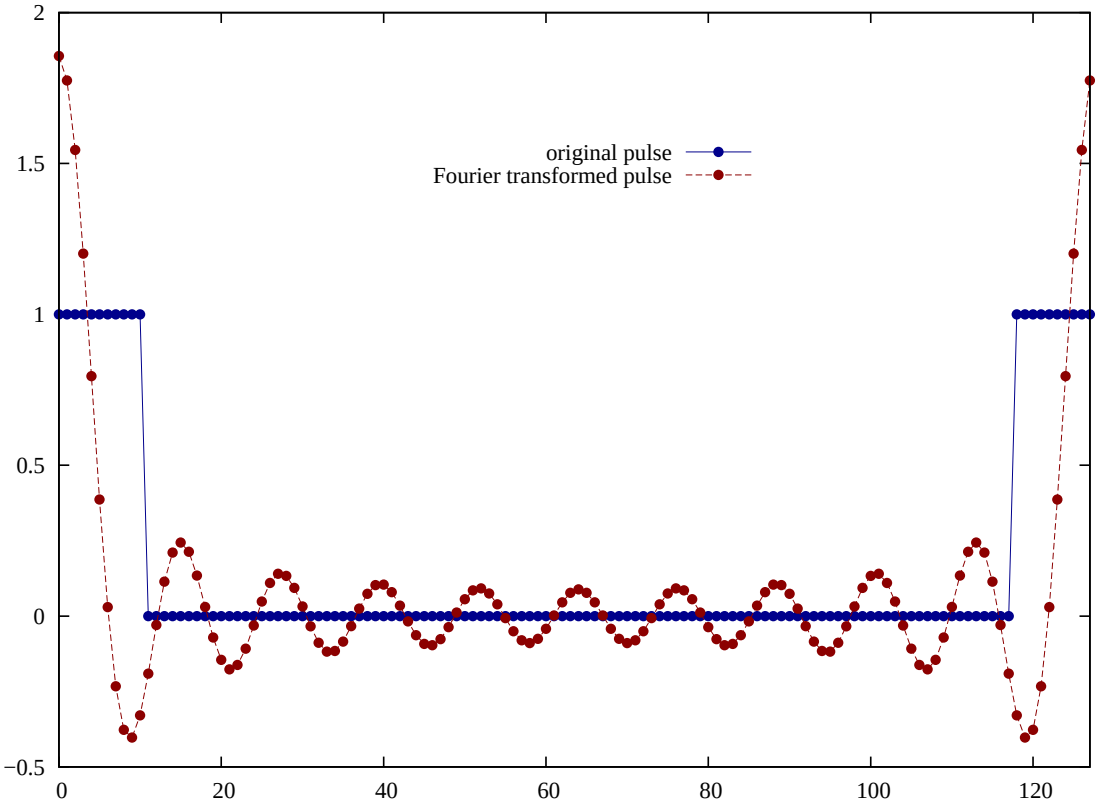
Esim. 40: (std11_gsl_fft_arma_main.cpp) FFT testiohjelma; tämä käyttää Armadillon vektoria.

```
// Compile:
// g++ -std=c++0x std11_gsl_fft_arma_main.cpp `gsl-config --libs`
#include <iostream>
#include <cmath>
#include <complex>
#include <fstream>
#define ARMA
#include "gsl_fft.hpp"
#include <iomanip>
#include <armadillo>

using namespace std;
using namespace arma;

// output help routine; works with many complex data types
template <class T>
void vector_out(std::ostream& stream, const T& v){
  stream<<setprecision(8)<<scientific;
  int i=0;
  for(auto x: v){ stream<<i++<<" "<<x.real()<<" "<<x.imag()<<endl; }
}
```

```
// short name for data type
typedef Col<complex<double>> vectype;


int main (void){
  const int n=128;
  const int n1= 10; // Keep less than n!
  vectype data(n);
  ofstream myfile("result");
  //
  // Symmetric pulse:  1111111111100...001111111111
  //                   11 ones            10 ones
  data.ones();
  for(int i=n1+1;i<n-n1-1;++i) data(i)=0.0;

  cout<<"before fft :\n";
  vector_out(cout,data);
  vector_out(myfile,data);
  myfile<<endl;
  // ----------------------
  // forward FFT
  int status = fft(data,1);
  // ----------------------
  if(status!=0){
    cout << "fft fails\n";
    return 1;
  }
  cout<<"after forward fft (divided by sqrt(n)) :\n";
  data /= sqrt(n);
  vector_out(cout,data);
  vector_out(myfile,data);
```

```
  // ----------------------
  // backward FFT
  status = fft(data,-1);
  // ----------------------
  if(status!=0){
    cout << "fft fails\n";
    return 1;
  }

  cout<<"after backward fft (divided by sqrt(n)) :\n";
  data /= sqrt(n);
  vector_out(cout,data);

  myfile.close();
  return 0;
}
```

FFT-ohjelman tulos:



FFT-data on taulukoitu tavallisen kieroutuneeseen tapaansa “kiertyvässä järjestyksessä” (*wrap-around order*).

std::valarray-luokasta

Kuten olet huomannut, std::vector-kontti ei ole ollenkaan kätevä numeriikassa - **kamalampaa voi tuskin kuvitella!** std::vector ei ole "vektori" geometrisessa mielessä, se on lista joka osaa venyä ja kutistua.

