

Short test of C++ knowledge

- What is a class?
- What is virtual function?
- What is template?
- What means explicit?

```
class A{
    int a;
public:
    explicit A(int a);
};
```

- What is implicit type conversion?
- What is return value optimization?
- What is the difference between "new" "new[]" and "operator new"?
- What is placement new?

```
class A;
void* memory = operator new(size);
a = new(memory) A;
```

- What is map?
- What is deque?

Programming in high-level languages

There is no "perfect" or "best" computer language. The choice strongly depends on problem one is solving.

For numerics, **speed** is of utmost importance:

- fastest: Fortran77 (no aliasing)
- fast: Fortran90, C, C++
- not so fast: Java, any interpreter (Perl, Python)

(aliasing: concurrent acces to a single entity causes synchronization problems and results in performace loss).

Human time necessary to write and mantain large scale projects:

- C++, Java, Python
- C, Fortran 90
- Fortran77

For some purposes like manipulating files, writing small scripts or working with strings of text (parsing) none of the above languages is a good choice. In this case, **interpreters** are preferred: Perl, Python,...

Historical overview on computer language development can be found at

http://en.wikipedia.org/wiki/Timeline_of_programming_languages

There are thousands of programming languages and new ones are created every year.

Compilers: Fortran, C, C++, Pascal, (Java), ...

Interpreters: Perl, Python, Mathematica, (Java), ...

Compilers: Code needs to be compiled and translated from the original language into machine-code (assembler). Machine code in form of an executable can be run. When you run the program, the computer executes the machine-code instructions in sequence.

- advantage: The machine-code can be very optimized and execution is fast.
- disadvantage:

– The original program is no longer connected to the machine-code, and mapping the

machine-code back to the original program can be very difficult (hard job for debugger, hard to connect run-time errors with the bugs in the original code)

- Each processor has a different machine code and thus requires a completely different compiler.
- The code can-not be changed on fly, much less flexible and usually harder to code.

Interpreters: Converts each line of a program into machine language as the statement is encountered. If a statement is encountered multiple times (as occurs in a loop) the machine must convert it to machine language each time.

- advantage:
 - Easy to detect bugs because original program is still closely related to execution.
 - No separate executable file need be stored and the same code usually runs on many platforms
 - Usually offer more flexibility and provide more powerful statements (regular expressions in Perl or Python, Simplify,Solve,... in Mathematica,...)
 - Code of the original program can be changed on the fly (can be very powerful)
- disadvantage: speed - Translating the same line of code into machine-code and then

executing the machine-code has quite a high overhead.

Usual strategy for commercial products is to separate the product into few layers:

- **low-level** routines (usually C, sometimes Assembles, for numerics Fortran)
- **core** (usually C++)
- **user interface** (tools producing html, sometimes java)

Most of the time, we will work on **core** part of numerical algorithms. Low level routines will be used (LAPACK,BLAS,root finding routines,...) and user interface is not going to be addressed. The programming language or the choice is therefore C++.

Object oriented designed is going to be empasized.

C++ in short

Many excellent tutorials on the WEB

- <http://cplus.about.com/od/beginnerctutorial/#more>
- <http://www.cprogramming.com/tutorial.html#c++tutorial>
- <http://www.cplusplus.com/doc/tutorial/>
- <http://www.icce.rug.nl/documents/cplusplus/>
- <http://www.josuttis.com/libbook/idx.html> (STL)
- <http://www.oonumerics.org/oon/> (OO-numeric)
- <http://cpplapack.sourceforge.net/> (CPP LAPACK)

The best way to learn is to actually write code.

1 Hello World

The course is taken from <http://cplusplus.about.com/od/beginnerctutorial>

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello World!\n";
}
```

Line 1 : `#include <iostream>`

As part of compilation, the C++ compiler runs a program called the C++ preprocessor. The preprocessor is able to add and remove code from your source file. In this case, the directive `#include` tells the preprocessor to include code from the file `iostream`. This file contains declarations for functions that the program needs to use, as well as available classes.

Line 2 : `using namespace std`

C++ supports the concept of name spaces. Essentially, this allows variables to be localized to certain regions of code. The command `using namespace std` allows all objects and functions from the standard input and output library to be used within this program without explicit qualifications.

Line 3 : `int main()`

This statement declares the main function. A C++ program can contain many functions but must always have one main function. `int` specifies the return type of `main` to be an integer. An explicit value may be returned using a return statement. In this case, 0 is returned by default.

Line 4 : {

This opening bracket denotes the start of the scope.

Line 5 : `cout<<"HelloWorld!\n" ;`

`cout` is a object from a standard C++ library that has a method used to print strings, numbers,... to the standard output, normally your screen. The `\n` is a special format modifier that tells the method to put a line feed at the end of the line. If there were another `cout` in this program, its string would print on the next line.

Line 6 : }

This closing bracket denotes the end of the program.

2 Variables and Constants

A variable represents a location in your computer's memory. You can put data into this location and retrieve data out of this location. Every variable has two parts, a name and a data type.

Unlike in Fortran, variable names are **case sensitive**.

Some example of data types are

- int: basic integer number of size 4 bytes on 32 bit systems with a precision of 10 digits
- double: real number of size 8 bytes
- float: real number of size 4 bytes
- complex: is not a simple data type but is part of STL (complex<double>).
- user defined types, typically classes. Most of variables are classes in OO code.

Remember: Choose long variable names that tell you something about the purpose of the variable.

Never choose similar names for variables because miss-taping mistakes will not be detected by compiler.

Example of declaration

```
int count;  
int number_of_students = 30;
```

This statements can appear anywhere in the program not only at the beginning.

It is a good practice to define variables close to the place where they are used especially if their primary purpose is being temporary variables. For example to swap two numbers

```
int temp=b;  
b = a;  
a = temp;
```

It is much easier to read and understand the code if most of variables are very localized and they are declared and go out of scope few lines later (are destroyed).

Remember: Try to make variables localized (Make a short range interaction problem rather than long range interaction problem)

In the old days of C, strings were simple C arrays of characters (character pointers):

```
char firstInitial = 'J';  
char name[] = "John";
```

Nowadays, one can use much more flexible and useful class string from STL

```
std::string firstInitial("J");  
std::string name("John");
```

Constants in old days of C were usually defined with preprocessor directive

```
#define pi 3.1415
```

This should be avoided in C++ and replaced by 'const double'

```
const double pi = 3.1415;
```

because debugger will know what `pi` is in the second case, but not in the first case.

The statement `const` is used also to emphasize that this variable is not going to be changed - can not be changed. Compiler will issue error if one tries to change it.

This is very usefull in declaring functions and classes. For example

```
double Circumference(const double& r){  
    return 2*M_PI*r;  
}
```

tells the reader that variable `r` will not be changed within the function. If function size is large, it is very usefull to see in declaration which variables are changed and which are not inside the function.

Remember: It is highly recommended to use `const` whenever possible.

3 Input and Output

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    int ID;
    cout << "Enter your name ";
    cin >> name;
    cout << "Enter your ID number ";
    cin >> ID;
    cout << "Hello " << name << " or should I say " << ID << endl;
    return 0;
}
```

The input operator, `>>`, is used to put data into variables in your program. It is also called the insertion operator. The output operator, `<<`, is used to direct output to standard output or standard error. It is also called the extraction operator. These operators also work with other streams including files.

