FYSA120

C++ Numerical Programming

Vesa Apaja

November 23, 2015

Contents

1 Course itinerary

- 8 Lectures
  - lecturer Vesa Apaja (YN212, Nanoscience Center 2nd floor)
- 4 Demos and 2 groups: Group 1 starts on Nov. 5, Group 2 starts on Nov. 3.
  - held in Physics Dep. Computer class **FL349**
  - Demos are supervised by **Sampsa Kiiskinen**
- warm-up demo 1 solved during exercise
- Requirements: At least half of the exercises of the demos 2,3,4,5, and 6, plus one programming task - **No exam.**
- You may use
  - a machine of your choice; you need a fairly recent C++ compiler, GSL, and armadillo libraries.
  - the Physics Department server `calc.phys.jyu.fi`. Log in using your JYUNET login name and password. In calc window, type

  ```
  module add gcc
  ```

  This loads a recent version of gcc along with the C++ compiler g++. The system compiler is outdated, thanks to the concervative ("dino-age is safer") package policy of RHEL.

1.1 A Brief History of C++

- 70's and 80's:
  - FORTRAN-77 annoyed users with its fixed-size tables
  - Pascal became popular, Borland introduced a fast compiler
  - Dennis Ritchie creates C language with dynamic tables

- From 1979 on: Bjarne Stroustrup creates C++ as a sort of "improved C" – but is it? The jury is still out. (The name "C++ was coined in 1983)

- Characteristic to C++:
  - strong typing of variables (supposedly) reduces programming mistakes
  - Object-oriented programming possible, though not forced
  data belonging together is collected together to an "object", that can be moved around as one big chunk.
  - Compiled language, for speed.

- Early 90's: No standard, compiler manufacturers make their own decisions

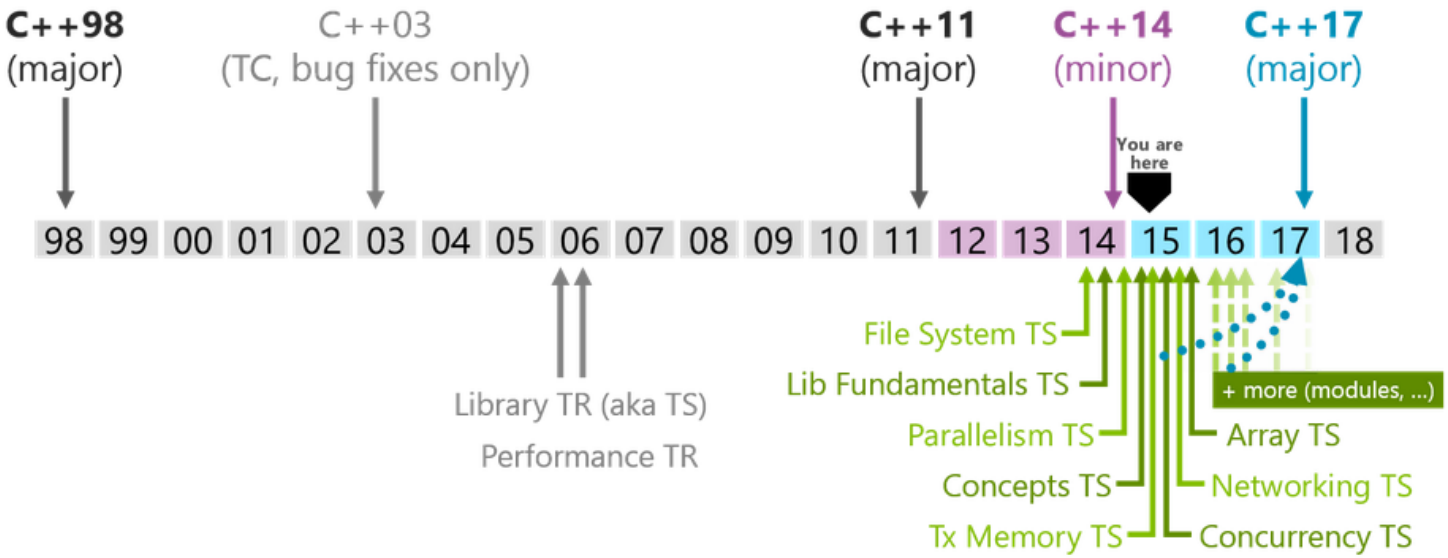
- 1998 : Horray! - C++ Standard!
  -An extensive standard library

- 2011 : C++11 Standard (working name was C++0x)
  - Easier ways to do very common tasks
  - C++ manuals written before about 2010 are better left to collect dust

C++ is a multi-purpose language and was never intended to specialize on numerics. You may also come across the fact that old languages have the burden of legacy code. Backwards compatibility and committee decision making tends to make old languages contain everything but the kitchen sink. Maybe we, the numerical people, would actually need that sink!

1.2 Future of C++

This timeline is given by the ISO C++ committee-
TS stands for "Technical Specification"; TR for "Technical Report":



1.3 How transferable is C++?

We'd better first settle what is meant with transferability. Do we transfer source code or binary? Java is well known for its tranferability, thanks to a clever combination of a programming language and the Java Virtual Machine JVM. C++ is "just" a programming language. The questions one should ask is how much work do you have to do to make a program 10 years from now. It's fairly common that people in science write the same task already a third time. Example: The first version was in fortran, then in C++, and now in Python, planning to write it in Haskell. Getting any faster? No, but the structure is nice and the user interface is cute! More modular and easier to expand? You hope! Why does one reprogram? It has to do with sociology, some programming languages are more fashionable.

1.4 About this document

The author is not a C++ guru – and never will become one :/

I see programming languages as tools that help me understand Nature, just like mathematics. Programming and mathematics deserve to be disciplines for their own sake, but I have no desire of becoming a car manufacturer or repairman, I just want to drive.

I spend some time in introducing you some C++11 features, it's a whole lot different to what old C++ used to be. As s general goal, I try to avoid pointers (at least naked pointers) and use references. C does not use references, so I may talk about pointers as well. Just because you may one day find yourselves on the drivers seat of a car with manual transmission. GSL is written in C, so that is at least one point where pointers are needed. Usually I put my faith in libraries, acknowledging they are also written by humans.

Specifically, I hope this course

(i) gives you self-confidence to write your own C++ program that solves a numerical task using a chosen library

(ii) helps you read C++ programs and sometimes even understand how they function
– on a good day even C programs may look familiar

It's easier to make a working code faster than to make a fast code to work.

A code, optimized for speed, can be a very unfriendly beast to debug. Most codes have portions that need not to be fast at all.

The English in these lecture notes is not perfect, but I remind you how Yoda, the Lucasfilm trademark character, the master of The Force, after over 874 years says "Your father he is, but defeat him you must".
[1]

[1]This is BTW called OSV or Object-Subject-Verb word order. C++ you learn!

1.5 Why C++, why not Java, fortran, Lisp, Haskell, ... French ?

Depends on what you want to do. Maybe French is the best language if you want to read old mathematical manuscripts or write poems. The learning curve in C++ is steep, and it was designed to do all but numerics! Fortran was designed for numerics, making it more compact. We would probably never have heard about Linus Torwalds had he written the linux operating system in fortran. Relax, if you know C++ you can more easily learn Java and fortran (I'm not talking about the ancient FORTRAN 77, phew, but fortran 2008 (fortran 2015 coming)). They all have similar structures like classes. This excludes functional languages, such as Haskell, because their way of thinking is quite different from the imperative languages C or C++. To mention a few differences, Java has no operator overloading[2], appreciated by numerical programmers. Speed is crucial, C++ programs can challenge and sometimes beat fortran. I suspect that a lousy fortran programmer can write faster code than a lousy C++ programmer. Speed comparisions age quickly, but you get hints of the situation [3].

> If you plan to run the program a few times, pay attention to time spent on programming.
> If you plan to run the program hundreds or millions of times, pay attention to time spent by the program.

Numerics is the latter. It's number crunching, squeezing the last drops of the computer juice.

[2] `http://javarevisited.blogspot.fi/2011/08/why-java-does-not-support-operator.html`

[3] 2009: `http://engineering.linkedin.com/49/linkedin-coding-competitions-graph-coloring-haskell-c-and-java`

1.6 Easy tasks

Yilun Zhang, Learning data science
70 upvotes by Chiraz Ben A, Gunnar Oehlschlägel, Jake Merchant, (more)

Hello world in:

Python:

```
print "Hello, world!"
```

Java:

```
public class HelloWorld {
    public static void main(String[] args) {
System.out.println("Hello, world!");    }
}
```

C++:

```
#include <iostream>
 int main(){    std::cout << "Hello, world!\n";
}
```

C:

```
#include <stdio.h>
int main(void){
printf("Hello, world!\n");
}
```

JS:

```
//myfile.js
console.log("Hello, World!");

***command line***
node myfile.js
```

MySQL:

```
DELIMITER $$CREATE FUNCTION hello_world() RETURNS TEXT COMMENT
DELIMITER ;
 SELECT hello_world();
```

Python + Numpy + Scipy is often an unbeatable combination

Eigenvalues and eigenvectors of a random 5x5 matrix:

```
import numpy as np
import scipy.linalg as la
A = np.random.randint(0, 10, 25).reshape(5, 5)
print("matrix A = \n",A)
e_vals, e_vecs = la.eig(A)
print ("eigenvalues = ",e_vals)
```

how-can-scipy-be-fast-if-it-is-written-in-an-interpreted-language-like-python

1.7 C++ in the Net

- www.ohjelmointiputka.net/oppaat.php (in Finnish)
  ("mureakuha" is out of action, competition killed it)
- www.cplusplus.com
- www.greenteapress.com/thinkcpp/ (Book, 2012)

Don't read old books; Anything written before 2011 should be taken with a grain of salt.
It's futile to force any "good" outlook for C++ programs, just **be systematic with your style**. When to use capital letters and when to use underscores is up to you. Think carefully what to put in *header files*, I'll speak up my opinion later. About style and good C++ practices, I recommend Bjarne Stroustrup's home page www.stroustrup.com. After all, he *created* the language.

1.8 C++ in Matlab or Octave

You can call a compiled C or C++ programs from Matlab using MEX files (Matlab executable files). Matlab can access C/C++ library routines and may work through a numerical bottleneck faster. See instructions in write-cc-mex-files.html.

Octave is a free Matlab-lookalike, with an almost identical syntax but different internals. Octave has interface to several languages, including C++. The compiled files used in Octave are called oct-files, see instruction in oct-files.

1.9 C or C++ in Python 3

Extending Python 3 with C++
Cython
ctypes

2 (A really) Brief introduction to C++

A list of some basic elements is in ***C++-sheet.pdf***.

Example 1: (basic/main1.cpp) Main program

```
#include <iostream>        // here cout is defined
using namespace std ;      // all names (like cout) come from this list
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
}                          // ALL ENDS
```

Story:

`include <iostream>`	Take `Standard Input / Output Streams Library`
	`Header` that defines the standard input/output stream objects
`using namespace std;`	unless told otherwise, the keyword is defined in `std` namespace
`main()`	parenthesis indicate, that the main program is a function (and can have arguments)
`int main()`	it returns an integer

The function `main()` can end with

`return 0;`

or

`return (0);`

so that the function `main()` gives out the integer 0. out where? Hmm, to the operating system. Safe to leave out entirely.

2.1 Meaning of `#include <iostream>`

The line `#include <iostream>` tells the compiler, that the code has something defined in the header `iostream`, which is part of the C++ standard library. [4].

Two ways to include headers:

```
#include<iostream>  // part of standard library
#include "myheader.h" // a home-made header
```

The difference is the header search path, the latter uses the current path first, after that the include path. To see what headers were included, try g++ -M code.cpp and to include a path g++ -Imypath code.cpp .

You get pretty far with these headers:

- `#include <iostream>`: input and output
  `cout, cin, cerr, clog` - here `c` means "console", which means "screen".
- `#include <iomanip>`: formatted output; if you like `printf()` forget this.
- `#include <cmath>` (C++) or `#include math.h` (C or C++) : mathematical functions
  `sin, cos, log, sqrt, pow`... x^2 is coded `pow(x,2)`
- `#include <complex>`: complex numbers
- `#include <fstream>`: file handling (also `ofstream, ifstream,...`)
- `#include <string>` : character strings
- `#include <random>` : (pseudo)random numbers
- `#include <functional>` : function objects

[4] Take a peek at what headers exist and what's in them : www.cplusplus.com/reference

Function styles

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

tai

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

if it's short,

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

I'll be using `double` precision real numbers a lot, as they are accurate enough for most needs.
Here the function `f` returns a number of type `double`, the argument `x` is also a `double`.

Example 2: (basic/fun_before.cpp) Function defined before `main()`

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

```
f(2.333)=0.482627
```

The compiler knows all it needs to know about the function `f()` when it's met in `main()`.

Example 3: (basic/fun_after.cpp) Function defined after `main()`

```
#include <iostream>
#include <cmath>
using namespace std ;
double f(double);
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)*cos(x*x);
}
```

When the compiler meets `f()` inside `main()`, it wants to know what it takes as arguments and what it returns. Therefore you need to provide it with a sneak peak

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

called **prototype** or **function declaration**; You have to give the prototype also if the function is defined in ***another file***. Failing to declare a function results the error message `fun_after.cpp:9: error: 'f' was not declared in this scope`. Here we encounter the concept of ***scope***.

C++11 standard made an addition, the return type of a function can be declared like this:

```
auto f(double) -> double; // trailing return type
```

Pretty useless in this example. If you, however, think of how the return types of class methods become clear only after going through all the types of the arguments, you may appreciate how this helps the compiler. There is another use of `->`: `foo->bar` means `(*foo).bar`.

2.2 Scope

The scope tells the range of visibility. Which parts of the program is an object defined, and also who may know the thing even exists[5]. The basic ideas are global objects, visible everywhere, and local objects, visible in a limited scope. The scope limits are curly brackets, boundaries of a ***block***:

```
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
{
```

A local loop variable exist only in the loop

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

When the program leaves the loop, the variable `i` ceases to exist. The last line is not allowed.

[5]Sounds modern, invented in the 50's.

2.3 Simple file operations

C++ idea: create a *stream*, either for writing (`ofstream`) or for reading (`ifstream`)

`o` as out, `i` as in and `f` as file.

To a stream we stuff things using the operator << and read with the operator >>.
The console output `cout` console input `cin` work like streams.

Example 4: (basic/fileio.cpp) write to a file

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

This declared and opened a stream in the one line with `ofstream`

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

The file `output` contains now the text "Text to file output" and

```
myfile.close(); // myfile's close; there may be many other close() functions
```

closes the file. The object `myfile` of the class `ofstream` has a method (member function) `close`.

3 Class

Say you want an object to be the state of the system. The state holds as variables the number of microstates N, their occupations $n_1, n_2, ..., n_N$, and the energy of the state E (and things you didn't come to think of at this stage). In addition, the value of E should be set only by the method `comp_energy()`. A collection like this can be stored as a ***class)***. A class is a collection of data and methods to manipulate the data.

A class is just an abstract type, a bit like `int` only tells what an integer is. You need an ***object***, a manifestation of the class. A class `Bird` may have an object `crow`, which tells the compiler that "a crow is a bird".

The method `comp_energy()` is conveniently a ***class method, a member function***.

Rough outlook:

```
class ClassName{
   // here private things
public:
   // here public things
   // and member functions, used to manipulate and show contents of private date
};  // this line may already contain objects of this class
```

One possible class describing the state of the system (just a draft, improve it later)

```
class SystemState{
  double E;
public:
  void comp_energy(); // compute energy
  double energy()const {return E;};    // read energy
} state1,state2;
```

This defines the class and immediately constructs two objects `state1` and `state2`. You can do this later as well, `SystemState state1;`

4 Member function with `const` or `noexcept` after it

C++11 has several "compiler directives", that facilitate code optimization.

```
Class MyClass {
  ::
public:
  void member1() const; // does not change any data member of the class (except
     mutable)
  void member2() noexcept; // does not throw any exceptions
  void memder3() const noexcept ; // does neither throw nor change any data member
     (unless mutable)
  :::
}
```

`throw` refers to a mechanism used to make exceptions, errors or warnings; More about this in chapter 21 for "throw-catch". `noexcept` can potentially lead to faster code, see chapter **??** In short, if a function cannot fail use `noexcept`. In numerics you seldom want failing functions.

5 Code rotting and `[[deprecated]]` functions

You want to replace an old function with a better one, but don't want to upset your boss by making his legacy code fail to compile. Use C++14 `deprecated`:

```
// C++14
[[deprecated]] void foo(int);
```

You can add a message, too: `[[deprecated(''Sorry Boss, newfoo replaces foo'')]]`
A call to `foo` causes g++ to emit
`warning: 'void foo(int)' is deprecated: Sorry Boss, newfoo replaces foo [-Wdeprecated-declarations]`
and a C++11 compiler emits the do-not-quite-understand message
`warning: 'deprecated' attribute directive ignored [-Wattributes]`

To work, the class `SystemState` still needs the definition of the method `comp_energy()`. If it's short, it can be defined *in situ*, on the spot in side the class (as we did for `energy()`), but you are free to define it later:

```
// member function of class SystemState
void SystemState::comp_energy(){
  E=3.12515e-10; // just anything for testing
}
```

The variable `E` used byt the method is exactly the same `E` as in the object, not just any `double E` ! Now the energy of the state can be computed and read,

```
int main()
{
  state1.comp_energy();         // compute state1's energy
  cout<<state1.energy()<<endl; // print state1's energy
  return 0;
}
```

This introduced one benefit of a class, ***data encapsulation***, which means that data can be protected from inadvertent changes or hackers. Just make it `private` and give only carefully chosen methods the permit to touch the data. Secondly, upon improving the code you can improve the methods without disrupting the workings of the rest of the code.

A class has properties that can be ***inherited***, meaning another class can inherit the properties and methods of a parent class. Smart base classes can be inherited and you can recycle code. In the C++ language itself a lot of things are inherited, for example `ofstream` header file inherits this chain of mre general purpose headers:

`ios_base` ← `ios` ← `ostream` ← `ofstream`

On the bottom there is a general-purpose class `ios_base`.

The next example – sorry that this has nothing to do with numerics – makes a class `Car` and an object `cadillac`.

Example 5: (basic/car.cpp) Car class

```
// How to create a Car class with member function get_color()
#include <iostream>
using namespace std;
class Car
{
  string color;  // private member
public:
  string get_color() const {return color;};
  Car(string col) {color=col;}     // constructor
  ~Car(){cout<<"Car demolished\n";}  // destructor
};

int main()
{
  Car cadillac("black_and_white");  // create a Car with a color
  Car ferrari("red");
  cout<<"cadillac is ";
  cout<<cadillac.get_color()<<endl; // get cadillac's color
  cout<<"ferrari is ";
  cout<<ferrari.get_color()<<endl;
  Car buick(cadillac);              // copy cadillac to buick: all properties
  cout<<"buick is ";
  cout<<buick.get_color()<<endl;
}
```

```
cadillac is black_and_white
ferrari is red
buick is black_and_white
Car demolished
Car demolished
Car demolished
```

`color` on luokan `Car` jäsen; se ei ole `public`, joten se on `private`. `get_color()` on *ainoa toimiva* menetelmä lukea auton väri

```
  cout<<buick.color<<endl;
```

tuottaa kääntäjän virheilmoituksen

```
car.cpp:5:10:  error:  'std::string Car::color' is private
```

Notice, how the class `Car` has a function with the very same name as the class itself:

```
Car()
```

is a class constructor, used to create an object (a Car) and to give it some properties. This constructor is used if and only if the color is given in parenthesis. The compiler looks how we try to create a Car and find a match from the list of possible ways to constructor a Car.