Numeerista dataa varten on `valarray`-luokka (ei kontti).
Lähin vastaavuus on

`std::valarray` ≈ `boost::ublas::vector`

Pitkään `ublas:vector` oli aivan ylivoimainen, koska se on koodatty käyttäen hyvin tehokasta *expression template*-tekniikkaa. Onneksi uusien kääntäjien valarray on koodattu samalla tekniikalla ja se on hyvin nopea. [11]

Oli miten oli, C++-ohjelmoijien keskuudessa std::valarray ei ole suosittu ja jotkut neuvovat karttamaan koko kyhäelmää. Tässä onkin vaikean ratkaisun paikka: käyttääkö jonkin ulkoisen kirjaston "vektoria" vai standardikirjaston valarrayta tai vector-konttia? Edellisen ongelma on se, ettei se ole standardoitu ja koko kirjasto voi lakata olemasta kun kehittäjät siirtyvät muihin töihin - tai mikä todennäköisempää, joku kirjoittaa tehokkaamman kirjaston ja kaikki ryntäävät sitä käyttämään. Valarray voidaan poistaa standardista. Teetpä miten hyvänsä, pelkäänpä ettei koodisi enää käänny 10 vuoden kuluttua. Kielen sisäänrakennettujen numeeristen datarakenteiden puutteellisuus on allekirjoittaneesta vakavin ongelma C++ numeriikan tiellä.

Oma suositukseni on: kirjoita koodia, joka hautaa päätöksen pois näkyvistä.

[11] Kyse on lopulta työpanoksesta: valarray on harvojen käyttäjien suosikki, eikä sen tehokkuuteen kääntäjää tehtäessä kannata laittaa liikaa vaivaa. Takana on ideologia "numeriikka tehdään kuitenkin fortranilla".

GSL: differentiaaliyhtälöt

Ratkaistaan yhtälö

$$y'(t) = -ty \quad , \quad y(0) = -1 \; , \quad \text{tarkka ratkaisu} y(t) = -2e^{-t^2/2} \; .$$

Esim. 41: (gsl_ode_simple.cpp)

```
// solve y'(t) = -t*y ,  condition y(0)  = -2
// compile :
// g++ --std=c++0x gsl_ode_simple.cpp `gsl-config --libs`
#include <iostream>
#include <iomanip>
#include <cmath>
#include <vector>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>
using namespace std;
int func (double t, const double y[], double f[], void *params){
  f[0] = -t*y[0] ; // y[0] = y,  f[0] = y' = dy/dt = -t*y
  return GSL_SUCCESS;
}
void output(double t, double y, double exact){
   cout<<fixed<<setprecision(16);
   static bool first=true;
   if(first) {
     cout<<setw(20)<<"t"<<setw(20)<<"gsl solution";
     cout<<setw(20)<<"exact solution"<<setw(20)<<"error\n";
     first = false;
   }
   cout<<setw(20)<<t<<setw(20)<<y<<setw(20)<<exact<<setw(20)<<y-exact<<endl;
}
```

```
int main (void)
{
  double exact;
  const int n=50;                    // # of points
  double t0 = 0.0, t1 = 10.0;        // time start and end
  double dt=(t1-t0)/(n-1);           // time step
  vector<double> tim(n-1);           // time interval start points
  {
  int i=0;
  for( auto & t: tim) {t=t0+i*dt; ++i;}  // t0,t0+dt,..,t1-dt
  }
  double y[1] = {-2.0};              // initial value; table with one entry

  gsl_odeiv2_system sys = {func, NULL, 1, NULL}; // not using a Jacobian, hence NULL
      pointer
  gsl_odeiv2_driver* driver =
    gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-13, 1e-13, 0.0);

  output(t0,y[0],y[0]);
  for (auto t: tim) {
    int status = gsl_odeiv2_driver_apply (driver, &t, t+dt, y);
    if (status != GSL_SUCCESS) {cout<<"FAILED near "<<t<<endl;return 1;}
    exact = -2.0*exp(-0.5*t*t);
    output(t,y[0],exact);
  }
  gsl_odeiv2_driver_free(driver);
  return 0;
}
```

Seuraava esimerkki on selvästi hankalampi. Se on poimittu suoraan GSL:n manuaalista. Ratkaistaan numeerisesti 2. kertaluvun epälineaarinen Van Der Pol oskillaattoriyhtälö,

$$x''(t) + \mu x'(t)(x(t)^2 - 1) + x(t) = 0$$

ja koska ohjelma ratkoo *1. kertaluvun* yhtälöitä, hajotetaan tämä kahdeksi kytketyksi 1. kertaluvun yhtälöksi määrittelemällä apumuuttuja y:

$$\begin{aligned} x'(t) &= y(t) \\ y'(t) &= -x(t) - \mu y(t)(x(t)^2 - 1) \end{aligned}$$

Ohjelma ratkaisee kaksi tuntematonta funktiota $\{x(t), y(t)\}$ piste pisteeltä alkaen annetuista alkuarvoista hetkellä $t = 0$. Kun yksi pistepari $\{x(t), y(t)\}$ tunnetaan, saadaan seuraava $\{x(t + dt), y(t + dt)\}$ käyttäen lähtöpistettä ja e.o. derivaattoja.

Käyn läpi ratkaisun, jossa on adaptiivinen askelpituus. Ohjelma koettaa päästä haluttuun absoluuttiseen tarkkuuteen lyhentämällä askeleen pituutta ratkaisufunktion tiukoissa mutkissa, joissa numeerinen ratkaisu helposti eksyy väärälle ratkaisukäyrälle.

