Module I
Introduction

Role of Programming Languages

Reasons for Studying Concepts of Programming Languages:

- ***Increased capacity to express ideas.***
- Those with only a weak understanding of natural language are limited in the complexity of their thoughts, particularly in depth of abstraction.
- The language in which Programmers develop software places limits on the kinds of control structures, data structures, and abstractions they can use; thus, the forms of algorithms they can construct are likewise limited.
- Awareness of a wider variety of programming language features can reduce such limitations in software development.
- Programmers can increase the range of their software development thought processes by learning new language constructs.

- Language constructs can be simulated in other languages that do not support those constructs directly. For example, a C programmer who had learned the structure and uses of associative arrays in Perl might design structures that simulate associative arrays in that language.

➢ ***Improved background for choosing appropriate languages.***

- Many professional programmers have had little formal education in computer science; rather, they have developed their programming skills independently or through inhouse training programs. Such training programs often limit instruction to one or two languages that are directly relevant to the current projects of the organization.
- Many other programmers received their formal training years ago. The languages they learned then are no longer used, and many features now available in programming languages were not widely known at the time.

- ***Increased ability to learn new languages***.

- The process of learning a new programming language can be lengthy and difficult, especially for someone who is comfortable with only one or two languages and has never examined programming language concepts in general.
- Once a thorough understanding of the fundamental concepts of languages is acquired, it becomes far easier to see how these concepts are incorporated into the design of the language being learned.
- For example, programmers who understand the concepts of object-oriented programming will have a much easier time learning Java than those who have never used those concepts.

- ***Better understanding of the significance of implementation***.

- In learning the concepts of programming languages, it is both interesting and necessary to touch on the implementation issues that affect those concepts.
- In some cases, an understanding of implementation issues leads to an understanding of why languages are designed the way they are.
- We can become better programmers by understanding the choices among programming language constructs and the consequences of those choices.
- Certain kinds of program bugs can be found and fixed only by a programmer who knows some related implementation details.
- Another benefit of understanding implementation issues is that it allows us to visualize how a computer executes various language constructs.
- In some cases, some knowledge of implementation issues provides hints about the relative efficiency of alternative constructs that may be chosen for a program.

- ***Better use of languages that are already known***.

- Many contemporary programming languages are large and complex.
- Accordingly, it is uncommon for a programmer to be familiar with and use all of the features of a language he or she uses.
- By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

- ***Overall advancement of computing.***
- Although it is usually possible to determine why a particular programming language became popular, many believe, at least in retrospect, that the most popular languages are not always the best available.
- In some cases, it might be concluded that a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.
- In general, if those who choose languages were well informed, perhaps better languages would eventually squeeze out poorer ones.

Programming Domains

- In this section, we briefly discuss a few of the areas of computer applications and their associated languages.

1. **Scientific Applications**

- The first digital computers, which appeared in the late 1940s and early 1950s, were invented and used for scientific applications.
- Typically, the scientific applications of that time used relatively simple data structures, but required large numbers of floating-point arithmetic computations.
- The most common data structures were arrays and matrices; the most common control structures were counting loops and selections.
- The early high-level programming languages invented for scientific applications were designed to provide for those needs.
- The first language for scientific applications was Fortran.
- ALGOL 60 and most of its descendants were also intended to be used in this area, although they were designed to be used in related areas as well.
- No subsequent language is significantly better than Fortran, so **Fortran** is still used.

2. **Business Applications**

- The use of computers for business applications began in the 1950s. Special computers were developed for this purpose, along with special languages.
- The first successful high-level language for business was **COBOL**, the initial version of which appeared in 1960. It is still the most commonly used language for these applications.

3. **Artificial Intelligence**

- Artificial intelligence (AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations.
- Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. Also, symbolic computation is more conveniently done with linked lists of data rather than arrays.
- The first widely used programming language developed for AI applications was the functional language LISP , which appeared in 1959.
- Most AI applications developed prior to 1990 were written in LISP or one of its close relatives.
- During the early 1970s, however, an alternative approach to some of these applications appeared—logic programming using the **Prolog** language.
- More recently, some AI applications have been written in systems languages such as C.

4. **Systems Programming**

- The operating system and the programming support tools of a computer system are collectively known as its systems software.
- In the 1960s and 1970s, some computer manufacturers, such as IBM, Digital, and Burroughs (now UNISYS), developed special machine-oriented high-level languages for systems software on their machines. For IBM mainframe computers, the language was PL/S, a dialect of PL/I; for Digital, it was BLISS, a language at a level just above assembly language; for Burroughs, it was Extended ALGOL.
- However, most system software is now written in more general programming languages, such as C and C++.
- The UNIX operating system is written almost entirely in C
- It is low level, execution efficient, and does not burden the user with many safety restrictions.