Object is a manifestation of a class, a thing that has all the properties defined in the class:

```
Car lada("black"); // Car is a class, lada is an object
```

Meaning "lada is a Car and it's black". When the compiler comes to this line, it fetches a constructor in the class `Car` answering the needs, a function that looks like this:

```
 Car(string)
```

Such a function exists, so the compiler will create a black lada. It may become as a surprise, that constructor of a colourless Car fails:

```
Car volvo;
```

fails, because `Car` has no constructor

```
Car()
```

By the way, this would be the so-called ***default constructor***, one made by the compiler if no other constructor is specified – after all, a class must have *some* constructor to be useful at all. The only thing a default constructor does, is to reserve some space for an object, nothing else. C++ language specifies that

> If you give a constructor for a class, the default constructor vanishes

This is exactly what happened in the example; a colourless Car construction was not there.

Objects can be copied,

```
Car buick(cadillac);
```

and `buick` picks the colour from the `cadillac`.
Wait a moment, what function did the copying? It was the ***copy constructor***, provided by the compiler automatically.

As you guessed, there is a method called ***destructor***, which deletes an object. It has the strange tilde sign,

```
~Car(){cout<<"Car demolished\n";}
```

Compilers create this automatically, then it's called a ***default destructor***. You can also provide one yourself, and include a clear message when an object is destroyed.

To summarize, a class has a set of building methods:
(add the prefix "default" if you let the compiler create a method)

constructor	how an object is created, when or if created
copy constructor	how an object is copied (resulting two similar objects with different name)
copy assignment	(=) makes a copy from right to left
move constructor	tells how the information of an object is transfered to another the object essentially pilfers (steals) the resources of the other object, leaving it in default state
move assignment	pilfer the resources on the right and gives them to the left
destructor	tells how an object should be destroyed

C++11 has a ***rule of five***, saying that if you **need** to write your own destructor, chances are your class is so wierd that you need to provide all five:

destructor
copy constructor
move constructor
copy assignment operator
move assignment operator

because none of the default version generated by the compiler meet the needs of your class.

[5]The example used a clumsy constructor, a better way would have been to use an ***initialization list***.

5.1 Delegating a Constructor *(read on spare time)*

C++11 introduced a way to ***delegate a constructor***. The idea is to let, for example, a constructor without any arguments to delegate the construction to another constructor with with a single argument by telling what a default argument is. The points is to avoid copying the same piece of code over and over again in all constructors.

Here is a bit artificial example, this doesn't save much code:

```
// C++98
class Sum
{
public:
   Sum(): a1(0), a2(0){s=a1+a2;}
   Sum(int i):a1(i),a1(0){s=a1+a2;}
   Sum(int i,int j):a1(i),a2(j){s=a1+a2;}
private:
   int a1,a2,s;
};
```

```
// C++11
class Sum
{
public:
   Sum(): Sum(0){}
   Sum(int i): Sum(i,0){}
   Sum(int i,int j):a1(i),a2(j){s=a1+a2;}
private:
   int a1,a2,s;
};
```

In the C++11 code the object `mysum`, to be created without an argument, `Sum mysum()`, is delegated to `Sum(0)`, who delegates it to `Sum(0,0)`.

This example used the ***initialization list***:

```
Sum(int i,int j):a1(i),a2(j){...}
```

which means that the value of `i` is given to `a1` and the value of `j` is given to `a2`.

5.2 Own data structures: pairs of numbers

Warning The example below is BAD, no need to re-invent the wheel.

Example 6: (numerics/class_pairs_of_numbers.cpp) A class for pairs of numbers

```
#include <iostream>
#include <cmath>
using namespace std;
// Class for pairs of numbers
class TwoDouble{
public:
  double x,y;
};
double distance(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="<<distance(point1,point2)<<endl;
  return 0;
}
// distance between two points (x,y)
double distance(const TwoDouble  & c1, const TwoDouble & c2) {
  return ( sqrt(pow(c1.x-c2.x,2) + pow(c1.y-c2.y,2) ) );
}
```

Notice how the name chosen for the function, `distance()`, is conveniently descriptive, but potentially dangerous! The namespace `std` is used in it's entirety, and who knows if there already is a function called `distance()` doing a completely wrong thing! Actually, C++98 does have `std::distance()` for iterators. Please check out the example `class_pairs_of_numbers2.cpp` to see how to use a common name like `distance()` safely. We'll come to name conflicts later in this course.

The previous example about pairs of numbers obviously works, but it is a **very common structure**. And common structure are found in ***C++ Standard Library)***. If you ever need a pair of – well, practically anything – use the standard structure `std::pair<type1,type2>`.

Example 7: (numerics/pairs_of_anything.cpp) `std::pair` and `std::make_pair`

```
// using std::pair to make pairs of almost anything
#include<iostream>
#include<vector>
int main()
{
  std::pair<int,int> iipair;
  iipair= std::make_pair(10,12);
  std::cout<<iipair.first<<" "<<iipair.second<<std::endl;

  std::pair<int,double> idpair(1,120.324);
  std::cout<<idpair.first<<" "<<idpair.second<<std::endl;

  // vector of pairs (int,vector)
  std::vector< std::pair<int,std::vector<double>> > ivpairs;
  std::vector<double> v{10.1,20.2,30.3,40.4}; // testing vector
  for(int i=0; i<10;++i)  ivpairs.push_back(std::make_pair(i,v));
  for(auto i:ivpairs) {
    std::cout<<"i="<<i.first<<" vector=";
    for(auto j:i.second) std::cout<<j<<" ";
    std::cout<<std::endl;
  }
}
```

`std::tuple` is a generalization of `std::pair` to more than two things of any type.

6 Templates - generic instructions and algorithms

A template is a generic model how things fed to the model should be handled. Templates are advertised as one of the best things C++ has to offer – as always some disagree. You'll benefit from them even though you have no idea how to write ones yourself, because ***most C++ libraries use templates to achieve excellent performance and multi-purpose applicability***.

> A template is an abstract model of what to do

`std::sort` is a template to sort arguments – many kinds of arguments, not just numbers
`std::pair` is a template to combine two things – many kinds of things
`std::swap` is in a template to swap two things – many kinds of things

For example `std::vector<int> x` tells the compiler "find a model for a `vector` in the `std` library and implement – "instantiate" – it using `int` as data type and call it `x`". The angular bracket is for ***template arguments***.

Basic templates are simple. Suppose you have a function that is written for `double`'s,

```
void f(double x)
{
  cout<<"argument is "<<x<<"\n";
  // do something more
}
```

but you know it should work the same way also for `int`'s and `string`'s, then options are to overload `f` (more on overloading functions in chapter 10) and manually write the three functions `double f(double x)`, `int f(int x)` and `string f(string x)` – much typing – or make a template:

```
template <typename T>
void f(T x)
{
  cout<<"argument is "<<x<<"\n";
  // do something more
}
```

Even this simple template is not omnipotent, there are lots of type `T` objects that cannot be printed by `std::cout`, and those give a long compiler error message.

For a more serious example, see `basic/simple_template.cpp` and `basic/static_assert.cpp`. The latter shows how to check types in compile time, to catch attempts to use a wrong template by mistake.

Skip the rest, unless you are familiar with templates or curious.
Take a peek at the series of articles An-Idiots-Guide-to-Cplusplus-Templates.

Still curious? See what strange things can be done at Type-Rich-Style-for-Cplusplus. The original source is Bjarne Stroustrup's Youtube lecture. It tells the story about C++11 code like this:

```
int main(){
        Speed spd = 1m/1.2s;
        Acceleration acc = 8m/3.3s2;
}
```

Own types `Speed` and `Acceleration` are defined simply with `typedef` or `using` (see 9.4). This we know. But the units! Stroustrup claims, that the template implementation of the unit system is so clever, that the compiler can do type checking and find programming mistakes that show as unit mismatch. Moreover, the compiled code has only the `double`'s but no units around after compilation. No speed penalty! Anyone who, after lengthy calculations on paper, has got the final result $energy = 234.4\ \mathrm{kg}^6/\mathrm{m}$ values greatly automated unit checking.

7 C++ libraries

7.1 C++ Template libraries – just a list

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

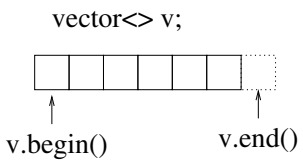
7.2 C++ Standard Library

C++ Standard Library comes with all C++ compiler suites. What, then is the Standard Template Library (STL) many people talk about? I rise my hands here. I've figured out this much: The STL existed before the Standard Library, and parts of it seem to be now in the Standard Library. People talk about STL and mean the Standard Library and also talk about the Standard Library when they mean STL. As an ***end user*** of the C++ products, I couldn't care less, and have taken the pragmatic approach to accept it's a library that comes with the compiler suite. I call thme both as *the C++ Standard Library*. Period. On late hours I simply think that

> STL means Standard Template Library (official)
> STL means STandard Library (inofficial, my own idea to save my boiling brain)

The Standard Library contains also the C Standard Library, so you are able to use some C features as well. But not all of C, notably C++ programmers avoid naked pointers to the extreme, and they are never really *needed* in C++. Some parts are templates (multi-purpose models), such as

- ***containers*** are collections of objects
  `vector, stack, deque, list, map, ...`
- ***iterators*** are for going through elements of containers
  `begin(), end()`



> `v.begin()` points to the begin of the container `v`
> `v.end()` points to *one step past* the end of the container `v`.

Why one step past the end? It was chosen like that because of cleaner loops:

```
for(auto it=v.begin();it<v.end()) {...}
```

- ***algorithms*** are frequently needed ways to process data
  `sort, find, min_element, max_element, reverse ...`

Example 8: (basic/std_swap.cpp) Swap two numbers using `std::swap`

```
#include <iostream>
#include <utility>
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
```

Someone familiar with C might be concerned about what values `a` and `b` have after the call to `swap`. How are the arguments passed to the function and what comes back?

8 C++ reference variables

Three ways to pass data to functions:

- ***Pass by value***

```
void dothis(int);  // function prototype
...
c=15;
dothis(c); // work with number 15, not with c; cannot change contents of c
```

- ***(C++) Pass by reference***

```
void dothat(int &);  //  function prototype
...
c=15;
dothat(c); // work with c, can change contents of c
```

- ***Pass a pointer***

```
void doodd(int *);  // function prototype
...
c=15;
doodd(&c);  // works with the pointer to c; can change c at will
```

Importrant: by looking at just the function call, there is no way to tell if the argument is passed-by-value or passed-by-reference! Only the function prototype (declaration) reveals which it is. The problem is, that the prototype can be on line 3467 in the header file myheader_for_this_job.hpp. In practise you end up guessing whether a function can change the contents of variable `c`: `dothis(c)` doesn't, but `dothat(c)` does!

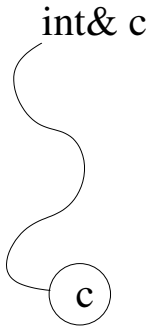
8.1 Why would a reference be safer than a pointer?

Imagine that the memory of a computer is a stack of drawers.
A pointer `int *c` means roughty "the integer c is in the upmost drawer" – or actually just "upmost drawer, take c from there"!. Such pointers may be doing funny mistakes:

- The upmost drawer may have a wrong number or not an integer but a mouse trap (real number)
- The upmost drawer can be bottomless (contain a pointer to the drawer below)
- The freaking drawer doesn't even exist (pointer to outside computer memory space ?)

A programmer is let to do all this without the compiler ever noticing any bad deeds. Running the program gives odd results. I once wrote a program that was supposed to do a simple calculation - result was something like "Full moon tonite".

C++ reference to variable `c`, `int& c`, is like a cord or a thin rope tied to the integer `c`. There is no place for mistake, the end of the cord always has `c`, because there is no way to detach the cord and tie it to anything else. A C++ reference cannot be detached from its variable.



Amuse yourself with these almost relevant examples:

Pass by value (C or C++):
Write the numbers 5 and 10 on a piece of paper, show it to you college and tell him/her to copy the numbers to *his/her own piece of paper in reverse order*. Swap failed: your paper still holds 5 and 10, not 10 and 5

Pass by reference (C++):
Write the numbers 5 and 10 on a piece of paper, hand it to you college *without letting go of the paper*. Tell him/her to swap the numbers on your paper, then pull the piece of paper back. Swap achieved! Robustness: Your college immediately sees if your hand is empty or there are no numbers on the paper.

Pass by pointer (C tai C++):
Write the numbers 5 and 10 on a piece of paper and tell your college the paper is in a specific drawer. Tell him/her to swap the numbers written on the paper and to put it back to the drawer, Swap achieved! Robustness: your college picks a wrong piece of paper from the drawer and wonders why you want to swap two telephone numbers.

Remarks:

1) If you have many colleges and/or 10000 numbers, copying them around may take a lot of time and paper.

> Prefer passing references, it's fast and economical.

2) For the `swap` to function properly, you can't pass numbers 5 and 10 by value; that is, copies of the values. Two working ways:

- C++ style: `swap` handles references function call: `swap(a,b);` (Ineffective code, does copying)

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

- C style: `swap` handles pointers function call: `swap(&a,&b);`

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

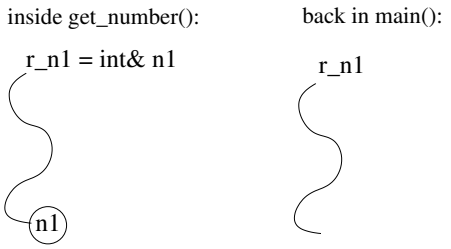
One benefit of passing by value is that the original value is safe, only a copy is sent out. In C++ you can give the reference `const` attribute, to protect it from changes. Caveat: some libraries seem to ignore this attribute. You can never be absolutely sure :(A malicious software package can also pretend it's using input by value only, but snatch the reference instead and change your input data! The function calls look the same. Not that I know of any such packages.

8.2 Can a reference be unsafe?

Alas, a reference is not completely safe, "it's still possible to shoot yourself in the leg". A ***dangling reference*** is one way to (mis)use an address of an object after the object has ceased to exist.

Example 9: (basic/dangling_reference.cpp) A dangling reference.

```
// Unsafe code example
// run with valgrind -q a.out
#include<iostream>
int & get_number()
{
  int n1 = 100;
  int & r_n1 = n1;
  return r_n1;
}
int main()
{
  int number = get_number();
  std::cout<<"number is "<<number<<"\n";
}
```



Think where the reference `r_n1` is referring to. The line

```
int & r_n1 = n1;
```

tells `r_n1` is a reference to an integer, which is `n1`. Nothing wrong with that, but `n1` is a ***local variable***, so it's no longer alive after you exit the function `get_number()`. After returning to `main()`, the reference `r_n1` points to where ***n1 used to be!*** What makes dangling references perilous, is that sometimes the reference happens to dangle over the right spot.

Example 10: (basic/dangling_reference3.cpp) A dangling reference that the g++ compiler will notice

```
// Unsafe code example
// g++ warns at complile time
#include<iostream>
int & get_number()
{
  int n1 = 100;
  return n1;
}
int main()
{
  int number = get_number();
  std::cout<<"number is "<<number<<"\n";
}
```

You don't necessarily need a function call to get a dangling reference. Example `basic/dangling_reference2.cpp` shows what happens if you store the reference to a vector element and then resize the vector. There is no guarantee the stored reference is pointing to a valid location any more.

valgrind a.out often detects such memory problems.

My own valgrind user manual: *"Any message from* ***`valgrind -q a.out`*** *means your code has a bug."*

8.3 `lvalue` and `rvalue`, and `rvalue` references

C++11 has actually *five* kinds of references, and I introduce two them so that you can read C++11 code :

- `lvalue` reference
  Marked with a single ambersand `&`, for example `int & a` . All references I've been using so far.

- `rvalue` reference
  Marked with a double ambersand `&&`, for example `T&&`. But wait, not all `&&`'s are rvalue references, read the next chapter.

There is also `xvalue`, `glvalue` and `prvalue`. I'm cutting corners, so for deeper understanding read, for example, c11-tutorial-explaining-the-ever-elusive-lvalues-and-rvalues by Danny Kalev.

Two things you need to consider: can the thing be ***moved*** and does it have an ***identity***.
Expressions, program statements that have a value, have long or short lifetimes. The long-living expressions have an identity, so that you can call them, the short-living expressions die soon and have no identity.

An `lvalue` is a non-movable expression or one with identity. It lives long or cannot be moved.
An `rvalue` is a movable expression or one without identity. It may be about to die, such as a ***temporary***[a].

[a] Evaluating `x=(a*c)+b` the compiler can create a temporary to store the result of `a*c`: that's an `rvalue`.

Simplified rule: `lvalue` is a variable, `rvalue` is a temp.

With an ***rvalue reference*** you can prolong the `rvalue`'s lifetime. For example,

```
int && temporarily_rich() { return 5000; } // number 5000 is an rvalue: no identity
temporarily poor&& =std::move(temporarily_rich);
std::cout<<poor<<std::endl;
```

8.4 The strange T&& and the *perfect forwarding problem*

This line of thinking is not my idea, it's been frequently used in teaching C++11 by the smart ones, like Scott Meyers and Eli Bendersky. I give here a try.

The task: Write a wrapper function `wrapper(x,y,z)` that *perfectly forwards* – here was the keyword – all arguments to function `f(x,y,z)`. Simple, ha? It's just a simple template:

```
// try one
template <typename T1, typename T1, typename T3>
void wrapper(T1 x, T2 y, T3 y){
  f(x,y,z);:
}
```

Then you realize that this will always call `f` by value, so the arguments `x,y,z` are copied over. This is no good if you can have declared `f(int & x, int & y, int & z)`. Clever you, you rewrite this version:

```
// try two
template <typename T1, typename T1, typename T3>
void wrapper(T1& x, T2& y, T3& y){
  f(x,y,z);:
}
```

Then you realize that this won't do if you can call `f` with rvalues (things that are movable or have no identity), like `f(1.0,3.0);`. The compiler complains about an invalid initialization of non-const reference from an rvalue. It cannot make a reference required by "`T1& x`" that points to number 1.0, because the number has no identity to refer to.

You could cure the problem by adding a `const` to an argument. To *any* argument, because `f` could have argument types `f(1.0,int& y, int& z)` or `f(int& x,3.0, int& z)` or ... !

At this stage you smell something burning. You would have to write overloaded wrapper functions for all possible combinations, in this case

```
wrapper(       T1& x,        T2& y,        T3& z)
wrapper(const  T1& x,        T2& y,        T3& z)
wrapper(       T1& x, const  T2& y,        T3& z)
wrapper(       T1& x,        T2& y, const  T3& z)
wrapper(const  T1& x, const  T2& y,        T3& z)
wrapper(const  T1& x,        T2& y, const  T3& z)
wrapper(       T1& x, const  T2& y, const  T3& z)
wrapper(const  T1& x, const  T2& y, const  T3& z)
```

This gets impossible with increasing number of arguments, for n arguments you would need 2^n overloads. And it only gets worse once you add the possibility of rvalue references. We could want `f(int x, int& y, int&& z)`, so to perfectly forward the `int&&` you would need more overloads.

