Programming, numerics and optimization

Lecture A-1: Preliminaries, programming basics I

Łukasz Jankowski
ljank@ippt.pan.pl

Department of Intelligent Technologies
Institute of Fundamental Technological Research
Room 4.32, Phone (+48)228261281 ext. 428

March 3, 2026[1]

[1] Current version is available at http://bluebox.ippt.pan.pl/~ljank.

Outline

Outline

Course objectives

Objective I

To introduce the principles of

- Numerical computations in general
  - number representations,
  - conditioning,
  - stability,
  - distinction between problems and algorithms, etc.
- Selected numerical techniques for
  - linear systems,
  - ordinary differential equations (ODEs),
  - optimization and structural optimization.

Course objectives

Objective II

To provide a foundation for implementing selected numerical techniques in C/C++ (or in any other programming language).

Matlab, Wolfram Mathematica (Maple, Scilab, Octave, Sage, etc.) are great software for rapid prototyping, small- to medium-scale general computations and visualization. Learn to use at least one of them![2]

However, you may soon encounter practical problems that are too large or run too slow to be fully coded in standard high-level packages. They need to be coded in a lower-level environment.

[2] Matlab and Wolfram Mathematica are available in IPPT PAN through network licenses. Contact me, if you are interested in Mathematica.

General motivation

1. Providing a broad picture of basic numerical techniques for linear systems, ordinary differential equations and (not only structural) optimization.
2. Increasing the general understanding of the internals/pitfalls of several frequently used algorithms. As a result, their common implementations (in Matlab, Mathematica, etc.) should be used more consciously and less in a black-box manner.
3. Providing a foundation for further work with C/C++ or other general-purpose programming languages.

Syllabus

15 lectures are planned in three threads:

1. Programming in C/C++ (4 lectures)
   - Basics (3)
   - Object-oriented programming (1)
2. Numerics (5 lectures)
   - Basics (1)
   - Linear systems (2)
   - Linear integral equations (1)
   - Integration of ODEs (1 lecture)
3. Optimization (6 lectures)
   - Unconstrained optimization (2)
   - Constrained optimization (1)
   - Heuristic methods (1)
   - Optimization in engineering, structural optimization (2)

Syllabus — Programming in C/C++

Programming in C/C++:

1. programming paradigms, code and header files, preprocessor, compiler, linker
2. types and variables, operators, control flow statements, functions
3. pointers, arrays, structures, lists, trees, command line arguments
4. object-oriented programming: objects and classes, creating and destroying objects, overloaded operators, STL vector class

Syllabus — Numerics

Numerics:

1. number representations, floating-point numbers, round-off errors, problems and algorithms, conditioning, (in)stability
2. Linear systems I: basics, direct methods (special matrices, factorizations and decompositions, Gaussian elimination), iterative methods (stationary, Krylov subspace)
3. Linear systems II: least-squares problems, conditioning, regularization, large systems
4. Linear integral equations
5. Integration of ODEs: basics (reduction to 1st order ODE, convergence, order stability), explicit and implicit one-step methods, multistep methods, Newmark method

Syllabus — Optimization

Optimization:

1. Optimization I: optimization basics (objective function, variables, constraints), sensitivity analysis
2. Structural optimization: structural reanalysis, examples
3. Unconstrained optimization I: stop conditions, line search and trust region methods, search directions, step size, 1D case
4. Unconstrained optimization II: zero-order methods, steepest descent, conjugate gradient methods, Newton and quasi-Newton methods, least-squares problems
5. Constrained optimization: hard and soft constraints, Lagrangian and KKT conditions, penalty functions, classification of problems, linear and quadratic programming
6. Heuristic methods: No Free Lunch Theorem, simple randomization, coupled local minimizers, Nelder-Mead method, simulated annealing, evolutionary algorithms, swarm intelligence, ...

Grading

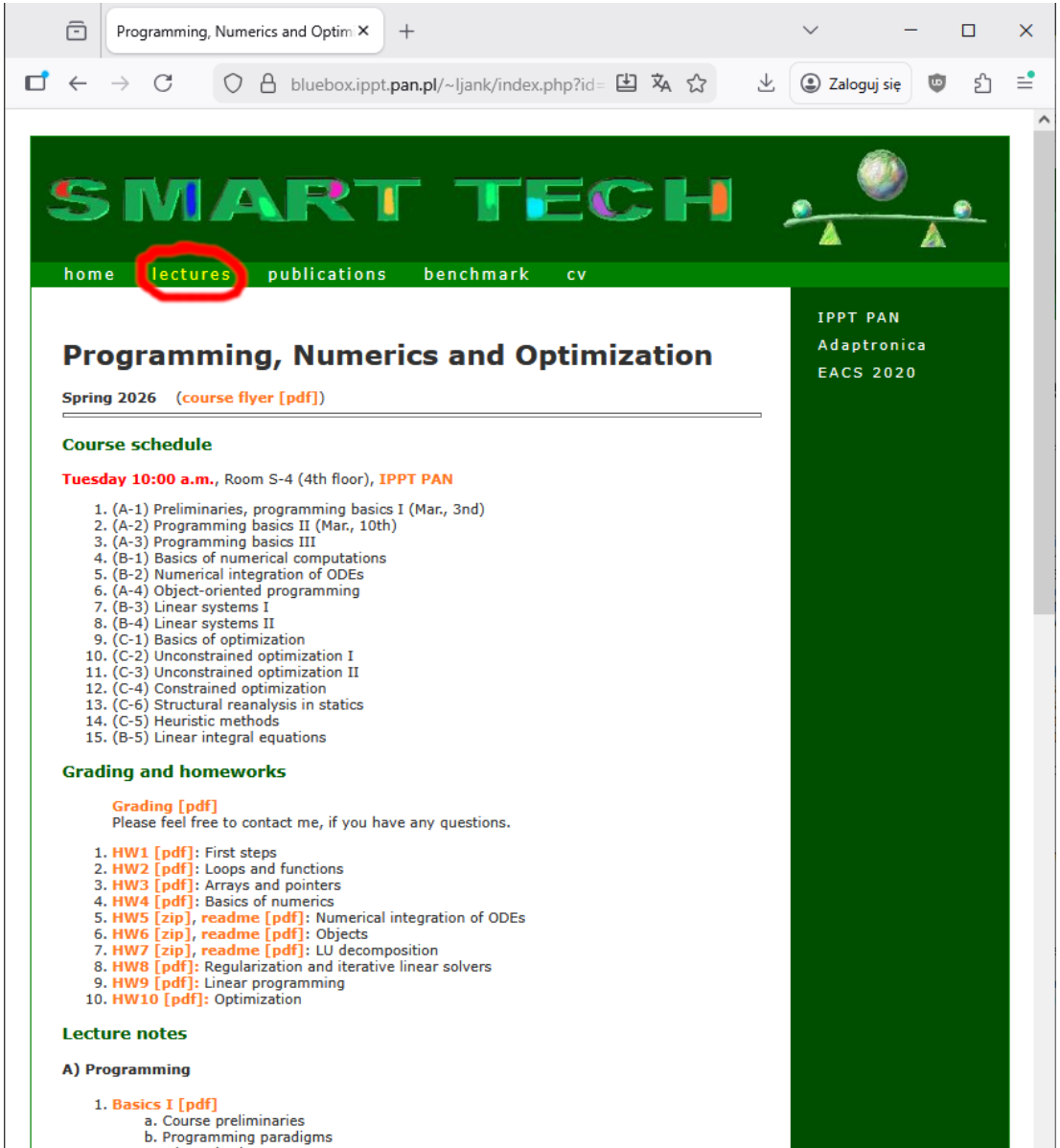
Grading will be based entirely on your homeworks (HWs). The first three HWs are obligatory, others you can freely choose from.