Melkeinpä suurin työ on pitää kirjaa matemaattisten merkintöjen ja ohjelman merkintöjen vastaavuuksista. Asioiden sotkemisen helpottamiseksi GSL manuaali käyttää seuraavaa yleistä matemaattista merkintää:

$$\frac{dy_i(t)}{dt} = f_i(t, y_1(t), ..., , y_n(t)) \quad , \quad i = 1...n \ .$$

On siis kolme merkintätapaa, (A) oma matikka, (B) GSL matikka ja (C) GSL ohjelma. Tässä vastaavuustaulukko tälle ongelmalle:

A	B	C
$x(t)$	$y_1(t)$	`y[0]`
$y(t)$	$y_2(t)$	`y[1]`
$x'(t) = y(t)$	$\frac{dy_1(t)}{dt} = f_1(t, y_1(t), y_2(t))$	$f[0]$
$y'(t) = -x(t) - \mu y(t)(x(t)^2 - 1)$	$\frac{dy_2(t)}{dt} = f_2(t, y_1(t), y_2(t))$	$f[1]$
$\frac{\partial x'(t)}{\partial x(t)} = 0$	$\frac{\partial f_1(t, y_1(t), y_2(t))}{\partial y_1(t)}$	`m[0,0]`
$\frac{\partial x'(t)}{\partial y(t)} = 1$	$\frac{\partial f_1(t, y_1(t), y_2(t))}{\partial y_2(t)}$	`m[0,1]`
$\frac{\partial y'(t)}{\partial x(t)} = -1 - 2\mu xy$	$\frac{\partial f_2(t, y_1(t), y_2(t))}{\partial y_1(t)}$	`m[1,0]`
$\frac{\partial y'(t)}{\partial y(t)} = -\mu(x^2 - 1)$	$\frac{\partial f_2(t, y_1(t), y_2(t))}{\partial y_1(t)}$	`m[1,1]`
$x''(t)$	$\frac{df_1(t, y_1(t), y_2(t))}{dt}$	`dfdt[0]`
$y''(t)$	$\frac{df_2(t, y_1(t), y_2(t))}{dt}$	`dfdt[1]`

Funktio `func()` (taulukon neljä ensimmäistä riviä):
laskee derivaatat $\{x'(t), y'(t)\}$ hetkellä t kun funktion arvot $\{x(t), y(t)\}$ tunnetaan. Derivaatat pannaan taulukoon `f`, alkioihin $\{$`f[0]`, `f[1]`$\}$.

Funktio `jac()` (taulukon neljä viimeistä riviä): Jacobin matriisi. Tätä tietoa tarvitaan vain korkean kertaluvun menetelmissä. Yksi pikku mutka: funktion `jac()` pitää laskea 1-ulotteinen taulukko `dfdy`, jossa on Jacobin matriisi näin: `m[i,j] = dfdy[i + dimensio * j]`-siksi oudot `gsl_matrix_*`.

Esim. 42: (gsl_func_jac.h)

```
int func (double t, const double y[], double f[], void *params){
  double mu = *(double *)params;
  f[0] = y[1];
  f[1] = -y[0] - mu*y[1]*(y[0]*y[0] - 1);
  return GSL_SUCCESS;
}
int jac (double t, const double y[], double *dfdy, double dfdt[], void *params){
  double mu = *(double *)params;
  gsl_matrix_view dfdy_mat
    = gsl_matrix_view_array (dfdy, 2, 2);
  gsl_matrix * m = &dfdy_mat.matrix;
  gsl_matrix_set (m, 0, 0, 0.0);
  gsl_matrix_set (m, 0, 1, 1.0);
  gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
  gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
  dfdt[0] = 0.0;
  dfdt[1] = 0.0;
  return GSL_SUCCESS;
}
```

Uudessa GSL-kirjastossa on differentiaaliyhtälöiden ratkaisuun siistit **driver-funktiot** . Käytitpä sitten drivereita (otsikkotiedosto `gsl_odeiv2.h`) tai suoraan funktioita (`gsl_odeiv.h`), on tehtävä esivalmisteluita:

- Valitaan integrointialgoritmi, eli miten lasketaan seuraava piste $\{x(t+dt), y(t+dt)\}$. Valinnanvaraa on: rk2, rk4, rkck, rk8pd, rk2imp, rk4imp, bsimp, rk1imp, msadams ja msbdf. Tavallisesti hyviä ovat mm. "rkf45" ja "rk8pd".
- Valitaan kontrollikriteeri, eli milloin on otettava lyhempiä askeleita. Esimerkiksi absol. virhe 10^{-6}, suhteellinen virhe 0
- Kootaan differentiaaliyhtälöiden kaikki tieto yhteen (tietorakenteen nimi on tässä `sys`):

```
gsl_odeiv_system sys = {func, jac, 2, &mu} ;
```

  Argumentteina on tiedot funktioista ja niiden derivaatoista, Jacobin matriisifunktio, kertaluku (2 kytkettyä yhtälöä) ja parametrit. Esimerkissä on vain yksi parametri, mutta yleisesti ne annetaan osoittimena `void * params` , joka osoittaa C-kielen `struct`-tietorakenteeseen.

GSL:n uusi driver-funktio yhdistää edellä olleet näin[12]:

```
gsl_odeiv2_driver * d =
      gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);
```

ja sitä käytetään myöhemmin näin:

```
int status = gsl_odeiv2_driver_apply (d, &t, ti, y);
```

jonne menee systeemi ja ratkaisualgoritmi `d`, välin alkupiste `t`, välin loppupiste `ti` ja alkuarvot `y`. Hetken `ti` ratkaisu tulee ulos niinikään taulukossa `y`.

[12]Driver-funktioita on itse asiassa tarjolla neljä (http://www.gnu.org/software/gsl/manual/html_node/Driver.html).