C++11 solves this task with the type cast `std::forward`:

```
// solution
template <typename T1, typename T2, typename T3>
void wrapper(T1&& x, T2&& y, T3&& z) {
    f(std::forward<T1>(x), std::forward<T2>(y), std::forward<T2>(z));
}
```

As Scott Myers says, `std::move` moves nothing and `std::forward` forwards nothing. They do a ***type cast***. `std::move` is a single-minded-Robin-Hood cast:

```
poor = std::move(rich);  //rich becomes an rvalue
                         // ''movable'', property ready to be stolen.
                         // std::move does this always.
poor = std::move(another_poor); // really *always* , a single-minded cast
```

How did `std::forward` do the wrapper task? First, `T&&` is not always an rvalue reference. There are cases where one, instead, does ***type deduction***. This recycling of `&&` is widely considered unfortunate, but live with it. Scott Meyers calls `T&&` a *universal reference*, some call it *forwarding reference*. To make sense out of a thing like `& &&` (an rvalue reference to a reference?) one has a rule:

> ***Reference Collapsing rule***: `&` always wins.
> `T` is `T`
> `T&` is `T&`
> `T&& &` is `T&`
> `T& &&` is `T&`
> `T&& &&` is `T&&`

These rules seem odd, but they work. Return to the wrapper task and see how the argument types are cast:

```
wrapper(1.0,int& y,int&& z);
// will call
f(std::forward<int &&>(1.0), std::forward<int& &&>(y), std::forward<int && &&>(z));
// will reference-collapse to
f(int&&, int& y, int&& z); // rvalue reference, reference, rvalue reference
```

This is the way we wanted `f`'s arguments to be. Task accomplished.

8.5 The `restrict` keyword in C *(read on spare time)*

You probably encounter this subjet in C coded scientific programs.
Wikipedia:

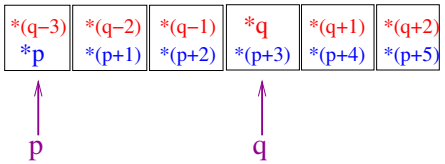
> In C or C++, as mandated by the strict aliasing rule, pointer arguments in a function are assumed to not alias if they point to fundamentally different types, except for char* and void*, which may alias to any other type. Some compilers allow the strict aliasing rule to be turned off, so that any pointer argument may alias any other pointer arguments. In this case, the compiler must assume that any accesses through these pointers can alias. This can prevent some optimizations from being made.
>
> In C99, the restrict keyword was added, which specifies that a pointer argument does not alias any other pointer argument.
>
> In Fortran, procedure arguments and other variables may not alias each other (unless they are pointers or have the target attribute), and the compiler assumes they do not. This enables excellent optimization, and is one major reason for Fortran's reputation as a fast language. (Note that aliasing may still occur within a Fortran function. For instance, if A is an array and i and j are indices which happen to have the same value, then A[i] and A[j] are two different names for the same memory location. Fortunately, since the base array must have the same name, index analysis can be done to determine cases where A[i] and A[j] cannot alias.)
>
> In pure functional languages, function arguments may usually alias each other, but all pointers are read-only. Thus, no alias analysis needs to be done.

In short: If you use pointers and you are sure two pointers never ever point to the same data, you can let compiler generate much faster C-code by using the `restrict` keyword.

Example: pointers `p` and `q` point to integers,

```
int* p;
int* q;
```

and will be pointing to elements of a table of integers like this:



If the code can have the situation, where, for example, the numbers `*(p+2)` and `*(q-1)` happen to be exactly the same element, you have ***aliasing***: Two pointers point to the same data. In other words, you had two different names for the same thing.
If you are absolutely sure this can never happen during the lifetime of pointers `p` and `q`, let the compiler know.
"I solemnly swear, not to write a line of code where a pointer points to the same object as `p`":

```
int *restrict p;
```

This is something the compiler can't deduce by itself, it has to trust your word for it . If you lie, the code may run, but the results are arbitrary.

In C, use the `restrict` keyword often, but with caution

Further information: demystifying the restrict keyword

9 C++ Standard Library: A closer look

9.1 `std::vector` container

`std::vector` containers are flexible and minimize the risks of dynamic memory allocations. But they are not vectors.

Example 11: (basic/vector.cpp) Create an empty container and push a few elements to it.

```
// Create empty vector container and push data in
#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              size of the container increased automatically
final capacity =8              more space reserved than needed, just in case you extend the vector
```

9.1.1 Iterators

Iterators are things pointing to elements of a container. A bit like custom-made pointers.

Example 12: (basic/vector2.cpp) Print all elements of a `std::vector` using iterator

```
// print all but 2 elements of a vector container using iterator (it)
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
int main()
{
  vector<double> x;
  // push i^2
  for(unsigned i=0;i<6;i++) x.push_back(pow(i,2));
  // use iterator to print all but the two last elements (!)
  for(auto it=x.begin() ; it!=x.end() - 2 ; ++it){
    cout<<*it<<' ' ;
  }
  cout<<endl;
  // To print *all* elements, use the range-for loop
  for(auto elem:x) cout<<elem<<' ';
  cout<<endl;
  return 0;
}
// output:
// 0 1 4 9
// 0 1 4 9 16 25
```

The keyword `auto it` is a message to the compiler: "figure out yourself the type of `it`, you can".

I recommend C++11 `auto` type; let the compiler figure out the type.

In the example `auto` spared you from typing the ugly type

```
vector<double>:: iterator it;  // type of an iterator
```

One way to fill a `std::vector` is `std::fill`; it's very fast if you need to reset an existing `std::vector` [6],

```
std::vector<double> y(5);
std::fill(y.begin(), y.end(), 1.0);     // math: y=(1.0,1.0,1.0,1.0,1.0)
std::fill (y.begin()+1,y.end()-2,2.0); // math: y=(1.0,2.0,2.0,1.0,1.0)
```

To create a `std::vector` and set values the cleanest ways are

```
std::vector<double> y(5,1.0);  // math: y=(1.0,1.0,1.0,1.0,1.0)
std::vector<double> y{1.0,1.0,1.0,1.0,1.0} // C++11 universal initialization
```

C++11 lets you initialize almost anything with the universal initialization {}.
C++11 has also a ***range-for*** or ***range-based-for*** loop:

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

The loop goes goes through all elements of `y` without you worrying about how many there are.

Be careful with the ambersand `&`:

```
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)
```

[6] Can you do algebra with the length of a `std::vector` ?
`x.resize(0)` sets the length to zero, so `x.size()` is 0. Ok, but `x.size()-1` is 18446744073709551615, not -1 !
Lengths are unsigned integers and can't store negative numbers.

9.1.2 Storing objects into `std::vector`

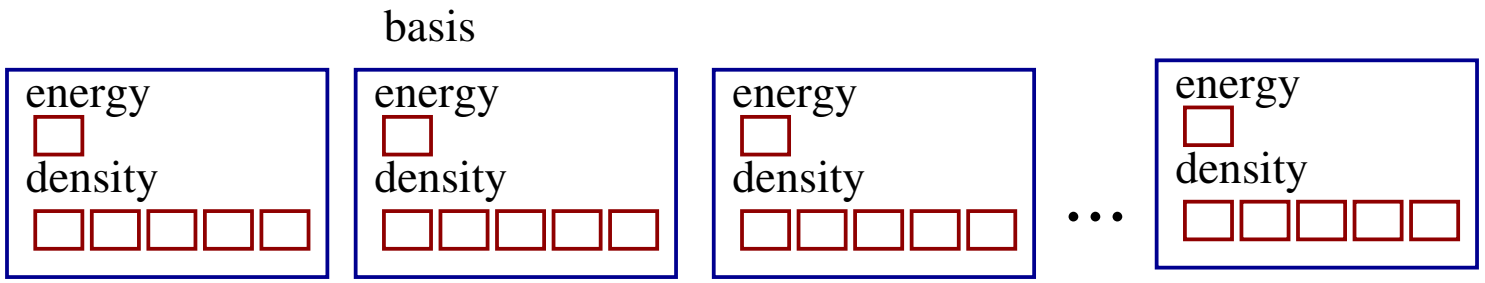
If you know C++ well, check out `numerics/better_vector_of_class_objects.cpp`.

Example 13: (numerics/vector_of_class_objects.cpp) `std::vector` storing objects

```
// g++ -std=c++11 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.clear();// REMEMBER THIS or 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;
  }
```

```
}
```

As you see, you can fill a `std::vector` with almost any type of data. Here the container contains object of the self-made type WaveFunction (blue boxes), and a number (energy) and a yet a `std::vector` (density).



These are exactly the same thing:

```
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;
};
```

9.2 Moving, not copying

You may have noticed that moving objects instead of copying them is a big thing in C++11. ***Moving*** is a Robin Hood operation,

```
poor=std::move(rich);
```

pilfers `rich` from it's resources and hands them to `poor`. Way more effective than copying money or gold!

The `std::move(rich)` does not actually move `rich`, it makes it *movable*. Many times the compiler cannot tell whether an object is movable, so with `std::move` you can tell that an object has resources that can be pilfered. In other words, `std::move` is for turning lvalues to rvalues (see section 8.3) so that you can call ***the move constructor***. The move constructor is the "Robin Hood code".

A move constructor and a move assignment look like this:

```
X::X(X&& other);              // C++11 move constructor
X& X::operator=(X&& other); // C++11 move assignment operator
```

You can be even more dramatic. If objects of a class should never be copied, you can forbid copying by ***deleting the copy constructor and the copy assignment*** (only in C++11):

```
class T {
        // ...
        T(const T&) = delete;// forbid copying
        T& operator=(const T&) = delete;
};
```

The next lengthy example tries to elucidate situations when an object is copied and when moved. For transparency, the copy and move constructors as well as the copy and move assignments print out a message so you can see which is invoked.

Example 14: (advanced_move_constructor.cpp) Copying or moving

```
#include <iostream>
#include <vector>


class T{
public:
  int count;
  std::vector<double> x;
  T() = default ; // constructor
  T(int count_, std::vector<double> x_): count{count_},x{x_} {} // constructor
  ~T() noexcept = default; // destructor
  // copy assignment
  auto operator=(T& rhs) & -> T&
  {
    x=rhs.x;
    count=rhs.count;
    std::cout << "-- copy assignment called --\n";
  }
  // copy constructor
  T (const T& rhs) : x(rhs.x), count(rhs.count)
  {
    std::cout << "-- copy constructor called --\n";
  }
  // move assignment
  auto operator=(T&& rhs) & noexcept -> T&
  {
    x=std::move(rhs.x);
    count=std::move(rhs.count);
    rhs.count = 0;
    rhs.x={};  //resizes x to 0 length
```

```
    std::cout << "-- move assignment called --\n";
  }
  //move constructor
  T(T&& rhs) noexcept : x(std::move(rhs.x)), count(std::move(rhs.count))
  {
    std::cout << "-- move constructor called --\n";
  }

};

int main()
{

  T x1{3,{1,2,2}};
  std::cout<<"T x2 = x1:\n";
  T x2;
  x2 = x1;
  std::cout<<"T x3(x1)\n";
  T x3(x1);  // same as T x3=x1
  std::cout<<"T x4 = std::move(x1)\n";
  T x4;
  x4 = std::move(x1);
  std::cout<<"T x5(std::move(x2))\n";
  T x5(std::move(x2));
}
```

9.3 `std::valarray` class

As you may have noticed, `std::vector` is far from ideal for numerics – ***you can hardly think of anything more horrendous!*** `std::vector` is not a mathematical vector, it's a list that can expand and shrink.

This led one to introduce to C++ the class `std::valarray` (not a container!). Recent compilers should produce as fast code with `std::valarray` as with `std::vector` , because the use the very efficient ***expression template*** technique. The closest relative to `std::valarray` is in the Boost library,

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

To be honest, `std::valarray` is not popular among C++ programmers and some recommend to stay away from it.

9.4 `typedef` and `using`: give a good name to your data type

IMHO the lack of built-in numerical data structures, such as vectors and matrices, prevents C++ to be a perfect language for numerics. I face a dilemma: How to code vectors and matrices? A few options:

1 Should I stick to `std::valarray` or `std::vector` and live with the deficiences? In a small project, that needs to be more portable than numerically less ambitious: Yes. In a serious numerical project: No, go to option 3.

2 Should I write a class for a vector myself?

> **Never** write your own classes for vectors or matrices.

You may see C++ numerics cources in the web and books telling you how to make a matrix class. Don't, you'll make mistakes, waste your time and get a slow program. Sure, these are deceivingly simple tasks at a first glance, but once you hit the speed bumps you can hear your tailpipe clank to the floor. Go to option 3.

3 Should I use a "vector" or "matrix" from an external (external to C++ standard) library?

> **YES**. But not directly, utilize `typedef` or `using`

This has the problem of being not part of the standard (Ha!), so the whole library may cease to conform with the standard in the future. The developers retire, or, more probably, someone writes a better and faster library and your vocal colleges push you to using that instead.

There is a way to overcome these difficulties. It's not a cure, but a workaround.

With `typedef` or `using` you can give a name to your own data structure
Write the code so that the decision about a data type is in one place, in case you change your mind.

```
typedef existing_type new_type_name;
```

Example:

```
typedef double v_data ;  // call double by the name v_data
typedef std::vector<v_data> stl_vector; // vector of v_data is called stl_vector
typedef std::vector<stl_vector > stl_matrix; // vector of vectors is a matrix
```

If you change your mind and want to store integers to `stl_vector` and `stl_matrix`, just change the first line to `typedef int v_data ;`
Another way to do the same:

```
using new_type_name=existing_type;
```

```
using v_data=double;  // call double by the name v_data
using stl_vector=std::vector<v_data>; // vector of v_data is called stl_vector
using stl_matrix=std::vector<stl_vector>; // vector of vectors is a matrix
```

Example: you find that using library smack (fictitious), that has a nice `smack::vector` , you could build upon this alias

```
using my_vector=smack::vector;
```

If you decide to replace `smack::vector` with a close relatives from library `crunge`, you edit these lines and not touching code using `my_vector`. Be cautious, `crunge::vector` and `smack::vector` may be incompatible and support different operations. Rather than inventing a name `my_vector`, it's better to use your own namespace `my` and hide the definitions there. Then `my::vector` is the thing to use. We'll come to this later.

9.5 Stream iterators *(read on spare time)*

Stream is an object representing an I/O *(input/output)* channel. Streams are considered one of the best qualities of C++ by C++ programmers (and one of the worst by C programmers). A stream iterator is something that goes through a stream. So a stream iterator is a boat. [7]

Think of the following task: Write a program that reads an arbitrary string of characters from the keyboard. Sounds simple, but in many programming languages this is very lengthy to do safely. Remember, user may hit any key and and keep on hitting for a month. Below is a C++ suggestion.

Example 15: (advanced/streamiter1.cpp) Read character to a `std::vector`

```
#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;
}
```

Use the keyboard to give characters and press ctrl-d when you are done. Here `std::copy` is an algorithm and I'm going to tell more about them next.

[7] istream = stream in, ostream= stream out

9.6 C++ Standard Library: algorithms and utilities

The C++ Standard Library has many useful methods, search algorithms, sorting algorithms etc. As soon as you know what you need, visit the pages

www.cplusplus.com/reference/algorithm
www.cplusplus.com/reference/utility

Examples:
If `v` and `w` are `std::vector` containers, and `it` is an iterator of that container type, then

`it = std::max_element(v.begin(),v.end())` returns the iterator `it` pointing to the largest element: the largest element is `*`it.
`std::sort(v.begin(),v.end())` sorts the container
`std::swap(v,w)` swaps the contents of containers `v` and `w`. C++11 has this in the header `<utility>`.

9.6.1 std::min_element, max_element, find , sort, reverse

Example 16: (numerics/algo_minmax.cpp) C++ Standard Library algorithms: minimum, maximum and search of an element

```
// Finding elements from a vector container
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

// utility to print out a vector
template<typename T>
void vector_out(const vector<T> v)
{
  for(auto x:v) cout<<x<<" ";
  cout<<endl;
}

int main()
{
  vector<double> v{1.4,1.6,0.2,1.8,0.1,1.5}; // or do many push_back's

  cout<<"original vector"<<endl;
  vector_out(v);

  auto it = min_element(v.begin(),v.end()); // vector<double>::iterator it;
  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;
}
```

Iterators may look a bit messy, but they are easily hidden from view. Especially, if all you want is the number where the iterator points at.