No.	Title	Points
HW 1	First steps (C++)	5
HW 2	Loops and functions (C++)	15
HW 3	Arrays and pointers (C++)	15
HW 4	Basic numerics	20
HW 5	Objects (C++)	20
HW 6	LU decomposition	25
HW 7	Regularization and iterative linear solvers	25
HW 8	ODE integration	25
HW 9	Linear programming	25
HW 10	Structural optimization	25
	Total	*200*

Points	Grade
91–100	3.0
101–110	3.5
111–120	4.0
121–130	4.5
131–200	5.0

Webpage

http://bluebox.ippt.pan.pl/~ljank



Lecture notes
Homeworks
Course schedule
etc.

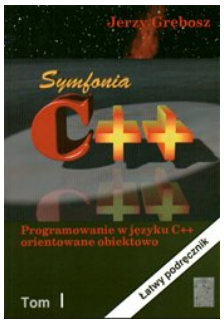
Questionnaire

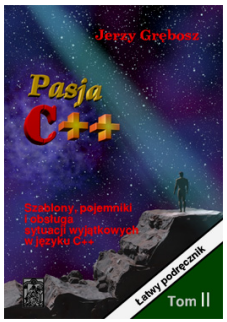
Your

- educational background,
- computer/programming skills,
- comments on the proposed schedule,
- wishes and expectations.

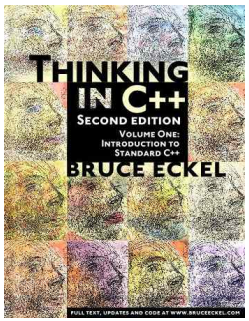
Outline

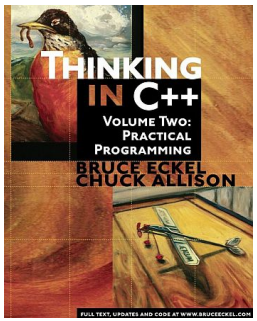
Reading material — C/C++, programming



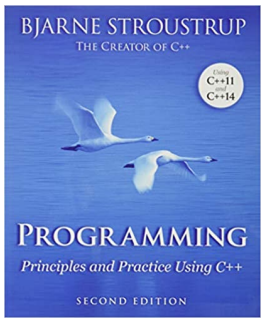


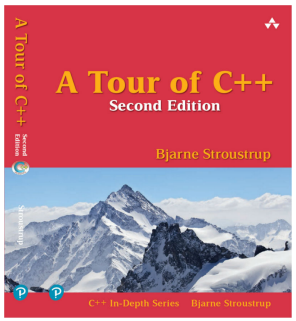
Jerzy Gręboszz,
Symfonia C++ (in Polish)





Bruce Eckel,
Thinking in C++,
once available at `http://mindview.net/Books`





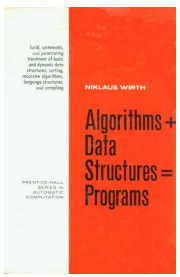
Bjarne Stroustrup,
Programming: Principles and Practice Using C++
A Tour of C++

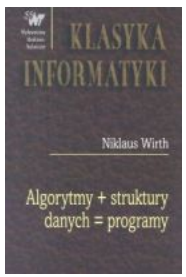
C++ reference: `https://en.cppreference.com/`

Reading material — C/C++, programming

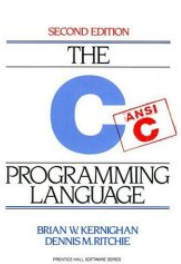
C++ Core Guidelines

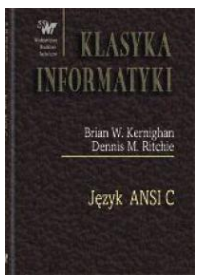
`https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`





Niklaus Wirth
Algorithms + Data Structures = Programs



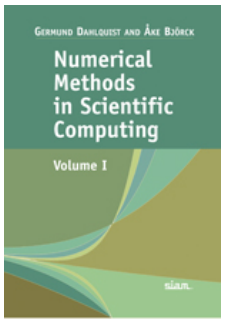


Brian W. Kernighan,
Dennis M. Ritchie,
ANSI C.