5. **Web Software**

- The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java.
- Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation.
- This functionality can be provided by embedding programming code in an HTML document.
- Such code is often in the form of a scripting language, such as JavaScript or PHP.

Language Evaluation Criteria

1. **Readability**

- One of the most important criteria for judging a programming language is the ease with which programs can be read and understood.
- Before 1970, software development was largely thought of in terms of writing code.
- In the 1970s, however, the software life-cycle concept was developed; coding was relegated to a much smaller role, and maintenance was recognized as a major part of the cycle, particularly in terms of cost.
- Because ease of maintenance is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages.
- The characteristics that contribute to the readability of a programming language are:

i. ***Overall Simplicity***

- The overall simplicity of a programming language strongly affects its readability.
- A language with a large number of basic constructs is more difficult to learn than one with a smaller number.
- Readability problems occur whenever the program's author has learned a different subset from that subset with which the reader is familiar.
- A second complicating characteristic of a programming language is feature **multiplicity**—that is, having more than one way to accomplish a particular operation. For example, in Java, a user can increment a simple integer variable in four different ways:

```
count = count + 1
count += 1
count++
++count
```

- A third potential problem is **operator overloading**, in which a single operator symbol has more than one meaning.
- It can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly.
- For example, it is clearly acceptable to overload + to use it for both integer and floating-point addition. In fact, this overloading simplifies a language by reducing the number of operators.
- This overloading simplifies a language by reducing the number of operators. However, suppose the programmer defined + used between single-dimensioned array operands to mean the sum of all elements of both arrays.
- This very simplicity, however, makes assembly language programs less readable. Because they lack more complex control statements, program structure is less obvious

In procedural programming languages like C, functions are orthogonal constructs. They can be defined independently of each other and can be called in any valid combination, without dependencies or conflicts. This allows programmers to compose complex programs by combining smaller functions in a modular and flexible manner.

ii. Orthogonality

- Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language.
- Furthermore, every possible combination of primitives is legal and meaningful.
- For example, consider data types. Suppose a language has four primitive data types (integer, float, double, and character) and two type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined.
- A lack of orthogonality leads to exceptions to the rules of the language.
- For example, in a programming language that supports pointers, it should be possible to define a pointer to point to any specific type defined in the language. However, if pointers are not allowed to point to arrays, many potentially useful user-defined data structures cannot be defined.

- Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

iii. Data Types

- The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability.
- For example, suppose a numeric type is used for an indicator flag because there is no Boolean type in the language.
- In such a language, we might have an assignment such as the following: timeOut = 1

  The meaning of this statement is unclear, whereas in a language that includes Boolean types, we would have the following:

  timeOut = true

  The meaning of this statement is perfectly clear.

iv. Syntax Design

- The syntax, or form, of the elements of a language has a significant effect on the readability of programs.
- Some examples of syntactic design choices that affect readability are:

- **Special words**.
  - Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, while, class, and for).
  - Especially important is the method of forming compound statements, or statement groups, primarily in control constructs.
  - Some languages have used matching pairs of special words or symbols to form groups.
  - C and its descendants use braces to specify compound statements. All of these languages suffer because statement groups are always terminated in the same way, which makes it difficult to determine which group is being ended when an end or a right brace appears.
  - Fortran 95 and Ada make this clearer by using a distinct closing syntax for each type of statement group. For example, Ada uses **end if** to terminate a selection construct and **end loop** to terminate a loop construct.
  - This is an example of the conflict between simplicity that results in fewer reserved words, as in C++, and the greater readability that can result from using more reserved words, as in Ada.
  - Another important issue is whether the special words of a language can be used as names for program variables. If so, the resulting programs can be very confusing. For example, in Fortran 95, special words, such as Do and End, are legal variable names.

- **Form and meaning.**
  - Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability.
  - Semantics, or meaning, should follow directly from syntax, or form.
  - In some cases, this principle is violated by two language constructs that are identical or similar in appearance but have different meanings, depending perhaps on context.
  - In C, for example, the meaning of the reserved word **static** depends on the context of its appearance.
  - If used on the definition of a variable inside a function, it means the variable is created at compile time.
  - If used on the definition of a variable that is outside all functions, it means the variable is visible only in the file in which its definition appears; that is, it is not exported from that file.