Sitten vain asetetaan alkuarvot ja ratkaistaan DY välillä t=0...100.

Esim. 43: (gsl_ode_part.cpp)

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_odeiv2.h>
#include "gsl_func_jac.h"
int main (void){
  double mu = 10;
  gsl_odeiv2_system sys = {func, jac, 2, &mu};
  gsl_odeiv2_driver * d =  gsl_odeiv2_driver_alloc_y_new (
              &sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);
  int i;
  double t = 0.0, t1 = 100.0;
  double y[2] = { 1.0, 0.0 };
  for (i = 1; i <= 100; i++) {
    double ti = i * t1 / 100.0;
    int status = gsl_odeiv2_driver_apply (d, &t, ti, y);
    if (status != GSL_SUCCESS) {
      printf ("error, return value=%d\n", status);
      break;
    }
    printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
  }
  gsl_odeiv2_driver_free (d);
  return 0;
}
```

GSL: interpolointi

Kätketään GSL-kirjaston käyttö otsikkotiedostoon, ohjelma käyttää `std::vector`-luokkaa. Ohjelma tunnistaa kaksi spline-tyyppiä:

- "cspline" (*natural cubic spline*)
  muutokset yhteen pisteiseen heijastuu csplinessa laajalle
  → epästabiili
  "Luonnollinen" (*natural*) viittaa siihen, että päätepisteiden toiset derivaatat oletetaan nolliksi.

- "akima" tai "Akima" (*natural Akima spline*)
  yhden pisteen muutoksen vaikutus jää paikalliseksi
  → stabiili

GSL interpoloinnissa on neljä vaihetta:

1) Varataan tila "kiihdytykselle", joka nopeuttaa interpolointia
   `gsl_interp_accel * accel = gsl_interp_accel_alloc () ;`

2) Varataan tila halutun tyyppiselle interpoloinnille
   `gsl_spline spline = gsl_spline_alloc( gsl_interp_cspline , ...)`

3) Alustetaan interpolointi tunnetuille pisteille (x, y) (nyt esim. vector-kontteja tai Armadillo vektoreita):
   `gsl_spline_init(spline,&x[0],&y[0],x.size()) ;`

4) Lasketaan interpoloidut `y`:n arvot vector-konttiin `yy` (iteraattori `posy`) pisteissa `xx` (iteraattori `posx`)
   `*posy++ = gsl_spline_eval(spline, *posx, acc)`
   Tässä `posy++` kasvattaa iteraattoria yhdellä heti käytön jälkeen, eli siirrytään seuraavaan `yy`:n alkioon. Myös 1. ja 2. derivaatta saataisiin samalla tavalla:
   `... = gsl_spline_eval_deriv(...)`
   `... = gsl_spline_eval_deriv2(...)`

Esim. 44: (gsl_spline.hpp)

```
// wrapper to GSL spline
// spline type "cspline" is nat. cubic spline
// spline type "akima" or "Akima" is Akima spline
// C-style string comparison!
#ifndef GSL_SPLINE_HPP
#define GSL_SPLINE_HPP
#include <iostream>
#include <cstring>
#include <vector>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

using namespace std;

void spline(const vector<double>& x,const vector<double>& y,
            vector<double>& xx, vector<double>& yy,const char* type){
  gsl_interp_accel* acc  = gsl_interp_accel_alloc();
  gsl_spline* spline;
  int choose=0;
  if(strcmp(type,"akima")==0) choose=1;
  if(strcmp(type,"Akima")==0) choose=1;
  switch (choose){
  case 0:
    spline = gsl_spline_alloc(gsl_interp_cspline, x.size());
    break;
  case 1:
    spline = gsl_spline_alloc(gsl_interp_akima, x.size());
    break;
  }
  gsl_spline_init(spline, &x[0], &y[0], x.size());
```

```
  vector<double>::iterator posx,posy;
  posy = yy.begin();
  for(posx=xx.begin();posx!=xx.end();++posx){
    *posy++= gsl_spline_eval(spline, *posx, acc);
  }
  gsl_spline_free(spline);
  gsl_interp_accel_free(acc);
}
#endif
```

Esim. 45: (gsl_spline.cpp) Pääohjelma testaamaan spline-headeriä

```
// spline test
#include <iostream>
#include <iomanip>
#include <vector>
// home made call to GSL spline
#include "gsl_spline.hpp"
using namespace std;
typedef vector<double> vec;


int main(){
  vec x(10),y(10);
  // known points (x,y)
  double t;
  t = -5.0;
  for(unsigned i=0;i<x.size();++i){
    x[i] = t; y[i] = 10.0-t*t;
    t += 1.0;
  }
  y[4] += 10.0; // one point up to demonstrate Akima spline
```

```
  cout<<fixed<<setprecision(8);
  for(unsigned i=0;i<x.size();++i)
    cout<<x[i]<<" "<<y[i]<<endl;
  cout<<"\n\n\n";

  // points to interpolate xx
  vec xx(100),yy(100);
  double dx = (x[x.size()-1]-x[0])/(xx.size()-1);
  for(unsigned i=0;i<xx.size();++i){ xx[i] = x[0]+i*dx;}

