CS-231 – LESSON 01 – PROGRAMMING PARADIGMS

GIUSEPPE TURINI

TABLE OF CONTENTS

STUDY GUIDE

Study Material:

- These slides and your notes.
- Your textbooks for the course prerequisites (CS-101 and CS-102).

Homework:

- Exercises to review course prerequisites (CS-101 and CS-102).
- Install and configure your IDE for C/C++ SW development.

Additional Resources:

- *"The C Programming Language (2nd Edition)."*
- *"A Tour of C++."*
- *"Functional Programming in C++."*

COURSE INTRODUCTION

Syllabus Overview:

- Contacts, schedule, office hours, topics, and learning objectives.
- Textbooks, course material, code repository, and technical tools.
- Course policies, attendance tracking and student assessment, and grading.

Student Survey:

- Professional activity (current and future jobs).
- Programming language experience (languages used and to do what).
- Learning expectations (academically and professionally).

Course Motivation:

- Designing SW, choose the programming language/paradigm properly.
- In maintaining code, understanding the programming paradigm used is critical.

PREREQUISITES REVIEW

Review of course prerequisites review for CS-231: *"Programming Language Paradigms"*:

- CS-101: *"Computing and Algorithms 1"*
  - **Basic Programming:** Variables and constants, data types, operators, functions, scope, debugging and testing, selection and iteration, arrays.
  - **Object-Oriented Programming (OOP):** Classes and objects, attributes and methods, references, arrays of objects, exceptions, input/output.
- CS-102: *"Computing and Algorithms 2"*
  - **Intermediate Programming:** OOP principles, modifiers, overloading and overriding, generics and interfaces, memory management, type casting.
  - **Data Structures:** Recursion, ADTs, lists, stacks and queues, trees, heaps, hashtables, graphs, array-based and reference-based implementations.

See: catalog.kettering.edu/coursesaz/undergrad/cs

PROGRAMMING PARADIGMS

This section presents the most common programming paradigms, focusing on imperative, procedural, functional, and object-oriented.

In particular, we will discuss:

- Generations of programming languages (1GLs, 2GLs, 3GLs, 4GLs, and 5GLs).
- Most common programming paradigms, and their characteristics.
- Imperative programming and declarative programming.
- Structured programming and procedural programming.
- Object-oriented programming.
- Functional programming.

GENERATIONS OF PROGRAMMING LANGUAGES

Programming languages can be classified into different generations.
This classification is somewhat based on design trends:

- 1st-generation programming languages (1GLs), also known as machine languages.
- 2nd -generation programming languages (2GLs), also known as assembly languages.
- 3rd-generation programming languages (3GLs): Java, C, C++, C#, etc.
- 4th-generation programming languages (4GLs): SQL, R, etc.
- 5th-generation programming languages (5GLs): Prolog, etc.

This "*Generational*" classification is not precise.

A better classification is based on programming paradigms.

PROGRAMMING PARADIGM

Programming Paradigm:	A style for structuring and conceiving the design and implementation of computer programs. It includes principles, concepts, and techniques to define the structure, organization, and flow of the code, as well as methodologies for problem-solving and computations.

A programming language can support one or more programming paradigms!

Programming paradigms can involve different programming characteristics.

Some paradigms describe the implications of the code execution (e.g., allowing side effects).

Other paradigms focus on the way code is organized (e.g., units including state and behavior).

There is some overlap between paradigms, but each paradigm has its own main features!

COMMON PROGRAMMING PARADIGMS

Two basic paradigms represent the different approaches to computer programming:

- **Imperative:** Code directly controls execution flow and state change. It focuses on how to execute, defining control flow as statements that change a program state.
- **Declarative:** Code declares properties of desired result, but not how to compute it. It focuses on what to execute, defining program logic, but not detailed control flow.

Other main programming paradigms represent core language characteristics:

- **Procedural:** Code organized as procedures that call each other. It specifies the steps a program must take to reach a desired state.
- **Functional:** Desired result declared as output of function evaluations. It treats programs as evaluating mathematical functions and avoids state and mutable data.
- **Object-Oriented (OO):** Code as objects (data and methods) with their interactions.

OTHER PROGRAMMING PARADIGMS

These are other types of programming that can be implemented using different paradigms:

- **Event-Driven:** Control flow determined by events. For example, an event could be: a sensor input, or a user action, or a message from another program or thread.
- **Automata-Based:** Program is a model of a Finite State Machine (FSM) or any other form of automation (robotics).