- `cin` - This object provides for input from the terminal (keyboard)
- `cout` - This object provides for output to the screen.
- `cerr` - This object provides unbuffered output to the standard error device, which defaults to the screen. Unbuffered means that any messages or data will be written immediately. With buffered input, data is saved to a buffer by the operating system, transparently to your program. When the buffer is full, everything in it is written out. This is more efficient because each write

requires a certain amount of overhead from the operating system. Writing out one large buffer has less overhead than writing out multiple smaller messages. The downside is that if a program crashes before the buffer is written, nothing in the buffer is output. Output via `cerr` is unbuffered to ensure that error messages will be written out.

- `clog` - This object provides buffered output to the standard error device, which defaults to the screen.

To redirect in bash standard output to file `stdout` and standard error to `stderr`, one needs to execute

```
./myprogram > stdout 2>stderr
```

4 Conditional Processing

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

int main()
{
    int number = 5;
    int guess;
    cout << "I am thinking of a number between 1 and 10" << endl;
    cout << "Enter your guess, please ";
    cin >> guess;

    if (guess == number){
        cout << "Incredible, you are correct" << endl;
    }else{
        cout << "Sorry, try again" << endl;
    }
    return 0;
}
```

To compare two operands, relational operators are used:

- == : equal
- != : not equal
- > : greater than
- >= : greater than or equal
- < : less than
- <= : less than or equal

Instead of many if-else statements, it is convenient to use **switch** statement

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

int main()
{
    int choice;
    cout << "What flavor ice cream do want?" << endl;
    cout << "Enter 1 for chocolate" << endl;
    cout << "Enter 2 for vanilla" << endl;
    cout << "Enter 3 for strawberry" << endl;
    cout << "Enter 4 for green tea flavor, yuck" << endl;
    cout << "Enter you choice: ";
    cin >> choice;
    switch (choice) {
        case 1:
            cout << "Chocolate, good choice" << endl;
            break;
        case 2:
            cout << "Vanillarific" << endl;
            break;
        case 3:
            cout << "Berry Good" << endl;
            break;
        case 4:
            cout << "Big Mistake" << endl;
            break;
        default:
            cout << "We don't have any" << endl;
            cout << "Make another selection" << endl;
    }
    return 0;
}
```

Multiple conditions can use the following operators:

&&	AND
	OR
!	NOT

An example of multiple condition statement:

$\text{if}(x < 7 \ \&\& \ y > 50 \ || \ z < 2) \dots$

5 Looping

Most often used loop statement in C or C++ is `for` statement and is also the fastest.

```
for (int i = 0; i<100; i++) cout << i << endl;
```

First 100 integer numbers can be printed to standard output also with `do-while` or `while` statement in the following way or

```
int i=0;
do {
    cout << i << endl;
} while (++i<100);
```

or

```
int i = -1;
while (++i < 100){
    cout << i << endl;
}
```

The important difference between the first and the second statement is the execution order and time of test statement (and also increment statement). In the first case, it is performed after printing and therefore the statement is always executed even if the condition is never true. In the second case, the statement will never execute if the condition is never true.

Homework

Chose one of the following two coding problems:

- a) Write algorithm to search an ordered table
- b) Simulate the bug population growth and produce bifurcation plot. You will need to instal "plotutils" package to run the code.

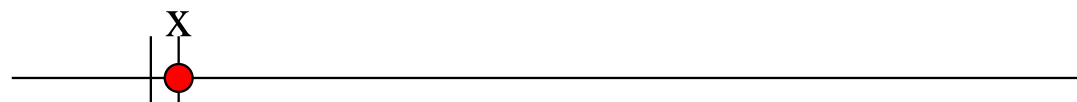
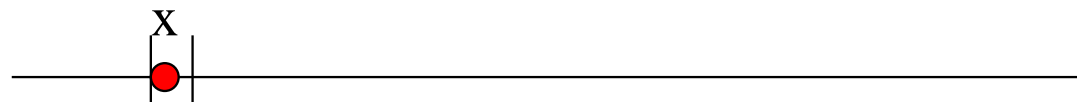
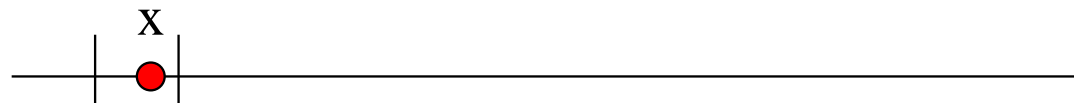
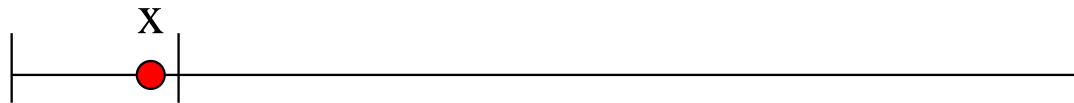
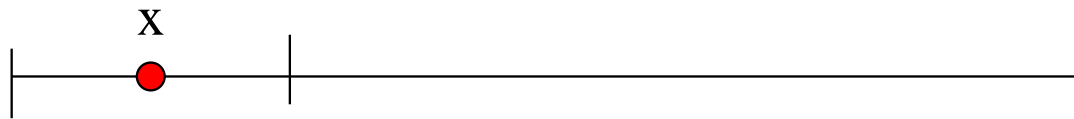
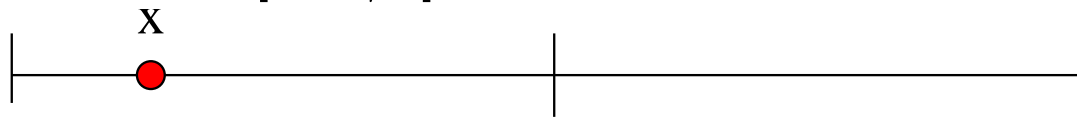
Searching an ordered table

- Write the bisection algorithm to search an ordered table
- Write optimized searching algorithm of an ordered table for the case of correlated lookups.

Giwen an ordered table of values ω_i and certain number x , we need to find index i such that the following is satisfied: $\omega_i < x < \omega_{i+1}$. In most of the cases, the bisection algorithm is most efficient and needs only $\log_2(N)$ lookups in the table, where N is the size of the array.