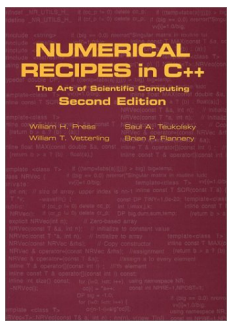
Reading material — C/C++, programming

Kernighan, Ritchie, ANSI C

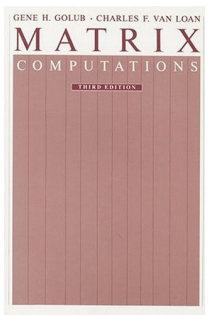
Reading material — Numerics



Germund Dahlquist, Åke Björck,
Numerical Methods in Scientific Computing, vol. 1,2



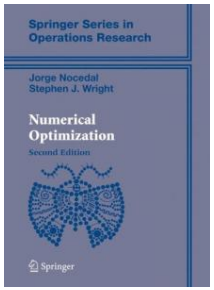
Numerical Recipes in C++ (C, Fortran),
`http://nr.com`.



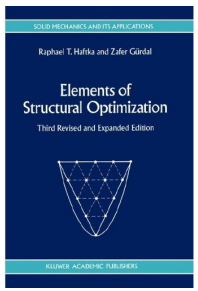
Gene H. Golub, Charles F. Van Loan,
Matrix computations.

MathWorld
`http://mathworld.com`

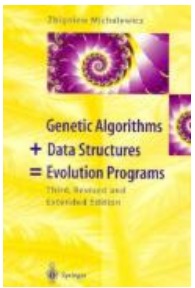
Reading material — Optimization



Jorge Nocedal, Stephen Wright,
Numerical Optimization



R. T. Haftka, Zafer Gürdal,
Elements of Structural Optimization.



Zbigniew Michalewicz,
Genetic Algorithms + Data Structures
= Evolution Programs.

Germund Dahlquist, Åke Björck,
Numerical Methods in Scientific Computing,
vol. 2, Chapter 10: *Iterative methods for linear systems*, pp. 469–560.

Outline

Programming paradigms

There is a certain number of fundamental programming concepts:

- record,
- named state,
- lexically scoped closure,
- sequentiality vs. concurrency/independence,
- observable nondeterminism,
- data abstraction, polymorphism and inheritance,
- exceptions, etc.,

which define the way of representing and handling the two basic elements of any program, that is the

- data and
- operations.

Programming paradigms

programming paradigm

A *programming paradigm* is a fundamental style of computer programming. Each paradigm supports a certain set of concepts.

There is a large number of paradigms (or wannabe paradigms). However, there is a fundamental opposition between

1. the *imperative programming*, which focuses on data, program state and control flow (how-to),
2. the *declarative programming*, which focuses on functions and declares relations (what-is).

