INTRODUCTION TO PRORAMMING ENVIRONMENT

Assignment

Student's Name: Md. Imtiaz Uddin Tanvir

Roll:MUH2111028M

Course lecturer: Mahbubul Alam

DEPT OF ICE

Programme

In computer science **Programme is** a sequence of instructions that a computer can interpret and execute .It contains a list of ingredients (called variables, which can represent numeric data, text, or images) and a list of directions (called statements) that tell the computer how to execute a specific task.

Programs are created using specific programming languages such as C++, Python, and Ruby. These are high level programming languages that are human-readable and writable. These languages are then translated into low level machine languages by compilers, interpreters, and assemblers within the computer system. Assembly language is a type of low level language that is one step above a machine language and technically can be written by a human, although it is usually much more cryptic and difficult to understand.

Examples of programs include Web browsers, word processors, e-mail clients, video games, and system utilities. These programs are often called applications, which can be used synonymously with "software programs."

On Windows, programs typically have an .EXE file extension, while Macintosh programs have an .APP extension.

When "program" is used as verb, it means to create a software program. For example, programmers create programs by writing code that instructs the computer what to do. The functions and commands written by the programmer are collectively referred to as source code. When the code is finished, the source code file or files are compiled into an executable program.

Programming

Computer **Programming** is a step by step process of designing and developing various sets of computer

programs to accomplish a specific computing outcome. The process comprises several tasks like analysis, coding, algorithm generation, checking accuracy and resource consumption of algorithms, etc. The purpose of computer programming is to find a sequence of instructions that solve a specific problem on a computer.

Computer Programming is very easy if it is appropriately managed. There are many computer programming languages available so finalizing the right programming language is not an easy task.

What is Programming Language?

As mentioned above, Computers understand instructions that are written in a specific syntactical form called a programming language. A programming language provides a way for a programmer to express a task so that it could be understood and executed by a computer. Some of the popular Programming languages are Python, C, C++, Java, etc.

TYPES OF COMPUTER LANGUAGES

A programming language defines a set of instructions that are compiled together to perform a specific task by the CPU (Central Processing Unit). The programming language mainly refers to high-level languages such as C, C++, Pascal, Ada, COBOL, etc.

Each programming language contains a unique set of keywords and syntax, which are used to create a set of instructions. Thousands of programming languages have been developed till now, but each language has its specific purpose. These languages vary in the level of abstraction they provide from the hardware. Some programming languages provide less or no abstraction while some provide higher abstraction. Based on the levels of abstraction, they can be classified into two categories:

1. **Low level language**
2. **High level language**

Low-level language

The low-level language is a programming language that provides no abstraction from the hardware, and it is represented in 0 or 1 forms, which are the machine instructions. The languages that

come under this category are the Machine level language and Assembly language.

Machine language: It is the language of machines, consisting of bits (1s and 0s) put together into chunks like bytes, a group of 8 bits, and lots of other larger sizes. It's highly unlikely you will ever have to write in machine language, but in the old days, we used to plot 1s and 0s on graph paper and then type them in, to make pictures appear on the computer screen.

Aseembly language:

It is a little easier than machine language, but not much! It uses more convenient numbers, symbols, and abbreviations to describe the huge strings of 1s and 0s, to make it both easier and more memorable to type in instructions. The computer knows that certain strings of numbers are commands, so assembly language lets us use English-like strings instead of numbers to refer to those. Plus, with assembly language lets us have access to all kinds of resources to organize our programming code. Then we'll tell a program called an assembler to assemble our instructions, which just means they get turned into 1s and 0s for us.

High-level language

This languages use English-like statements and symbols, and are independent of the type of computer you are using. You can even put in lots of English labels and comments to help remember what the instructions are doing. This makes your programs much easier to read and modify. There are far more high-level languages than any other type of computer language, each one tailored for a certain kind of use.

History of programming language

The first computer programming language was created in 1883, when a woman named Ada Lovelace worked with Charles Babbage on his very early mechanical computer, the Analytical Engine. While Babbage was concerned with simply computing numbers, Lovelace saw that the numbers the computer worked with could represent something other than just amounts of things. She wrote an algorithm for the Analytical Engine that was the first of its kind. Because of her contribution, Lovelace is credited with creating the first computer programming language. As different needs have arisen and new devices have been created, many more languages have followed.

Generations of Programming Languages

Generations of programming got better with time so we started writing down its changes. so that we can know how much programming languages have evolved from the beginning.