For the array of values with index $[0, N]$, the algorithm check if x is bigger or smaller than point $\omega_{N/2}$? If $x > \omega_{N/2}$, search proceeds in the range $[N/2, N]$, otherwise search

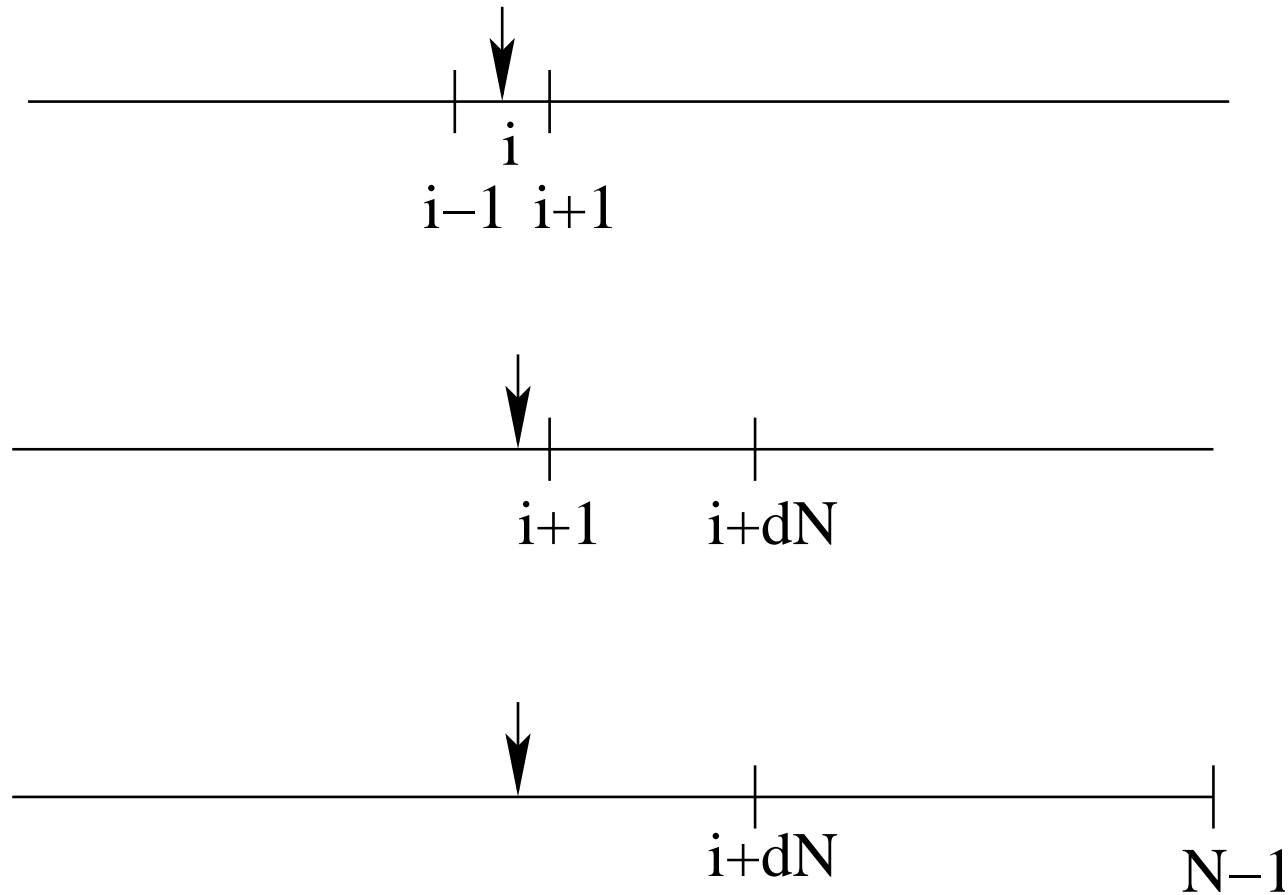
proceeds in $[0, N/2]$. The following figure explains the sequence of lookups and decisions.



If consecutive lookups x_k are correlated and most of lookups occur in a small region of the table (an example is interpolation of logarithmic grids, or, searching for zero of a tabulated function), it is advantageous to search in the immediate neighborhood of the previous point first. The probability that the new point x_{k+1} is close to the previous point x_k is very large. If the previous point x_k was between ω_i and ω_{i+1}

- Check whether the point is smaller or bigger than the previous ω_i . If it is bigger, proceed as follows
 - Check if the new point is between $\omega_i < x < \omega_{i+1}$
 - In the next step check if x is in the range $\omega_{i+1} < x < \omega_{i+dN+1}$ where dN is some number of the order of 10. If yes, use bisection in this small interval.
 - If the above fails, apply bisection in the interval $[i + dN + 1, N]$
- If the point x_k is smaller than ω_i , do the following
 - Check if the new point is between $\omega_{i-1} < x < \omega_i$
 - In the next step check if x is in the range $\omega_{i-dN-1} < x < \omega_{i-1}$ where dN is some number of the order of 10. If yes, use bisection in this small interval.
 - If the above fails, apply bisection in the interval $[0, i - dN - 1]$

The following figure demonstrates the idea
previous



The skeleton of the program is given below. You need to fill in few lines of the algorithm in the function `bisection` and `find_ordered`.

```
#include <iostream> // for printing
#include <cstdlib> // for drand48
#include <algorithm> // for sort routine

inline int bisection(double x, const double om[], int ia, int N)
{ // Bisection in the interval between ia and N
  // looking for index i such that om[i]<x<om[i+1]
  // We know that the search is necessary only in the region om[ia]<x<om[N-1]!
  // Be careful that 0<i<N! Otherwise coredump might occur.
  .....
  return ...;
}

const int dN=10;

inline int find_ordered(double x, double om[], int ia, int N)
{ // The optimized search for correlated lookups
  // Index ia is the index from the previous lookup. Most probably,
  // the new point x is close to om[ia].
  .....
}

int main()
{
  using namespace std;

  const int N=10000; // size of array is fixed at this value
  double om[N]; // real array of numbers is initialized

  srand48(time(0)); // random number generator from cstdlib is initialized to unique value

  for (int i=0; i<N; i++) om[i] = drand48(); // array is filled with random numbers
```

```
sort(om, om+N); // array is sorted using STL call

// Testing the first part of the assignment
{
    double x0=0.5;
    int ia = bisection(x0, om, 0, N);
    cout<<"The number preceding "<<x0<<" is "<<om[ia]<<" and following "<<om[ia+1]<<endl;
}

// Testing the second part of the algorithm
{
    int M=100*N; // testing sufficiently large number of casses to find possible bugs
    int ia=0;    // start searching table at the beginning in the absence of knowledge
    for (int i=0; i<M; i++){
        double x = drand48();           // random number to be searched in the table
        ia = find_ordered(x, ia, om, N); // actual searching routine

        if (x>om[0] && x<om[N-1]){      // interpolation possible only if number is inside the interval
            // otherwise extrapolation needs to be used
            if (!(om[ia]<x && x<om[ia+1])) // if you see the following message, the code does not yet work!
                cerr<<"Failed at point "<<om[ia]<<" "<<x<<" "<<om[ia+1]<<endl;
        }
    }
}
return 0;
}
```

Bug population and bifurcation plot

Imagine a bunch of insects reproducing generation after generation. We start with N_0 bugs, in the next generation we have N_1 and after i generations we have N_i of them.

The birth rate is proportional to the number of living bugs as long as their number is not too big. When they populate the entire region, the competition for a finite food supply tends to limit their number to maximum N_* . A plausible model for the bug population is

$$\frac{\Delta N_i}{\Delta t} = \lambda N_i (N_* - N_i) \quad (1)$$

To simplify units, we define

$$\mu = 1 + \lambda N_* \Delta t \quad (2)$$

$$x_i = \frac{N_i}{N_*} \frac{\lambda \Delta t}{\lambda \Delta t + \frac{1}{N_*}} \quad (3)$$

Now, $\mu > 1$ and $x_i \in [0 - 1]$. Bug population becomes

$$x_{i+1} = \mu x_i (1 - x_i) \quad (4)$$

We are interested in the bug population after a long period of time. The equation has two fixed points:

$$x_i = 0 \quad (5)$$

$$x_i = 1 - \frac{1}{\mu} \quad (6)$$

The fixed point is stable if

$$\left| \frac{dx_{i+1}}{dx_i} \right| < 1 \quad (7)$$

In case of our bugs, the derivative is $\mu(1 - 2x_i)$. The point of no bugs is stable for $\mu < 1$ (not possible in this case). For the second fixed point, the derivative is $2 - \mu$ and is therefore stable at $\mu < 3$.