Example 17: (numerics/easyusemin.cpp) C++ Standard Library algorithms: another version of minimum search

```
// Finding min element; trying to simplify usage
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

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

```
minimum element = 1.3
```

Be cautious when coding specialized versions of C++ Standard Library algorithms. Here I used the namespace `my` to protect the home made function `min_element`. This is necessary, because somebody else may use that simple, descriptive name and once you incorporate his/her program to your you end up with a name conflict.

9.6.2 `std::swap` is a template

Previously given `swap` using reference variables is for swapping one type of things only. What if we want to swap two real numbers or two other type of variables? Terribly boring to write each variable type it's own version of `swap`.

Write a ***template***. Actually it's been done already, `std::swap(a,b)` looks like this. Like many standard library codes, this too were refurnished in C++11. The user sees only the improved performance.

Example 18: (advanced/swap_template.cpp) std::swap template

```
template <class T> void swap (T& a, T& b)
{
  T c(std::move(a)); a=std::move(b); b=std::move(c);
}
template <class T, size_t N> void swap (T &a[N], T &b[N])
{
  for (size_t i = 0; i<N; ++i) swap (a[i],b[i]);
// C++98 version was
//{
//  T c(a); a=b; b=c;
//}
```

`template`	this is a ***template***
`class T`	means `T` is ***a class***; no need to know what class!
`void swap`	means swap returns nothing
	no point writing
	`result = swap(a,b); // nonsense`
`T c(std::move(a))`	create class `T` object `c` and ***move*** the contents of `a` to object `c`
	`std::move` make sure you use the correct `move` function from the `std` namespace
	`std::move(a)` fix `a` to a movable object (one whose contents can be stolen).

What can be swapped with this template:
(C++11)"Type `T` shall be move-constructible and move-assignable (or have swap defined for it)" [8]

[8] C++11 replaced copy (`T c(a)`) for move (`T c(std::move(a))`): search the net for "C++ move semantics".

Many C++ Standard Library algorithms are compact and efficient. Take for example this one:
(this page has only utility functions)

Example 19: (numerics/algo_permutations.cpp) : C++ Standard Library permutation algorithm

```
// Finding permutations
// using C++ Standard Library algorithm next_permutation
//
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

// utility to print out a vector
template<typename T>
void vector_out(const vector<T> v)
{
  for(auto x:v) cout<<x<<" ";
  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()); // make sure next_permutation starts ok
  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
```

There are $N!$ permutations, computed for checking in the ***recursive function*** `factorial()` [9]. Now

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

does all the work, all else is supporting code. It generates the next permutation of elements and returns `true`, until it no longer finds a new permutation and returns `false`. The loop

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

goes on until the test `while (true)` changes to `while (false)`.

[9] A recursive function calls itself; The implementation is poor, there is no test that the result fits an `int`.

Physics example:
Generate all many-body states with a fixed number of fermions.
Each spin state can hold 0 or 1 fermions (Pauli rule). If you have 4 spin states and 2 fermions, you can have states occupied as (0011),(0101),(0110),(1001),(1010),(1100) in total 6 different many-body states.

Example 20: (numerics/algo_fermion_states.cpp) Fermion states by permutation

```
// Finding fermion basis states
// using C++ Standard Library 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;
  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
```

What would for example 110010 stand for? Electrons can have spin up ($\uparrow$) or down ($\downarrow$), so the states could be coded as occupations of up-down pairs:

110010 would mean $\uparrow\downarrow\, 00 \uparrow 0$,

meaning the 1st one-body state has up and down electrons, the 2nd is empty, and the 3rd has a spin-up electron. Each state has only zeroes and ones, so it would be very economical to encode them in binary.

9.7 Header guards and namespace encapsulation

This section demonstrates how to write a function for home-made statistics, with C++ Standard Library algorithms and without. This is just for demonstration, there are much better statistical library routines than this. Here we push numbers to a `std::vector` and compute the statistical mean and standard deviation of the data in the function `get_stats()`. Let's start with a header. Headers are for the compiler, with some information to you; Java doesn't need them.

Example 21: (numerics/mystatistics1.hpp) First attempt as a header for `get_stats()`

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

The ***preprosessor directives***

```
#ifndef MYSTATISTICS_HPP
#define MYSTATISTICS_HPP
...
#endif
```

make up a ***header guard***. They make sure this piece of code is not processed more than once. To use this header, stored in the file `numerics/mystatistics.hpp`, put in the beginning of the program the line

```
#include "mystatistics.hpp"
```

This line may well be in many program units, hence the header guard: `ifndef` stands for "if not defined". If `MYSTATISTICS_HPP` is not defined, define `MYSTATISTICS_HPP` and process the rest. The next time the compiler tries to `# include mystatistics.hpp` it already has the code processed and does nothing.

The file suffix `.hpp` is one way to tell that this is a header file, not to be processed unless `include`'d. In C one has the suffix `.h`, and it will also do in C++. ***In C++ the common practise is to put to headers only function declarations***.

What if `get_stats()` is part of a huge pile of code, where another `get_stats()` happens to exist, with the same type of arguments? You get an ***ambiquity error*** also called ***name collision***. Let's use ***namespace encapsulation***. Define our own namespace, where `get_stats()` lives, so that we can be sure which of the many `get_stats()` should be invoked.

Example 22: (numerics/mystatistics.hpp) A better `get_stats()` header

```
#ifndef MYSTATISTICS_HPP
#define MYSTATISTICS_HPP
#include <vector>

namespace mystat
{
  void  get_stats(const std::vector<double> &, double & , double & );
}

#endif
```

A remark about style and good habits: refrain from taking the whole `std` namespace unnecessarily,

```
using namespace std; // don't do this
namespace mystat
{
...
}
```

This is impolite, because if you ever give this header to a college to be used in his/her code, the poor fellow gets the whole `std` namespace, wanted or not. This easily leads to to name collisions, if the college was not carefully protecting his/her cute and short function names, such as `get()`.

Namespace encapsulation saves the day, `mystat::get()` is always recognizable as a `mystat` function and `myclasses::vector` is your `vector`, not `std::vector` .

The next task is to code the function itself. I present here two styles, the first utilizes `std` algorithms, the other plain C++11 loops.

The principle of `get_stats` that uses `std` algorithms:

> The algorithm `std::accumulate` computes the sum of elements (unless told to do something else). The container `std::vector` has has method `size()`, which gives the number of elements. For standard deviation we compute how much elements differ from the average using the algorithm `std::transform`. There is also `std::bind2nd`, which is left for the reader. It's there because `std::transform` can do only operations for one or two elements, so the problem of feeding in the average as well is solved using a `lambda` function (old days one used `std::bind2nd`, but it's deprecated).

If you at this point feel exhausted, don't be alarmed. I feel the same!

Example 23: (numerics/mystatistics.cpp) `get_stats()` using `std` algorithms

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

namespace mystat
{
  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
    // deprecated in C++11:
    // transform(x.begin(),x.end(),xx.begin(),bind2nd(minus<double>(),average));
    // C++11 version using a lambda function
    transform(x.begin(),x.end(),xx.begin(),[average](const double & elem) {return
       elem-average;});
    sigma = sqrt(inner_product(xx.begin(),xx.end(),xx.begin(),0.0)/(N-1));
  }
}
```

Example 24: (numerics/mystatistics_c++11.cpp) `get_stats` using plain C++11

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

namespace mystat_C11
{
  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));
  }
}
```

I used the namespace `mystatC11` to tell from the previous `srd` version. The header in file `mystatistics_c++11.hpp` has the same namespace.

You may disagree, but to me this version is much easier to grasp that the C++ Standard Library version. The code and the math formulas are more easily compared. The *range-for* loop is neat and safe.

For completeness, I present the main program to test both variants.

Example 25: (main_mystatistics.cpp) Main program to test `mystatistics.cpp` and `mystatistics_c++11.cpp`

```
// compile: g++ -std=c++11 main_mystatistics.cpp mystatistics_c++11.cpp
//                         mystatistics.cpp utility.cpp
#include <iostream>
#include <vector>
#include "mystatistics.hpp"
#include "mystatistics_c++11.hpp"
#include "utility.hpp"
int main()
{
  using namespace std;
  vector<double> x{5.5,6.3,5.6,8.1,9.7,10.0};
  double ave1,ave2,sigma1,sigma2;

  cout<<"x = \n";
  myutil::vector_out(x);
  cout<<"std algoritm version: vs. C++11 version:"<<endl;
  mystat::get_stats(x,ave1,sigma1);     // std  version
  mystat_C11::get_stats(x,ave2,sigma2); // C++11 version
  cout<<"           average = "<<ave1<<" vs. "<<ave2<<endl;
  cout<<"standard deviation = "<<sigma1<<" vs. "<<sigma2<<endl;
}
//std algoritm version: vs. C++11 version:
//           average = 7.53333 vs. 7.53333
//standard deviation = 2.02452 vs. 2.02452
```

9.8 `std::complex`: complex numbers and arithmetics

All common complex operations (multiplication (*), division (/), addition (+), real part (real()) etc. are there already.

Example 26: (numerics/complex_ex.cpp) Basic operations with complex numbers

```
#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;
}
```

In math, the multiplication of complex numbers is

$$\begin{aligned} x &= a + i\,b \qquad y = c + i\,d \\ xy &= ac - bd + i(ad + bc)\ . \end{aligned}$$

So how does the compiler know that if `x` and `y` are type `std::complex`, then `x*y` means this operation? It's called ***operator overloading***, but let's first take a look at the simpler ***function overloading***.

10 Function overloading, optional arguments and default arguments

Function overloading in C++ means you can assign different, but related tasks under one function name. This is nothing new, in math the exponent of a real number x is $\exp(x)$ and the exponent of a complex number c is $\exp(c)$. Even $\exp(M)$ of matrix M is under the same name exp().

C has no function overloading, only math functions have been overloaded. Designers of C++ have apparently a different opinion of good practise.

The compiler has to distinguish which task you want the function to perform: It does this by means of the types and number of arguments – a bit like in math.

Moreover, function overloading makes two things possible:

- ***Optional arguments*** are ones you don't always have to give.
  For example, `estimate(x)` may do a slightly different calculation than `estimate(x,a)`. No need to call them `estimate1(x)` and `estimate2(x,a)`.

- ***Default arguments***: unless given, the argument has its default value.
  Imagine the boredom and messy program if you always have to call the function
  `myfun(x,y,alpha,beta,gamma);`
  with all five arguments even though you know you in most cases have `alpha=1;beta=5;gamma=14.5;`. In C++ you can set these values as defaults, and call the function simply
  `myfun(x,y);`

Important difference:

- Optional arguments *are used in the function only if they are present.*
  Their presence or absence causes different code to be executed

- Default arguments *are always used in the function* and they must have some values, default or given.

Don't overdo function overloading.
There's no real benefit in calling two functions by the same name if their tasks have nothing in common.

Example 27: (basic/function_overload.cpp) Function arguments a and b are optional

```
// Function integ() can do two different things, depending on arguments
#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
```

Example 28: (basic/function_overload2.cpp) Functions second argument is by default 1.0

```
// Function fun has a default value 1.0 for the second parameter b
#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
```

Default value is given *only* in the function declaration. This can make the default value hard to find! [10]

[10] Technically, it's possible to set default values in function *definition*, but I strongly advice you not to. You code will not be.

11 Operator overloading – all for readability

In section 9.8 we learned, that the complex number multiplication is done correctly by the `*` operator. The way this was achieved is ***operator overloading***: an operator can be told to do a slightly different operation depending on the data type.

> Operator overloading can greatly improve code readability

Operator overloading is something you don't have to learn to do, but you will appreciate it if someone has done a good job overloading operators. As an example, without overloading, adding two complex numbers `c1` and `c2` would read something like this:

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

With overloaded `+` operation it reads

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

The compiler does change the `+` to function call, but it's out of sight. The code is readable and just like math.

> If you overload an operator, make sure it works as expected

This is related to

> ***The law of least astonishment***: The program should behave in a way that least astonishes the user. [11]

[11] Steve Oualline, *How Not To Program in C++*.

Here is a story of a code that didn't obey the *law of least astonishment*:

> I was surfing the net one day for examples on how operators can be overloaded, and came across with one that overloaded the `+` operator for complex numbers. Upon testing, I found that the sum `c3 = c1 + c2` really is computed correctly. I thought the implementation was ok and used it in my code. Then I *was astonished* to get wrong results! No, not because `c3` was computed wrong, no-hou. It was because I didn't come to think that the operation `c3 = c1 + c2` was changing also the value of `c1`! It did, and in the code that followed `c1` had a wrong value.

A classic piece of bad code is this attempt to use a macro for squaring:

```
// BAD CODE, *Never* use #define, this is just for educational purposes
#define SQUARE(x) ((x)*(x))
int a=2;
int b=SQUARE(a++); // you would think this squares 2 and *then* increments it by one
                   // No. The result is (2)*(3)=6. Astonishing!
```

There are some rules and limitations to operator overloading:

- Think of operators as functions with one or two arguments. They are called unary and binary operators. If your operation needs two arguments, take one existing binary operator and overload that.
- You can't invent new operators
  ("`my_clever_new_operator`" is no good)
  Only some of the existing ones can be overloaded:

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

- The order of execution prevails ($*$ is executed before $+$)

You may find tempting to overload an operator to compute powers. Especially since the math form x^y and the function call `pow(x,y)` look so very different. Resist the temptation and use `pow` or you'll be astonished.

Curiosity:
In case you are interested, fortran has operator overloading, too. Funnily, in C++ you can't invent your own operators, in fortran you can overload existing ones if you apply it to your own data type, but if you apply it to an existing data type you *must* invent a new operator :) So working on 2D tables `double A(:,:),B(:,:)`, where the data type exists, you must invent an operator, such as `.x.` (the strange colours come from C++ settings):

```
! fortran
interface operator(.x.)
   module procedure multMatrix
end interface operator(.x.)
...
function multMatrix(lhs,rhs) result(res)
```

and use it as `D=A.x.B.x.C`. It's also simple to create you own matrix type and overload * ,

```
! fortran
type matrix
   double, pointer:: data(:,:)
end type Matrix
interface operator(*)
   module procedure multMatrix
end interface operator(*)
...
function multMatrix(lhs,rhs) result(res)
```

and write `D=A*B*C`. but then you have to dig the `data` from the objects (in fortran it's done `A%data`).

Example 29: (advanced/myclass_overload.cpp) operator << overloaded to print self-made class objects

```
// How to teach << to print a self-made class object
#include <iostream>
#include <vector>
#include <iterator>
#include <iomanip>
#include <fstream>
class MyClass
{
  double a,b;
  std::vector<double> v;
public:
  // universal initialization
  MyClass(double a_,double b_,std::vector<double> v_): a{a_},b{b_},v{v_}{}
  // overload << and tell it how to print MyClass objects
  // define << as friend to let it access private data
  friend std::ostream& operator <<(std::ostream& os, const MyClass& obj);
};

std::ostream& operator<<(std::ostream & os, const MyClass& obj)
{
  using namespace std;
  os<<fixed<<setprecision(8); // some I/O manipulation
  os<<"a="<<obj.a<<" b="<<obj.b<<endl;
  os<<"v=";
  for(auto ele:obj.v) os<<ele<<" ";
  return os;
}
//continues
```

```
int main()
{
  using namespace std;
  MyClass testclass{2.2,3.1,{1.0,2.0,3.0}}; // C++11 universal initialization
  cout<<testclass<<endl;  // uses overloaded <<
  ofstream out("MyClass.out");
  out<<testclass<<endl; // same output to file
}
// on screen and in file "MyClass.out":
// a=2.20000000 b=3.10000000
// v=1.00000000 2.00000000 3.00000000
```

The overloaded operator `<<` can also write to a file

Without overloading, `cout<<testclass` cannot work. As s reward, we have a clean output of objects of a self-made class without a visible call to a function. This may sound a small achievement, but in numerics you learn to appreciate clean formulas. Assume you have a matrix A, vectors a, b and c, and hte job is to compute $c = Ab + a$. With overloading, this will at best look likes

```
c=A*b+a;
```

which is easier to decipher than a nested function call,

```
c=vec_add(matrix_multiply(A,b),a);
```

Now that you asked, the overloaded operator `<<` is compiled with this logic:
Operation `cout<<testclass` means "find from the class of the object `cout` (that'll be `ostream`) the method `<<` and call it with arguments `(cout,testclass)`". This translates to something not completely unlike "`(ostream.<<)(cout,testclass)`". Next, the compiler searches the methods `(ostream.<<)` to find a function, whose arguments are `(ostream&, MyClass&)` or without the ambersand `&`. It can find one, as a method with the `friend` attribute, the one we just wrote.

12 C++ Standard Library: More algorithms

12.1 std::for_each

The algorithm `std::for_each` performs a given operation to all elements. As arguments you give the beginning, the end and what to do.

Example 30: (numerics/vector_print_foreach.cpp) `std::for_each` and printing (some) elements of a `std::vector`

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void doubleout(double y) { cout << " " << y; }
int main () {
  vector<double> x{1.1,2.2,3.3};
  cout << "x vector: \n";
  //  for_each (x.begin(), x.end(), doubleout);
  for_each (x.begin(), x.begin()+2, doubleout);
  cout<<endl;
  return 0;
}
```

This applies `doubleout()` to all elements. In general, the applied function can be anything, as long it's declared `anything(double x)`, i.e., it must eat doubles.

12.2 When to use `std::for_each` ?

Now you may wonder what more **`std::for_each`** has to offer than the range-for loops in C++11. After all, you can print all elements of a container more neatly with a range-for loop:

Example 31: (numerics/vector_print_range_for.cpp) Range-for loop and printing all elements of a **`std::vector`**

```
#include <iostream>
#include <vector>
using namespace std;
void doubleout(double y) { cout << " " << y; }
int main () {
  vector<double> x{1.1,2.2,3.3};
  cout << "x vector: \n";
  for(auto elem:x) doubleout(elem);
  cout<<endl;
  return 0;
}
```

This is just as good as the **`std::for_each`** example, and compiles faster!

This is where `for_each` shines:

- Access only part of the elements

```
for_each(x.begin(),x.begin()+2,doubleout()); // output two values from the
   beginning
```

- Access all elements or some elements and use anything a class can contain

  Example 32: (numerics/for_each_limited.sum.cpp) `std::for_each` can do things range-for loops can't

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

struct LimitedSum {
  void operator()(int i) { if (i > 1) sum += i;}
  int sum{0};
};

int main() {
  std::vector<int> x{1,2,3,4};
  LimitedSum lim = std::for_each(x.begin(), x.end(), LimitedSum());
  std::cout << "Limited sum = " << lim.sum << "\n";
}
```

  Notice how we access the member of the class `LimitedSum` after a call to `std::for_each`: The return value of `for_each` is the very same class object that the function object in the argument is. Function object are discussed more in chapter 13.3

  Without storing the return value we had no access to the data member `sum`. See the details on the next page.

12.3 `std::for_each` in detail

One way `std::for_each` can be implemented is this:

Example 33: (advanced/foreach_template.cpp)

```
// std::for_each works essentially like this
template<class Iter, class Func>
Func for_each(Iter first, Iter last, Func f)
{
  while (first!=last) {
    f (*first);
    ++first;
  }
  return f;      //  C++11: return move(f);
}
```

The third argument `f` can be a class – as long as `f(*first)` is defined! This hints that the name of a class can sometimes be used the same way as a function; they are called function objects or functors.

> `for_each` argument third is `class Func f`, and the return value is the same `f`

The "pass-by-value feature"

Notice also how the third argument is passed by value, as `class Func`. This means the argument object `Func` has a one-way ticket, it does not return as an argument. But the function does return it as a return value. So in as argument, out as a return value. Example: If you send in a function object and change something in that object, you have to use the return value object. The example `numerics/foreach_functor2.cpp` shows the principle, it computes the cosines of elements (done by the function object) and collect their sum as a data member in the object.

The gcc version 4.9.1 has the file `.../include/c++/4.9.1/parallel/for_each.h`. Here the "parallel" gives away that `for_each` can be parallelized: All elements are pushed separately though the same function, so why not do it in parallel.

The page Draft of Technical Specifaction tells just how draft the parallel part is still in 2015:

Working Draft, Technical Specification for C++ Extensions for Parallelism

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad fomatting.

This kind of put me off and I had to go for coffee.

12.4 `std::generate` algorithm

One way to generate values to a container.

Example 34: (numerics/generate_random_vector.cpp) Fill a `std::vector` with random numbers

```
#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;
  }
}
```

The next chapter takes a look at why `generate` may be dangerous for random number generation.

12.5 C++ Standard Library algorithms – take care of copies

As the previous example showed, `std::generate` is a nice way to fill a container with random numbers. There is a potential risk, however. The functioning of `std::generate` is equivalent to this:

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

The third argument is `Generator gen`, and `Generator` is a class. ***The generator is passed by value, so a copy of gen is used***. The compiler creates a copy of `gen` using the copy constructor of the class `Generator`.

`std::generate` may copy the third argument.

`std::for_each` may copy of the third argument.

⇒ ***The random number generator is copied, too***.

Why not copy a random number generator (rng)? A rng is just another program. Given a ***seed***, it can produce a nearly random number sequence. The sequence depends only on the seed, the algorithm is deterministic and the same seed gives exactly the same number sequence. That's why it's often called a *pseudo* random number generator. If you copy the rng, you have two identical "number mills". If you compare the two number sequences they produce, you find that within each sequence the numbers are (almost) random, but the numbers in the two sequences are badly correlated. Turning the crank of two similar mills in a simulation code can give you exciting, but wrong, results. [12]

Every generator, that is not giving a constant output, must have a ***state***. In other words, a generator that can give non-constant output has to remember where it is. A clock that cannot keep track of time is a stopped clock. Copying a ***stateful*** generator has to be done with care.

[12] Ah. why not give the generator a new seed and get a new random sequence, different from the other? That way you would have several number mills with different seeds. Bad idea. The rng algorithms have been tested to produce a number sequence random only in relation to numbers within the same sequence. The basic problem is that the *seeds* that determine the sequences are not random. Ok, why not run one mill with a "mother seed" to give you seeds for many rng's? That too, has not been tested to give sufficiently random results. Algorithmic generation of (pseudo) random numbers is a tricky thing.

12.6 C++ Standard Library algorithms – stateful objects and `std::ref`

Algorithms `std::for_each()` and `std::generate()` are not useless in context of stateful objects. There is a simple remedy to the copy problem: For stateful objects, use a *reference wrapper*

```
std::vector v(100);
std::generate (v.begin(), v.end(), std::ref(gen));  // gen is stateful object
```

`std::ref` is a helper function to generate a std::reference_wrapper, meaning (see `std::ref` or `std::reference_wrapper`)
`std::ref(10)` is `std::reference_wrapper<int>`
Example 35: (numerics/generate_id_vector.cpp) Fill a `std::vector` with unique id numbers from an id generator.