As the programming languages got better and better we started every programming language generation's goal was to become better than their previous generations, as a result, we got better and more user friend programming languages.
The programming language in terms of their performance and upgrades can be grouped into five different generations,

1.First generation languages (1GL)

2.Second generation languages (2GL)

3.Third generation languages (3GL)

4.Fourth generation languages (4GL)

5.Fifth generation languages (5GL)

First Generation Language (Machine language)

The first generation language, which is commonly known as a low-level programming language. it is called low-level language because these languages were used to program the computer system at a very low level. i.e, at the machine level. here machine level means, the programmer only write programs with the binary numbers(0 and 1). So you can imagine how hard it would be to write programs in 1Gl languages.
advantages of First Generation Language

1. Since there was no middle man(compiler, interpreter), computers can easily understand the code.

2. Processing was faster in 1GL

3. The programs written in these languages efficiently utilize the memory because it was possible to keep track of each bit of data.

Second Generation language (Assembly Language)

Just like first-generation languages, second-generation languages is also known as a low-level programming language.It was slightly better than the first generation language. In the assembly language, symbolic names are used to represent the opcode and the operand part of the instruction.

Advantages of second-generation programming language

1.With 2GL, It was easy to develop understand and modify the program, programs developed in these languages compared to those developed in the first generation programming language were way better.

2. The programs written in these languages are less prone to errors and therefore can be maintained with a great case.

3.Writing code was a lot easier now.

Third generation language(High- level language)

The languages of this generation were considered as high-level programming languages. Because it was designed and developed to reduce the time, cost, and effort needed to develop different types of software applications. considering the internal architecture of the computer system.

Advantages of third-generation programming language

1. It was more user friendly than the previous 2 generation languages.

2. These languages are less prone to errors they are easy to maintain.

3. development with these languages was faster.

Fourth-generation language (Very High-level Languages)

Just like third-generation languages, it is also considered as a high-level programming language. And just like those previous three languages it was also designed and developed to reduce the time, cost, and effort needed to develop different types of software applications.

Advantages of fourth-generation languages

1. These programming languages allow the efficient use of data by implementing the various database.

2. They require less time, cost, and effort to develop different types of software applications.

3. Developed in these languages are highly portable as compared to the programs developed in the previous generations of languages.

Fifth-generation language (Artificial Intelligence Language)

The fifth-generation language which is considered Artificial Intelligence Language is very different from the previous language ,as the name suggests it is used to develop Artificial Intelligence and Artificial Neural Networks

Advantages of fifth-generation languages

1. These languages can be used to query the database in a fast and efficient manner.

2. In this generation of language, the user can simply communicate with the computer system.

3. In this generation of language, the user can simply communicate with the computer system.

Now we have used five generations of computers. In future generation computers have the power to access things like the human brain by taking inputs in different ways. The future generation of computers performs the tasks very faster than previous generations and devices may be invisible or smaller in size.

Translator, Assembler and Compiler:

Translator, Compiler and **Assembler** are all software programming tools that convert code into another type of code, but each term has specific meaning. All of the above work in some way towards getting a high-level programming language translated into machine code that the central processing unit (CPU) can understand. Examples of CPUs include those made by Intel (e.g., x86), AMD (e.g., Athlon APU), NXP (e.g., PowerPC), and many others. It's important to note that all translators, compilers, interpreters and assemblers are programs themselves.

Translator

The most general term for a software code converting tool is "**translator**." A translator, in software programming terms, is a generic term that could refer to a compiler, assembler, or interpreter; anything that converts higher level code into another high-level code (e.g., Basic, C++, Fortran, Java) or lower-level (i.e., a language that the processor can understand), such as assembly language or machine code. If you don't know what the tool actually does other than that it accomplishes some level of code conversion to a specific target language, then you can safely call it a translator.

Assembler

An assembler translates a program written in assembly language into machine language and is effectively a compiler for the assembly language, but can also be used interactively like an interpreter. Assembly language is a low-level programming language. Low-level programming languages are less like human language in that they are more difficult to understand at a glance; you have to study assembly code carefully in order to follow the intent of execution and in most

cases, assembly code has many more lines of code to represent the same functions being executed as a higher-level language. An assembler converts assembly language code into machine code (also known as object code), an even lower-level language that the processor can directly understand.

Interpreter

An interpreter translates code into machine code, instruction by instruction - the CPU executes each instruction before the interpreter moves on to translate the next instruction. Interpreted code will show an error as soon as it hits a problem, so it is easier to debug than compiled code.