To summarize:

- for $\mu \in [1..3]$ stable fixed point is $x_i = 1 - 1/\mu$.
- for $\mu > 3$ there is no fixed point. We can find points with double period (attractors or cycle points) by requiring $x_{i+2} = x_i$.

Bifurcation plot is obtained by plotting population of bugs x_i (versus growth rate μ) after a long period of time when it reaches a steady state.

The algorithm is:

- start with any bug population $x_i \in [0...1]$
- simulate few hundred bug generations to reach steady state
- after steady state is reached, plot next few hundred generations of bugs x_i on the screen at the point (x_i, μ) .
- scan value of μ between $[1...4]$ and try few different initial populations. The bifurcation should be independent of the initial population.

6 Pointers

Memory of a computer is enumerated and each variable is stored at certain unique address. Usually one is interested in the value of the variable which is the **content** of memory slot. The **address** of the same memory slot is called pointer to the variable.

Pointers provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages (Fortran). However, there is a drawback of this flexibility, called **aliasing** which slows down the execution of the program for 10%-20%.

Some operations are possible only by using pointers. However, in good C++ program, programmer **should almost never see** a pointer (or need to care how to manipulate pointers). This should be hidden inside basic classes which are very seldom reimplemented (for example vector, matrix, map,...). When they are, they should be heavily tested first, and only then used in writing numerical algorithms. In this way, most painful **memory fault bugs** are avoided (**Very serious Fortran90 problem**).

Remember: Pointers are most frequent source of program bugs. Avoid using them if not necessary.

Simple examples of pointer usage:

```
int count = 5;
int *ptr = &count // Stores the address of count.
int count_copy = *ptr; // pointer is dereferenced and the contents of memory
// is assigned to new variable
*ptr = *ptr + 2; // contents of memory is changed. count is now equal to 7.
// count_copy is still 5
ptr = ptr + 2; // ptr now points to new location 2*4=8 bytes after variable count.
*ptr = 3; // This might cause core dump. The contents of unallocated memory
// is changed. The results is unpredictable.
```

The most important use of pointers for us is in allocating memory for data structures: arrays, matrices,... But we, as users, do not necessary need to implement these structures. We can use them.

The usual static C arrays are defined in the following way

```
double A[100]; // vector of size 100
```

The problem of static C type of arrays is that the size needs to be known at compile time (just like fortran77). This is not convenient for most of algorithms.

Dynamic arrays are allocated with operator "new" in C++

```
double *A = new double[N];  
.....  
delete[] A;
```

Here N can be any integer number determined during execution. Don't forget to deallocate memory with delete every time using new. Number of new and delete statements should always exactly match.

It is much more convenient to use predefined class of type vector from STL. The same statement with vector is

```
vector<double> A(N);
```

User does not see pointers although they are used in this constructor. User does not need to deallocate memory either, because the class is destroyed automatically when variable A goes out of scope.

7 File Input and Output

```
#include <iostream>
#include <fstream> // Includes definitions of the file input-output streams
using namespace std;

int main()
{
    ofstream myFile("~/out.txt"); // Creates an output file stream object named myFile
    if (! myFile)                // Always test file open.
    {
        cout << "Error opening output file" << endl;
        return -1;                // Program is terminated
    }
    myFile << "Hello World" << endl; // Actual writting
    myFile.close();                // It is advisable to close the file in order that
                                   // computer flushes output
    return 0;                      // In most compilers, this is not really necessary
                                   // because destructor flushes output anyway.
}
```

Many options can be used for opening data files

- out - Open a file or stream for insertion (output).
- in - Open a file or stream for extraction (input).
- app - Append rather than truncate an existing file. Each insertion (output) will be written to the end of the file

- `trunc` - Truncate existing file (default behavior)
- `ate` - Opens the file without truncating, but allows data to be written anywhere in the file
- `binary` - Treat the file as binary rather than text. A binary file has data stored in internal formats, rather than readable text format. For example, a float would be stored as its internal four byte representation rather than as a string.

An example of opening output binary file for appending:

```
ofstream myFile("SomeFileName", ios::out | ios::app | ios::binary);
```

To combine modes, one needs to use symbol `|` ("or")

Need to know:

- By default, if a file does not exist it is created.
- By default, a file will be a text file, not a binary file.
- By default, an existing file is truncated.

One can read and write from the file in the same way as from standard input and output by `<<` and `>>`. There are many other very usefull advanced functions for reading and writting including `getline`, `ignore`, `get`, `peek`, `putback`, `read`. Check them out when you need them.

There exist so-called input-output manipulators which control how data is formatted. Most often, a precision of the output stream is changed (increased). The function `width` than needs to be used to take enough space between two real numbers for nicer output.

```
#include <iostream>
#include <iomanip> // manipulators defined
.....
cout.precision(16);
cout<<"variable a = "<<setw(25)<<a<<endl;
```

8 References

The real usefulness of references is when they are used to pass values into functions. They provide a way to return values from the function without using pointers.

But first we'll examine how to create and use references.

```
int val;           // Declares an integer
int &rVal = val;   // Declares a reference to the integer val
```

Notice the use of the "&" before the reference name. It indicates that rVal is a reference rather than an ordinary object. A reference is not a unique object. It is merely an alias or synonym for another object.

```
int val = 5;      // Declares an integer
int &rVal = val;  // Declares a reference to the integer val.
rVal = 6;        // val is now also 6 since rVal is actually the same variable as val
val = 7;        // val and rVal are both 7 now
```

Reference cannot be reassigned. It has to be assigned when declared (line two above) and from then on, the reference is always bound to the same variable. Pointer, on the other hand, can be reassigned and pointed somewhere else.

9 References in a function call

C++ has an important advantage over C because it supports function call by reference. In C, one needs to pass pointers instead. In fortran, on the other hand, there is no possibility to pass variable by value and everything is called by reference.

In C++ one has both choices for passing arguments to the function: call by value or call by reference.

This is call by reference

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

This is call by value

```
int max(int a, int b){
    return (a>b) ? a : b;
}
```

This could alternatively be written with constant references

```
int max(const int& a, const int& b){
    return (a>b) ? a : b;
}
```

Usual call by value is discouraged in this case since one could miss-use variables inside the function body

```
int max(int& a, int& b){
    a = 3;           // a is changed not only locally but also globally.
    return (a>b) ? a : b;
}
```

Call by value in the case of swap would not work. The following code does not modify variables a or b:

```
int swap(int a, int b){
    int temp = b;
    b = a;
    a = temp;
}
```

10 Object oriented programming

Wikipedia: *Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several techniques from previously established paradigms, including inheritance, modularity, polymorphism, and encapsulation. Even though it originated in the 1960s, OOP was not commonly used in mainstream software application development until the 1990s. Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support OOP.*

*Object-oriented programming's roots reach all the way back to the creation of the Simula programming language in the 1960s, when the nascent field of software engineering had begun to discuss the idea of a **software crisis**. As hardware and software became increasingly complex, how could software quality be maintained? Object-oriented programming in part addresses this problem by strongly emphasizing **modularity** in software.*

Creators of Simula programming language (Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo) were working on ship simulations, and were confounded by the combinatorial explosion of how the different attributes from different ships could affect one another. The idea occurred to group the different types of ships into different classes of objects, each class of objects being responsible for defining its own data and behavior.)