2. **Writability**

- Writability is a measure of how easily a language can be used to create programs for a chosen problem domain.
- Most of the language characteristics that affect readability also affect writability.
- This follows directly from the fact that the process of writing a program requires the programmer frequently to reread the part of the program that is already written.
- The most important characteristics influencing the writability of a language are:

i. ***Simplicity and Orthogonality***

- If a language has a large number of different constructs, some programmers might not be familiar with all of them.
- This situation can lead to a misuse of some features and a disuse of others that may be either more elegant or more efficient, or both, than those that are used.
- Therefore, a smaller number of primitive constructs and a consistent set of rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives.
- A programmer can design a solution to a complex problem after learning only a simple set of primitive constructs.

ii. Support for Abstraction

- Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored.
- Abstraction is a key concept in contemporary programming language design.
- This is a reflection of the central role that abstraction plays in modern program design methodologies.
- The degree of abstraction allowed by a programming language and the naturalness of its expression are therefore important to its writability.
- Programming languages can support two distinct categories of abstraction, **process** and **data**.

- A simple example of **process abstraction** is the use of a subprogram to implement a sort algorithm that is required several times in a program. Without the subprogram, the sort code would need to be replicated in all places where it was needed, which would make the program much longer and more tedious to write.
- As an example of **data abstraction**, consider a binary tree that stores integer data in its nodes. Such a binary tree would usually be implemented in a language that does not support pointers and dynamic storage management with a heap, such as Fortran 77, as three parallel integer arrays, where two of the integers are used as subscripts to specify offspring nodes.

  In C++ and Java, these trees can be implemented by using an abstraction of a tree node in the form of a simple class with two pointers (or references) and an integer. The naturalness of the latter representation makes it much easier to write a program that uses binary trees in these languages than to write one in Fortran 77.
- The overall support for abstraction is clearly an important factor in the writability of a language.

iii. Expressivity

- Expressivity in a language can refer to several different characteristics.
- In a language such as APL (Gilman and Rose, 1976), it means that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program.
- More commonly, it means that a language has relatively convenient, rather than cumbersome, ways of specifying computations.
- For example, in C, the notation count++ is more convenient and shorter than count = count + 1.
- Also, the and then Boolean operator in Ada is a convenient way of specifying short-circuit evaluation of a Boolean expression.
- The inclusion of the for statement in Java makes writing counting loops easier than with the use of while, which is also possible.
- All of these increase the writability of a language.

3. Reliability

- A program is said to be reliable if it performs to its specifications under all conditions.
- Several language features that have a significant effect on the reliability of programs in a given language are:

i. Type Checking

- Type checking is simply testing for type errors in a given program, either by the compiler or during program execution.
- Type checking is an important factor in language reliability.
- Because run-time type checking is expensive, compile-time type checking is more desirable.
- Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs.
- The design of Java requires checks of the types of nearly all variables and expressions at compile time. This virtually eliminates type errors at run time in Java programs.
- In the original C language language, the type of an actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function.
- For example, because the bit string that represents the integer 23 is essentially unrelated to the bit string that represents a floating-point 23, if an integer 23 is sent to a function that expects a floating-point parameter, any uses of the parameter in the function will produce nonsense. Furthermore, such problems are often difficult to diagnose. The current version of C has eliminated this problem by requiring all parameters to be type checked.

ii. Exception Handling

- The ability of a program to intercept run-time errors, take corrective measures, and then continue is an obvious aid to reliability. This language facility is called exception handling.
- Ada, C++, Java, and C# include extensive capabilities for exception handling, but such facilities are practically nonexistent in many widely used languages, including C and Fortran.

iii. Aliasing

- Aliasing is defined by having two or more distinct names that can be used to access the same memory cell.
- It is now widely accepted that aliasing is a dangerous feature in a programming language.
- Most programming languages allow some kind of aliasing—for example, two pointers set to point to the same variable, which is possible in most languages. In such a program, the programmer must always remember that changing the value pointed to by one of the two changes the value referenced by the other.

iv. ***Readability and Writability***

- Both readability and writability influence reliability.
- A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural approaches. Unnatural approaches are less likely to be correct for all possible situations.
- The easier a program is to write, the more likely it is to be correct.
- Readability affects reliability in both the writing and maintenance phases of the life cycle.
- Programs that are difficult to read are difficult both to write and to modify.

4. Cost

- The total cost of a programming language is a function of many of its characteristics.
- **First**, there is the cost of training programmers to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers.
- **Second**, there is the cost of writing programs in the language. This is a function of the writability of the language.
- Both the cost of training programmers and the cost of writing programs in a language can be significantly reduced in a good programming environment.
- **Third**, there is the cost of compiling programs in the language. A major impediment to the early use of Ada was the prohibitively high cost of running the first-generation Ada compilers. This problem was diminished by the appearance of improved Ada compilers.
- **Fourth**, the cost of executing programs written in a language is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler.