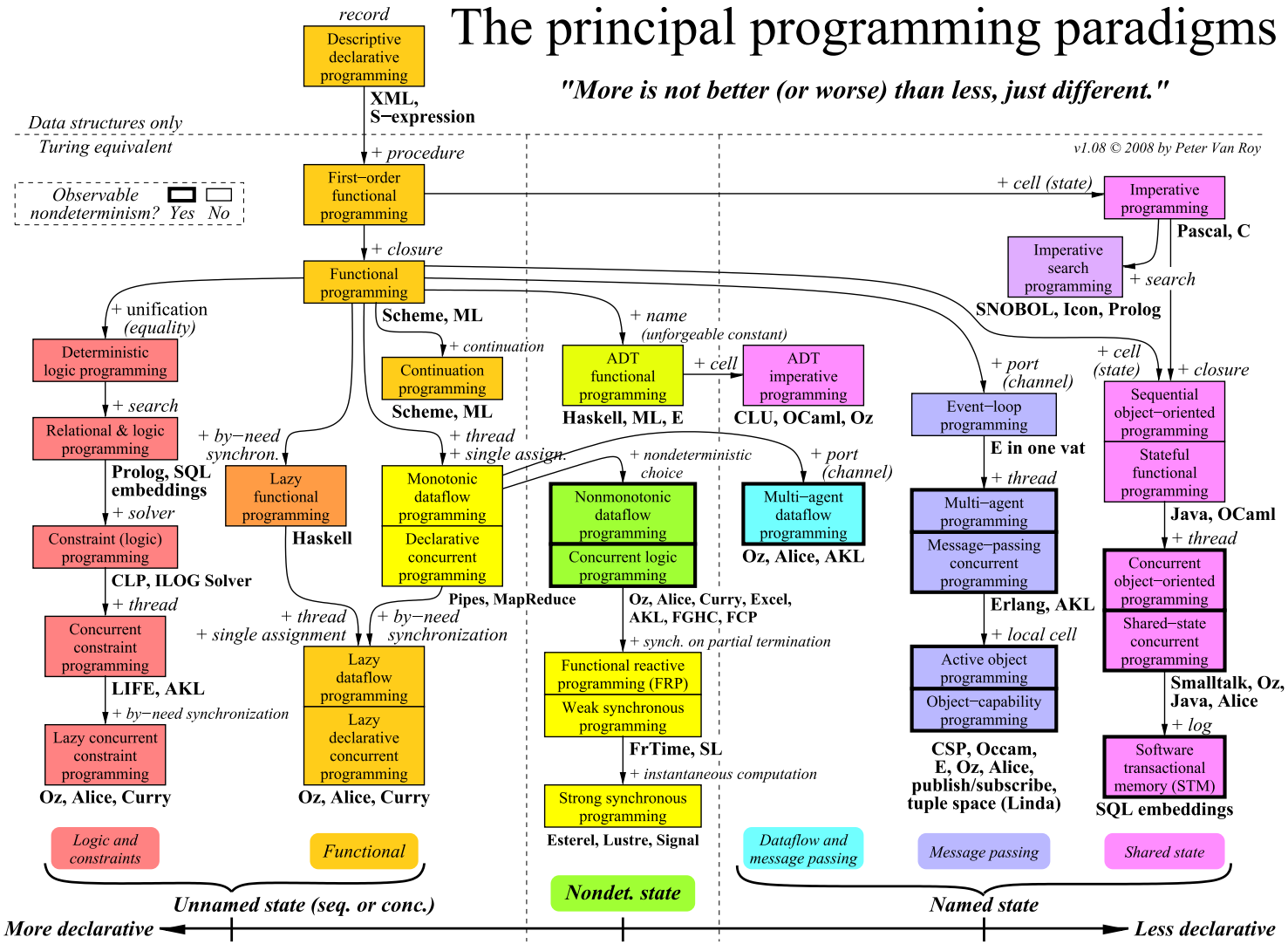
Programming paradigms

The most commonly used paradigms seem to be:

- procedural (C, Pascal),
- object-oriented (C++, Java, Smalltalk),
- array-based (APL, J),
- functional (Haskell, Erlang, J),
- logic (Prolog).

A programming language can be designed to support one particular programming style, but also a certain subset of styles (e.g. one for small applications and another one for large systems).

Programming paradigms[3]



[3]Peter van Roy, "Programming Paradigms for Dummies: What Every Programmer Should Know"

Imperative programming

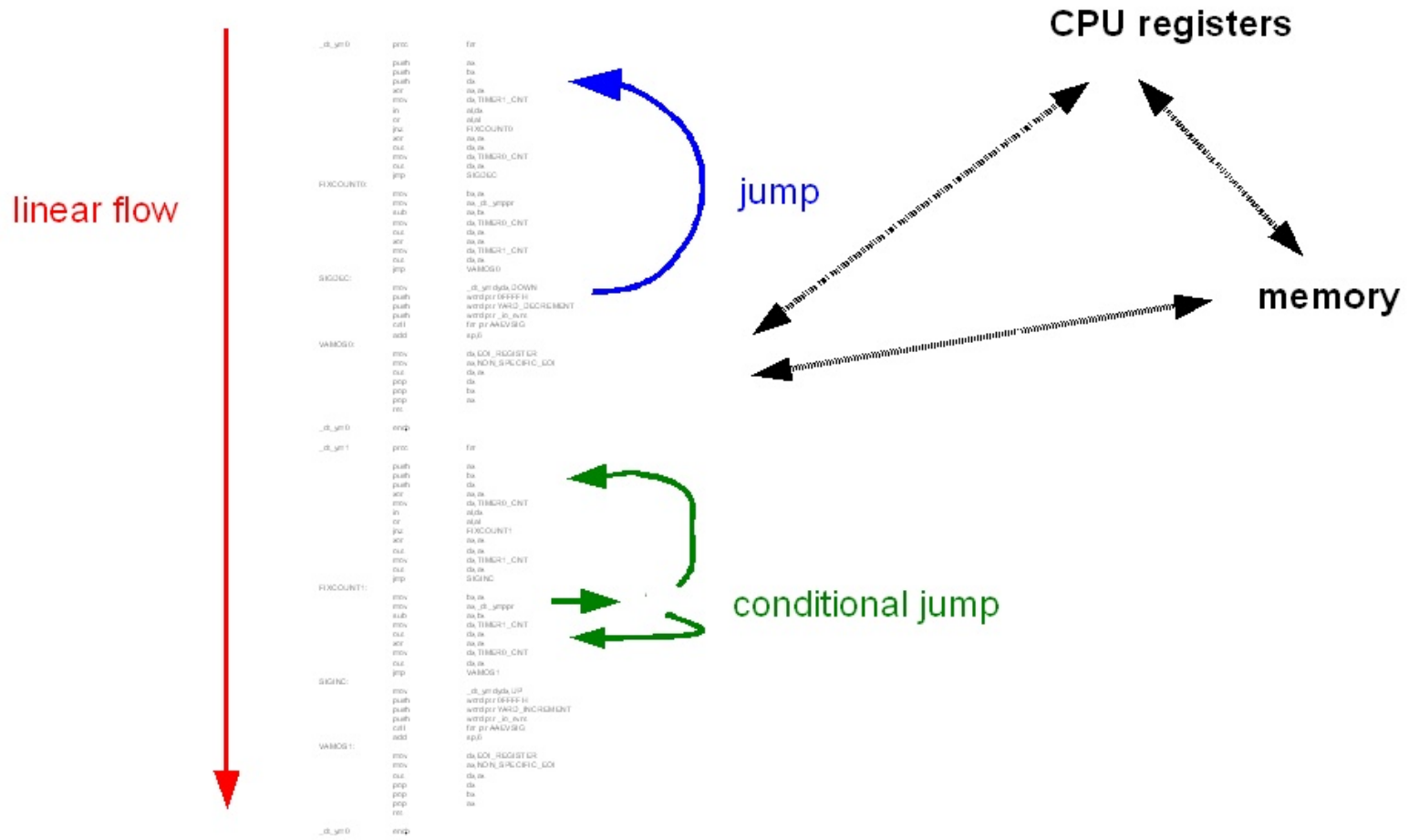
Imperative-style programs describe

- what is to be computed
- along with and inseparable from the details *how* it is to be computed (implementation): control flow, data storage, etc.

All programs, irrespective of the paradigms, are ultimately transformed for execution to machine code, which is

- imperative style and
- very inconvenient for programming.