Object-oriented programming may be seen as a collection of cooperating objects, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

*Object-oriented programming is intended to **promote greater flexibility and maintainability** in programming, and is widely popular in large-scale software engineering. By virtue of its strong emphasis on modularity, object oriented code is intended to be simpler to develop and easier to understand later on, lending itself to more direct analysis, coding, and understanding of complex situations and procedures than less modular programming methods*

Classes are a software construct that can be used to emulate a real world object. It can be defined by its attributes and by its actions. (a cat has attributes such as its age, weight, and color. It also has a set of actions that it can perform such as sleep, eat and complain (meow).)

The difference between a class and an object can be confusing to beginners.

- A class is data type that encapsulates data and abilities.
- An object is a particular instance of a class.

In case of simple integer

```
int x;
```

Class is "int" and object is x.

The goal in designing classes is to have all the relevant attributes and abilities encapsulated by the class. Any class contains

- class members - its data
- class methods - its functions

Designing good classes is very challenging even for experienced programmers. The designing part of programming is the most important part and some effort should be devoted to carefully define the problem and choose "best" classes for the problem at hand.

What is a good choice of classes?

Classes should not be too big.

Classes should interact as little as possible

The interaction should be simple.

Choosing a "good" set of classes is like finding quasiparticles of an interacting many-body problem.

Example: Suppose we need to manipulate functions on a non-regular grid, lets say interpolation of a function at any point x and we store functions f_i at some frequencies ω_i . In addition, suppose we need to evaluate two or more functions at the same point x .

One choice of classes would be *function+grid*. The advantage is that there is *no interaction* when doing interpolation because both ω_i and f_i are inside the same class. Disadvantage is that the class is *large*. Second, usually there are *many functions* that share the *same grid* and there is no point to duplicate the grid.

If two separate classes are constructed: *"function"* class, and *"grid"* class, and linear interpolation is needed at point x
$$f(x) = f_i + (f_{i+1} - f_i)(x - \omega_i)/(\omega_{i+1} - \omega_i),$$
 how is class "function" going to access class "grid" to do the job?

One needs to allow communication between "function" and "grid". The straightforward implementation would be to connect function object with grid object when constructor of function is called, and keep a *pointer to grid inside the function*. In this way one opens a big window off possible interactions between function and mesh class and it is very hard for client to understand what function might do to mesh. The *interaction* between them through pointer can be very *strong* and not transparent.

There are many other possibilities like implementing member function for class function which would take grid object and floating point number x and return $f(x)$. A disadvantage is that user needs to be careful which functions and grids fit together. More importantly, this would not allow optimization in case two or more functions need to be evaluated at the same point.

One could also create a function which takes object grid and many objects of type function and returns interpolated values.

Another possibility which makes interaction much more transparent to client is by *exchange of a small class* (class that *mediates interaction*). In this way, we will allow only very *simple type of interaction* between function and grid and the two big classes are otherwise completely *independent entities*. The interaction mediating class contains only one integer i and one floating point number $(x - \omega_i)/(\omega_{i+1} - \omega_i)$. User thus gives point x to grid, which returns interaction class and interaction class can then be used on many functions to get value at point x . It is very common that one needs value of *many functions at the same point x* and in this way one does not need to recalculate the interpolation for each function separately.

Class interface should be complete and minimal.

A complete interface is one that allows clients to do anything they might reasonably want to do.

A minimal interface, on the other hand, is one with as few functions in it as possible, one in which no two member functions have overlapping functionality. Clients can do whatever they want to do, but the class interface is no more complicated than absolutely necessary.

Encapsulation: Users can see an object as a black box with a well-defined interface to a set of functions in a way which hides their internal workings.

Good choice of classes should have a natural syntax, an intuitive semantics.

They should be as simple to understand as standard types are, for example string, vector,...

One usually discovers that the choice of classes **was bad** only when the coding is finished, or when one comes back to the same code after a **few monts** and does not understand anymore **what** each **class can be used for**.

```
class dcomplex{
private:
    double re, im;
public:
    dcomplex (double r = 0, double i = 0): re (r), im (i) {}
    dcomplex conj() const {return dcomplex(re,-im);}
    double real() const { return re; }
    double imag() const { return im; }
    dcomplex& operator+= (const dcomplex& r);
    dcomplex& operator-= (const dcomplex& r);
    friend dcomplex operator+ (const dcomplex& x, const dcomplex& y);
};
```

Access specifiers are

- **private** indicates that a member may be accessed only by class methods and not by other parts of the program.
- **public** indicates that a member is accesible from the "outside world".
- **protected** allows access to the derived classes. For the rest of the world, members are hidden.

Members "re" and "im" can not be accessed directly by users of dcomplex type. They can be accessed only through member functions real() and imag().

Good class design should always enforce **data hiding**, i.e., limiting access and manipulation of members only through methods. The interface to the class is public and the data is private.

Data hiding accomplishes two goals

- Users need not be concerned with the internal representation of the data
- If the internal representation of the data is modified, any code using this class need not be modified.

There are two ways to define methods:

- within the class definition (small methods, automatically inlined)
- outside of the class (larger methods)

```
inline dcomplex& dcomplex::operator+= (const dcomplex& r)
{
    re += r.re;
    im += r.im;
    return *this;
}
```

Inlined functions are faster (avoid overhead of function call) but compilation is slower.

Constructor can have "member initialization list". It is a comma-separated list of members along with their initial values, separated by a colon from the end of the parameter list.

```
inline dcomplex operator+ (const dcomplex& x, const dcomplex& y)
{
    return dcomplex(x.re + y.re, x.im + y.im);
}
```

Friend function is a function which is allowed to access private and protected members.

Note that it is 'return by value' function. 'Return by reference' would not work. It is important to return temporary object of unnamed form to allow return value optimization.

Functions can be overloaded (multiple functions with the same name) Each version of the function must differ in either the number and/or type of its arguments.

