Computer simulations of telecommunication networks;

C++ introduction

Dmitry Petrov, PhD

dmitry.petrov@magister.fi
Agenda

- Network Modeling and Simulations
  - About Magister Solutions
  - When and why to use simulations?
  - Selected models and examples

- C++ introduction
  - Basics of C++ programming
  - Running C++ programs and examples
  - Main concepts
  - Useful materials and home work
Network Modeling and Simulations
# Magister Solutions – Quick facts

## Quick Facts

<table>
<thead>
<tr>
<th>Resources</th>
<th>~30 employees within main company, incl. 2 professors, 12 doctors and 8 Ph.D. students. Roughly 40 employees when including subsidiaries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Services</td>
<td>R&amp;D, simulators and test-beds, training and advisory and software services. Main focus on telecommunication technologies</td>
</tr>
<tr>
<td>Technology competence</td>
<td>Wireless communication technologies, incl. UMTS (3G), HSPA and HSPA+ (3.5/3.75G), LTE, LTE-A and WiMAX (3.9G/4G), Wi-Fi, cognitive networks, DVB-RCS, DVB-S2, positioning systems (GPS)</td>
</tr>
<tr>
<td>R&amp;D tool expertise</td>
<td>System designing; simulator development (C/C++) and maintenance; simulations; KPI analysis tool development (Perl, Matlab, C/C++); network and device KPI analysis</td>
</tr>
<tr>
<td>Publications</td>
<td>Over 110 international scientific publications</td>
</tr>
<tr>
<td>Patents</td>
<td>Co-authoring in over 20 patents covering WCDMA, HSPA, LTE, LTE-A</td>
</tr>
</tbody>
</table>
Magister Solutions

Mobile, satellite and security technologies
Simulators and test-beds

**Your Current System**

- Competing solutions
- Change in the market situation and needs
- Need to reduce cost. Through more efficient operations

**External and Internal Pressures for Change**

**MAGISTER SOLUTIONS**

Modeling and simulating your system with a software program

Most efficient solutions for your needs
Simulators and test-beds – Own tools

NS-3 with additional modules for LTE, wireless mesh and satellite communication
What is a model?

- Modeling is a process of producing a model.
- A model is a representation of the construction and working of some system of interest.
- A model is similar to but simpler than the system it represents.
- Close approximation to the real system and incorporate most of its salient features.
- Judicious tradeoff between realism and simplicity.
What is a simulation?

- Simulation of a system is the **operation of a model** to the system
- Model can be reconfigured and experimented with
- Usually this is impractical to do in the system it represents
- Simulation is a tool
  - to evaluate the performance of a system, existing or proposed, under different configurations of interest and over long periods of real time
“Essentially, all models are wrong, but some are useful”

Applications of simulations

- Analysis of **air pollutant dispersion** using atmospheric dispersion modeling
- Design of complex systems such as **aircraft and also logistics** systems
- Design of **Noise barriers** to effect roadway noise mitigation
- **Flight simulators** to train pilots
- **Weather** forecasting
- Simulation of other computers is **emulation**.
- Forecasting of prices on **financial markets**
- **Behavior of structures** (such as buildings and industrial parts) under stress and other conditions
- Design of **industrial processes**, such as chemical processing plants
- **Strategic Management** and Organizational Studies
- Reservoir simulation for the **petroleum engineering** to model the subsurface reservoir
- **Process Engineering** Simulation tools.
- **Robot simulators** for the design of robots and robot control algorithms
- **Urban Simulation Models** that simulate dynamic patterns of urban development and responses to urban land use and transportation policies
- **Traffic engineering** to plan or redesign parts of the street network from single junctions over cities to a national highway network, for transportation system planning, design and operations
- **Modeling car crashes** to test safety mechanisms in new vehicle models
How may networks be studied?

- Measurements from real devices / networks
  - Protocol analysers and monitors
  - Drive test with specialized equipment
  - Measurements from real devices
    - $ and time consuming, new devices not always available

- Test networks / emulation
  - Access to all equipment / software
  - Open-source software
    - Expensive and not large enough

- Mathematical analysis
  - Matlab, pen’n’paper
    - Too many simplifications in complex dynamic systems
Why to simulate?