  // interpolate using natural cubic spline
  spline(x,y,xx,yy,"cspline");
  for(unsigned i=0;i<xx.size();++i)
    cout<<xx[i]<<" "<<yy[i]<<endl;
  cout<<"\n\n\n";
  // interpolate using natural Akima spline
  spline(x,y,xx,yy,"Akima");
  for(unsigned i=0;i<xx.size();++i)
    cout<<xx[i]<<" "<<yy[i]<<endl;

  return 0;
}
```

GSL: Monte Carlo integrointi

Lasketaan likiarvo N-ulotteiselle integraalille yli hyperkuution,

$$\int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 ... \int_{a_N}^{b_N} dx_N \quad f(x_1, x_2 ... x_N) \ .$$

GSL sisältää kolme tapaa:

1) Plain - arvotaan satunnaispisteitä hyperkuution sisältä; karkein näistä kolmesta

2) MISER - (Press & Farrar) ns. stratified sampling-algoritmi, yrittää keskittyä alueisiin, joista tulokseen tulee suurin virhe

3) VEGAS - (Lepage) yrittää keskittyä alueisiin, jotka vaikuttavat tulokseen eniten

Ohjelma laskee integraalin

$$\frac{1}{\pi^3} \int_0^\pi dx \int_0^\pi dy \int_0^\pi dz \frac{1}{1 - \cos(x)\cos(y)\cos(z)} = 1.39320392... \ .$$

Ohjelman toiminta tarvitsee seuraavat tiedot:

- Satunnaislukugeneraattorin - algoritmi, jolla tuotetaan satunnaislukuja
- Funktion jota integroidaan - vaatii huolellisuutta:

```
 Data Type: gsl_monte_function
This data type defines a general function
with parameters for Monte Carlo integration.

double (* f) (double * x, size_t dim, void * params)
this function should return the value f(x,params)
for the argument x and parameters params,
where x is an array of size dim giving
the coordinates of the point where the function is to be evaluated.
size_t dim the number of dimensions for x.
void * params a pointer to the parameters of the function.
```

Esim. 46: (gsl_monte.carlo.cpp)

```
#include <iostream>
#include <iomanip>
#include <string>

#include <gsl/gsl_math.h>
#include <gsl/gsl_monte.h>
#include <gsl/gsl_monte_plain.h>
#include <gsl/gsl_monte_miser.h>
#include <gsl/gsl_monte_vegas.h>

using namespace std;

double func (double *x, size_t dim, void *params);
void display_results (string title, double result, double error);

int main ()
{
  const size_t dim=3;
  double result,error;
  string method;
  gsl_rng *r;
  gsl_monte_function G = { &func, dim, 0 };
  double a[dim] = { 0, 0, 0 };
  double b[dim] = { M_PI, M_PI, M_PI };
  size_t calls = 500000;


  // random number generator
  gsl_rng_env_setup ();
  r = gsl_rng_alloc (gsl_rng_default);
```

```
{
method="plain";
gsl_monte_plain_state *s = gsl_monte_plain_alloc (dim);
gsl_monte_plain_integrate (&G, a, b, dim, calls, r, s,
                           &result, &error);
gsl_monte_plain_free (s);
display_results (method , result, error);
}


{
method="MISER";
gsl_monte_miser_state *s = gsl_monte_miser_alloc (dim);
gsl_monte_miser_integrate (&G, a, b, dim, calls, r, s,
                           &result, &error);
gsl_monte_miser_free (s);
display_results (method , result, error);
}


{
method="VEGAS warmup";
gsl_monte_vegas_state *s = gsl_monte_vegas_alloc (dim);
// warmup
gsl_monte_vegas_integrate (&G, a, b, dim, 10000, r, s,
                           &result, &error);
display_results (method, result, error);
method="VEGAS";
```

```
  do
    {
      gsl_monte_vegas_integrate (&G, a, b, dim, calls/5, r, s,
                                 &result, &error);
      display_results (method, result, error);
      method = "VEGAS continue";
    }
  while (fabs(gsl_monte_vegas_chisq(s)-1.0) > 0.5);

  gsl_monte_vegas_free (s);
  }
  return 0;
}
double func (double *x, size_t dim, void *params)
{
  double A = 1.0 / (M_PI * M_PI * M_PI);
  return A/(1.0 - cos (x[0]) * cos (x[1]) * cos (x[2]));
}
void display_results (string title, double result, double error)
{
  const double exact = 1.393203929685676859184246260325 5;

  cout<<setiosflags(ios::fixed);
  cout<<setfill('=')<<setw(50)<<"="<<endl;
  cout<<"Method : "<<title<<endl;
  cout<<setfill('=')<<setw(50)<<"="<<endl;
  cout.precision(8);
  cout<<"result = "<<result<<" +/- "<<error<<endl;
  cout<<" exact = "<<exact<<endl;
  cout<<"|diff| = "<<fabs(result-exact)<<endl;
}
```

Tulostus