```
#include<iostream>
#include<algorithm>
#include<functional>
#include<vector>
using namespace std;
class IdGen{
  int id;  // object's state
public:
  IdGen(): id(0) {cout<<"constructed an IdGen\n";}; // constructor
  int operator()(void) {return(id++);} // functor
}idgen;
int main() {
  vector<int> v(20),w(20);
  generate(v.begin(), v.end(), std::ref(idgen)); // try these *without* std::ref
  generate(w.begin(), w.end(), std::ref(idgen));
  std::cout<<"v = \n";
  for(auto x:v) cout<<x<<" ";  cout<<"\n";
  std::cout<<"w = \n";
  for(auto x:w) cout<<x<<" ";  cout<<"\n";
}
```

13 A few things that can potentially speed up your code

You may be interested to check out Wiki: C++ Performance improving features. If you are a good programmer, find out what "Move Semantics" and "Perfect Forwarding" mean in C++, and dive into the pool of "Smart Pointers". Read books and postings by Scott Meyers!

13.1 `noexcept`: no-throw quarantee

Recommendation: Use frequently in numerical code

If your function never throws an exception, the keyword `noexcept` may let the compiler to optimize your code more. [13] If it does throw, your code will terminate - Ha, you lied! Still, `noexcept` is one of the very latest features of C++, so don't believe just any blog posts, just give it a try. Usage:

```
void myfunction() noexcept
{
  //...
}
```

13.2 `constexpr`: Compile time constant expressions

Recommendation: Use frequently in numerical code

Not quaranteed to give any speedup, but an interesting concept. Computation of factorials recursively can be traditionally done like this:

```
// Common way to compute a factorial
int factorial ( const int n ) {
  int fact = 1 ;
  if ( n <= 1 )
    return 1 ;
```

[13] We return to throw-catch in chapter 21.

```
  else
    fact = n*factorial ( n - 1 ) ; // recursion
  return fact ;
}
```

C++11 is able to perform tasks during compilation, coded as ***template metaprogramming*** or with ***constexpr***.

```
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);  // 13! is computed in compile time
   ...
}
```

If you look at the assembly code after compilation (`g++ -S code.cpp` and look at `code.s`) you see that indeed fact13 is set equal to 6227020800.

> `constexpr` means that everything the expression **may** be known at compile time.
> If all is known, everything is done in compile time. If not, the function is treated as a normal function, evaluated in run time.

It a win-win situation! Although compilation takes longer. Numerical constants are, by definition, already constant expressions, so no need to replace `const`'s with `constexpr`'s there.

Extra information:
`const` means that, once initialized, the value cannot be changed. `constexpr` is more than `const`, it's a *constant expression*. Functions declared `constexpr` can compute the allocator template parameter of e.g. `std::list<>` (see stackoverflow:difference-between-constexpr-and-const).

If you know C++ metaprogramming and Haskell, you might be interested to read

What Does Haskell Have to Do with C++?

For curiosity, the same task with template metaprogramming:

```
// Template metaprogram
#include<iostream> // just for testing
template<int N>
struct Factorial {
  static const long int value = N * Factorial<N-1>::value;
};
template<>
struct Factorial<1> {
  static const long int value =1;
};
int main()
{
  const long int fact13 = Factorial<13>::value;
  std::cout<<fact13<<std::endl;
}
```

The number 13 is fed to `Factorial` as a template parameter: `Factorial<13>`. The compiler ***instantiates*** the template `Factorial<13>`, which tells to instantiate the template `Factorial<12>` and so on, until `Factorial<1>` is instantiated. At this points the compiler notices that there is a ***template specialization*** corresponding to argument `<1>`, and the recursive instantiation ends. Compilers can't handle very deep recursive instantiation, and builtin integer data type can't hold large factorials anyhow.

13.3 Function objects (functors)

Recommendation: Use frequently in numerical code

Function objects, or ***functors*** for short, are rather popular in numerics. You can define a function in a class that is not a method (member function). This function is called when the class name is used as a function name. Don't be put off, this is actually very simple.

Example 36: (numerics/functor.cpp) A function object to compute $\sin(x)$, $\cos(x)$ and $\tan(x)$

```
// Function object - functor
#include <iostream>
#include <cmath>
class TrigFuns{
public:
  double operator()(double x) {
    std::cout<<"x="<<x<<std::endl;
    std::cout<<"sin(x)      cos(x)      tan(x)\n";
    std::cout<<sin(x)<<" "<<cos(x)<<" "<<tan(x)<<std::endl;
  }
};
int main()
{
  TrigFuns comp;
  comp(10.0);
  return 0;
}
```

Here `comp` is an object, but used as if it were a function – hence it's a function object.

Why use a function object? Why not an ordinary function?

1) Function objects are not passed as pointers, so the compiler can easily *inline* it (insert to its place). As you saw in exerc.1.pdf, one way to send a function to a function is as a function pointer:

```
void apply(double (*g)(double),double x)  {std::cout<<g(x)<<std::endl;}
```

but it's also simple to wrap the function in a class and make it a function object,

```
#include <iostream>
#include <cmath>
class Func{
public:
  double operator()(double x) { return sin(x);}
};
void apply(Func g,double x) {std::cout<<g(x)<<std::endl;}

int main()
{
  Func f;
  apply(f,10.0);
  return 0;
}
```

2) A class can contain data members, such as counters, accessible only to class methods. A function object can easily perform complicated tasks. For example, a function object can compute the cosine of all input and, simultaneously, compute the sum of these cosines – this is done in `numerics/forarch_functor2.cpp`.

Example 37: (numerics/foreach_functor.cpp) A function object used with `std::for_each`

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

class TakeCos{
public:
  void operator()(double& x){ x=cos(x); } // function object
};

int main () {
  vector<double> x{1.1,2.2,3.3};
  cout << "vector x      : ";
  for(auto e:x) cout<<e<<" ";
  cout<<endl;
  for_each(x.begin(), x.end(), TakeCos());
  cout << "vector cos(x) : ";
  for(auto e:x) cout<<e<<" ";
  cout<<endl;
  return 0;
}
```

If the task is this simple, or it's supposedly used only here, it's more convenient to use a ***lambda function***, introduced later in chapter 24. One example of a lambda function was already in `numerics/mystatistics.cpp`.

13.4 Four ways to pass a function to a function

1) One way is to pass a ***function pointer***:

```
double f(double x) {return 1.0 + exp(10.0*x);} // sample integrand
double integrate( double (*f) (double), double a, double b)
{
 // integrate function f(x) from a to b
}
// use like
integrate(f,1.0,2.0); // integrate f(x)
```

2) Wrap the function in a class and make it a ***function object***. Write `integrate` as a template.

```
class MyFunc{
     public:
             double operator()(double x) {return 1.0 + exp(10.0*x);}; //
                define a function object
};
template<class T>
double integrate(T f, double a, double b){
  ... use function f() as usual...
}
// use like
integrate(Myfunc(),1.0,2.0); // integrate MyFunc class function object
```

Notice how this code does not have any explicit `MyFunc` class objects. The compiler instantiates the template `integrate`, with typename `T` for `MyFunc`, and replaces the function `f()` with `MyFunc::operator()` as an *inline* function). This makes the code very fast!

3) C++11: Pass the function as a ***function object of the class template*** *std::function*
Recommended method

```
        double f(double x) {return 1.0 + exp(10.0*x);} // sample integrand
        double integrate(const std::function<double(double)> &f, double a,
           double b){ // integrate function f(x) from a to b}
        // use like
        integrate(f,1.0,2.0); // integrate f(x)
```

If the function has been declared earlier, you can let the compiler deduce the types (now `double(double)`)

```
    double integrate(const std::function<decltype(f)> &f, double a, double b)
```

4) Pass the function pointer as a ***template parameter***:

```
        double f(double x) {return 1.0 + exp(10.0*x);} // sample integrand
        template<double Tfunc(double)>
                double integrate(double a, double b){
                ... use function Tfunc() as usual...
                }
// use like
integrate<f>(1.0, 2.0); // integrate function f
```

Examples of each style is in the file `numerics/function_to_function_speed_test.cpp`. The speed depends on whether the compiler inlines a function pointer; recent ones do. IMHO, methods 2 and 3 looks best and conform with the C++ general goal to *"avoid naked pointers"*.

13.5 Cache data

Don't recompute the same data over and over again. Use *cached* values.

13.6 Use `emplace_back` instead of `push_back`

Recommendation: Use always if available

`emplace_back` will contruct the value at the end of the `std::vector`, while `push_back` will contruct it someplace else and move it to the vector. The latter simply has to be slower.

Example 38: (numerics/emplace_vector.cpp) How `emplace_back` beats `push_back` in speed.

```
#include <iostream>
#include <cmath>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct MyThing
{
  int idat;
  std::vector<double> x;
  MyThing(int idat_,vector<double> x_) noexcept : idat{idat_},x{x_}{}
};
```

```
int main()
{
  const int N=1000000;
  vector<MyThing> v;
  auto t0 = high_resolution_clock::now();

  for (auto i=0;i<N;++i)
    {
      v.push_back(MyThing(i,vector<double>{1.0,2.0,3.0,4.0,5.0,6.0}));
    }
  auto t1 = high_resolution_clock::now();
  auto d = duration_cast<milliseconds>(t1-t0);
  cout <<"vector push_back took: "<<d.count() << " ms\n";

  v.clear(); // remove reservations in the container

  t0 = high_resolution_clock::now();
  for (auto i=0;i<N;++i)
    {
      v.emplace_back(i,vector<double>{1.0,2.0,3.0,4.0,5.0,6.0});
    }
  t1 = high_resolution_clock::now();
  d = duration_cast<milliseconds>(t1-t0);
  cout <<"vector emplace_back took: "<<d.count() << " ms\n";

}
```

`emplace_back` can also call the constructor **with parameters needed in contructing**! You don't even consruct the value of the `std::vector` element, you just tell *how* it should be made.

Example 39: (numerics/emplace_parameter_arguments.cpp) `emplace_back` can construct with parameters.

```
//
// emplace_back with parameter arguments
//
#include <iostream>
#include <cmath>
#include <vector>

using namespace std;

struct MyObj{
  double value;
  MyObj(const double & par,double angle) noexcept : value{par*sin(angle)} {}
};

int main()
{
  const int N=1000000;
  const double par=3.866;
  vector<MyObj> v;
  for (auto i=0;i<N;++i)
    {
      v.emplace_back(par,i*M_PI/N); // not emplacing an object, but forwarding
         parameters to constructor
    }
  cout<<"first 10 values:\n";
  for (auto it=v.begin();it<v.begin()+10;++it) cout<<(*it).value<<" ";
  cout<<"\n";
}
```

13.7 If available, use the methods of containers rather than algorithms

The general idea is that a container-specific algorithm can take advantage of the container properties and gain speed. Sometimes generic algorithms are not available at all; the `list` container has no random access iterator, so `std::sort` can't work. Instead, there is a `sort` method in `list`:

```
...
std::list<int> mylist;
// fill mylist
mylist.sort() ; // calls the sort member function
...
```

13.8 Expression templates

This chapter gives you an idea how complicated it is to master C++ and why I never can. What I hope you to learn from all of this is **use good libraries**. What do expression templates do faster than a `for`–loop? If you ever find out, please tell me. If you have no desire to become a C++ guru, skip this chapter and do something useful.

All this said, here goes:

> *The basic idea behind expression templates is to use operator overloading to build parse trees*

Expression templates were used by Todd Veldhuizen in his matrix library Blitz++ in mid 90's, ever since applied in practically all numerical C++ libraries (see Todd Veldhuizen: "Techniques for Scientific C++"). [14] Early expression templates were only able to cure one bad C++ side effect, namely that one should not create temporaries in every corner. [15] But this only ***taught C++ behave reasonably***. Since then we have learned quite a bit and realized that avoiding temporaries is not the only essence in speed.

Where did the "avoid temporaries" goal come about? C++ has this wonderful thing called operator overloading. The problem is that if you apply is straightforwardly, the compiler easily creates temporaries. Consider how the addition operator can be translated to function calls:

```
D = A+B+C means add(A,B)+C, so set temporary M=add(A,B), finally set D = add(M,C)
```

If `A,B,C` and `D` fill the fast ***cache memory***, then the intermediate result `M` "drops" something out of cache to slow RAM – 10x slower or more. Another example is adding three vectors, and using only one component,

```
D[0] = (A+B+C)[0];
```

You can easily expand this the most effective way,

```
D[0] = A[0]+B[0]+C[0];
```

but a compiler can be so stupid that it adds up the whole million element vector `A+B+C`, puts it to `D` and only *then* looks for `D[0]`!

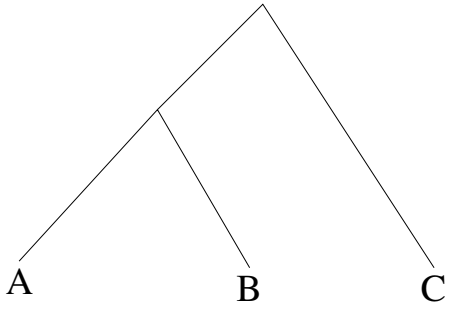
[14] Blitz++ still works behind scenes: Last time I looked, Scipy (Scientific Python) module used parts of the Blitz++ library.

[15] `Valarray` tries to avoid intermediates (temporaries) using "proxy" objects. Most libraries prefer the expression template technique due to its generality.

Continue reading if you the previous discussion didn't drop out of your cache.
Think how templates can help to avoid temporaries. If a compiler meets a line of code it can't immediately recognize, it starts looking for a suitable template ("model"). Finding one, it ***instantiates*** the template, i.e. brings it alive. This template can itself instantiate other templates and so on. Apart from instantiating another template, a template can instantiate another copy of itself (recursive template). This recursion is in the heart of many expression templates. For identification, templates have ***template parameters***, which tell exactly what kind of model to instantiate: [16]

```
template <parameters>
   blaablaa() {...}
```

Letting templates instantiate new templates lets one express the addition of objects as a tree:



The vertices could represent + operators. Let `A,B,C,D` be class `Array` objects (just some class, data could be in a `std::vector`). Then the tree could be

```
Array A, B, C, D;
D = A + B + C;
first + should translate as expression X<Array,plus,Array>() + C;
second + should translate as expression X<X<Array,plus,Array>,plus,Array>();
```

Here `X` is a vertex for objects "left" and "right" the operator "plus" in between. The code `advanced/recursive_template.cpp` shows how such a tree structure `X<X<Array,plus,Array>,plus,Array>` is created. The example `advanced/expression_template.cpp` computes the sum of `std::vector` 's using expression templates.

[16] Excuse me for mixing template arguments and template parameters, I honestly don't know their difference.

What do we gain?

Expression templates can delay the evaluation until the "=" sign is reached (lazy evaluation)

The compiler goes through the whole tree at compilation time, instantiating vertices `X`, as many as needed. Reaching the end it knows exactly what is needed (such as only `D[0]`).

The idea is that operations return expressions, not results.

Expressions are something that can be further manipulated during compilation.

Compiling the code with a recursive template essentially generates changes to the code, or "the code changes itself". This is the essence of ***template metaprogramming***: templates can generate new code during program compilation. Alas, compilation takes longer.

[16] Also "template template arguments" exist. IBM explains, on their linux compiler pages
http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Ftemplate_arguments.htm
rather aptly, I should say:

A template argument for a template template parameter is the name of a class template.

14 The best random number generator

If only the best is good enough:

The fastest quantum random number generator to date. Credit: ICFO

Delivers a very random number in every nanosecond. This was needed in a quantum entanglements experiment at Delft (More in einstein-god-dice)

15 C++11 generation of (pseudo)random numbers

A random number generator is a program, that, given a seed (say 23525176471263), produces a sequence of numbers that appears random. It's like a number mill: In goes the seed and out comes flour of random numbers. Always the same output.

Stages to invoke C++11 random number generation:

1) Include the headers

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

2) Choose the type of random number algorithm

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

Here `gener` is my name for the generator; only this appears in the code and changing this single line I can change to another generator (`linear_congruential_engine`, `subtract_with_carry_engine`),

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

3) Choose from available distributions
`uniform_real_distribution` (parameters start and end) `normal_distribution` (parameters meand value and variance).
Uniform distribution `unif_dist` U[0,1) is created like this:

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

and a normal distribution like this (the name `normal_dist` is my own):

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

4) Give the generator a seed **only once**
(suffix `u` means `unsigned`)

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

or pick the seed from system clock

```
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  )
```

or rely on `std::ramdom_device`

```
gener.seed(std::random_device{}());
```

This is not without problems, see a discussion in cpps-random_device.html

5) EITHER (i) bind the generator and the distribution together:

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

and use the combination like this:

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

OR (ii) get the random number directly from a call to `distribution(generator)` :

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

Here the random number is `random`.

15.1 `std::bind` to simplify usage

Before continuing with random numbers, lets look what `auto normal_random = std::bind(normal_dist, gener)` did: It created a function `normal_random()` that means "call `normal_dist` with argument `gener`".

Example 40: (basic/bind_example.cpp) `std::bind` is versatile

```
// How to use bind to
// a) change the order of arguments
// b) turn a 3 argument function to a 2 argument function
//
#include<iostream>
#include <functional>
void  f(const double & x,const double & y,const double & z)
{
  std::cout<<"called f with arguments "<<x<<" "<<y<<" "<<z<<std::endl;
}
int main()
{
  using namespace std::placeholders;  // for _1, _2
  f(1.1,2.2,3.3);
  // a) change the order of arguments with bind:
  auto invf=std::bind(f,_3,_2,_1);   // bind return type is std::function
  invf(1.1,2.2,3.3);
  // b) create a two-argument function from f
  auto g=std::bind(f,_1,_2, 333.3); // bind 3rd argument to fixed value 333.3
  g(10.1,20.2);
}
```

The next examples are a set of easy-to-use helper functions to initialize a generator and get uniform distribution (`unirand()`), normal distribution (`gaussrand` and `gaussrand2()`) or exponentially distributed (`exprand()`) random numbers.
The goal here was to hide all inconveniences to the header `numerics/random.hpp`:

```
#include "random.hpp"
...
double random = gaussrand() // clean and simple
```

The header is by no means perfect, especially the initialization is clumsy.

esl

Example 41: (numerics/random.cpp) C++11 random number generation

```
// 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();
}
```

16 Boost library

Boost library has served as a test bench for ideas, that may be included in a future C++ standard. C++11 took many of its new features from Boost. It offers many extensions to the C++ standard library: ***algorithms, special functions, differential equation solvers and many more.***

The documentation is

`http://www.boost.org/doc/libs/`

and a very interesting math part is in

`http://www.boost.org/doc/libs/?view=category_Math`

The examples are thorough, e.g. how to use your own vector type in a equation is shown here. As long as your data structure conforms with a standard or a Boost container you can just throw it into Boost. Especially the library odeint for solving ordinary differential equations is comprehensive.