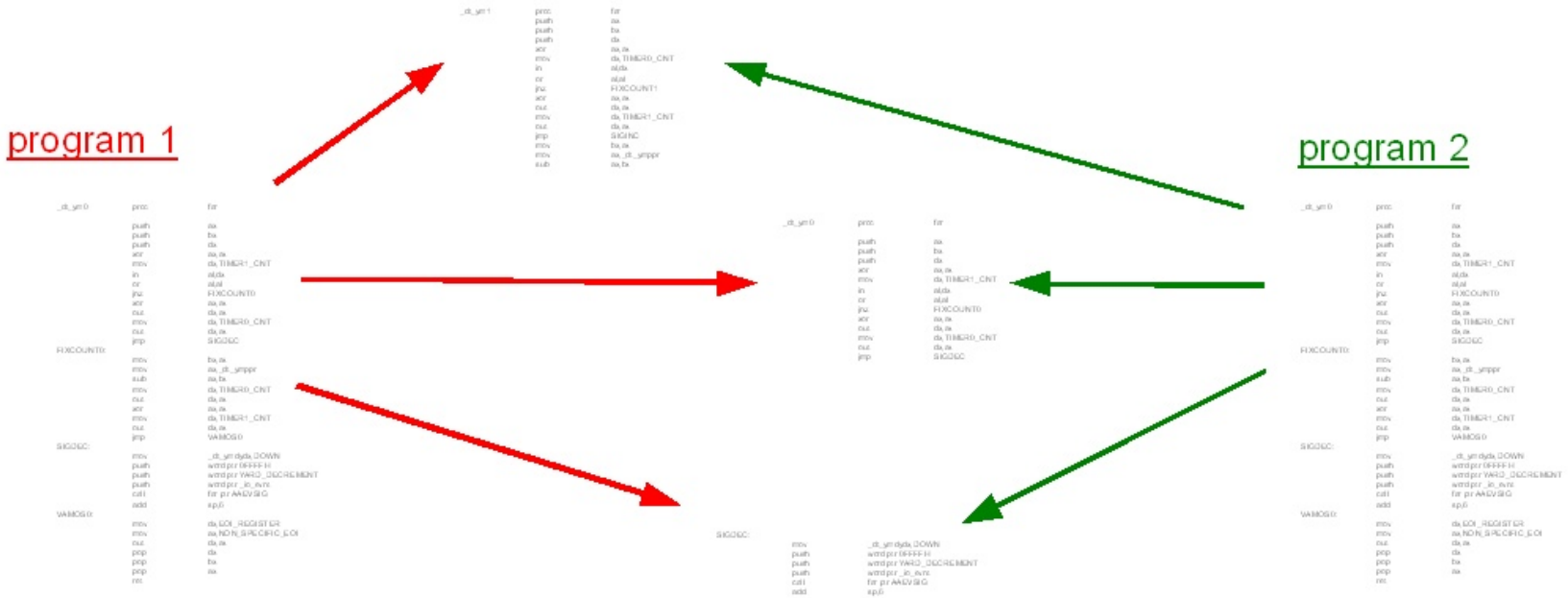
Imperative programming — assembly



Procedural programs

The imperative paradigm allowed *procedural* programming to evolve.

- *Reusable code* parts are collected and separated in procedures (functions, routines) for repetitive or later use.
- The programs got structured.