```
================================================
Method : plain
================================================
result = 1.41220870 +/- 0.01343586
 exact = 1.39320393
|diff| = 0.01900477
================================================
Method : MISER
================================================
result = 1.39132158 +/- 0.00346056
 exact = 1.39320393
|diff| = 0.00188235
================================================
Method : VEGAS warmup
================================================
result = 1.39267259 +/- 0.00341041
 exact = 1.39320393
|diff| = 0.00053134
================================================
Method : VEGAS
================================================
result = 1.39328139 +/- 0.00036248
 exact = 1.39320393
|diff| = 0.00007746
```

Physical problem: Spin-1/2 Heisenbergin ketju

Spin-1/2 Heisenberg chain

Spin-1/2 Heisenbergin ketjun Hamiltonin operaattori koostuu spin-spin vuorovaikutuksista kahden peräkkäisen spinin välilläs,

$$\begin{aligned}\hat{H} &= J\sum_{i=1}^{N}\mathbf{S}_i\cdot\mathbf{S}_{i+1} = J\sum_{i=1}^{N}[S_i^xS_{i+1}^x + S_i^yS_{i+1}^y + S_i^zS_{i+1}^z] \\ &= J\sum_{i=1}^{N}[\frac{1}{2}(S_i^+S_{i+1}^- + S_i^-S_{i+1}^+) + S_i^zS_{i+1}^z]\ .\end{aligned}$$

Tässä muutettiin spinin x-komponenttia mittaava operaattori S^x ja y-komponenttia mittaava operaattori S^y operaattoreiksi
$S^{\pm} = S^x \pm iS^y$, jotka ovat ns. nosto- ja laskuoperaattorit. Tämä muutos tehtiin, koska meidän pitää sopia mitä kantatiloja käytämme. Sovitaan että käytämme kantaa $|s,m\rangle$, joka on kokonaisspinin neliön S^2 ominaistila ($S^2|s,m\rangle = s(s+1)|s,m\rangle$) ja S^z:n ominaistila ($S^z|s,m\rangle = m|s,m\rangle$). Tässä $\hbar \equiv 1$.

Nyt nosto ja lasku on siis spinin z-komponentin nostoa ja laskua:

$$S^{\pm}|s,m\rangle = \sqrt{s(s+1)-m(m\pm 1)}|s,m\pm 1\rangle$$
$$S^{z}|s,m\rangle = m|s,m\rangle$$

Nyt spin on 1/2 ja saadaan

$$S^{+}|\frac{1}{2},\frac{1}{2}\rangle = 0; \quad S^{+}|\frac{1}{2},-\frac{1}{2}\rangle = |\frac{1}{2},\frac{1}{2}\rangle$$
$$S^{-}|\frac{1}{2},\frac{1}{2}\rangle = |\frac{1}{2},-\frac{1}{2}\rangle; \quad S^{-}|\frac{1}{2},-\frac{1}{2}\rangle = 0 \ .$$

Lyhyesti sanoen sovimme kvantitusakseliksi z-akselin ja kirjoitimme Hamiltonin operaattorin niin, että siitä saadaan matriisiesitys.

Kirjoitetaan tilat niin, että pidämme kirjaa spinin z-komponentista (m), systeemin tila on siis ketju spin-ylös tai spin-alas arvoja. Kaikkiaan on siis 2^N mahdollista spin-kombinaatiota. Koodataan tilat binäärimuotoon,

$$|\downarrow,\downarrow,...,\downarrow,\downarrow\rangle = |00...00\rangle = |0\rangle$$
$$|\downarrow,\downarrow,...,\downarrow,\uparrow\rangle = |00...01\rangle = |1\rangle$$
$$|\downarrow,\downarrow,...,\uparrow,\downarrow\rangle = |00...10\rangle = |2\rangle$$
$$|\downarrow,\downarrow,...,\uparrow,\uparrow\rangle = |00...11\rangle = |3\rangle$$
$$...$$
$$|\uparrow,\uparrow,...,\uparrow,\uparrow\rangle = |11...11\rangle = |2^N-1\rangle$$

Kirjoitetaan Hamiltonin operaattori näiden tilojen kannassa matriisina. Lasketaan matriisielementit tilojen a ja b avulla

$$H_{ab} \equiv \langle a|\hat{H}|b\rangle, \quad a,b = 0...2^N-1 \ .$$

Sitten laskemaan! Kaikki nollasta poikkeavat arvot ovat tilojen i ja $j = i + 1$ välisiä (muita ei ole Hamiltonissa). Haluamme periodiset reunaehdot, eli ketju on itse asiassa silmukka, joten päässä kun $i = N$ (ketjun "päässä") niin $j ='' i + 1'' = 1$ (ketjun "alussa"). Tämän voi tehdä modulo-operaationa, $j = mod(i + 1, N)$ tai C++-koodissa `j=(i+1)%N`.

Seuraavissa tapauksissa tulee nollasta poikkeava H_{ab}:

1) $S_i^z S_j^z$-termi: tilojen pitää olla täysin samat eli $a = b$.
Silloin $H_{ab} = H_{aa}$ ja saadaan neljä tapausta:

$$H_{aa} = \langle ...0_i...0_j...|S_i^z S_j^z|...0_i...0_j...\rangle = (-\frac{1}{2})(-\frac{1}{2}) = \frac{1}{4}$$
$$H_{aa} = \langle ...0_i...1_j...|S_i^z S_j^z|...0_i...1_j...\rangle = (-\frac{1}{2})(\frac{1}{2}) = -\frac{1}{4}$$
$$H_{aa} = \langle ...1_i...0_j...|S_i^z S_j^z|...1_i...0_j...\rangle = (\frac{1}{2})(-\frac{1}{2}) = -\frac{1}{4}$$
$$H_{aa} = \langle ...1_i...1_j...|S_i^z S_j^z|...1_i...1_j...\rangle = (-\frac{1}{2})(-\frac{1}{2}) = \frac{1}{4}$$

2) $\frac{1}{2}(S_i^+ S_j^-)$-termi: Tilasta b laitetaan j:s ykkönen nollaksi, sitten i:s nolla ykköseksi ja päädytään tilan a. Siis b on tila, jossa on a-tilan spinit i ja j käännetty ja j:s spin on ylös. Silloin

$$H_{ab} = \langle ...1_i...0_j...|\frac{1}{2}(S_i^+ S_j^-)|...0_i...1_j...\rangle = \frac{1}{2} \ .$$

3) $\frac{1}{2}(S_i^- S_j^+)$-termi: Tilasta b laitetaan j:s nolla ykköseksi, sitten i:s ykkönen nollaksi ja päädytään tilaan a. Siis b on tila, jossa on a-tilan spinit i ja j käännetty ja j:s spin on alas. Silloin

$$H_{ab} = \langle ...0_i...1_j...|\frac{1}{2}(S_i^- S_j^+)|...1_i...0_j...\rangle = \frac{1}{2} \ .$$

Algoritmi:
Käydään läpi kaikki tilat a. $H_{aa} = \frac{1}{4}$ jos $b = a$ ja tilan a bitit i ja j ovat samat (1. ja 4. tapaus kohdasta 1). $H_{aa} = -\frac{1}{4}$ jos $b = a$ ja bitit i ja j eivät ole samat (2. ja 3. tapaus, kohta 1). Lopuksi $H_{ab} = \frac{1}{2}$, jos b saadaan a:sta kääntämällä spinit i ja j.

Pseudokoodi: tilat bittiesityksessä
Koodi: A. Sandvik (Boston University).