Can have default arguments. Defaults must be specified starting at the right most parameters.

```
class function{
protected:
    double *f;
    int N;
public:
    function(): f(NULL), N(0) {};// default constructor needed for arrays.
    explicit function(int N_); // No conversion allowed
    function(const function&); // Copy constructor
    ~function(); // Destructor
    int size() const { return N;}
    double& operator[](int i) // returns reference, thus allows modifying class
    { if(i>=N) cerr<<"Out of range in function[]"<<endl; return f[i];}
    const double& operator[](int i) const// returns const ref. Needed when declared const
    { if(i<N) cerr<<"Out of range in function[]"<<endl; return f[i];}
}
```

A constructor is called whenever an object is defined or dynamically allocated using the "new" operator. It is typically used to obtain resources such as memory

```
inline function::function(int N_)
{ f = new double[N_];}
```

An object's destructor is called whenever an object goes out of scope or when the "delete" operator is called on a pointer to the object. The purpose of the destructor is clean up.

```
inline function1D::~~function()
{ delete[] f; f=NULL;}
```

Why explicit in declaration of constructor?

```
int DoSomething(const function& f);  
....  
DoSomething(2); // Error, constructor is explicit and conversion not possible
```

Few points to stress

- A constructor is a method that has the same name as its class.
- A destructor is a method that has as its name the class name prefixed by a tilde, .
- Neither constructors nor destructors return values. They have no return type specified.
- Constructors can have arguments.
- Constructors can be overloaded.
- If any constructor is written for the class, the compiler will not generate a default constructor.
- The default constructor is a constructor with no arguments, or a constructor that provides defaults for all arguments.
- The container classes such as vector require default constructors to be available for the classes they hold. Dynamically allocated class arrays also require a default constructor. If any constructors are defined, you should always define a default constructor as well.
- Destructors have no arguments and thus cannot be overloaded.

Remember: If constructor allocates memory, destructor needs to deallocate the same memory in the same way.

Dynamic allocation of memory

- `new` operator
 - most often used.
 - Allocates any object - allocates memory and calls constructor
 - Example: `string *ps = new string("Memory Management");`
- `new[]`
 - allocates an array
 - Example `string *ps = new string[10];`
 - Don't forget to use `"delete[]"` instead of simple `"delete"`;
- `operator new` (equivalent to `malloc` in C)
 - returns `void*`
 - it returns a pointer to raw, uninitialized memory. No constructor called
 - Example: `void* memory = operator new(size);`
 - Don't forget to call `operator delete`: `operator delete(memory)`
- "placement" `new`
 - raw memory is already allocated, one constructs an object in the existing raw memory (calls constructor)
 - Example: A is a class type `a = new(memory) A;`
 - Don't forget to call destructor "by hand": `a.~A();`

Copy constructor

A copy constructor is a [special constructor](#) that takes as its argument a reference to an object of the same class and creates a new object that is a [copy](#).

```
function(const function&); // Copy constructor
```

By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created. This is called a [member wise](#) or [shallow copy](#). This is often not satisfactory.

In the above example, default *copy constructor* would only create new pointer (`double* f`) which would point to the same location in memory. When the original object is destroyed, the memory-data is gone and the copy of the object, created by the *copy constructor*, is corrupted.

Remember: Whenever your constructor allocates memory, default copy constructor is not enough.

Write your copy constructor.

```
inline function::function(const function& m)
{
    resize(m.N);
    std::copy(m.f, m.f+N, f);
}
inline void function::resize(int n)
{
    if (n>N){
        if (f) delete[] f;
        f = new T[n];
    }
    N = n;
}
```

Remember: Always prefer pass-by-const-reference to pass-by-value.

The meaning of passing an object by value is defined by the copy constructor of that object's class. This can make pass-by-value an extremely expensive operation.

The following peaces of code give the same result, but the second is much slower. It needs two more copy-constructor calls

```
inline dcomplex operator+ (const dcomplex& x, const dcomplex& y)
{
    return dcomplex(x.re + y.re, x.im + y.im);
}
```

and

```
inline dcomplex operator+ (const dcomplex x, const dcomplex y)
{
    return dcomplex(x.re + y.re, x.im + y.im);
}
```

In case of large arrays or matrices, the overhead can be enormous.

Arrays of objects

Default constructor is necessary for creating arrays of objects. Needed also when STL standard containers are used for storage.

```
function f1; // Simple object. Calls default constructor. Object is empty (of size 0)
function f2(N); // Simple object. Calls constructor which allocates space for N numbers.
function fa[3]; // Array of functions. For each function, default constructor is called
std::vector<function> fb(M); // Vector of functions. Again default constructors are called
function fc[3] = {10,10,10}; // Rarely used and not very usefull. Calls non-default const.
```

Static members

Static members provide a way to create objects that are shared by an entire class, rather than being part of a particular instance of the class, an object. Static methods provide a way to access and update static members.

```
class function{
protected:
    static const int max_size = 1000;
    static int default_size;
public:
    static int Set_default_size(int def_size){default_size=def_size;}
    ...
}
```

Homework

- Implement your own class Vector.

The declaration of the vector class is:

```
class Vector{
    int N;        // size of the vector
    double *m;   // pointer to data
public:
    ///// constructors & destructor
    Vector(int N, double def_val=0); // constructor takes size as an argument
    Vector();                       // default constructor is called when no arguments are gi
    Vector(const Vector& A);         // copy constructor necessary for return by value
    ~Vector();                       // desctructor necessary because we have pointers to deal
    ///// other class members
    double operator[](int i) const;  // array access operator for reading
    double& operator[](int i);       // array access operator for writting
    int size() const {return N;}     // returns size
    void resize(int N);              // resizes existing vector
    void random();                   // fills with random numbers
};
```

Implementation of Vector class is empty.

The testing code is available to download (matrix.cc) as well as matrix class (matrix.h). check

http://www.physics.rutgers.edu/~haule/509/src_prog/C++/homework2/ for details

Inheritance

check out <http://www.cs.bu.edu/teaching/cpp/inheritance/intro/>

Classes attempt to model real world entities - have various relationships which help to simplify the task of understanding the systems. One of them is *Inheritance*.

Creating or deriving a new class using another class as a base is called *inheritance*. The new class created is called a **Derived class** and the old class used as a base is called a **Base class** in C++ inheritance terminology.

C++ inheritance is very similar to a **parent-child** relationship. When a class is inherited all the functions and data member are inherited, although not all of them will be accessible by the member functions of the derived class.

- The derived class will inherit all the features of the base class
- The derived class can also add its own features
- It can also override some of the features (functions) of the base class
- Exeptions to the above rules:
 - The constructor and destructor of a base class are not inherited
 - the assignment operator is not inherited
 - the friend functions and friend classes of the base class are also not inherited.

In the **derived class**, only **protected** and **public** members of the base class are **accessible**. The **private members** of the base class are **not accessible** by a derived class.

Remember: Most of data should be protected rather than private if a class is used as a base class.

```
class vehicle{
protected: // members accesible also for Car
    string colorname; // Car has colorname and
    int number_of_wheels; // number_of_wheels
public:
    vehicle(); // constructor is not inherited
    ~vehicle(); // destructor is not inherited
    void start(); // start, stop and run exists also
    void stop(); // in Car, but they can be changed
    void run(); // if necessary
};
class Car: public vehicle{
protected:
    char type_of_fuel; // Char has one more member
public:
    Car();
};
```

Improper inheritance:

make the base class weaker

```
class Bird{
public:
    void fly();
    . . .
};
```

eliminate the proposed inheritance relationship

```
class Penguin : public Bird{
    // cannot fly
}
```

Optimization

- Be aware of 80/20 rule
- Use successive memory (row-major) order
- Understand the origin of temporary objects
- Always prefer pass-by-const-reference to pass-by-value
- Facilitate the return value optimization
- Consider using op= instead of stand-alone op
- Avoid using virtual functions, multiple inheritance, virtual base classes, exception handling
- Consider alternative libraries

Further reading : *Effective C++* and *More Effective C++* by Scott Meyers

Templates

Templates provide a way to represent a wide range of general concepts and algorithms and simple ways to combine them (STL).