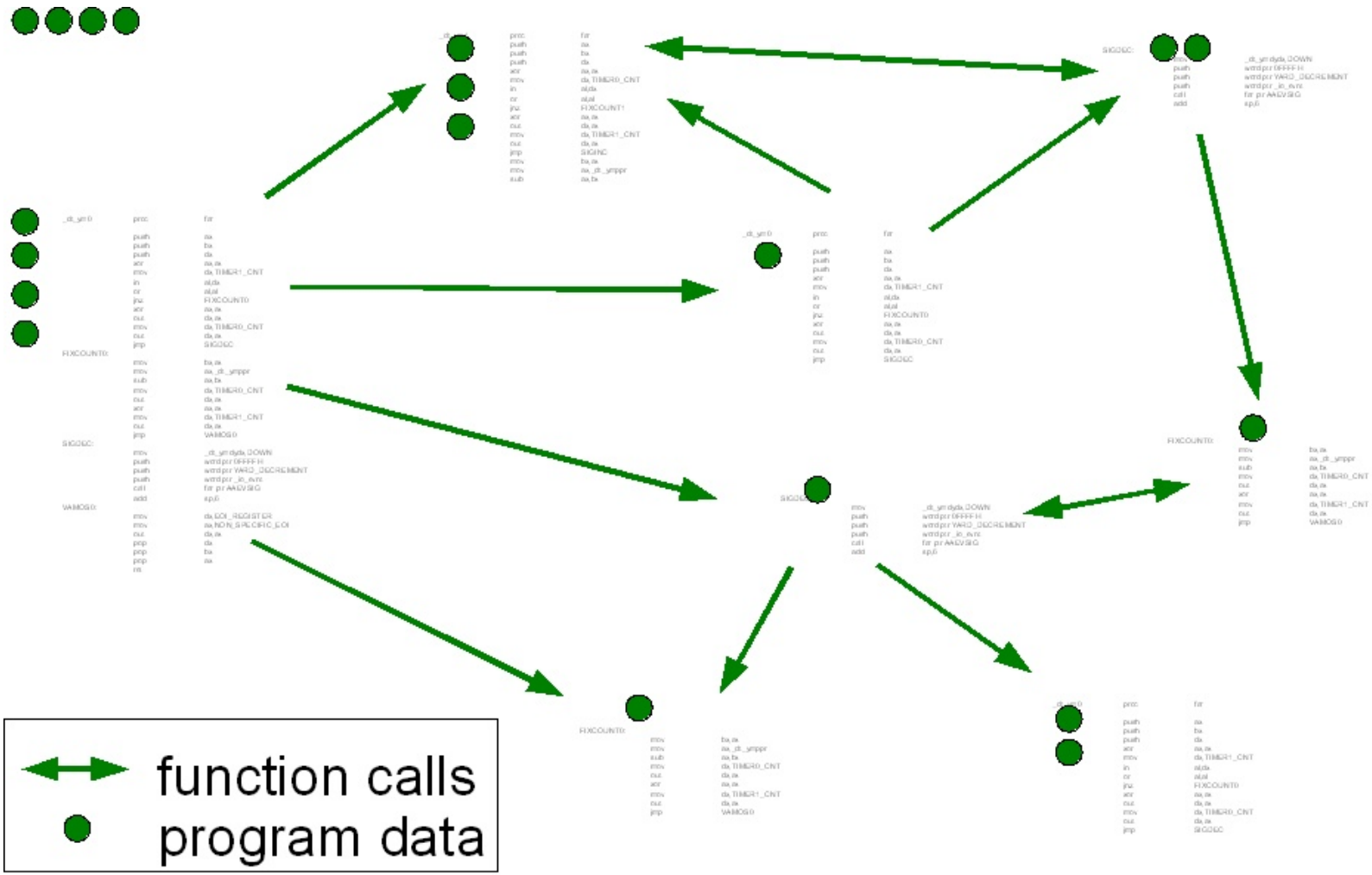
Procedural programs

In larger procedural programs, the

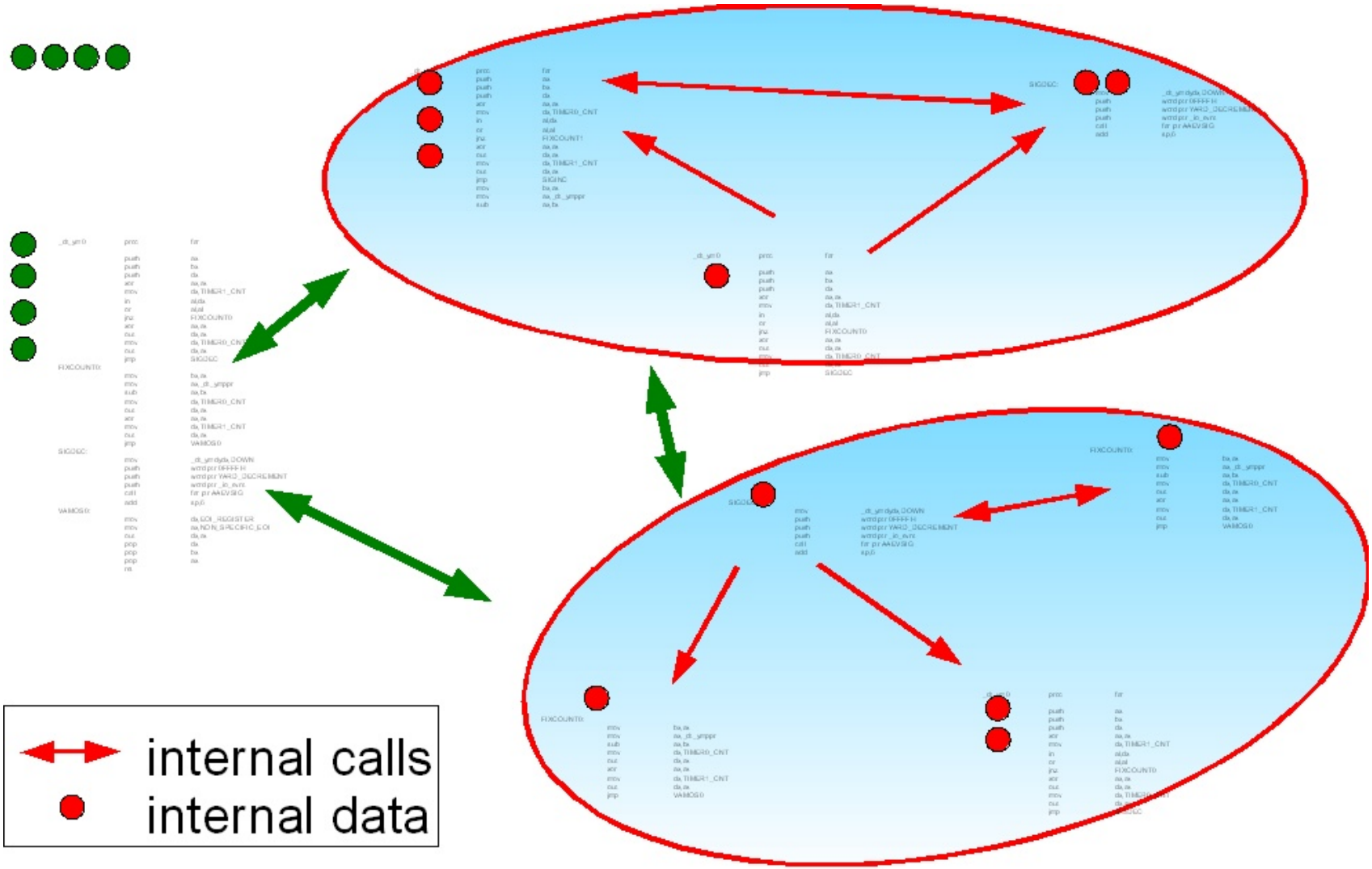
- the data and
- the structure of the program (procedure call order)

tend to be increasingly scattered and hard to manage.

Procedural programs



Object-oriented programs



Object-oriented paradigm

Object-oriented languages group data and operations together in re-usable *objects*, which represent entities from the real world being modelled (data abstraction).