```
H=0 // nollaa H
do a=0...2^N-1   //tilat a
  do i=0...N       // spin i
    j = (i+1)%N  // j=i+1 periodiset reunaehdot (i=N, niin j=1)
    if ( a[i]=a[j] )
      H(a,a)=1/4
    else
      H(a,a)=-1/4
      b=flip(a,i,j) // käännä a:n bitit i ja j, saat tilan b
      H(a,b)= 1/2
     end if
  end do
end do
```

STL:ssä on binäärilukujen tallennukseen kätevä kontti `bitset`. Alla `Matrix` on Boostin Hermiten matriisi,

```
typedef ublas::hermitian_matrix<cmplx, ublas::lower> Matrix;
```

ja Hamiltonin matriisin täyttämisen voi tehdä näin (vertaa edellisen sivun pseudokoodiin):

```
void FillHermitianMatrix(Matrix& H){
  std::bitset<std::numeric_limits<int>::digits> abit;
  std::bitset<std::numeric_limits<int>::digits> bbit;
  int i,j,a,b;
  int NN=H.size1();
  int N=log2(NN);
  H.clear(); // zero H (Boost matrix H)
  for(a=0; a<NN; ++a){
    abit = std::bitset<std::numeric_limits<int>::digits>(a);
    for (i=0; i<N; ++i){
      j = (i+1)%N; // periodic boundary conditions
      if (abit[i] == abit[j]){
        H(a,a) += 0.25 ;
      }
      else{
        H(a,a) -= 0.25;
        bbit = abit;
        bbit.flip(i);
        bbit.flip(j);
        b = static_cast<int>(bbit.to_ulong());
        H(a,b) = 0.5;
      }
    }
  }
}
```

Hamiltonin diagonalisointiin voi käyttää Lanczos-algoritmia (esim. IETL-kirjastosta).

C++: hyödyllisiä pikkuasioita

- Määrittele vakiot

```
const double Vakio=3.4443;
```

  vältä `#define Vakio 3.4443` määrittelyä, kääntäjä ei näe nimeä "Vakio": preprosessori hautaa sen ja tekee helposti useita kopioita samasta numerosta.

- Suosi

```
const
```

  määrittelyä: et pääse vahingossa muuttamaan suureen arvoa.

[12] Suosittelen kirjaa Scott Meyers: *"Effective C++"*.

Miten yhdistetään numeroita merkkijonoon, esimerkiksi parametreja tiedoston nimeen? Käytä `stringstream`-oliota. Nimensä mukaan se käsittelee string-oliota kuin se olisi stream.

Esim. 47: (stringstream_ex.cpp) Tehdään tiedostonimet `tulos_`, perässä on reaaliluku

```
#include <iostream>
#include <sstream>
#include <fstream>
#include <math.h>
using namespace std;
int main()
{
  stringstream s;
  const string str = "res";
  for (unsigned i=1;i<5;i++){
    s<<str<<"_"<<cos(i);
    ofstream out(s.str().c_str());
    s.str(""); // clears s
    out.close();
  }
  return 0;
}
/* opened files
res_0.540302
res_-0.416147
res_-0.989992
res_-0.653644
*/
```

Lambda-funktiot
(*lambda functions/expressions*)

Jos pysytte C++:n parissa kannattaa opetella lambda-funktiot tai -muodot. Ne ovat kivoja. Ja nopeita: lambda-funktio luo funktio-objektin (*function object*), joka kääntäjän on helppo sijoittaa inline-koodiksi.

Uudessa C++-standardissa C++11 [13], on määritelty **nimeämättömät funktiot**, jotka ovat ns. funktionaalisen ohjelmoinnin perusteita. Boost-kirjastossa on oma lambda-osionsa.

Käyttötarkoitus: usein on kätevä määritellä "pikafunktio" siinä missä sitä tarvitaan, eikä sille kannata keksiä nimeä.
Bonus: Ei tarvitse kirjoittaa luokkaa funktio-objektille - jota voisi vahingossa (väärin)käyttää toisaalla koodissa
Mutka: g++ -kääntäjälle pitää antaa optio `-std=c++0x`, eli

```
g++ -std=c++0x  code.cpp
```

Esim. 48: (autolambda.cpp) Tehdään pikafunktio `func`

```
//g++ -std=c++0x autolambda.cpp

#include <iostream>
using namespace std;

int main()
{
  auto func = [] () { cout << "Hello world\n"; };
  func();
}
```

[13] Vanha nimi oli c++0x eli "C plusplus nolla x"; virallinen version C++11 standardoitiin 12. elokuuta 2011.

Esim. 49: (lambda1.cpp) Tulostetaan kontin alkiot ja niiden kolmannet potenssit lambda-funktioon välittyy arvo `(int n)`, kontin alkiot eivät muutu.