Boost is in most parts a *header-only* library. In linux, this means you just copy it to, say, directory `/home/myusername/boost_1_59_0`. The example program `chaotic_system.cpp` resides in `/home/myusername/boost_1_59_0/libs/numeric/odeint/examples/`, and it can be translated like this (bash):

```
mkdir tmp
cd tmp
export BOOSTDIR=/home/myusername/boost_1_59_0/
ln -s $BOOSTDIR/libs/numeric/odeint/examples/chaotic_system.cpp .
ln -s $BOOSTDIR/libs/numeric/odeint/examples/gram_schmidt.hpp .
g++ -std=c++11 -O3 -I$BOOSTDIR chaotic_system.cpp
```

Of course, installing Boost to your *include* path makes life simple – if you have enough admin rights.

```
Boost examples in calc.phys.jyu.fi:
module add gcc
g++ -I/usr/local/boost_1_59_0 /usr/local/boost_1_59_0/libs/numeric/odeint/examples/solar_system.cpp
a.out > solar_system.dat
and in gnuplot
p 'solar_system.dat' u 2:4 w l," u 5:7 w l," u 8:10 w l," u 11:13 w l," u 14:16 w l," u 17:19 w l
```

16.1 Boost: ordinary differential equations (ODE)

If your equation is nth order (second order or higher) you first separate it to n coupled first order differential equations (details in 22.3). Then use your great mathematical intuition and tell if the solution has kinks or is otherwise badly behaving. Choose the integration algorithm, the "stepper", that solves the next point.

Fixed-step-size routines: Simple and fast, accuracy of the solution is your responsibility.
Adaptive-step-size routines: Try to reach an accuracy goal (absolute and relative error limit).

Boost has a seasoned library for solving ODE'S, as does GSL. Boost calls the current values the "state", which contains $\{x(t), x'(t), ...\}$. Hence the type `state_type`.

Example 42: (numerics/boost_ode_simple.cpp) Fixed step-size solution using boost::numeric::odeint

```
#include <iostream>
#include <boost/numeric/odeint.hpp>
#include <fstream>
using namespace std;

using state_type = std::vector<double> ;
using stepper_type = boost::numeric::odeint::runge_kutta4<state_type> ;

/* Solves x''(t) = -x(t) - gam*x'(t), split to coupled
    x'(t) = y(t)
    y'(t) = -x(t) - gam*y(t)
   notation: x(t)=x[0], y(t)=x[1], x'(t) = dxdt[0], y'(t) = dxdt[1]
*/
void harmonic_oscillator(const state_type &x, state_type &dxdt, const double /*t*/)
{
  const double gam=0.15;
  dxdt[0] = x[1];
  dxdt[1] = -x[0] - gam*x[1];
}
```

```
int main()
{
  stepper_type stepper;
  state_type x ={1.0,2.0} ; // initial values, x(0) = 1, x'(0) = 2
  // integrate all the way to final time:
  //integrate_const( stepper , harmonic_oscillator , x , 0.0 , 100.0 , 0.01 );
  //cout<<"final point "<<x<<endl;

  // use method do_step to follow the solution step by step
  ofstream out("ode.dat");
  const double dt = 0.01;
  for( double t=0.0 ; t<100.0 ; t+= dt )
    {
      stepper.do_step( harmonic_oscillator , x , t , dt );
      cout<<t<<" "<<x[0]<<" "<<x[1]<<endl;
      out<<t<<" "<<x[0]<<" "<<x[1]<<endl;
    }
  out.close();
}
```

Example 43: (numerics/boost_ode_adaptive.cpp) Adaptive step-size solution using boost::numeric::odeint

```
// solve y'(t) = -t*y ,  condition y(0)  = -2
// Adaptive integration using Boost::numeric::odeint
// compile :
// g++ --std=c++11 boost_ode_simple.cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <boost/numeric/odeint.hpp>

using state_type= std::vector<double>;
using stepper_type= boost::numeric::odeint::runge_kutta_cash_karp54<state_type>;

namespace my{
  using namespace std;
  void system(const state_type& y, state_type& dydt, const double t)
  {
    dydt[0] = -t*y[0];
  }
  void output(double t, double y, double exact){
    cout<<fixed<<setprecision(16);
    static bool first=true;
    if(first) {
      cout<<setw(20)<<"t"<<setw(20)<<"Boost 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()
{
  using namespace std;
  const double e_abs=1e-15,e_rel=1e-15; // absolute and relative error goal
  auto stepper = boost::numeric::odeint::make_controlled<stepper_type>(e_abs, e_rel)
      ;
  state_type y ={-2.0} ; //condition y(0)=-2 gives y(t) = -2.0*exp(-0.5*t*t)
  const int n=50; // solve 50 points
  const double t1 = 0.0, t2=10.0;
  const double dt = (t2-t1)/(n-1);

  for(int i = 0; i<n ; ++i)
    {
      auto t = t1+i*dt;
      auto exact = -2.0*exp(-0.5*t*t);
      my::output(t,y[0],exact);
      auto steps =integrate_adaptive(stepper, my::system, y, t, t+dt, 0.01);
      // cout<<"steps "<<steps<<endl; // how many sub-division steps were needed
    }
}
```

Such a simple `my::system` begs to be replaced with a lambda expression:

```
auto steps =integrate_adaptive(stepper,
           [](const state_type& y, state_type& dydt, const double t)
           {dydt[0] = -t*y[0];}
           , y, t, t+dt, 0.01);
```

This same ODE will be solved using GSL in chapter 22.3.

17 Formatted output

It's important to keep numerical output readable. The following example is mixing columns pretty badly:

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

and this is easily the default outcome. Another thing to remember is that the method may limit the accuracy of the result, not the number of stored and filed decimals. It's up to you not to publish 6 decimals if the method is reliable only up to 2 decimals.

17.1 Formatted output using `#include <iomanip>`

The next shows how a `std::cout` object eats output manipulations: width of the field (10), the output form (fixed), and the number of decimals (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;
```

This is awful, but at least the same formatting works also with writing to a file; it's a stream just as `cout`.

```
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
```

See `numerics/output_formatting.cpp` for more examples. Beware, that after formatting some setting are still on (`fixed`), while some are immediately forgotten (`setw`)!
C has the famous `printf` function, which is quite readable, but not considered a good C++ practise. I suggest that if you are already good with `printf` use it. [17]

[17]Pros of a `stream` object is type safety, the compiler can (always?) tell if a data type cannot be sensibly dealt with, and flexibility, it works the same way on screen and on file.

17.2 `printf`, type safety and variadic functions and templates

There are several attempts to code a type-safe `printf` in C++, for example by Andrei Alexandrescu (see type-safe printf discussion). Boost has worked on the issue quiet a while, see on choices to be made in printf.

I suggest that as soon as you define a class, think of how the objects should be printed. Write a member function to do that, or overload `<<` to do the job. The former way looks like this:

```
class Thing{
        mytype data;  // some data type, can be self-defined as here
public:
        print() { ..some code ..} // how to print data
};
int main()
{
        Thing blob{ ..input data.. }; // create and initialize a Thing
        blob.print();    // output the Thing
}
```

You can improve this example easily to use arbitrary `stream` for output and hide the formatting to the member function. Same for reading, as `blob.read()`.

By the way, `printf` is an example of a function, that can have a variable number of arguments, this is known as

variadic function

C++11 introduced a template, that can have any number of template arguments,

variadic template
`template <typename ...Ts>`

Here the `...` is *literally* in the code, it's known as the ***ellipsis***.

This can be quite handy and I'm just learning to appreciate it. It's also template meta-programming and that makes us all feel groovy. Note: The C++ Standard Library has already a nice variadic template, `std::tuple`.

Example 44: (advanced/variadic_template.cpp) Variadic template to sum squares of numbers.

```
// Any number of arguments to sum_of_squares()
#include <iostream>
#include <cmath>
using std::cout;
using std::pow;

double sum_of_squares() {return 0; }// end of recursion

template <typename T,typename ...Ts>
double sum_of_squares(T first, Ts ... rest)  // double output to be sure
{
  // pow takes a double or float as base, int base will be implicitly converted
  return pow(first,2) + sum_of_squares(rest ...);   // recursive
}
int main()
{
  cout<<sum_of_squares(1,2)<<"\n";
  cout<<sum_of_squares(1,2,3,4,5)<<"\n";
  cout<<sum_of_squares(1.1,2.2,3.3)<<"\n";
  cout<<sum_of_squares(1,2.1,3.1)<<"\n";
}
// In the coming standard C++1y (g++ -std=c++1y) you could write
//     auto sum_of_squares(T first, Ts ... rest)
```

For more examples, see http://en.wikipedia.org/wiki/Variadic_template, which contains also an example of how one could improve `printf` in C++11. Don't worry about runtime speed

C++ variadic templates are expanded at compile time. [18]

[18] In C, variadic functions are resolved at run time, so there *is* a speed penalty.

18 Linear algebra – which library to use?

THE answer is LAPACK and BLAS, written in fortran and available in www.netlib.org. Processor manufacturers have their optimized versions (ACML,MKL) and an independent, freely distributed automatically optimized ATLAS. C versions (CLAPACK, CBLAS) are heavily used in GSL[19]. GSL, the Gnu Scientific Library, offers an extensive package of C-numerics.

There are optimized LAPACK and BLAS libraries by Intel (MKL) and AMD (ACML). To be able to use them you have to

- learn how to call fortran subroutines from C++ (tricky business)
- find a third party *wrapper*, C++ code that calls fortran or the C versions of the libraries.

Calling old fortran-77 (FORTRAN) library routines is pretty safe, but calling, for example, new fortran-95 parts of the MKL library has spawned a warning:

> *Caution*
> *Avoid calling BLAS 95/LAPACK 95 from C/C++. Such calls require skills in manipulating the descriptor of a deferred-shape array, which is the Fortran 90 type. Moreover, BLAS95/LAPACK95 routines contain links to a Fortran RTL.*

[19] The netlib versio of CLAPACK was "f2c'd", translated from fortran to C. I find this approach and buying with photocopied money more than a bit troublesome. Why not use the real thing?

A C++ programmer would prefer a package written in C++. There was once LAPACK++, but the project faded away around year 2000. It was superseced by TNT (Template Numerical Toolkit), which too looks pale. Boost is very much alive, but mostly not numerics stuff. The Boost collection `boost::ublas` is outdated, limited and the interface is too far from math notation (only my personal opinion).

Here is an incomplete list of projects providing well-coded C++ algorithms:

- **OpenBlas** and **GotoBlas2** (original code by Kazushige Goto) is optimized to processor architectures (Nehalem, Sandy Bridge, Haswell, AMD Bulldozer, Piledriver etc.)
- **Armadillo** (NICTA (Australia),) 2014-10-30 Version 4.500 (Singapore Sling)
  can utilize MKL and OpenBlas, to mention a few.
- **MTL4** (Simunova, a small c++ software company in Dresden) (speed tests)
- **IT++**(Sweden; version 4.2.0 came out 2010 – long break –version 4.3.1 came out 6.7.2013)
- **Blaze** 20.6.2014 version 2.1
  A project initiated by Klaus Iglberger, main developers in Erlangen, Germany.
  A link to a fine article on Smart Expression Templates.
- Trilinos (Sandia) A huge collection of methods; actively maintained, version 11.12.1 Oct. 24th 2014.
- **Eigen** Eigenvalue problem specialist – version 3.2.2 came out 4.8.2014
- IETL (The Iterative Eigensolver Template Library) is another eigenvalue specialist. Uses Boost, LAPACK, BLAS, ATLAS and possibly Blitz++. The web page was updated 2004, so I wonder if there is any development..

Armadillo, MTL4 and Blaze have a simple interface, close to Matlab/Octave. They use generic programming and boldly apply the *expression template* technique to optimize e.g. matrix and vector operations. In this group Blaze is a newcomer – and blazingly fast! It works parallel in shared memory machines, such as your laptop.

> Header-only libraries only need to be copied to your machine.
> Include the library path in compilation is all you need to do.

18.1 Armadillo examples

Example 45: (numerics/arma_matrix_multi.cpp) Armadillo: Matrix product (source: Armadillo web page)

```
#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;
}
```

This calls internally a CBLAS routine, since the attempt to compile as

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

gives the error message (among a plethora of other lines)

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

Here "dgemm" stands for a LAPACK/BLAS subroutine name meaning "double general matrix matrix". You get CBLAS as part of GSL or OpenBlas or ATLAS:

```
g++ -std=c++11 arma_matrix_multi.cpp `gsl-config –libs`
```

```
g++ -std=c++11 arma_matrix_multi.cpp -lopenblas
```

```
g++ -std=c++11 arma_matrix_multi.cpp -L/usr/lib64/atlas/ -lcblas
```

Example 46: (numerics/arma_eigenvalues.cpp) Armadillo: Eigenvalues of a symmetric matrix

```
#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);  // a way to make a symmetric matrix
  cout<< "A= \n"<<A<<endl;;

  eig_sym(eigval, eigvec, A); // this does all work

  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;
}
```

Armadillo calls the LAPACK routine `dsyev` ("double symmetric eigenvalue"), so link, e.g., with `-llapack`.

One possible output:

```
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

...
```

18.2 Blaze example

Example 47: (numerics/blaze1.cpp) Blaze: Simple vector and matrix operations (source: Blaze web page)

```
// https://code.google.com/p/blaze-lib/wiki/Getting_Started
// "An Example Involving Matrices"
// added main()
#include <blaze/Math.h>

using namespace blaze;

int main()
{

  // Instantiating a dynamic 3D column vector
  DynamicVector<int> x( 3UL );
  x[0] =  4;
  x[1] = -1;
  x[2] =  3;

  // Instantiating a dynamic 2x3 row-major matrix, preinitialized with 0. Via the
     function call
  // operator three values of the matrix are explicitly set to get the matrix
  //   ( 1  0  4 )
  //   ( 0 -2  0 )
  DynamicMatrix<int> A( 2UL, 3UL, 0 );
  A(0,0) =  1;
  A(0,2) =  4;
  A(1,1) = -2;

  // Performing a dense matrix/dense vector multiplication
  DynamicVector<int> y = A * x;
```

```
  // Printing the resulting vector
  std::cout << "y =\n" << y << "\n";

  // Instantiating a static column-major matrix. The matrix is directly initialized
     as
  //   (  3 -1 )
  //   (  0  2 )
  //   ( -1  0 )
  StaticMatrix<int,3UL,2UL,columnMajor> B( 3, 0, -1, -1, 2, 0 );

  // Performing a dense matrix/dense matrix multiplication
  DynamicMatrix<int> C = A * B;

  // Printing the resulting matrix
  std::cout << "C =\n" << C << "\n";
}
```

```
g++ -std=c++11 -Ipath_to_blaze blaze1.cpp
```

calc:

```
module add gcc
g++ -std=c++11 -I/usr/local/blaze blaze1.cpp
```

19 Calling C or fortran from C++

Here are some principles about mixing fortran, C and C++.

Example 48: Fortran subroutine `dgemm` is part of BLAS, it computes the matrix-matrix product.

```
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,*)
```

Declaration in C++:

```
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"
  means "compile C style".
  It tells the C++ compiler to forget about function overloading and identify the function by its name only , just like in C,
- The compiled fortran subroutine is in object code file `dgemm.o` or inside a library with an unerscore: `dgemm()` is shown as `dgemm_()`.
- fortran is always "pass by reference", so all arguments have to be pointers. Here `double* a` points to the start of a double array.

As if this were not enough, there is one more problem: matrix storage in fortran (and in Matlab) is ***column major***, but in C/C++ ***row major***. Below is a figure about the two storage habits.

$$\text{math notation:} \quad \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

fortran/Matlab row major : 1 5 2 6 3 7 4 8

C/C++ column major : 1 2 3 4 5 6 7 8 .

Both are natural, but moving between languages you have to change indexing before and after a call to `dgemm`. For square matrices this means transpose. If you have defined a static array in C-style like this

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

the fortran call must include `const int* n` .

Finally, by default indices begin from 0 (C/C++) or from 1 (fortran)

fortran : 1 → N, meaning `V(1),V(2)...,V(N)`

C++ : 0 →(N-1), meaning `V(0),V(1)...,V(N-1)`

20 Fixed-size arrays in C++: plain array and `std::array`:

Two different things that can be called "arrays".

- Plain, C-style array. A static array is made like this:

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

Dynamic arrays are created with keywords `new` and deleted with `delete`. I'm not telling you how to use plain arrays, because they don't conform with C++ containers and their memory management is manual labour. Deleting a two-dimensional, dynamic array `array2` is *not* simply `delete[][] array2`. Below are two examples.

Example 49: (basic/array_to_function.cpp) Passing an C array to function in C++ style

```
#include <iostream>
using namespace std;
void f(int d[],const int sized){ // C++ style
  // empty [] tells compiler d is an 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 ;
}
```

Example 50: (basic/array_to_function.cpp) Passing an C array to function in C style

```
#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 ;
}
```

- There is one more data type suitable for numerical data in C++, `std::array`. It's a container, for start. From cppreference, `std::array` can be initialized like this:

```
  std::array<int, 3> a1{ {1,2,3} };  // double-braces required
  std::array<int, 3> a2 = {1, 2, 3}; // except after =
  std::array<std::string, 2> a3 = { {std::string("a"), "b"} };
```

  Notice how `std::array` type and dimension is defined as template parameters! Since it's a container, it has many built-in methods just like `std::vector` has.

  Fixed size makes compile time checks possible, since C++11 also template arguments can be checked. For that C++ has `std::static_assert` (see `basic/static_assert.cpp`). `std::static_assert` can greatly benefit code development, by letting you know if you, by mistake, contradict your own ideas.

21 Exception handling with `throw` and `catch`

The C++ Standard Library has a class dedicated to exception handling, `std::exception`. One part of it is `runtime_error`. One way to make your own error handling process is to *inherit* the class,

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

and add a new property.
In numerics exceptions are often simple, I'm using this handling:

```
  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";
    ...
  }
```

Error happens if `test` is `true`, and the exception with message "test failed" is thrown and we leave the function. Later the exception is caught with `catch` [20]

[20] The function `my_function()` throws and exception and wishes some exception handling routine will take care of it properly.

If `throw` is executed, the function execution terminates; it's still more gentle than halting the *program* execution. Any code after `throw` is not executed and essentially the function does not return at all (so no need to have a `return` in a function that only throws).

In the example the message is output to stream `std::cerr`, similar to `std::cout`, but specialized for error outputs. This makes it possible to separate error output from normal output. In bash, typing

```
a.out 2> program.err
```

causes normal `std::cout` output to screen and `std::cerr` output to go to file `program.err`. There is also `std::clog` for log outputs.