A procedural program is in fact a *group of tasks* (procedures) to compute, while an object-oriented program might be seen as a *collection of cooperating objects*, which

- receive/send messages and process data,
- reflect the structure of the system being modelled.

In this way object-oriented languages support modularity by separation, encapsulation and protection of data and operations.

Object-oriented languages — two views

Since 1990s object-oriented languages are in a wide use in mainstream software development.

+ By focusing on objects and their interactions rather than on computing tasks, object-oriented languages follow the natural way people perceive the world and deal with it.
– *Object-oriented programming is popular in big companies, because it suits the way they write software. At big companies, software tends to be written by large (and frequently changing) teams of mediocre programmers. Object-oriented programming imposes a discipline on these programmers that prevents any one of them from doing too much damage. The price is that the resulting code is bloated with protocols and full of duplication. This is not too high a price for big companies, because their software is probably going to be bloated and full of duplication anyway.* /Paul Graham, `http://www.paulgraham.com/noop.html`/

Object-oriented languages — two views

– Object-oriented programming enforces an additional abstraction layer, which is often unnecessary and hindering in small projects.

\+ It is useful in large projects as it

- follows the natural way people perceive the world and manage its complexity,
- considerably simplifies management and maintenance of the system.

\+ Even small projects may benefit from specialized reusable objects.

Declarative languages

Functional and logic languages are examples of *declarative languages*, which

- specify only *what* is to be computed (done, shown, etc.) and
- leave the implementation details (sequencing of computation, organization of the data storage) to the interpreter/compiler.

For example, in *logic languages* (like Prolog) the programmer represents the problem to be solved by *declaring a set of logical relations*, which are then tackled by a theorem prover or a model generator.

Array-processing languages

While other languages apply operations to scalars, which can be then explicitly grouped in higher-dimensional data structures, array-processing languages (vector languages: APL, J, K, Q) apply all operations transparently to vectors, matrices, and higher dimensional arrays.

- Each function must have a *rank*, which is defined separately with respect to its all arguments. The rank specifies the dimensionality of the "unit" argument.
- If an argument to a function is of a higher dimensions than the corresponding rank, implicit looping is performed. In practice, almost all loops can be treated implicitly, which makes programs very concise and even terse.

Array-processing languages — example

mean of an array in C++

```
double mean(double tab[], int no){
    double s=0;
    for(int i=0; i<no; ++i)
        s += tab[i];
    return s/no;
}
```

mean of a list in J

```
mean =. +/ % #
```

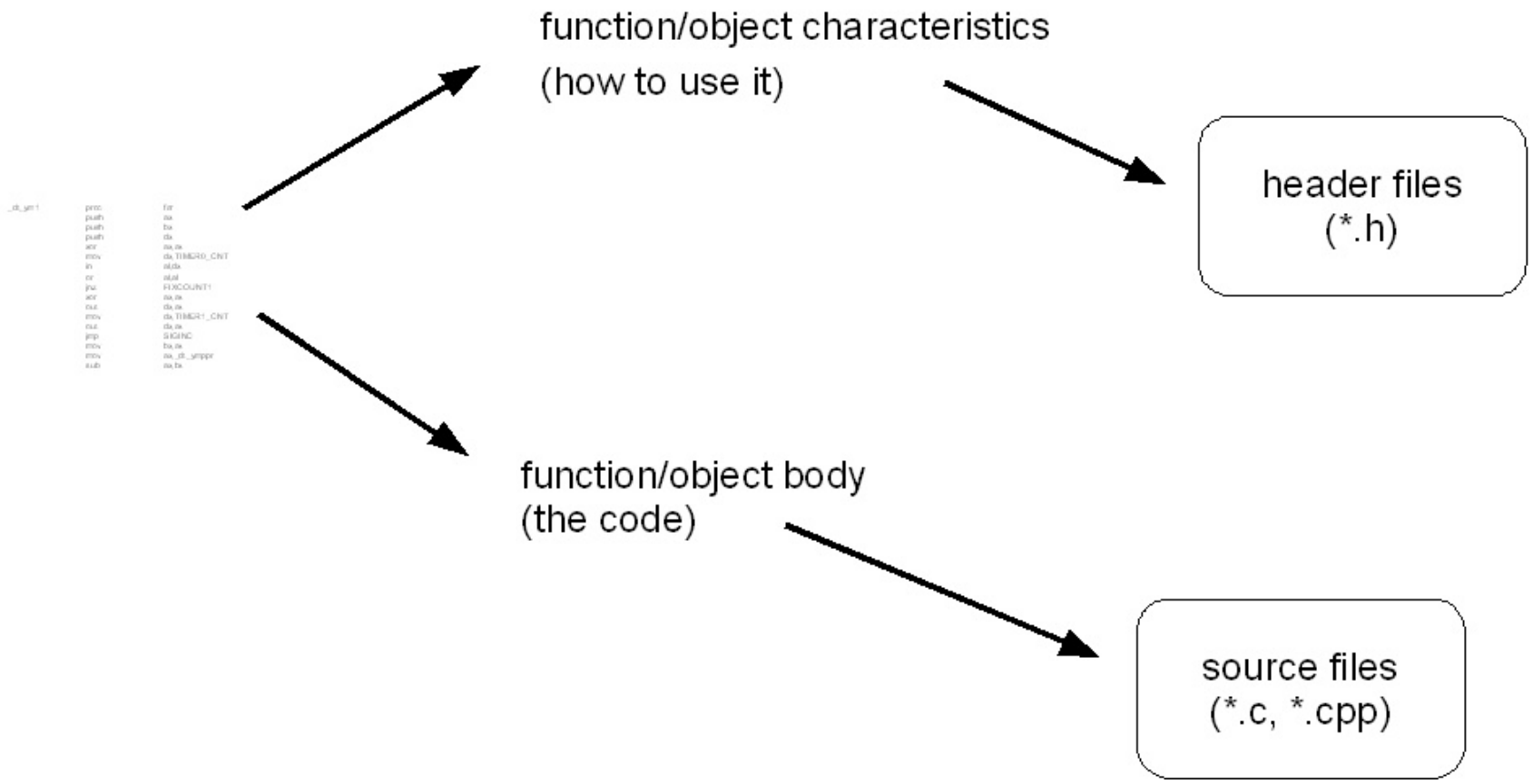
www.jsoftware.com

- The C++ version is applicable only to 1D arrays.
- The J version computes the mean item of a list, whatever type it is, so it can be also applied to matrices to compute the mean row, to 3D arrays to compute the mean 2D matrix, etc.
- In J, the means of all the rows of a matrix can be also computed by an explicit modification of the rank: `mean"1`.