- Develop and test vast amount of potential solutions for current and future challenges without excessive costs

- Wireless
  - Reproducibility
  - Fidelity (especially, real-time constraints)
  - Radios may not exist or be available
  - Field tests in realistic conditions cost $$$
  - Scenarios with desired parameters

- Scalability
  - 10,000+ nodes?
  - For smaller configurations, execution time
When to simulate

- When the **analytical model/solution is not possible or feasible**.
- Many times, simulation results are used to **verify analytical solutions** in order to make sure that the system is modeled correctly using analytical approaches.
- **Dynamic systems**, which involve randomness and change of state with time.
- **Complex dynamic systems**, which are so complex that when analyzed theoretically will require too many simplifications. In such cases, it is not possible to study the system and analyze it analytically.
New technology path

R&D
- Concept
- Test network
- Commercialization
- Planning
- Commitment
- Optimization

Explotation

Analytical model

Simulations

© 2015 Magister Solutions Ltd
Simulation workflow

Simulation model
- Identify the problem
- Formulate the problem
- Collect and process real world data
- Formulate and develop a model
- Validate the model

Simulation experiment
- Select appropriate experimental design
- Establish experimental condition for runs
- Perform simulation runs

Simulation analysis
- Interpret and present results
- Recommend further course of action

- parameter tuning
- simulation scenario tuning
- model debugging
What can be achieved with simulations?

- Test new and advanced features and their gains
- Study wide range of scenarios
- Identify bottlenecks in processes
- Prevent under or over-utilisation of resources
- Safe and chaper way to evaluate the side effects
- Optimize the performance of the system
- Balance expenses and improvements in user experience
Revolutionary applications in mobile industry

1. Need for unlimited telephony
   1. Paradigm shift: personalization of communications
2. Two-way paging
3. Internet browsing (WiFi)
4. Merge laptop function for a mobile use
5. New paradigm shift:
   “Instead of consumer going to the Internet, the Internet will come to them”*
   1. Thirst for data will continue
   2. Form and context of communications
   3. Clever content discovery
   4. M2M and MTC + steering

Simulator types

- Different ways of dividing simulators
  - Running principle: discrete event, continuous, ...
  - Level of detail: static, quasi-static, dynamic, ...
  - Modeling purpose: wireless radio access, wired, ...
  - Confidentiality: proprietary, open
Simulators and Network Planning Tools

- NPTs:
  - Focus on radio part:
    - propagation and interference models
    - Real maps
  - Radio planning and optimization
  - Some posses simulation possibilities:
    - Traffic modeling

- Simulators:
  - Representation of a real system
  - Both Radio and Core planes (end2end)
  - Detailed realization of protocols and features
  - Statistics for specific UE, eNB, link
  - RB or even symbol resolution on PHY
  - Packet tracing
Simulator types

- **Emulation**: The process of designing and building hardware or firmware (i.e., prototype) that imitates the functionality of the real system.

- **Monte Carlo simulation**: Any simulation that has no time axis. Monte Carlo simulation is used to model probabilistic phenomena that do not change with time, or to evaluate non-probabilistic expressions using probabilistic techniques.

- **Trace-driven simulation**: Any simulation that uses an ordered list of real-world events as input.

- **Continuous-event simulation**: In some systems, the state changes occur all the time, not merely at discrete times.

- **Discrete-event simulation**: A discrete-event simulation is characterized by two features: (1) within any interval of time, one can find a subinterval in which no event occurs and no state variables change; (2) the number of events is finite.
## Static vs. dynamic simulators

<table>
<thead>
<tr>
<th></th>
<th>Static simulators “Snapshot”</th>
<th>Quasi-Static simulators</th>
<th>Dynamic simulators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithms</strong></td>
<td>Simplified and limited algorithms, e.g no outer power control</td>
<td>Extensively modelled RRM</td>
<td>Fully modelled RRM</td>
</tr>
<tr>
<td><strong>Path Loss</strong></td>
<td>Fixed</td>
<td>Fixed</td>
<td>Varied every timeslot (slot, TTI, etc.)</td>
</tr>
<tr>
<td><strong>Time Domain</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Mobility</strong></td>
<td>Static</td>
<td>Static</td>
<td>Moving randomly</td>
</tr>
<tr>
<td><strong>Computational Complexity</strong></td>
<td>Fast</td>
<td>Middle</td>
<td>Slow</td>
</tr>
<tr>
<td><strong>USE Cases</strong></td>
<td>Interference study between systems, etc.</td>
<td>RRM study excluding mobility, etc.</td>
<td>Mobility Study, RRM Study, etc.</td>
</tr>
</tbody>
</table>
Link and System level simulations