22 Gnu Scientific Library (GSL)

GSL in wikipedia

GSL is free and written in C. For C++ users the linkage is made easy,

> *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++.*

Example 51: (numerics/gsl_bessel.cpp) Bessel function $J0$

Compilation `g++ gsl_bessel.cpp -lgsl -lgslcblas` or `g++ gsl_bessel.cpp` `` `gsl-config --libs` ``
(try `gsl-config --libs` in linux)

```
#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);
  printf("J0(%g) = %.18f\n", x, y);
  // or using iomanip:
  //cout<<fixed<<setprecision(18);
  //std::cout<<"J0("<<x<<") = "<<y<<endl;
  return 0;
}
// J0(5) = -0.177596771314338264
```

22.1 GSL: statistics

`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`.

Example 52: (numerics/gsl_statistics.cpp) Arithmetic mean and standard deviation

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

Passing a `std::vector` as a pointer `&v[0]` is not pretty, but does the job. All that the GSL function needs is the start address and the length of the data; elements of a `std::vector` are stored in consecutive memory slots.

22.2 GSL: Fast Fourier Transform (FFT)

I want to transform complex `data` with length 2^N, $N \in \mathbb{Z}_{>0}$, like this:

```
fft(data,direction) // direction = 1 forward, -1 backward
```

Here `direction` is 1 for a Fourier transform and -1 for an inverse transform. Notice that I want to keep this syntax no matter what library does the FFT. The decision to use GSL's FFT is made in a `#include`, which loads the header given below.

The variable type of `data` is flexible, and the template works at least for `std::vector` , `std::array` (fixed size, not plain array), `std::valarray`, armadillo vector, and Blaze vector. Armadillo vectors needs some special care. The compiler option `-DARMA` chooses the first option in

```
#ifdef ARMA
       // Armadillo data has no method size, use n_elem
       n = data.n_elem ;
#else
       n = data.size() ;
#endif
```

This solves the problem of missing `size()` method, but it's a hacky solution.

Example 53: (gsl_fft.hpp) A possible GSL FFT header

```
#ifndef GSL_FFT_HPP
#define  GSL_FFT_HPP
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

namespace my_GSL_FFT
{
  template <typename T>
  int fft(T& data, int direction){
    int status;
    const size_t stride=1;
    size_t n;
#ifdef ARMA
    // Armadillo data has no method size(), use n_elem
    n = data.n_elem;
#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
```

22.2.1 Passing a pointer to complex data

C++11 complains if you try to take the address of a complex number like this:

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

Here `data[0].real()` is an ***rvalue***, a temporary whose life is about to end (see chapter 8.3). It does not have an identity that lives long enough to be used. The first complex element *does* have an identity, an `lvalue`, so I tell the compiler to use that and pretend it's pointing to a `double` using `reinterpret_cast`. The type change done by `reinterpret_cast` is in your responsibility, it basically tells the compiler "trust me, I know what I'm doing".

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

Armadillo vector does not conform with a standard container, specifically it has no `size()` method. The least I want is to add a third argument to the clean call `fft(data,direction)` to tell the data size! I have chosen to use the ***preprocessor*** and
`#define ARMA`
if I'm using Armadillo vectors. Test code `numerics/gsl_fft_arma_main.cpp` is shown below, one using `std::vector` is in `numerics/gsl_fft_main.cpp`.

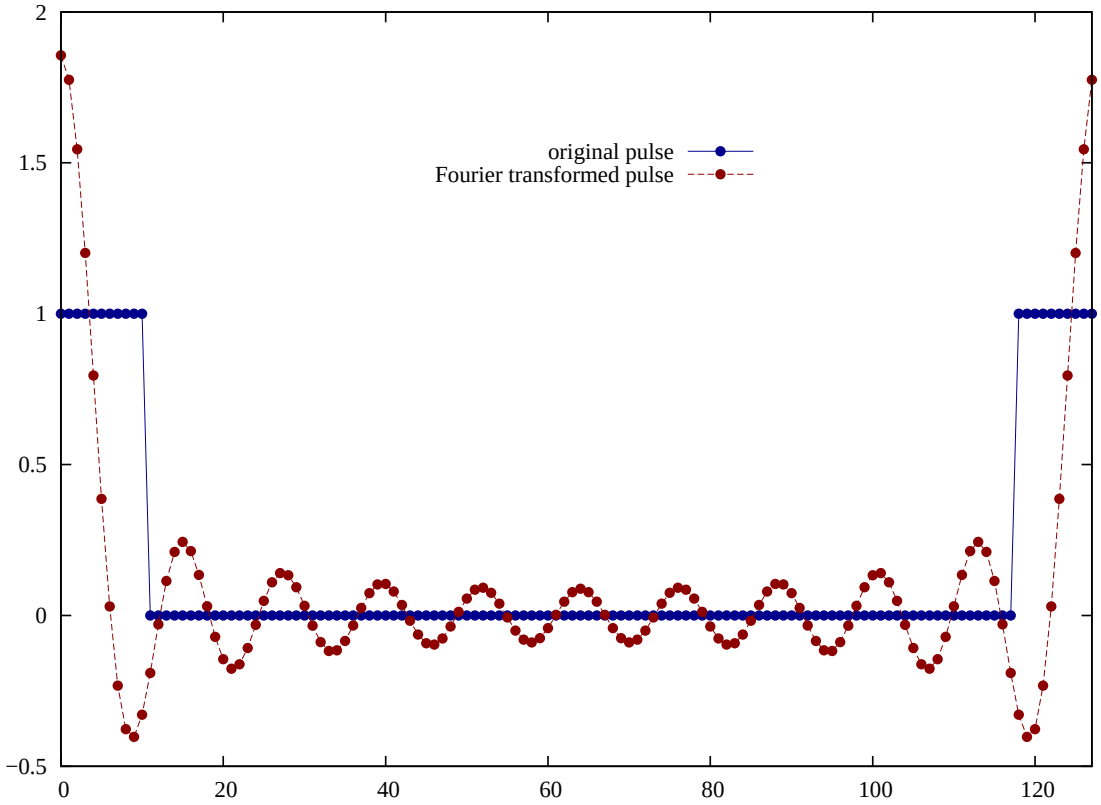
Example 54: (numerics/gsl_fft_arma_main.cpp) FFT test using Armadillo; modified from GSL manual C version.

```
// Compile: (the ‘ below is printed wrong in latex)
// g++ -std=c++11 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;
using my_GSL_FFT::fft;
// short name for data type
using vectype=Col<complex<double>>;


// overload << to print complex numbers in vectype as I like them here
std::ostream& operator<<(std::ostream & os, const vectype & v)
{
  using namespace std;
  os<<scientific<<setprecision(8); // some I/O manipulation
  int i=0;
  for(auto x: v){os<<setw(5)<<i++<<" "<<setw(20)<<x.real()<<" "<<setw(20)<<x.imag()
     <<endl; }
  return os;
}
```

```
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;++i) data(i)=0.0;

  cout<<"before fft :\n";
  cout<<data;
  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);
  cout<<data;
  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);
  cout<<data;
  myfile<<data;
  myfile.close();
  return 0;
}
```

Result:



The data is represented, as usual in FFT, in *wrap-around order*. You can easily write utilities to do the wrapping from "natural order" and unwrapping to natural order – move data or use an index table.

22.3 GSL: differential equations

Let's solve a simple equation, one that we can solve analytically,

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

Example 55: (numerics/gsl_ode_simple.cpp)

```
// solve y'(t) = -t*y ,  condition y(0)  = -2
// g++ -std=c++11 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 ()
{
  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
  double y[1] = {-2.0};              // initial value; table with one entry

  gsl_odeiv2_system sys = {func, nullptr, 1, nullptr}; // not using a Jacobian
  auto driver = gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-13, 1
     e-13, 0.0);

  output(t0,y[0],y[0]);
  for (int i = 0; i<n; ++i) {
    auto t = t0+i*dt; // begin of t interval
    if(gsl_odeiv2_driver_apply (driver, &t, t+dt, y) != 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;
}
```

The next one is more involved, it's directly from the GSL manual. The 2nd order, nonlinear Van Der Pol oscillator equation is

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

The numerical solver is for *1st order* equations, so we split this 2nd order equation to two coupled 1st order equations. Define a new variable y,

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

An n:th order differential equation is solved as n coupled 1st order differential equations

The code solves two unknown functions $\{x(t), y(t)\}$ point by point, starting from initial values at $t = 0$. Once a point $\{x(t), y(t)\}$ has been solved, the derivatives in that point can be computed and the next point $\{x(t+dt), y(t+dt)\}$ can be solved, and so on.

The Van Der Pol oscillator position $x(t)$ can have very sharp turns for some values of μ. Near sharp turns we apply ***adaptive stepsize***: In order to maintain numerical accuracy, the algorithm takes shorter steps in t. [21]

[21] If this is neglected, the solution goes astray after every steep turn. It will follow *a* solution curve, but not the one we started with, until after the next turn it picks yet another new curve. Since the numerical accuracy is limited anyhow, you can be sure this happens sooner or later in t.

The biggest challenge is bookkeeping. We have (i) the math on paper, (ii) 1st order differential equation notation in GSL and (iii) the notation in GSL solver. The GSL manual uses this general notation:

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

Don't spend too much time studying this, but tabulated, the three notations are related like this:

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]`

Implementation:

- The function `func()`(4 top rows of the table) computes $\{x'(t), y'(t)\}$ from known values $\{x(t), y(t)\}$. The results are stored to table `f`, elements {`f[0],f[1]`}.
- The function `jac()` (4 bottom rows of the table) hold the Jacobi matrix. This information is used only in higher order solvers – the more knowledge, the less function evaluations. A twist to bookkeeping: `jac()` is supposed to fill a 1-dimensional table `dfdy`, where the Jacobi matrix is stored as $\mathtt{m}[\mathtt{i},\mathtt{j}] = \mathtt{dfdy}[\mathtt{i} + \mathtt{dimensio} * \mathtt{j}]$. Hence the odd calls to `gsl_matrix_*`. If you find this hard to follow, fill `dfdy` as you please. A plain `for` loop is not a bad idea.

Example 56: (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;
}
```

GSL has ***driver functions*** in the header `gsl_odeiv2.h`, which are much easier to use than the basic functions in the header `gsl_odeiv.h`. Whichever, some preparatory steps need to be done.

- Choose the integration algorithm, that is, how the next point $\{x(t+dt), y(t+dt)\}$ is computed. Plenty of choice for this *stepper* here: `rk2, rk4, rkck, rk8pd, rk2imp, rk4imp, bsimp, rk1imp, msadams` and `msbdf`.
- Choose the control criterion, that is, when are shorter steps needed. For example absolute error 10^{-6}, relative error 0
- GSL wants all information about the problem at hand in one data structure of type `gsl_odeiv2_system`:

```
gsl_odeiv2_system sys = {func, jac, 2, &mu} ;
```

  This collects information about the functions, their derivatives, Jacobi matrix, order and the parameter(s). Any number of parameters are passed as a pointer
  `void * params`.
  This naked pointer doesn't let the compiler check your data properly, so better be careful.

A GSL driver function is set up [22]

```
auto solver =
       gsl_odeiv2_driver_alloc_y_new (&sys, gsl_odeiv2_step_rk8pd, 1e-6, 1e-6, 0.0);
```

and it's called later like this

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

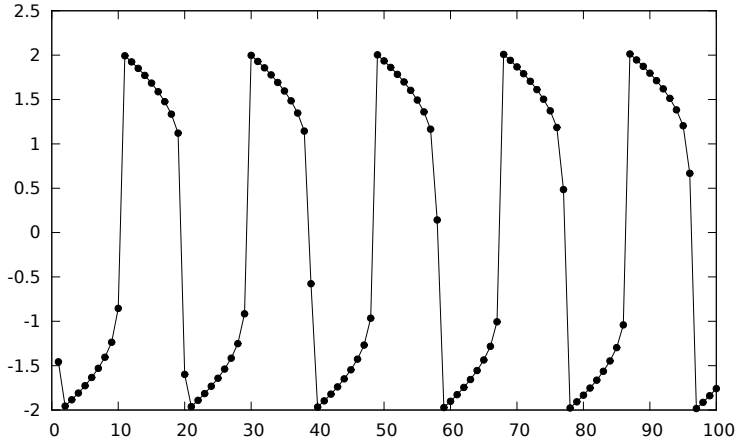
Passed values are the algorithm `solver`, start point `t`, end point `ti` and initial values `y`. The solution at time `ti` is in the return value of `y`.

[22] There are four drivers to choose from.

Set initial values and find the solution between t=0...100.

Example 57: (numerics/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};
  auto solver =  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 (solver, &t, ti, y);
    if (status != GSL_SUCCESS) {
      printf ("error, return value=%d\n", status);
      break;
    }
    // C printf is here a lot shorter than C++ iomanip would be
    printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);
  }
  gsl_odeiv2_driver_free (solver);
  return 0;
}
```



Van Der Pol oscillator with $\mu = 10$. The line is just a linear interpolation between solved points.

22.4 GSL: interpolation

It's convenient to hide the GSL-specific parts to a header. I chose to use `std::vector` for data. The example recognizes two spline types, not very ambitious :^)

- ***"cspline" (natural cubic spline)***
  Changes in one point causes non-local changes to the spline → unstable
  "Natural" refers to setting end point derivatives to zero.
- ***"akima" or "Akima" (natural Akima spline)***
  Changes in one point causes only local changes to the spline → stable

Steps:

1) Reserve space for accelerator (speeds up searches)

```
auto accel = gsl_interp_accel_alloc() ;
```

2) Reserve space for interpolation

```
gsl_spline spline = gsl_spline_alloc(gsl_interp_cspline, ...);
```

3) Initialize the interpolation with known points (x, y) (now in a `std::vector`):

```
gsl_spline_init(spline,&x[0],&y[0],x.size()) ;
```

4) Compute the interpolated y values to `yy` (iterator `posy`) at points `xx` (iterator `posx`)

```
*posy++ = gsl_spline_eval(spline, *posx, acc);
```

If needed, the 1st and 2nd derivative could be computed:

```
... = gsl_spline_eval_deriv(...);
... = gsl_spline_eval_deriv2(...);
```

Example 58: (numerics/gsl_spline.hpp)

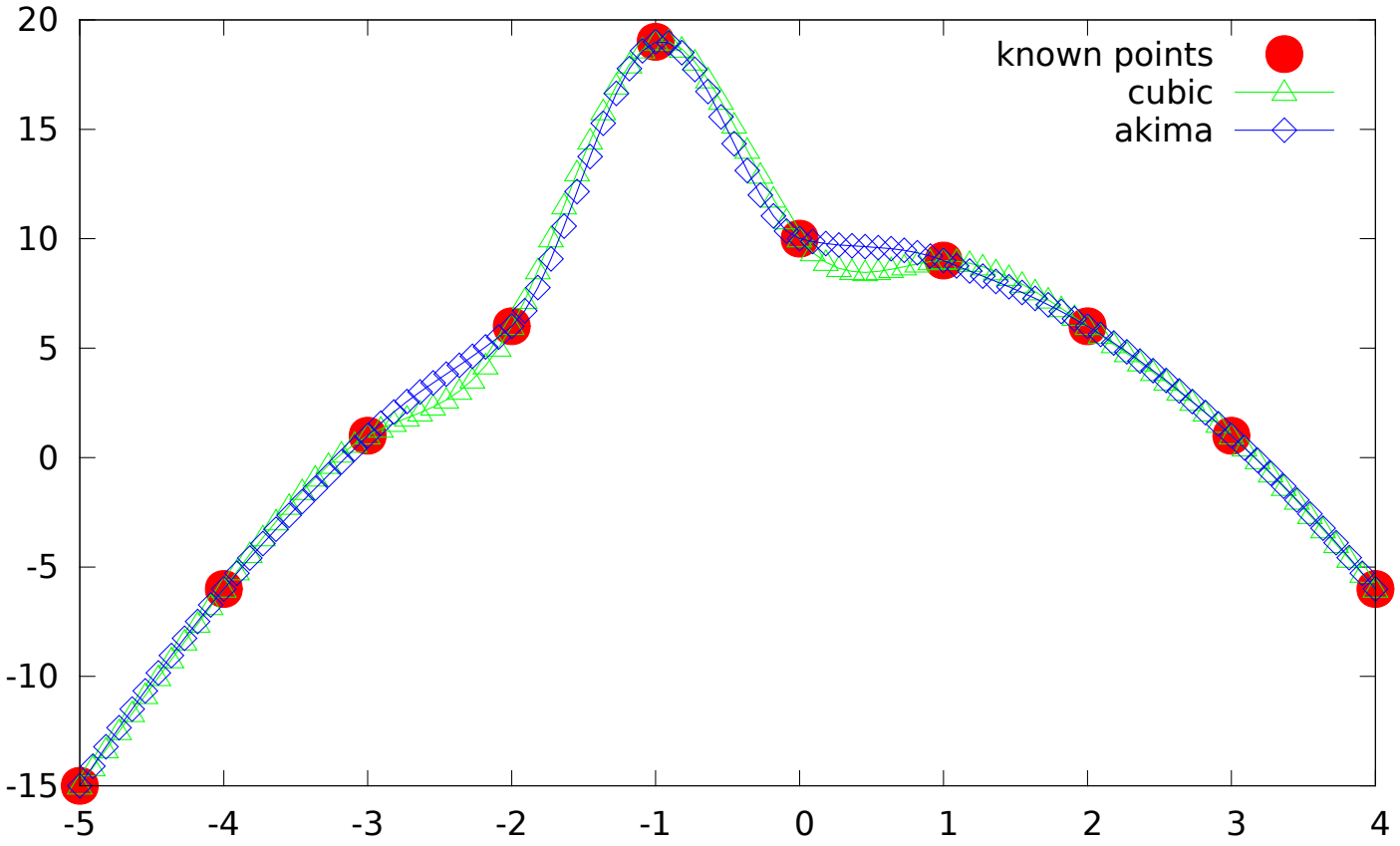
```
// wrapper to GSL spline
// spline type "cspline" is natural cubic spline
// spline type "akima" or "Akima" is natural Akima spline
#ifndef GSL_SPLINE_HPP
#define GSL_SPLINE_HPP
#include <iostream>
#include <cstring>
#include <vector>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_spline.h>

namespace my
{
  using namespace std;
  using vec= vector<double>;

  void spline(const vec& x,const vec& y, vec& xx, vec& yy, const string& type){
    auto acc  = gsl_interp_accel_alloc();
    gsl_spline* spline;
    int choose=0;
    if(type.compare("akima")==0 || type.compare("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());
```

```
    auto posy = yy.begin();
    for(auto posx = xx.begin() ; posx != xx.end() ; ++posx){
      *posy++= gsl_spline_eval(spline, *posx, acc);
    }
    gsl_spline_free(spline);
    gsl_interp_accel_free(acc);
  }
}
#endif
```

Example 59: (numerics/gsl_spline.cpp) Test code for GSL spline header

```
// spline test g++ -std=c++11 gsl_spline.cpp `gsl-config --libs`
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>
// home made call to GSL spline
#include "gsl_spline.hpp"
using namespace std;
typedef vector<double> vec;

void output(const vec& x, const vec y)
{
  cout<<fixed<<setprecision(8);
  for(auto i=0;i<x.size();++i)
    cout<<x[i]<<" "<<y[i]<<endl;
  cout<<"\n\n\n";
}

int main(){
  vec x(10),y(10); // known points (x,y)
```

```
  double t;
  t=-5.0;
  for(unsigned i=0;i<x.size();++i){  // you could use std::generate
    x[i] = t;
    y[i] = 10.0-t*t;
    t += 1.0;
  }
  y[4] += 10.0; // lift one point up to demonstrate locality/nonlocality
  output(x,y);

  // interpolated points 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
  my::spline(x,y,xx,yy,"cspline");
  output(xx,yy);

  // interpolate using natural Akima spline
  my::spline(x,y,xx,yy,"Akima");
  output(xx,yy);
  return 0;
}
// a.out>tt
// gnuplot
// p 'tt' i 0 w p pt 7 ps 1.5 t'known points','' i 1 w lp pt 8 t'cubic','' i 2 w lp
   pt 12 t'akima'
```


known points
cubic
akima
20
15
10
5
0
-5
-10
-15
-5
-4
-3
-2
-1
0
1
2
3
4

22.5 GSL: Monte Carlo integration

Let's compute an N dimensional integral over a hypercube,

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

GSL offers three ways (see Wikipedia Monte Carlo sampling)

1) Plain - sample random points inside thye hyper cube; a very crude method

2) MISER - (Press & Farrar) stratified sampling algorithm; sample points from areas that give largest error

3) VEGAS - (Lepage) combines both stratified sampling and importance sampling (sample points from areas that affect the result the most)

The example code evaluates the integral

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

Prerequisites:

- Random number generator
- Integrand:

```
 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.
```

The integrand is passed as a function pointer. ROOT (CERN C++ package) includes, among plenty of other things, Monte Carlo integration and a wrapper to pass a function object to GSL.