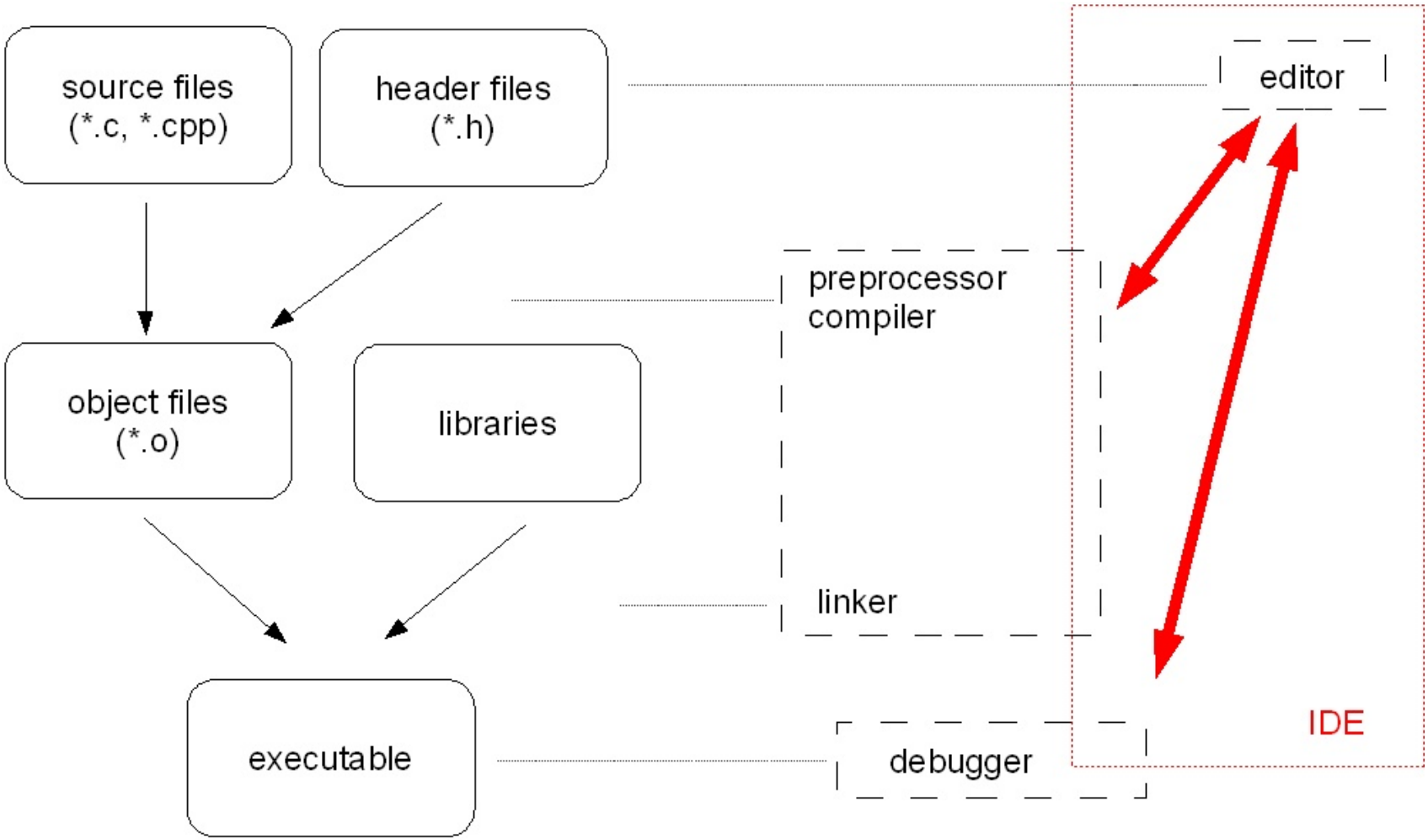
```
quicksort=: (($:@(<#[), (=#[), $:@(>#[)) ({~ ?@#)) ^: (1<#)
```

Outline

Source and header files



From source files to executables



Integrated Development Environments (IDE)

- Microsoft Visual C++
  `http://www.microsoft.com/express`
- CLion `https://www.jetbrains.com/clion/`
- Code::Blocks `http://www.codeblocks.org`
- ...
- Geany `http://www.geany.org`
- Eclipse + CDT `http://www.eclipse.org`
- Netbeans `http://www.netbeans.org/features/cpp`
- Linux: any text editor + gcc

Or any other.

A simple example

Example

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

/* A very simple example,
 * to illustrate basic concepts.
*/

int main() {
    cout <<"Hi, that's me! " <<endl;
    cout <<"How are you?";

    // Dev C++ users might add: system("pause");
    // Return value 0 means terminated ok, no errors.
    return 0;
}
```

Outline

Homework 1 (5 points)

First steps

1. Install on your computer a compiler/IDE of your choice.
2. Write and compile a simple code introducing yourself (e.g. printing 'My name is...').
3. Send me (`ljank@ippt.pan.pl`) your source file (*.cpp) and information on your configuration (system, IDE).