- **Structured:** An imperative programming paradigm with more logical program structure (structograms, indentation, no or limited use of goto statements).
- **Reactive:** A declarative programming paradigm in which the desired result is declared with data streams and the propagation of change.
- **Logic:** A declarative programming paradigm in which the desired result is declared as the answer to a question about a system of facts and rules.

COMPARISON OF PARADIGMS

PARADIGM	TRAITS	LANGUAGES
Imperative	Direct assignments, common data structures, global variables.	C, C++, Java, Python, etc.
Declarative	4th-generation languages, spreadsheets, report program generators.	SQL, regular expressions, etc.
Procedural	Local variables, sequence, selection, iteration, and modularization.	C, C++, Java, Python, etc.
Functional	Lambda calculus, compositionality, formula, recursion, ref. transp., no side effects.	C++, C#, Java (8+), Python, etc.
Object-Oriented	Objects, methods, info hiding, abstraction, encapsulation, polymorphism, inheritance.	C++, C#, Java, Python, etc.
Event-Driven	Main event listening loop, event handlers, asynchronous processes.	JavaScript, Visual Basic, etc.
Automata-Based	State enumeration, control variable, state changes, isomorphism, state-transition table.	Abstract State Machine Language.

IMPERATIVE PROGRAMMING

Imperative Programming: A programming paradigm of software that uses statements that change the program state.

An imperative program consists of commands for the computer to perform.

Most computer languages use imperative programming.

These are imperative programming languages: Algol, Basic, C, C++, C#, COBOL, Fortran, Java, PHP, Python, Ruby, etc.

Note: Imperative programming is used in contrast to **declarative programming** (which focuses on the program result, without specifying details of how to achieve it).

DECLARATIVE PROGRAMMING

Declarative Programming: A programming paradigm of software that expresses the logic of a computation without describing its control flow.

Usually declarative languages attempt to minimize side effects.

Declarative programming may greatly simplify writing parallel programs.

These are declarative programming languages: SQL, regular expressions, Prolog, OWL, SPARQL, Datalog, XSLT, etc.

Note: Declarative programming is used in contrast to **imperative programming** (which implements algorithms in explicit steps.)

IMPERATIVE VS DECLARATIVE CODE

Example: Consider the problem of selecting all even values from a list of integers.

The following are code examples in Java-like pseudocode.

With imperative programming we tell the computer the operations to perform (substantially an implementation step-by-step of our algorithm).

```
List<int> list = new List<int> { 1, 2, 3, 4, 5 };
List<int> res = new List<int>();
foreach(int val in list) { // Imperative code.
   if( val % 2 != 0 ) { res.Add(val); } } // Imperative code.
```

With declarative programming we tell the computer the result we want (but we do not specify the details on how to compute it).

```
List<int> list = new List<int> { 1, 2, 3, 4, 5 };
List<int> res = list.Where( val => val % 2 != 0); // Declarative code.
```

STRUCTURED PROGRAMMING

Structured Programming: A programming paradigm aimed at improving computer programs by extensively using structured control flow constructs (selections, iterations, blocks, and functions).

Structured programming is frequently used with deviations in some cases (e.g., exception handling).

Note: Even if the *"goto"* statement has now largely been replaced by structured constructs (selections, and iterations), few programming languages are purely structured.

The most common deviation is the *"return"* statement. This results in multiple exit points, instead of the single exit point required by pure structured programming.

PROCEDURAL PROGRAMMING

Procedural Programming: A programming paradigm (classified as a subtype of imperative programming) that involves implementing the behavior of a computer program as procedures (functions or subroutines) that call each other.

A procedural program is a series of steps that forms a hierarchy of calls to its constituent procedures.

Procedural languages include: C, C++, C#, Java, Lisp, PHP, Python, etc.

SW development practices are often focused on procedural programming. For example:

- **Modularity** is organizing procedures into separate modules, each with its own purpose.
- **Scoping** is minimizing the scope of variables and procedures.
- **Sharing** is supported by procedures with well-defined interfaces and self-contained.

OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP): A programming paradigm (classified as a subtype of imperative programming) based on the concept of objects (and classes).

Objects can contain data (attributes or properties) and operations (functions or methods). Classes are user-defined types, used to describe/create objects.

Significant OOP languages include: C++, C#, Java, JavaScript, Objective-C, PHP, Python, etc.

Many of the most widely used programming languages are multi-paradigm and they support object-oriented programming to a greater or lesser degree.

Typically, OOP is supported in combination with imperative programming, procedural programming and functional programming.

FUNCTIONAL PROGRAMMING

Functional Programming:	A programming paradigm (classified as a subtype of declarative programming) where programs are constructed by applying and composing functions. In functional programming, functions are treated as first-class citizens (have names, can be passed as arguments, and returned as results).