Templates provide direct support for generic programming, that is, programming using types as parameters.

Most importantly, templates are instantiated at compile-time. For numerics this is of extreme importance because there is **no performance cost**. The price one pays is **longer compilation time**. With increases of computational power, this is hardly a problem nowadays (Expression templates, however, might slow down compilation dramatically).

The simplest example of a template

```
template<class T>
inline T sqr(T x)
{
    return x*x;
}
```

One could also write `<typename T>` instead of `<class T>`. Type T can be any type, either double, int, complex or any user defined type one could think of. The only requirement for user-defined type is that it implements multiplication (`operator*`).

Usage of templates is extremely simple

```
int ia = 2;
int ib = sqr(ia); // Compiler implements version of sqr for type int
double da = 2.0;
double db = sqr(da); //Compiler implements new version of sqr for type double
complex<double> ca(2.0,0);
complex<double> cb = sqr(ca); //Compiler implements new version of sqr for type complex
```

Implementing template is thus equivalent to implement function for each class separately (much more user work).

More importantly, one can write general algorithms using templates and abstraction is done in the process of coding a template.

One example are standard containers. One can use vectors of any type, also user defined type, maps of any type (very powerful objects implementing hashes). We can also improve the definition of our container function above and make it a template. We will need functions of type double, complex, int, and many more.

```
template <class T>
class function{
protected:
    T *f;
    int N;
public:
    function(): f(NULL), N(0) {};// default constructor needed for arrays.
    explicit function(int N_); // No conversion allowed
    function(const function&); // Copy constructor
    ~function(); // Destructor
    int size() const { return N;}
    T& operator[](int i) // returns reference, thus allows modifying class
    { if(i>=N) cerr<<"Out of range in function[]"<<endl; return f[i];}
    const T& operator[](int i) const// returns const ref. Needed when declared const
    { if(i<N) cerr<<"Out of range in function[]"<<endl; return f[i];}
}

template <class T>
inline function<T>::function(int N_)
{ f = new T[N_];}

function<double> fund(N); // A function with N double numbers is declared
function<dcomplex> func(N) // Function of complex numbers
```

```
function<function<int> > fun2(N); // Two dimensional function of integers
vector<function<complex<double> > > func3; // Two dimensional complex function
map<dcomplex,function<dcomplex> > hash;//sorted associative containers containing
//unique key/value pairs.
```

Root finding routines were always a real pain until templates "were born". Why?

With templates, one can write a general purpose root-finding routine that works with any user define class. Here is an example

```
template <class functor>
inline double zeroin(
    const double ax,          // Specify the interval the root
    const double bx,          // to be sought in
    functor& f,               // Function under investigation
    const double tol)         // Acceptable tolerance
{.....}
```

The routine can be tested with a simple function like this

```
double f_simple(double x){return ....};
```

but can also be used with any objects which defines operator()

```
class Complicated{
    .....
public:
    double operator()(double x);
};
```

```
Complicated cmp;
double solution = zeroin(a,b,cmp,1e-12);
```

Why is this so useful? Because we can avoid using global variables. Data can be encapsulated in certain object, and the object just needs to implement `operator()` and zero can be found without giving hidden data to outside world.

We will see very nice example in "Density functional theory" part of the course where solution of Schroedinger equation is used to find bound states - energy at which solution of Schroedinger equation satisfies boundary conditions. The variable that one varies is the energy of the state and its value has to be such that the wave function is zero at origin. Data to solve the Schroedinger equation are encapsulated and hidden inside certain class and a static function can not access them. To use usual root finding routines, one would need to make many variables (or whole class) global. The solution with templates is simple: the class needs to implement "operator()" and zero can be found with the above routine.

For demonstrative purposes, let's consider simple case of cubic function which depends on variables a , b , c and d :

$f(x) = a + bx + cx^2 + dx^3$. Variables $a...d$ are determined in some complicated way somewhere inside a function or class member. We certainly do not want to declare them as global variables. However, we can not evaluate $f(x)$ without knowing them. The solution with templates is simple. In the place where root needs to be found, we write

```
int a,b,c,d;
Get_abc_complicated(a,b,c,d,...);
functor pack(a,b,c,d);
double solution = zeroin(start_interval, end_interval, pack, tolerance);
```

class functor needs to be defined somewhere above

```
class functor{
    double a,b,c,d;
    public:
    functor(double a_, double b_, double c_, double d_) : a(a_), b(b_), c(c_), d(d_){};
    double operator()(double x){return a+x*(b+x*(c+x*d));}
};
```

There are many nice example of generic algorithms implemented in STL, for example sort. If we define new data type, and want to sort an array of objects, we do not need to implement sorting routine. It is implemented in STL. All we need to do is to define operator less_than and we can use STL sort. Here is an example:

```
class A{
    ....
    friend bool operator<(const A& m, const A& n);
};
bool operator<(const A& m, const A& n) {...}

.....
vector<A> v(N);
....
sort(v.begin(),v.end());
```

Homework

- Promote your Matrix and Vector class to template, which can store any type of data.
- implement member function `C.Product(A,B)` and `C = A*B` which calls lapack subroutine `dgemm` for real matrix multiplication and `zgemm` for complex matrix multiplication.

The Standard Template Library

The STL has three fundamental concepts: containers, iterators, and algorithms.

Containers hold collections of objects. Iterators are pointer-like objects that let you walk through STL containers just as you'd use pointers to walk through built-in arrays. Algorithms are functions that work on STL containers and that use iterators to help them do their work.

At its core, STL is very simple. It is just a collection of class and function templates that adhere to a set of conventions. There's no big inheritance hierarchy, no virtual functions, none of that stuff. Just some class and function templates and a set of conventions to which they all subscribe.

Which leads to another revelation: STL is extensible. You can add your own collections, algorithms, and iterators to the STL family.

Some of the containers implemented are

- sequences containers
 - `vector`
 - `list` - optimized for insertion and deletion of elements. The `operator[]` not provided
 - `deque` - optimized for insertion at the beginning and end. It has `operator[]`.
 - `stack` - `push_back` and `pop_back` fast, no `operator[]`
 - `queue` - allows insertion at the back and extraction at front
- associative
 - `map` - sequence of (key/value) pairs which provide fast retrieval based on key
 - `multimap` - map which allows duplicate keys
 - `set` - like map but without values. Only keys
 - `multiset` - duplicate keys
- `valarray` - **vector optimized for numerics computations**
- `string`
- `bitset` - set of flags (might be useful for ED)

The truth is that `valarray` was not implemented in gcc until recently. Even now, it seems not to be really optimized.

Please, check out your compiler whether `valarray` is faster than `vector` and how much.

Typical *algorithms* implemented in STL are: finding certain element in container, sorting, finding minimal, maximal element, finding permutations of elements, copying of a collection of elements, search and replace, transformation of all elements according to certain rule,...

Perl

Perl is very suitable for small scale projects of the order of 100 lines of code which are not computationally intensive - is **interpreter**. It offers much more flexibility than C++ or other "older" languages.