- **Link Level:**
  - 1 UE – 1 eNB connection is simulated
  - 1 subcarrier / 1 time slot resolution
  - Mainly physical layer procedures: coding, modulation, complicated channel models, channel estimation etc.
  - Provides input for system level simulations
  - Actual data transmission

- **System Level:**
  - Hundreds of UEs - tens of eNBs
  - Resource Block (RB) resolution and more simple channel models (pathloss + shadowing + fast fading)
  - Includes full protocol stack and many other procedures
  - TBER is defined by BLER-SINR curves form Link level
  - Simulation of received SINRs
Link and system level simulators

Link Level parameters:
- Modulation
- Coding
- Interleaving
- Puncturing
- Repetition
- Joint detection parameters
- Diversity
- Channel model
- etc.

Inputs from link level:
- Average C/I vs. BER/FER/BLER
- Local C/I vs. raw BER
- Raw BER vs. coded BER/FER/BLER
- Link level gains for joint detection
- etc.

System level parameters:
- Cell layout
- Environment
- Traffic model
- Mobility model
- Multiple access related parameters
- Power amplifier ACLR
- Receiver selectivity
- RRM parameters
- Channel model
- etc.

Outputs from system level simulation:
- spectrum efficiency
- users/cell
- kbps/MHz/cell
- Noise rise in the base station
- UE and Node-B TXP and RXP
- UE and Node-B TXP and RXP maps
- BER/FER/BLER distributions
- BER/FER/BLER maps
- etc.
Simulator input, i.e. parameterization

- Basically everything within a simulation may/need to be parameterized
  - Number of terminals
  - Simulation length
  - Scenario and topology
  - Mobility
  - Traffic
  - RRM specifics
  - ...

© 2015 Magister Solutions Ltd
Simulation scenarios

- Simulation scenarios are often standardized by standardization organizations for specific purposes, e.g.
  - Results verification
  - Problem validation
Simulation scenarios (cont.)

UE was placed randomly on the circle and it takes a random drive direction towards the pico.

[3GPP TR 36.839]

Simulation scenarios (cont.)

- Springwald
  - ISD varies being about 500m in minimum and about 1500m in maximum

- Realism
  - Varying load per BS
  - Mobility problems, coverage holes
  - UL / DL imbalance issues

Signal propagation, fading types

- Radio propagation
  - Empirical mathematical formulation for the characterization of radio wave propagation as a function of frequency, distance and other conditions

- In practice, most simulation studies use empirical models that have been developed based on measurements taken in various real environments
  - Hata model
  - COST 231 extension to Hata model
  - COST 231 Walfish-Ikegami model
Fading types (cont.)

- **Slow fading**
  - Shadow fading is caused by **obstacles in the propagation path** between the UE and the eNodeB and can be interpreted as the irregularities of the geographical characteristics of the terrain introduced with respect to the average pathloss obtained from the macroscopic pathloss model.
  - Some part of the transmitted signal is lost through **absorption, reflection, scattering, and diffraction**.

- **Fast fading**
  - Fast fading occurs if the channel impulse response changes rapidly within the symbol duration.
  - Fast fading occurs when the coherence time of the channel is smaller than the symbol period of the transmitted signal.
  - This causes frequency dispersion or time selective fading due to Doppler spreading.
  - Fast Fading is due to reflections of local objects and the motion of the objects relative to those objects.
Propagation, slow and fast fading
Propagation map with slow fading

Dominance map
Traffic models

- Defined by standardization organizations
  - Infinite buffer
  - FTP
  - Constant Bit Rate (CBR)
  - HTTP models (bursty)
  - AMR codecs for VoIP
Mobility models

- Mobility models represent the movement of mobile users, and how their location, velocity and acceleration change over time.
- In certain scenarios the mobility plays an important role, e.g. mobile ad hoc networks.
- Several ways of categorizing mobility models:
  - Traces and synthetic mobility models
  - Entity and group mobility models
  - Human, animal and vehicle mobility models
  - Normal situation and special situation mobility models
- Usual factors:
  - Speed: \([\text{speed}_{\text{min}}; \text{speed}_{\text{max}}]\)
  - Direction: \([0, \pi]\)
  - Time or distance before making the next turn
Mobility models