Functional programming is very different from imperative programming:

- **Side Effects:** Functional programming avoids side effects, which are used in imperative programming to implement state and I/O.
- **Higher-Order Functions:** Functions of functions are rarely used in imperative programming (see references/pointers to functions).

SETUP IDE FOR C/C++ PROGRAMMING

In this course, each student is free to select any C/C++ IDE.
This freedom means that each student is also responsible for the IDE configuration.

Once an IDE is configured, any student should learn how to:

- Compile code (in multiple modes, if available).
- Execute code (in multiple modes, setting run arguments, if available).
- Use breakpoints (placing them, and running code line-by-line, etc.).
- Inspecting variables at runtime (expanding objects/ADTs, checking references, etc.).

The official course IDE for C/C++ SW development is *"Eclipse IDE for C/C++ Developers."*
What follows is a brief guide to configure the course IDE.

WHY ECLIPSE FOR C/C++ PROGRAMMING

Why Eclipse is the official course IDE for C/C++ programming?

- Eclipse is an open-source IDE supported by IBM.
- Eclipse is popular for Java, but also supports C/C++, PHP, Python, Perl, etc.
- Eclipse is cross-platform (Windows, Linux and Mac OS).

See: www.eclipse.org

See: www.eclipse.org/downloads/packages

See: *"Eclipse - Eclipse IDE for C/C++ Developers"*

See: *"Nanyang Technological University – How To Install Eclipse for C/C++ Programming"*

INSTALLATION OF A C/C++ COMPILER

To use Eclipse for C/C++ programming, you need a C/C++ compiler.

On Windows, you could install either MinGW GCC or Cygwin GCC.

Note: Choose MinGW if you are not sure (it is simpler, lighter, and easier to configure).

See: *"Nanyang Technological University – How To Install Eclipse for C/C++ Programming"*

On Linux ...

See: *"How to Compile and Run C/C++ Code on Linux"*

On Mac OS ...

See: *"Mac Tutorial to Set Up C/C++ Compiler"*

INSTALLATION OF ECLIPSE FOR C/C++

There are 2 ways to install Eclipse for C/C++:

- If Eclipse is already installed on your machine:
  - Install the package *“Eclipse IDE for C/C++ Developers”* (see instructions online).
- If Eclipse is **not** already installed on your machine:
  - Download *"Eclipse IDE for C/C++ Developers"* (see link).
  - Then, unzip the downloaded file into a directory of your choice.

You do **not** need to do any configuration!

Note: If the binaries of the C/C++ compiler are included in the PATH environment variable (on Windows), *"Eclipse IDE for C/C++ Developers"* automatically configure itself.

See: *“Nanyang Technological University – How To Install Eclipse for C/C++ Programming”*

FIRST C/C++ PROGRAM

Follow this steps to check that you can compile/run C/C++ code:

- Launch *"Eclipse IDE for C/C++ Developers."*
- Select your workspace (main working directory)
- Create a new C/C++ project for each C/C++ application (see online instructions).
- Code your first C/C++ program, for example:

```c
// Source code file name: HelloWorldC.c
#include <stdio.h> // To use external code (in other C files).
int main() { // Entry point in C.
   printf("Hello, World!\n"); // Printing text on console.
   fflush(stdout); // Force printf to print on console (otherwise is buffered).
   return 0; // Default return value for success.
} // End main function.
```

See: *"Nanyang Technological University – How To Install Eclipse for C/C++ Programming"*

REFERENCES

- *“Wikipedia - Programming Language Generations”*
- *“Wikipedia - Programming Paradigms”*
- *“Wikipedia – Comparison of Programming Paradigms”*
- *“Wikipedia – Imperative Programming”*
- *“Wikipedia – Declarative Programming”*
- *“Wikipedia – Structured Programming”*
- *“Wikipedia – Procedural Programming”*
- *“Wikipedia – Object-Oriented Programming”*
- *“Wikipedia – Functional Programming”*

- *“Nanyang Technological University – How To Install Eclipse for C/C++ Programming”*

APPENDICES

- 1GLs, 2GLs, 3GLs, 4GLs, and 5GLs.
- Syntactic Sugar.
- Side Effects.
- Referential Transparency.

1ST-GENERATION PROGRAMMING LANGUAGES

First-generation programming languages (1GLs), also known as machine languages, were the earliest languages used to program computers.

Machine language is the only programming language that the computer can understand directly (without translation). It is a language made up of entirely 1s and 0s. Each machine language is a low-level language, and must be written considering the individual characteristic of a given processor (machine-dependent or hardware-dependent).