- **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce. If little or no optimization is done, compilation can be done much faster than if a significant effort is made to produce optimized code.
- The **fifth** factor in the cost of a language is the cost of the language implementation system.
- A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of becoming widely used.
- For example, the high cost of first-generation Ada compilers helped prevent Ada from becoming popular in its early days.
- **Sixth**, there is the cost of poor reliability.
- If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high.
- The failures of noncritical systems can also be very expensive in terms of lost future business or lawsuits over defective software systems.
- The **final consideration** is the cost of maintaining programs, which includes both corrections and modifications to add new functionality. The cost of software maintenance depends on a number of language characteristics, primarily readability.
- A number of other criteria could be used for evaluating programming languages such as **portability**, **generality**, and **well definedness.**

Influence on Language Design

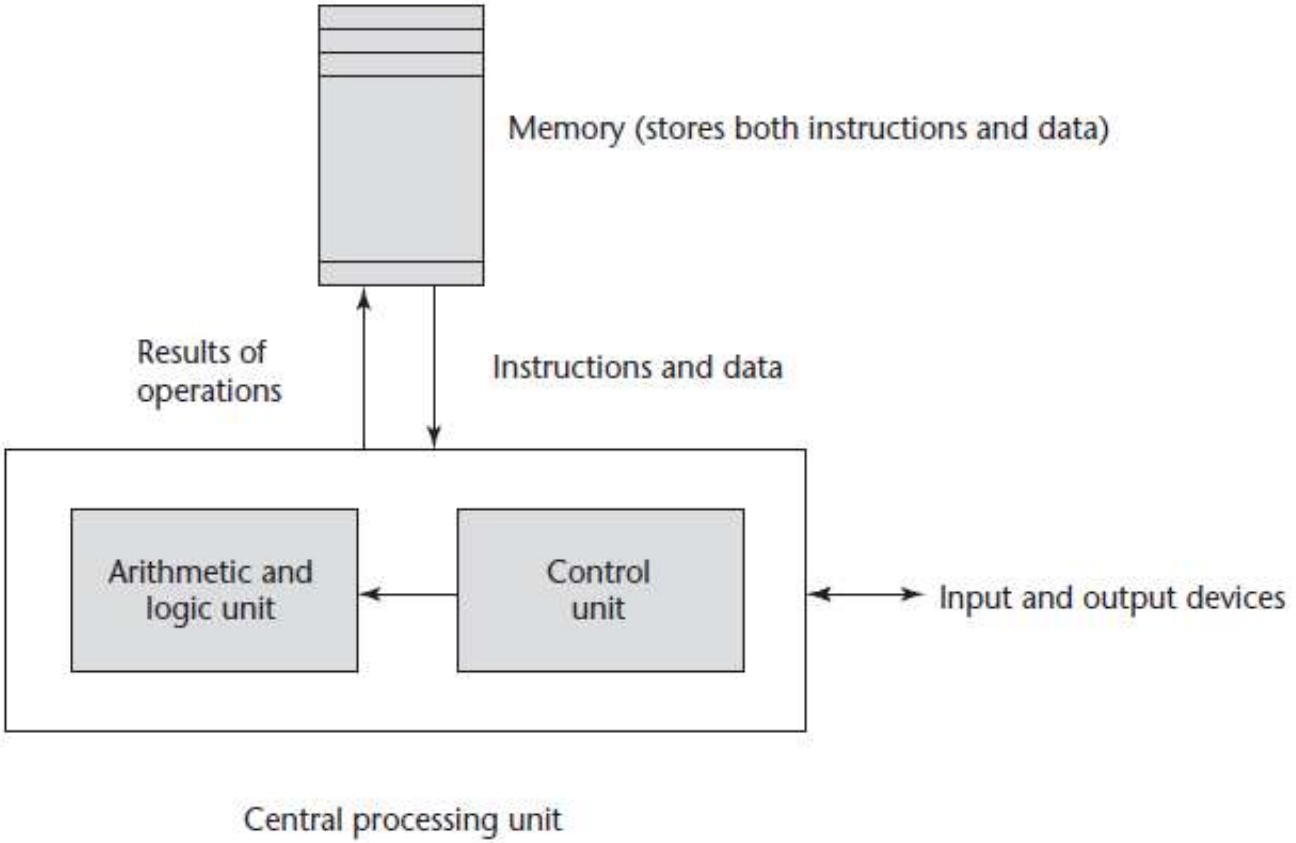
- several factors influence the basic design of programming languages.