Random walk

Random waypoint
New features in existing networks

Carrier Aggregation

DC capable
UE

Spectrum 1

Spectrum 2

5 MHz

Multipoint Tx

PA3 Channel (Softer Handover Users)

Mean Softer HO User BR Gain (%)

0 5 10 15 20 25 30 35

0
5
10
15
20
25
30
35
40
45
50
Users per cell

0 100 200 300 400 500 600 700 800

0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

CDF

Burst Throughput Distribution, 500 m ISD, TU6, RoT target 6 dB

Single Carrier, Burst Throughput Distribution, 500 m ISD, TU6, RoT target 6 dB

0
5
10
15
20
25
30
35
40
45
50
Users per cell

Mean Softer HO User BR Gain (%)
Converged Infrastructure for Emerging Regions (CIER) challenges

- Limitations to OPEX and CAPEX due to limited financial capabilities
- Lack of wired infrastructures
- Huge distances between localities
- Lack of reliable energy supply
- Lack of well-trained service personnel
- Sparsely populated areas
- Challenging socioeconomic and political conditions
Simulation scenario

TestBed integration

Mesh node
Real-time demo

Gateways

Mesh

Edge

Display

NEF player

Computer

NS3 Simulator

Tap IF

Tap dev.

+ side load

Emu dev.

Wire/Wired

RX/Play video

Play/TX video

Display

© 2015 Magister Solutions Ltd
Enhanced Multicarrier Techniques for Professional Ad-Hoc and Cell-Based Communications

- Main objectives of **EMPHATIC**:  
  - Enhanced multicarrier techniques (FB-MC)  
  - Flexible radio platform, based on variable FB processing  
  - Co-existence of radio frequency bands together with Professional Mobile Radio (PMR)
COgnitive network ManageMent under UNcErntainty (COMMUNE) project

LTE Network

Cell-specific re-configurations

{JSON}

HTTP / REST interface

Configuration manager
- Mapping solution to network configuration parameters

OSGi

Solution

Recovery analysis
- Find best matching solution from the knowledge base (KB) using case-based reasoning algorithm
- Enables automated verification and extension of human input knowledge in self-healing.

Anomaly analysis

Phase 1: Anomaly Detection
Detect anomalous UE reports using K-Nearest Neighbors algorithm

I. Training stage
Learn about normal behavior

II. Testing stage
Find behavior different from normal

Phase 2: Statistical KPI analysis
Find abnormal deviations of KPIs

Cell-level statistical KPI analysis

Phase 3: Statistical post-processing
Calculate abnormality magnitude

Cell level, KPI based abnormality magnitude calculation

Anomaly report

{JSON}

UE level KPIs
- N best RSRPs
- SINR
- Number of users per cell

Anomaly report

{JSON}

© 2015 Magister Solutions Ltd
Results of Cognitive Self-Healing in LTE

Problematic region (cell 29)

Coverage hole is significantly reduced after compensation
NEF player, Demo

1. **Savings in development and maintenance**
   No need to develop or maintain separate graphical user interfaces (GUI) or visualization tools for different simulators.

2. **Easy integrate and take into use**
   Optimized C++ based NEF-library can be easily integrated with different simulators. Simulations can be visualized with QT based NEF-Player in Windows and Linux environments.

3. **Faster results analysis and development cycle**
   Efficient simulation visualization helps the analyze system / network performance for R&D and marketing purposes for both technical and non-technical audience.

See demo at: [http://youtu.be/LIPUHyXHsz0](http://youtu.be/LIPUHyXHsz0)
C++ introduction
Programming languages*

- **Procedural** (C, Fortran, Pascal, Basic)
  - expresses the procedure to be followed to solve a problem
- **Functional and Logical** (Lisp, Scheme, Haskell)
  - the primary focus is on the return values of functions, and side effects and other means storing state are strongly discouraged.
  - allow the computer to reason about the consequences of statements declared by programmer
- **Object-oriented** (C++, C#, Java)
  - the world is a collection of objects that have internal data and external means of accessing parts of that data.
- **Scripting** (Perl, PHP, Python)
  - often procedural and may contain elements of object-oriented languages, but are typically not meant to be full-fledged programming languages (no var. declaration, no compile-time type checking, etc.)