Programs in machine language have fast execution speed and efficient memory usage!

See: en.wikipedia.org/wiki/first-generation_programming_language

See: en.wikipedia.org/wiki/machine_code

2ND-GENERATION PROGRAMMING LANGUAGES

Second-generation programming languages (2GLs), also known as assembly languages, were designed to make software development easier and more efficient.

Assembly languages are low-level and machine-dependent. They include mnemonic operation codes and symbolic memory addresses (instead of 1s and 0s).

Programs in assembly language have fast execution speed and efficient memory usage!

See: en.wikipedia.org/wiki/programming_language_generations
See: en.wikipedia.org/wiki/second-generation_programming_language
See: en.wikipedia.org/wiki/assembly_language

3RD-GENERATION PROGRAMMING LANGUAGES

Third-generation programming languages (3GLs) are more machine-independent (portable) and more programmer-friendly.

3GLs include: Java, C, C++, C#, PHP, Perl, BASIC, Pascal, Fortran, ALGOL, COBOL.

These are the main characteristics of a 3GL:

- High-level and human-friendly (syntax closer to human language).
- Machine independence (more portable compared to predecessors).
- Abstraction (more abstract facilitating the programmer, not the computer).
- Non-essential details (automation of non-essential tasks).
- Structured and object-oriented (support of structured and OOP).

See: en.wikipedia.org/wiki/programming_language_generations
See: en.wikipedia.org/wiki/third-generation_programming_language

4TH-GENERATION PROGRAMMING LANGUAGES

A fourth-generation programming language (4GL) is a high-level computer programming language that builds upon 3GLs.

4GLs include: SQL, R, ABAP, Unix Shell, PL/SQL, Oracle Reports, Halide, etc.

These are the main characteristics of a 4GL:

- High-level of abstraction (more programmer friendly).
- Designed for big data (large collections of information).
- Advanced features (DB, math optimization, GUI, web dev).

See: en.wikipedia.org/wiki/programming_language_generations

See: en.wikipedia.org/wiki/fourth-generation_programming_language

5TH-GENERATION PROGRAMMING LANGUAGES

A fifth-generation programming language (5GL) is based on problem-solving (not writing algorithms). It uses AI to make the computer solve a problem without the programmer.

5GLs include: Prolog, OPS5, Mercury, CVXGen, Geometry Expert, etc.

These are the main characteristics of a 5GL:

- Advanced abstraction (computer solves problems, not the programmer).
- Declarative syntax (declarative paradigm instead of procedural paradigm).
- Supporting natural language (human friendly).
- Highly automated (code generation, optimization, error handling).
- AI integration, and distributed and parallel computing.

See: en.wikipedia.org/wiki/programming_language_generations
See: en.wikipedia.org/wiki/fifth-generation_programming_language

SYNTACTIC SUGAR

In programming, syntactic sugar is syntax designed to make code easier to read/write. Usually it is a shorthand for code that could also be expressed in a more verbose form. A construct in a language is syntactic sugar if it can be removed from the language without any effect on what the language can do (functionality and expressive power remain the same).

Example: Augmented assignment (compound assignment) operators.

```
a += b; // Syntactic sugar, equivalent to: a = a + b;
```

Example: Method calling in object-oriented programming (OOP).

```
obj.method( par1, par2, par3 );
// Syntactic sugar, equivalent to: method( obj, par1, par2, par3 );
```

See: en.wikipedia.org/wiki/syntactic_sugar

SIDE EFFECTS

In programming, a statement/function/expression has a **side effect** if it modifies some variables outside its local environment, so: the statement/function/expression has an observable effect other than its primary effect.

Example: Consider the assignment operator in C.

```
a = b; // Assignment is also an expression that evaluates to b.
// The side effect here is storing the right-value into the left-value.
```

Example: Consider a method in Java.

```
public class MyClass {
   private int att = 0;
   public method( int arg ){ this.att += arg; }
    // Method changes the object state (side effect).
}
```

See: en.wikipedia.org/wiki/side_effect_(computer_science)

SIDE EFFECTS (2)

Referential transparency means that a expression (operation or function call) can be replaced with its value.

Referential transparency requires that the expression (operation or function call) is **pure**: deterministic (always giving the same value for the same input) and side-effect free.

Note: The absence of side effects is a necessary condition (but not sufficient) to achieve **referential transparency**.

Temporal side effects are side effects caused by the time taken for an operation to execute.

Note: Temporal side effects are usually ignored when discussing side effects and referential transparency.

See: en.wikipedia.org/wiki/side_effect_(computer_science)