It is extremely useful for working with files and text manipulation: transforming or changing files, changing strings in many large files,... It is used in many web based applications (cgi-bin) and even supports object-oriented (OO) programming. Many graphical user interfaces which run in web browsers are written in perl.

Perl comes with very useful man-pages. Type

```
man perl
```

and for short introduction

```
man perlintro
```

and you can learn perl in few minutes. Perl has probably the biggest resources and largest community on the web of all programming languages.

HelloWorld program can not be simpler than that

```
#!/usr/bin/perl
print "Hello, World!\n";
```

Each perl program can start with a line containing path to perl interpreter. To execute this script, one just needs to add executable flag to script (`chmod +x name_of_your_script`) and run it. Alternatively, one can omit first line and run it with perl prepended

```
perl name_of_your_script
```

Below are some simple examples of using perl variables, arrays and hashes

```
$a = 5; # scalar variable
@b = (1,2,3); # array variable
%c = ("apple", "red", "banana", "yellow"); # hash, just like map in C++
$sum = $b[0]+$b[1]+$b[2]; # summation of elements
print "Color of apple is ", $c{"apple"}, "\n"; # access hash elements
@kys = keys(%c); # stores all keys to array
print "fruits are @kys\n"; # and prints them
```

Perl is so popular because it is very **easy to use** and mainly because of its powerful built-in support for **text processing**. This powerful machinery is called "**regular expressions**". It is very simple to write rules (regular expressions) which can find certain substrings, replace certain substrings or write some reports out of complicated data which needs to be analyzed. Building blocks are symbols which match various characters

- `.` - a single character
- `\s` - a whitespace character (space, tab, newline)
- `\S` - non-whitespace character
- `\d` - a digit (0-9)
- `\D` - a non-digit
- `\a` - word character (az, AZ, 09, -)
- `\W` - a non-word character
- `[aeiou]` - matches a single character in the given set
- `[^aeiou]` - matches a single character outside the given set
- `(foo|bar|baz)` - matches any of the alternatives specified
- `^` - start of string
- `$` - end of string

Quantifiers (how many of the previous thing you want to match on)

- `*` - zero or more of the previous thing
- `+` - one or more of the previous thing
- `?` - zero or one of the previous thing
- `{ 3 }` - matches exactly 3 of the previous thing
- `{ 3 , 6 }` - matches between 3 and 6 of the previous thing
- `{ 3 , }` - matches 3 or more of the previous thing

Regular expression for a real number:

```
$r_n = "-?[0-9]+\.[0-9]*[e|E]?-?[0-9]*";
```

A trivial example of perl usage

```
# This loop reads from STDIN, and prints non-blank lines
while (<>) {
    next if /^$/;
    print;
}
```

line 1 diamond operator <> is equivalent to <STDIN> and reads a line from STDIN and stores into variable \$_

line 2 this line is equivalent to `if (/^$/) {next;}`. If empty line is matched, the rest of lines in the loop are skipped

line 3 prints variable \$_

Next example reads any number of files containing few columns and transformes them in "user defined way". We will first give examples of usage of the script and than code the script. Script called `strans` can do the following

```
strans datafile1 '#1' '2*#2' '#3/2.'
strans datafile1 '#1' 'abs(#2+#3*i)' 'Re(1/(#2+#3*i))' 'Im(1/(#2+#3*i))'
strans datafile1 datafile2 '#1' '(#2+#2:2)**2 + (#3+#2:3)**2' '(#4+#2:4)*ferm(#1/0.01)'
```

line 1 prints first three columns of data file and multiplies the second column with 2 and divides the third column by 2

line 2 second and third column are treated like real and imaginary part of a complex function. First column is copied, second column prints absolute value of the complex function, the third and fourth columns print real and imaginary part of the inverse of the complex function, respectively.

line 3 two data files are merged in the following way: first column of the first file is copied, the second column is the sum of second columns squared, the third is the sum of third coulums squared, the fourth column is sum of fourth columns multiplied by the fermi function

The following script will work in the above specified way

```
#!/usr/bin/perl
use Math::Trig;      # includes trigonometric functions
use Math::Complex;  # includes support for complex functions

push @files, "<&STDIN" unless -t STDIN; # stdin is treated like any other file
foreach (@ARGV){
    if (-s $_){          # if file exists, it must be function name
        push @files, $_;
    } else{
        push @trans, $_; # otherwise it is an expression for trans
    }
}

foreach (@trans)
{
    # in any user defined rule, it replaces symbol # by spl in the following way
    #   #2:3  -> $spl[1][2]
    #   #3    -> $spl[0][3]
    s/\#(\d+):(\d+)/\${spl}\[\${1-1}\]\[\${2-1}\]/g;
    s/\#(\d+)/\${spl}\[0\]\[\${1-1}\]/g;
}

foreach (@files){ open $_, $_; } # All files being transformed are opened

@{$spl[0]} = ( ' ' ); # object spl is initialized
```

```

# Main loop
LB: while (defined($spl[0][0])){ # goes over all columns
    $i=0;
    foreach $f (@files){ # goes over all files
        $_ = <$f>; # reads one line from this file
        if (/\/#\//){ # if the file is commented, just print it and do nothing
            print; $_ = <$f>; # read next line in this case
        }
        @{$spl[$i++]} = split(' ', $_); # saves the contents of the file into two dimensional
    } # array spl[i][j] containing j-th column of i-th file
    if (not defined($spl[0][0])){ # If first column does not exist, we reached end-of-file
        last LB; # or end of continuous lines region of file -> finish
    }
    $line = ""; # the output line initialized
    foreach $col (@trans){ # each column of each file is being processed
        $val = eval $col; # the code for this column, as prepared above, is being
        $line .= $val . " "; # evaluated (core step)
    } # the transformed column is glued together into a line
    print "$line\n"; # Finally, print out the resulting row
}

sub ferm{ return 1.0/(exp(@_[0])+1);} # An example of fermi function which can be used
sub nbose{ return 1.0/(exp(@_[0])-1);} # and bose function

```

Interpreters like Perl or Python are very useful for writing wrapper script for fast plotting of data. The script most often calls other plotting programs like `gnuplot` or `xmgrace`. One such script is provided in this lecture (find it on the web). Examples of its use are

```
plot data
plot -u1:2,1:3 -wlp -g data*
plot -x-1:1 -y-10:10 -uall data[1-2]
plot -x-1:1 -xt0.2 -cpcolor -u1:3 data[1-2]
plot -u1:2 -wlp data1 -u1:3 -wp data2
```

line 1 plots xy plot with first column as x and second column as y

line 2 plots second and third column of all files, whose name starts with data, using first column as x. Uses lines and points (default uses lines only). It also adds gridlines (`-g`)

line 3 plots all columns in both data1 and data2. The first column is again used for x axes. The x range is set between -1 to 1 and y range for -10 to 10.

line 4 plots data1 and data2 using column 1 and column 3. The xtics are set to 0.2 and x range is again set between -1 and 1. Results are not plotted to screen but rather printed using printer specified in pcolor script.

line 5 plots data1 using first and second column and data2 using first and third column. The first curve is plotted with lines and points while the second uses points only.