*A programming language* is a formal constructed language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs to control the behavior of a machine or to express algorithms.
C++ history

- **C language** was developed in 1972 by Dennis Ritchie at Bell Telephone laboratories, primarily as a systems programming language.
- In 1973, Ritchie and Ken Thompson rewrote most of the UNIX operating system using C.
- In 1983, the American National Standards Institute (ANSI) formed a committee to establish a formal standard for C.
- In 1990 the International Organization for Standardization adopted ANSI C (with a few minor modifications). This version of C became known as **C90**.
- In 1999, the ANSI committee released a new version - **C99**. It adopted many features which had already made their way into compilers as extensions, or had been implemented in C++.
- **C++ was developed by Bjarne Stroustrup** at Bell Labs as an extension to C, starting in 1979. C++ was ratified in 1998 by the ISO committee, and again in 2003 (called C++03)
- A new version of the standard, known as **C++11** has been made available...
Processing a C++ program

C++ program

Editor

Preprocessor

Preprocessor directives begin with #

Compiler

Library

Translation into machine language (object program); checks the rules of language.

Linker

Combination with other programs provided by the SDK -> executable

Loader

Load into main memory

Execution
Basics of a C++ Program

- **Syntax** - rules specify which statements (instructions) are legal.
- **Function** is a group of code statements which are given a name.
- **Variable** is a portion of memory to store a value.
  - Each variable needs a *name that identifies* it and distinguishes it from the others.
  - Program needs to be aware of *data type* stored in the variable.
- **Constants** are expressions with a fixed value.
- We can begin to operate with variables and constants by using *operators*.
- **Namespaces** subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.
- In general, every .cc (.cc, .cpp, .CPP, .c++, .cp, or .cxx) file should have an associated *header* .h (.hh) file. Order of inclusion:
  - dir2/foo2.h.
  - C and C++ system files.
  - Other libraries' .h files.
  - Your project's .h files.
// my first program in C++
#include <iostream>
using namespace std;

int main()
{
    cout << "My first C++ program." << endl;
    return 0;
}

- Install Ubuntu modules:
  - sudo apt-get install build-essential g++

- Compile:
  - gcc -o test1 test1.cc -lstdc++
  - g++ -o main -I /source/includes main.cpp
    // -I option to specify an alternate include directory

- Run
  - ./test1
C++ IDEs

- Text editor (gedit, notepad++, etc.)
- QT: [http://qt-project.org/](http://qt-project.org/)
- Eclipse CDT (C/C++ Development Tooling): [https://eclipse.org/cdt/](https://eclipse.org/cdt/)
Data types

- Data types – set of values together with a set of operations

C++’s Data Types

Simple

- Integral: char, short, int, bool, long, unsigned...
- Floating point: float, double, ...

Structured

```cpp
enum Color { RED, GREEN, BLUE };  
Color r = RED;
```

```cpp
struct product {  
    int weight;  
    double price;  
};
```

```cpp
product apple;  
product banana, melon;  
apple.weight = 50;  
apple.price = 0.20;
```
# Initialization and static casting

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

int main ()
{
    int a=5;              // initial value: 5
    int b(3);              // initial value: 3
    int c{2};              // initial value: 2 C++11
    int result;            // initial value undetermined

    a = a + b;
    result = a - c;
    cout << result;

    string mystring;
    mystring = "This is the initial string content";
    cout << mystring << endl;

    /* magical function returns the age in years */
    int age = getAge();
    /* magical function returns the number of visits */
    int pain_visits = getVisits();

    float visits_per_year = pain_visits / age;
    // float visits_per_year = pain_visits / static_cast<float>(age); // float(age);

    return 0;
}
```
If ... else statements in C++

```cpp
#include <iostream>

using namespace std;

int main() {
    int age;  // Need a variable...