Example 60: (numerics/gsl_monte.carlo.cpp)
(see http://www.gnu.org/software/gsl/manual/html_node/Monte-Carlo-Examples.html)

```
// g++ -std=c++11 -Wall gsl_monte.carlo.cpp `gsl-config --libs`
#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";
  auto 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";
  auto 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";
  auto 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)
{
  static const 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;
}
```

Output:

```
==================================================
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
```

23 Physics example: Spin-1/2 Heisenberg chain

The Hamiltonian of a spin-1/2 chain has spin-spin interactions of two nearest neighbour spins,

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

Here operators S^x and S^y, the operators measuring spin x and y components, respectively, were combined to rising and lowering, a.k.a ladder, operators $S^{\pm} = S^x \pm iS^y$. The basis states used here are $|s, m\rangle$, which are the eigenstates of S^2 (spin squared),

$$S^2|s, m\rangle = s(s+1)|s, m\rangle$$

and the eigenstates of S^z

$$S^z|s, m\rangle = m|s, m\rangle \ .$$

Units are chosen so that $\hbar \equiv 1$. Rising and lowerin spin z component works like this:

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

The Heisenberg chain has spin 1/2 and the states are

$$|\frac{1}{2}, \frac{1}{2}\rangle = |\uparrow\rangle$$
$$|\frac{1}{2}, -\frac{1}{2}\rangle = |\downarrow\rangle \ .$$

The only difference is in the spin z component (m), so we keep track on that only. The system state is a chain of spin up and down values. The ladder operators change the state,

$$S^+|\uparrow\rangle = 0; \quad S^+|\downarrow\rangle = |\uparrow\rangle$$
$$S^-|\uparrow\rangle = |\downarrow\rangle; \quad S^-|\downarrow\rangle = 0 \ .$$

Obviously there are 2^N possible spin combinations for a chain N spins long. These can be economically coded in binary,

$$
\begin{aligned}
|\downarrow,\downarrow,...,\downarrow,\downarrow\rangle &= |00...00\rangle = |0\rangle \\
|\downarrow,\downarrow,...,\downarrow,\uparrow\rangle &= |00...01\rangle = |1\rangle \\
|\downarrow,\downarrow,...,\uparrow,\downarrow\rangle &= |00...10\rangle = |2\rangle \\
|\downarrow,\downarrow,...,\uparrow,\uparrow\rangle &= |00...11\rangle = |3\rangle \\
&... \\
|\uparrow,\uparrow,...,\uparrow,\uparrow\rangle &= |11...11\rangle = |2^N-1\rangle
\end{aligned}
$$

Next write the Hamiltonian in the basis of these states as a matrix. The matrix elements between states a and b are

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

We impose periodic boundary conditions and tie the chain to a loop, so that if $i = N$, then $j = i + 1 = 1$. This can be done neatly using the modulo operation, $j = mod(i + 1, N)$ or in C++ `j=(i+1)%N`.

The Hamiltonian matrix has plenty of zeros. The non-zero elements H_{ab} are only the following:

1) Term $\langle a|S_i^z S_j^z|b\rangle$: states must be the same, $a = b$.
Thus $H_{ab} = H_{aa}$ and this happens in four cases:

$$\text{case 1: } H_{aa} = \langle ...0_i...0_j...|S_i^z S_j^z|...0_i...0_j...\rangle = (-\frac{1}{2})(-\frac{1}{2}) = \frac{1}{4}$$
$$\text{case 2: } H_{aa} = \langle ...0_i...1_j...|S_i^z S_j^z|...0_i...1_j...\rangle = (-\frac{1}{2})(\frac{1}{2}) = -\frac{1}{4}$$
$$\text{case 3: } H_{aa} = \langle ...1_i...0_j...|S_i^z S_j^z|...1_i...0_j...\rangle = (\frac{1}{2})(-\frac{1}{2}) = -\frac{1}{4}$$
$$\text{case 4: } H_{aa} = \langle ...1_i...1_j...|S_i^z S_j^z|...1_i...1_j...\rangle = (-\frac{1}{2})(-\frac{1}{2}) = \frac{1}{4}$$

2) Term $\langle a|\frac{1}{2}(S_i^+ S_j^-)|b\rangle$: The only non-zero element from this term is

$$\text{case 5: } H_{ab} = \langle ...1_i...0_j...|\frac{1}{2}(S_i^+ S_j^-)|...0_i...1_j...\rangle = \frac{1}{2} \ .$$

3) Term $\langle a|\frac{1}{2}(S_i^- S_j^+)|b\rangle$: The only non-zero element from this term is

$$\text{case 6: } H_{ab} = \langle ...0_i...1_j...|\frac{1}{2}(S_i^- S_j^+)|...1_i...0_j...\rangle = \frac{1}{2} \ .$$

Now we are ready to compute!

Algoritm:
Go through all states a.
Cases 1 and 4: Set $H_{aa} = \frac{1}{4}$ if $b = a$ and the bits i and j in states a and b are equal
Cases 2 and 3: Set $H_{aa} = -\frac{1}{4}$ if the bits i and j are not equal in states a and b
Cases 5 and 6: Set $H_{ab} = \frac{1}{2}$, if flipping spins i and j in state a gives state b **Pseudocode**

Code: A. Sandvik (Boston University).

```
H=0 // set all elements of H to zero
do a=0...2^N-1   // N-body states a
  do i=0...N       // spin i
    j = (i+1)%N  // j=i+1 periodic boundary conditions  (if i=N, j=1)
    if ( a[i]=a[j] )
      H(a,a)=1/4
    else
      H(a,a)=-1/4
      b=flip(a,i,j) // flip bits i ja j in a to get state b
      H(a,b)= 1/2
     end if
  end do
end do
```

The Hamiltonian matrix is large, full diagonalization may be slow, but a few lowest eigenvalues can be found effectively using, e.g., the Lanczos algorithm (for example IETL has one).

The C++ Standard Library has a convenient container for binary data, `std::bitset`. Next choose a C++ matrix representation, such as Boost Hermitian matrix: [23]

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

```
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 , only Boost
  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]){  // or: if(((a & 1<<i)>>i) == ((a & 1<<j)>>j)) {
        H(a,a) += 0.25 ; // only Boost
      }
      else{
        H(a,a) -= 0.25;
        bbit = abit;    // or: b = a^(1 << i | 1 << j);
        bbit.flip(i);
        bbit.flip(j);
        b = static_cast<int>(bbit.to_ulong());
        H(a,b) = 0.5;
      }
    }
  }
}
```

[23] The "`or:`" comments show how to do it using bitwise operators – funny, effective and unreadable for non-experts.

23.1 Add numbers to file names

Sometimes it's convenient to have file names contain numerical parameters. One option is to use `stringstream` objects, see `basic/stringstream_ex.cpp`. It's not convenient, because one has to dig a "C string" out of the `stringstream` object `s` using `s.srt().c_str()` [24] An easier way is shown below.

Example 61: (basic/string_to_filename.cpp) Open files `res_`, suffix is a real number.

```
// Principle: how to get a computed number to a file name
#include <iostream>
#include <fstream>
#include <math.h>
#include <string>
using namespace std;
int main()
{
  const string str = "res_";
  for (unsigned i=1;i<5;i++){
    ofstream out(str+to_string(cos(i)));
    out.close();
  }
}
/* opened files
res_0.540302
res_-0.416147
res_-0.989992
res_-0.653644
*/
```

[24] A valid file name has to end with null character ("\0") like a character string in C.

24 Lambda functions/expressions

I have already mentioned lambda functions or expressions in a few examples. They are nice. They are fast. A lambda function is compiled to a function object and easily *inlined*. Boost has it's own lambda's, parts of the ideas came to be in C++11.

Usage: Define a "temporary" function *on the spot*, exactly where it's used. It can remain unnamed.
Bonus: No class needed to get a function object. The lambda remains local and you can't misuse it elsewhere.

The brackets `[]` in a lambda are for capturing things from outside. Here is a list of what is captured and how:

```
[]        capture nothing
[x, &y]   capture x by value, y by reference
[&]       capture all external variables by reference
[=]       capture all external variables by value
[&, x]    capture x by value, all else by reference
[=, &x]   capture x by reference, all else by value
```

Example 62: (advanced/autolambda.cpp) A locally applied function `func`

```
//g++ -std=c++11 autolambda.cpp

#include <iostream>
using namespace std;

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

Example 63: (advanced/lambda1.cpp) Output a container and the cubes of the elements

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

using namespace std;

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

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

Note: `int n` is passed by value.

Example 64: (advanced/lambda2.cpp)

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

Tulostus:

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

Note: The elements are passed by reference to complex conjugation.

Example 65: (advanced/lambda4.cpp) Calculate the product of `std::vector` elements.

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

```
product = 6
```

Note: `prod` is captured by reference.

Example 66: (advanced/lambda5.cpp) Chop all elements below some limit to zero.

```
#include <iostream>
#include <vector>
#include <algorithm>
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);
   for(auto x:v) cout<<x<<"\n";
   const double limit=0.4;
   cout<<"chopped "<<vector_chop(v,limit)<<" values below "<<limit<<endl;
   for(auto x:v) cout<<x<<"\n";
}
```

25 openmp parallel programming

Your computer most probably has multiple cores or threads. `openmp` offers an easy way to parallelize loops, see, for example, `http://bisqwit.iki.fi/story/howto/openmp/`

Example 67: (numerics/openmp_ex.cpp) Openmp parallel loop; too small a task to get any speed-up, though.

```
// g++ -std=c++11 -pipe -O3 -march=native -fopenmp -mfpmath=sse -msse2 openmp_ex.cpp
#include <iostream>
#include <cmath>
#include <vector>
int main()
{
  using namespace std;
  vector<double> tab(200);
  const int N=tab.size();
#pragma omp parallel for
  for(int i=0; i<N; ++i) // threads have their own private i
    {
      tab[i] = sin(2*M_PI*i/N);
      //cout<<i<<endl; // will produce mess, but shows parallelism
    }
  //cout<<endl;
  for(auto t:tab) cout<<t<<" ";
  cout<<endl;

}
```

The ***preprosessor directive*** `#pragma omp parallel for` tells what to parallelize.

You need to read more from a better source, but here are some other pragmas:

```
#pragma omp parallel // Somewhere later comes a parallel computation
#pragma omp parallel num_threads(3) // parallel over 3 threads
#pragma omp for // following for-loop is parallel
#pragma omp for private(k) // each thread has it's own copy of k
#pragma omp for shared (m) // all threads use the same m
#pragma omp for ordered schedule(dynamic) // parts of for must happen in due order
#pragma omp ordered // following statement must be done in ordered fashion
```

A loop variable, such as `i` in the previous example, is automatically "private", meaning each thread get its own value of `i` from the openmp scheduler. The scheduler decides who computes what.

In linux, the number of thread is set to four usign ***environment variables***:

```
setenv OMP_NUM_THREADS 4    (csh or tcsh shell)
export OMP_NUM_THREADS=4    (bash shell)
```

These are for run-time only, you can change these as needed without re-compiling the code.

```
> export OMP_NUM_THREADS=2
> a.out
> export OMP_NUM_THREADS=16
> a.out
```

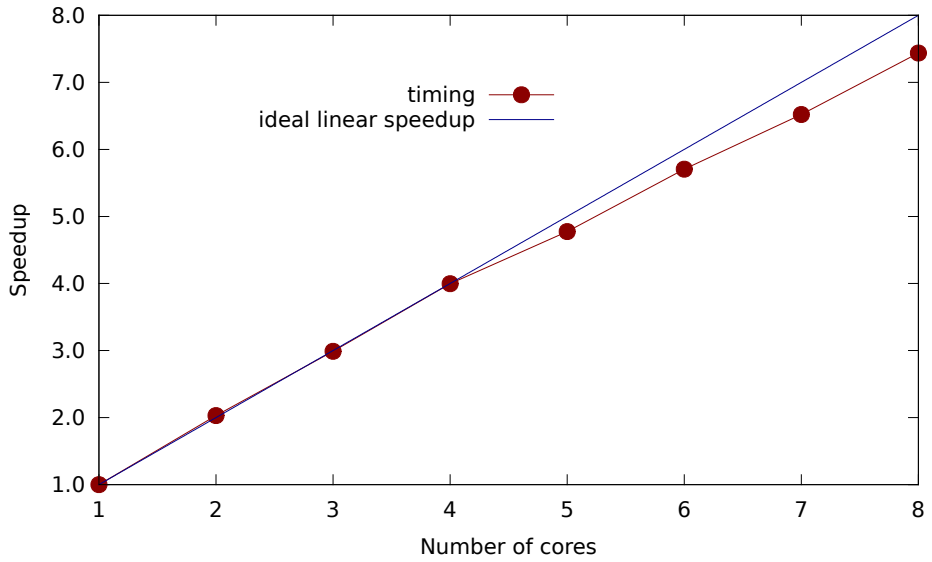
runs `a.out` first using two threads (cores) and then using 16 cores.
The function `omp_set_num_threads(16)` does the same, overriding OMP_NUM_THREADS. See the next example.

OMP_NUM_THREADS and `omp_set_num_threads()` are only suggested maximum number of threads.

Example 68: (numerics/openmp_reduction.cpp) `for`-loop reduction with timing.

```
#include<iostream>
#include<vector>
#include<cmath>
#include<fstream>
#include <omp.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
int main()
{
  const int n=100000000;
  ofstream fileout("timing");
  double time,reftime;
  for(int nc=1; nc!=5; nc++) {
    omp_set_num_threads(nc); // override OMP_NUM_THREADS
    double sum=0;
    auto t0 = high_resolution_clock::now();
#pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < n; i++) sum+=1/cos(i);//thread sums put together in the end
    auto t1 = high_resolution_clock::now();
    auto d = duration_cast<milliseconds>(t1-t0);
    time = d.count()/1000.0;
    cout<<"sum = "<<sum<<"   cores "<<nc<<"  timing "<<time<<" s"<<endl;
    if(nc==1) reftime = time;
    fileout<<nc<<" "<<time<<" "<<reftime/time<<endl;
  }
  fileout.close();
}
```

numerics/openmp_reduction.cpp: Timings on AMD FX-8350 Bulldozer eight-core processor.



cores	time (seconds)	speed-up factor
1	4.336	1
2	2.135	2.0309
3	1.451	2.9882
4	1.085	3.9963
5	0.908	4.7753
6	0.760	5.7052
7	0.665	6.5203
8	0.583	7.4373

The apparent super-optimal speed-up for two cores is due to inaccurate one-core timing used as a reference.

OpenMP has some overhead, don't expect speed-ups for short tasks.

26 Tips and tricks

Here `:::` means "some lines of code". The triple dot "..." has a C++ meaning, it's an "ellipsis" :)

- How to pause execution:

```
cout<<"PRESS ENTER TO CONTINUE\n";
cin.ignore();
```

- Infinite loops can be typed as `while(true) {:::}`, or more to the point, as

```
#define ever (;;)
:::
for ever
{
 // break to get out
 :::
}
```

  I heartily recommend *stackoverflow.com* discussions. ⇐ ***This is THE tip!***
  As what comes to having fun with C++, this one is absolutely priceless:
  what-is-the-worst-real-world-macros-pre-processor-abuse-youve-ever-com .

- (Linux) Long compiler messages can be piped to `more` (or `less`); Notice the "&" after the pipe character

```
g++ -std=c++11 program.cpp |& more
```

- (Linux) It's tedious to type `g++ -std=c++11 ...` all the time. Write a makefile, use an IDE, or set an alias,
  `alias g++='g++ -std=c++11'` (bash, put this line to $HOME/.bashrc and type ". $HOME/.bashrc")
  `alias g++ 'g++ -std=c++11'` (csh or tcsh, put this line to ${HOME}/.cshrc and type "source ${HOME}/.cshrc)
  and you only need `g++ program.cpp`

- `std::numeric_limits` (from `<limits>`) tells what traits a type has – hence they are "type_traits" For example, `std::numeric_limits<int>::max()` gives 2147483647. If you have in mind a class that can use an optimized algorithm and another related one that cannot, you can give the objects a trait. Based on that trait, the code can automatically decide if the optimized or the default algoritm is applied. See `advanced/traits.cpp`.

27 Some more C++ in the net

- Genetic algorithms are used for optimization
  Genetic-Algorithm-Library
- Bartlomiej Filipek gives examples how to use the standard algorithms instead of raw loops:
  Top-Beautiful-Cplusplus-std-Algorithms-Examples

28 Farewell words for C++ numerical programmers

- Prefer C++ Standard Library ***containers*** over dynamic arrays. Don't transform one to another.
- For math matrices and vectors, use ***Armadillo, MTL4, Blaze ...***, don't code them yourself.
- Use ***C++ Standard Library algoritms***, not loops
- ***Forget type safety***, it's the least of C++ numerics problems.
- ***Move, don't copy.*** For random number generators, use `std::ref()` in `std::generate`.
- Pass functions as ***function objects***, ***lambda functions*** or use `std::function`. Avoid pointers to functions.
- ***Keep the "daily stuff", the interface code, clean.*** Hide ugly details (library calls) to headers.

```
fft(data,1,n); // overloaded fft(), library call in a header
gsl_fft_complex_radix2_forward(data,1,n); // Same job. This is why I don't use C
```

- Use ***namespace protection*** for you own functions: `my::get_data()`
- ***Templates are you friends.***
- ***Use operator overloading to make cleaner code.*** Test all aspects.

```
outfile << myobject; // overloaded <<, more readable than a function call
A = (B+M)*c ; // overloaded * and +, B,M complex matrices,c real.
A = multiply_complexmat_realvec(sum_complexmat_complexmat(B,M),c); //C? Oh no!
```

- Make data types short and descriptive with `typedef` or `using`.

```
   using my_cvec = lib1::part3::section4::vector<complex<double>> ;
```

- ***A fast algorithm in a slow computer computes faster than a slow algorithm in a fast computer.***

Those are my principles, and if you don't like them... well, I have others.
Groucho Marx