```
// g++ -std=c++0x -Wextra lambda1.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;

int main()
{
   vector<int> v;
   v.push_back(11); v.push_back(22);v.push_back(33);
   for_each(v.begin(),v.end(), [](int n) {cout << n<<" "<<pow(n,3) <<endl;});
}
```

```
11 1331
22 10648
33 35937
```

Esim. 50: (lambda2.cpp)

Kompleksikonjugoinnissa lambda-funktioon välittyy viite
`(complx& c)`,
kontin alkiot `c` muuttuvat rivillä `c=conj(c);`

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <complex>
using namespace std;
int main()
{
  typedef complex<double> cmplx ;
  vector<cmplx> v;
  v.push_back(cmplx(11,12)); v.push_back(cmplx(22,21));v.push_back(cmplx(33,32));
  // print elements
  for_each(v.begin(),v.end(), [](cmplx c) {cout<<c<<" ";});
  cout<<endl;
  // take complex conjugate of elements
  for_each(v.begin(),v.end(), [](cmplx& c) {c = conj(c);});
  // print them
  for_each(v.begin(),v.end(), [](cmplx c) {cout<<c<<" ";});
  cout<<endl;
}
```

Tulostus:

```
(11,12) (22,21) (33,32)
(11,-12) (22,-21) (33,-32)
```

Esim. 51: (lambda3.cpp)
Yksinkertainen laskutoimitus
syöttää muuttujiin `x` ja `y` heti arvot 4 ja 5

```
#include <iostream>
int main()
{
   using namespace std;
   int n = [] (int x, int y) { return x + y; }(5, 4);
   cout << n << endl;
}
```

Tulostus:

```
9
```

Mitä on lambda-funktion hakasuluissa? Funktio voi **siepata** funktion itsensä ulkopuolella määriteltyjä suureita. Tässä lista mitä milloinkin siepataan:

```
[]        mitään ei siepata
[x, &y]   x:n arvo siepataan, y:n viite siepataan
[&]       kaikki käytetyt ulkoiset muuttujat siepataan viitteinä
[=]       kaikki käytetyt ulkoiset muuttujat siepataan arvoina
[&, x]    x siepataan arvona, muut viitteinä
[=, &x]   x siepataan viitteenä, muut arvoina
```

Esim. 52: (lambda4.cpp) Lasketaan tulo vector-kontin alkioista.

Siepataan muuttuja `prod` viitteenä lambda-funktioon

```
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
   using namespace std;
   vector<double> v;
   v.push_back(1.0);v.push_back(2.0);v.push_back(3.0);
   double prod=1.0;
   for_each(v.begin(),v.end(), [&prod] (double x) { prod *= x;});
   cout <<"product = "<< prod << endl;
}
```

Tulostus:

```
product = 6
```

Esim. 53: (lambda5.cpp)
Leikataan kaikki alle rajan (0.4) jäävät vector-kontin alkiot nolliksi.

```
//g++ -std=c++0x lambda5.cpp utility.cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include "utility.hpp"
using namespace std;
typedef vector<double> vec;

void rand_vector(vec& v){
  generate(v.begin(),v.end(), [](){return rand()/((double)RAND_MAX);});
}

int vector_chop(vec& v, const double lim){
  int count = 0;
  for_each(v.begin(),v.end(), [&count,lim](double& x){if(x<lim) {x=0.0; count++;}});
  return count;
}

int main()
{
   vec v(7);
   rand_vector(v);
   vector_out(v);
   const double limit=0.4;
   cout<<"chopped "<<vector_chop(v,limit)<<" values below "<<limit<<endl;
   vector_out(v);
}
```

Miten visualisoin datani?

Kymmeniä hyviä tapoja.

Gnuplot:

- Helppo, nopea ja rajallinen kyvyiltään
- ei liity mitenkään C++:aan
- `octave` käyttää kuvantekoon gnuplot:a

VTK:

- Hiukan mutkikas, mutta melkein rajaton
- C++ toteutus, voit kirjoittaa visualisoinnin mukaan koodiisi
- http://www.vtk.org

Matlab, Origin ... Kun olet saanut numerosi tiedostoon, voit valita minkä hyvänsä ohjelman.

Läksiäissanat

Jos jotain jäi mieleesi, niin toivottavasti nämä:

- Käytä vector-konttia, älä dynaamisia taulukoita - ellet ole hyvin tottunut `new`, `delete` -käyttäjä
- Käytä Boostia, Armadilloa tai IETL:a, varsinkin matriiseille
- Käytä algoritmeja, älä silmukoita
- Komentoriviargumentit: niele C++ ylpeytesi ja käytä C:n muunnosfunktioita `atoi()`, `atof()` ...
- Älä muuntele taulukoita konteiksi ja päinvastoin. Käytä konttia ja kutsu C-funktioita lähettämällä osoitin `&kontti[0]` tai `&kontti.begin()`. Jos tämä näyttää rumalta, määrittele osoitin `*pkontti=&kontti[0]` ja lähetä `pkontti` C-funktioon.
- Jos osaat, käytä nopeutta vaativissa töissä funktio-objekteja, älä funktioita. Tai: käytä lambda-funktiota.
- Kirjoita ohjelma yleisellä tasolla niin, ettei se oleta tietyn kirjaston käyttöä. Hautaa kirjastofunktion käyttö otsikkotiedostoon.
  Periaate: ohjelmakoodeissa `*.cpp` lukee

```
fft(data,forward); // siisti
```

  `fft.hpp`-tiedostossa lukee

```
gsl_fft_complex_radix2_forward(data,1,n); // ruma
```