    cout << "Please input your age: ";  // Asks for age
    cin >> age;  // The input is put in age
    cin.ignore();  // Throw away enter
    if (age < 100) {
        cout << "You are pretty young!\n";  // Just to show you it works...
    }
    else if (age == 100) {
        cout << "You are old\n";  // Just to show you it works...
    }
    else {
        cout << "You are really old\n";  // Executed if no other statement is
    }
    cin.get();
}
```

Boolean operations: !( 1 || 0 );  !( 1 || 1 && 0 );  !( ( 1 || 0 ) && 0 );
Loops

// continue loop example
#include <iostream>
using namespace std;

int main ()
{
    for (int n=0; n>0; n--)
    {
        if (n==5) continue;
        cout << n << "", ";
    }
    cout << "liftoff!\n";
}

Other statements: while (n>0) {... } 
switch (x) { case 1: ... default: ... }
An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

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

void printarray (int arg[], int length) {
    for (int n=0; n<length; ++n)
        cout << arg[n] << " ";
    cout << "\n";
}

int main ()
{
    int firstarray[] = {5, 10, 15};
    int secondarray[] = {2, 4, 6, 8, 10};
    printarray (firstarray,3);
    printarray (secondarray,5);
    return 1;
}
```

Other examples:
- char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
- char myword[] = "Hello";
- int jimmy [3][5]; // 2D array
- array<int,3> myarray {10,20,30}; //#include <array>
Pointers

- **Variables** - locations in the computer's memory which can be accessed by their identifier (their name).
- **Address-of operator**: `foo = &myvar;`
- **Dereference operator**: `bar = *foo;`
- **Declaring pointers**: `double * decimals;`

```
const char * foo = "hello";
```

```
foo
1776

1772 1773 1774 1775 1776 1777
'h' 'e' 'l' 'l' 'o' '\0'

1702 1703 1704 1705 1706 1707
foo
```
```cpp
#include <iostream>
using namespace std;

int main ()
{
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;

    p1 = &firstvalue;  // p1 = address of firstvalue
    p2 = &secondvalue; // p2 = address of secondvalue
    *p1 = 10;          // value pointed to by p1 = 10
    *p2 = *p1;         // value pointed to by p2 = value pointed by p1
    p1 = p2;           // p1 = p2 (value of pointer is copied)
    *p1 = 20;          // value pointed by p1 = 20

    cout << "firstvalue is " << firstvalue << endl;
    cout << "secondvalue is " << secondvalue << endl;
    return 0;
}
```

What values will be in firstvalue and in secondvalue?
Constants are expressions with a fixed value.

Const values:

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

const double pi = 3.14159;
const char newline = '\n';

int main ()
{
    double r=5.0; // radius
double circle;

circle = 2 * pi * r;
cout << circle;
cout << newline;
}
```

Pointer to constant data:

```cpp
void print_all (const int* start, const int* stop)
{
    const int * current = start;
    while (current != stop) {
        cout << *current << '\n';
        ++current; // increment pointer
    }
}
```

Question:

```cpp
int x;
int * const p_int = &x;
```
**Class** is an element of expanded concept of data structures: like data structures, they can contain **data members**, but they can also contain **functions as members (methods)**.

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

class Rectangle {
private:
  int width, height;
public:
  void set_values(int x, int y) {
    width = x;
    height = y;
  }
  int area() { return width*height; }
};

void Rectangle::set_values(int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values(3,4);
  cout << "area: " << rect.area() << 
  return 0;
}
```

**Contractors:**

```cpp
Rectangle::Rectangle() {
  width = 5;
  height = 5;
}

Rectangle::Rectangle(int a, int b) {
  width = a;
  height = b;
}
```

```cpp
Rectangle rect (3,4);
Rectangle rectb;
```

**Pointers:**

```cpp
Rectangle obj (3, 4);
Rectangle * foo, * bar;
foo = &obj;
bar = new Rectangle (5, 6);
cout << "*bar's area: " << bar->area() << \\
```

© 2015 Magister Solutions Ltd
Inheritance, polymorphism

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

class Polygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a; height=b;
        }
};

class Rectangle: public Polygon {
    public:
        int area () {
            return width * height;
        }
};

class Triangle: public Polygon {
    public:
        int area () {
            return width * height / 2;
        }
};

int main () {
    Rectangle rect;
    Triangle trgl;
    Polygon * ppoly1 = &rect;
    Polygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << '\n';
    cout << trgl.area() << '\n';
    return 0;
}
```
Class templates

```cpp
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
    public:
        mypair (T first, T second)
        {a=first; b=second;}
        T getmax ()
    };