1. **Computer Architecture**

- The basic architecture of computers has had a profound effect on language design.
- Most of the popular languages of the past 50 years have been designed around the prevalent computer architecture, called the **von Neumann architecture**, after one of its originators, John von Neumann (pronounced "von Noyman"). These languages are called **imperative languages**.
- In a von Neumann computer, both **data and programs** are stored in the same memory.
- The central processing unit (CPU), which executes instructions, is separate from the memory.
- Therefore, instructions and data must be transmitted, or piped, from memory to the CPU.
- Results of operations in the CPU must be moved back to memory.
- Nearly all digital computers built since the 1940s have been based on the von Neumann architecture.

Figure 1.1

The von Neumann computer architecture



- Because of the von Neumann architecture, the central features of imperative languages are **variables**, which model the memory cells; **assignment statements**, which are based on the piping operation; and the **iterative form of repetition**, which is the most efficient way to implement repetition on this architecture.
- Operands in **expressions** are piped from memory to the CPU, and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment.

- **Iteration** is fast on von Neumann computers because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction. This efficiency discourages the **use of recursion for repetition.**
- The execution of **a machine code program** on a von Neumann architecture computer occurs in a process called the **fetch-execute cycle**.
- ✓ As stated earlier, programs reside in memory but are executed in the CPU.
- ✓ Each instruction to be executed must be moved from memory to the processor.
- ✓ The address of the next instruction to be executed is maintained in a register called the **program counter**.

- The fetch-execute cycle can be simply described by the following algorithm:

  initialize the program counter

  repeat *forever*

  fetch the instruction pointed to by the program counter

  increment the program counter to point at the next instruction

  decode the instruction

  execute the instruction

  end repeat
- The “decode the instruction” step in the algorithm means the instruction is examined to determine what action it specifies.
- In a computer system in which more than one user program may be in memory at a given time, this process is far more complex.

2. Programming Design Methodologies

- The late 1960s and early 1970s brought an intense analysis, begun in large part by the structured-programming movement, of both the software development process and programming language design.
- An important reason for this research was the shift in the major cost of computing from hardware to software, as hardware costs decreased and programmer costs increased. Increases in programmer productivity were relatively small.
- The new software development methodologies that emerged as a result of the research of the 1970s were called *top-down design and stepwise refinement*.
- The primary programming language deficiencies that were discovered were *incompleteness of type checking and inadequacy of control statements (requiring the extensive use of gotos).*

- In the late 1970s, a *shift from procedure-oriented to data-oriented program design methodologies* began.
- Data-oriented methods emphasize data design, focusing on the use of abstract data types to solve problems.
- The first language to provide even limited support for data abstraction was SIMULA 67
- The latest step in the evolution of data-oriented software development, which began in the early 1980s, is *object-oriented design*.
- Object-oriented methodology begins with *data abstraction*, which *encapsulates* processing with data objects and controls access to data, and adds *inheritance* and *dynamic method binding*.
- Object-oriented programming developed along with a language that supported its concepts: *Smalltalk* (Goldberg and Robson, 1989).
- Although Smalltalk never became as widely used as many other languages, support for object-oriented programming is now part of most popular imperative languages, including Ada 95 (ARM, 1995), Java, C++, and C#.

Conflicting criteria in programming language evaluation can lead to design trade-offs, where improving one aspect of the language comes at the expense of another

Language Design Trade-Offs

- The programming language evaluation criteria provide a framework for language design. Unfortunately, that framework is self-contradictory.
- Two criteria that conflict are *reliability and cost of execution*.
- For example, the Java language definition demands that all references to array elements be checked to ensure that the index or indices are in their legal ranges. This step adds a great deal to the cost of execution of Java programs that contain large numbers of references to array elements. C does not require index range checking,
- so C programs execute faster than semantically equivalent Java programs, although Java programs are more reliable. The designers of Java traded execution efficiency for reliability.
- Conflicting criteria that leads directly to design trade-offs.

In the example, Java prioritizes reliability by enforcing index range checking, while C prioritizes execution efficiency by omitting such checks. This trade-off implies that Java programs are more reliable but can incur higher execution costs, while C programs execute faster but may be more prone to runtime errors related to array access.

- The conflict between *writability and reliability* is a common one in language design.
- The pointers of C++ can be manipulated in a variety of ways, which supports highly flexible addressing of data.
- Because of the potential reliability problems with pointers, they are not included in Java.