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () {
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
```
#include <iostream>

using namespace std;

class Base { virtual void dummy() {});
class Derived: public Base { int a; };

int main()
{
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast.\n";

    return 0;
}
Pointers to functions
- passing a function as an argument to another function.

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

int addition (int a, int b)
{ return (a+b); } 

int subtraction (int a, int b)
{ return (a-b); } 

int operation (int x, int y, int (*functocall)(int,int)) 
{
    int g;
    g = (*functocall)(x,y);
    return (g);
}

int main ()
{
    int m,n;
    int (*minus)(int,int) = subtraction;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout <<n;
    return 0;
}
```
C++11 features

- Automatic type definition:
  ```
  int x = 3;
  auto itr = address_book.begin();
  auto y = x;
  ```

- Ranged for loops:
  ```
  vector<int> vec;
  vec.push_back( 10 );
  vec.push_back( 20 );
  for (int &i : vec )
  {
    cout << i;
  }
  ```

- Lambda functions
- Performance improvements
- The new C++ memory model and the feature it supports: multithreading, etc...
- $ g++ -std=c++11 your_file.cpp -o your_program $
Useful links and books

- http://www.cplusplus.com/
- http://www.learncpp.com/

- *Accelerated C++: Practical Programming by Example*, Andrew Koenig & Barbara E. Moo, 2000
Home work

Write a simple histogram class, called Histogram, which collects samples into bins of equal width. For this exercise, the histogram is defined by a lower (L) and upper (H) boundary and bin width (W). The interval of numbers, on which the histogram operates, is half open:

\[ [L; H[ \]

i.e. the upper boundary is not included. Similarly for bins:

the first bin collects samples in the interval \([L; L+W[\),
the second bin collects from interval \([L+W; L+2W[\),
... etc.

The last bin collects from interval \([H-W, H[. (See the figure)

If the difference between the upper and lower boundary \(H - L\) is not multiple of bin width \(W\), increase the upper boundary \(H\) so, that \((H-L)\) is fully divisible by \(W\). In addition to that, the histogram collects number of samples below the lower boundary (<\(L\)) and above or equal to the upper boundary (≥\(H\)).
Home work (cont.)

- Write the following operations:
  - constructor with low, high boundaries and optional bin width. If the bin width is not given, assume it is 1.
  - copy constructor
  - assignment operator
  - destructor - if needed, depends on your implementation
  - function for adding a new sample into the histogram (A)
  - function returning the total number of samples (T)
  - function returning the number of samples below the lower bound (B)
  - function returning the number of samples above the upper bound (V)
  - function returning the average of all samples in the histogram - including those below and above (M)
  - function returning true, if there are no samples in the histogram
  - function for clearing all values
  - function returning the number of bins - for testing
  - function returning the number of samples in certain bin - for testing
  - operator « for writing the histogram out to a stream (must work with std::ostream). For each bin, write the following on one line: bin lower boundary TAB number of samples in that bin TAB cumulative number of samples up to and including this bin. Write also one line for samples below and above the histogram’s interval. See the output below for an example.
Home work (cont.)

- Organize your code in the following way:
  - Makefile OR compile.sh - the make file or shell script shall compile, link and start your program. If you use the online compiler, you don't need to write this file.
  - histogram.h - header file with class declaration
  - histogram.cc - implementation file with function definitions
  - main.cc - main file, which will contain the following code:

Select appropriate names for all functions. All samples will have type double. Assume there will be no NaNs or infinite samples. Use standard C++, you can use STL if you want. No other library is permitted. Your code must compile with GNU g++ 4.1.x or higher (you need at least 4.7.x if you use C++11).
```cpp
#include <iostream>
#include "histogram.h"

int main()
{
    Histogram h(1, 7, 1)
    Histogram g = h;
    h.A(20);
    h.A(10);
    h.A(2);
    h.A(5);
    h.A(-5);
    g = h;
    std::cout << "Total samples: " << h.T() << ", below: " << h.B() 
               << ", above: " << h.V() << ", mean: " << h.M() << std::endl;
    std::cout << g << std::endl;
    return 0;
}
```

Expected output:

```
Total samples: 5, below: 1, above: 2, mean: 6.4
below 1 1
1 0 1
2 1 2
3 0 2
4 0 2
5 1 3
6 0 3
above 2 5
```

! substitute the proper names of your functions for the letters!
Thank you for attention!

Thanks to my colleagues: Jani Puttonen, Petri Eskelinen, Anotoine Trux, Fedor Chernogorov, Janne Kurjenniemi, Timo Hämäläinen and others as well!

Dmitry Petrov, 
dmitry.petrov@magister.fi