An interpreter does not create an independent final set of source code - source code is created each time it runs. Interpreted code is slower to execute than compiled code.

Interpreted languages include JavaScript, PHP, Python and Ruby. Interpreted languages are also called scripting languages. These are ideal for using within dynamic web applications. They are used for client-side and server-side coding, as they are small programs that are executed within the browser.

Difference between Translator, Assembler and Interpreter:

Parameters	Translator	Assembler	Interpreter
Conversion	It converts higher level code into another high-level code or lower-level such as assembly language or machine code.	It converts programs written in the assembly language to the machine language or binary code.	It also converts the program-developed code into machine language or binary code.
SCANNING	It scans the entire program before converting it into a program in target language as output.	It translates the program line by line to the equivalent machine code.	It converts the source code into the object code then converts it into the machine code.
Error detection	Detects and reports the error during translation.	It detects errors in the first phase, after fixation the second phase starts.	It translates the program line by line to the equivalent machine code

Problem solving steps

Most people engage in problem solving every day. It occurs automatically for many of the small decisions that need to be made on a daily basis.

For example, when making a decision about whether to get up now or sleep in for an extra 10 minutes, the possible choices and the relative risks and benefits of obeying the alarm clock or sleeping later come automatically to mind.

Larger problems are addressed in a similar way. For example: "I have tasks that need to be done by the end of the week. How am I going to get them all done on time?"Sometimes, it is not enough to just cope with the problems – they need to be solved .After considering the possible strategies, 1 is chosen and implemented. If it proves to be ineffective, a different strategy is tried. People who can define problems, consider options, make choices, and implement a plan have all the basic skills required for effective problem solving.Sometimes following a step-by-step procedure for defining problems, generating solutions, and implementing solutions can make the process of problem solving seem less overwhelming.

Six step guide to help you solve problems:

Step 1: Identify and define the problem

State the problem as clearly as possible. For example: "I don't have enough money to pay the bills."
Be specific about the behaviour, situation, timing, and circumstances that make it a problem. For example: "I need to pay the phone and gas bills, and I don't have enough money to cover both this month."
Step 2: Generate possible solutions
List all the possible solutions; don't worry about the quality of the solutions at this stage.
Try to list at least 15 solutions, be creative and forget about the quality of the solution.

If you allow yourself to be creative you may come up with some solutions that you would not otherwise have thought about.

Step 3: Evaluate alternatives

The next step is to go through and eliminate less desirable or unreasonable solutions.

Order the remaining solutions in order of preference.Evaluate the remaining solutions in terms of their advantages and disadvantages.

Step 4: Decide on a solution

Specify who will take action.

Specify how the solution will be implemented.

Specify when the solution will be implemented. For example: tomorrow morning, phone the gas company and negotiate to pay the gas bill next month.

Step 5: Implement the solution

Implement the solution as planned.

Step 6: Evaluate the outcome

Evaluate how effective the solution was.

Decide whether the existing plan needs to be revised, or whether a new plan is needed to better address the problem.

If you are not pleased with the outcome, return to step 2 to select a new solution or revise the existing solution, and repeat the remaining steps.

Data types

A data type, in programming, is a classification that specifies which type of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. A string, for example, is a data type that is used to classify text and an integer is a data type used to classify whole numbers.

Data Types tells the Memory Management Unit (MMU) how much memory is required to store the data before the program compiles.
Each Data Type has a memory size that is pre-defined by the programming language.

Different Data Types

Many programming languages have their own pre-defined data types which can be used by simply typing the keywords assigned to them. Few programming languages have their own different Data Types and their characteristics are unique.

Basic Data Types :

- Int – includes all whole numbers
- Char – includes all
- String – comprises of a set of characters
- Float – contains decimal point numbers
- Double – stores fractional
- Boolean – contains one of 2 possible values only.

Classification of Data Types

Built-in Type/Primitive Type

Data Types that are pre-defined and provided by each Programming Language.

- Int
- Char
- Float
- Double
- Boolean

User-Defined Type

These Data Types are defined by the users in the program as per their needs.

- Structure
- Array
- Class
- Enumeration
- Union

Composite/Derived Type

These data types are formed by using basic data types i.e. using one or more built-in data types.

- Array
- Function
- Pointer
- Record
- Set

OPERATORS

There are various types of operators discussed in this appendix. The following types are provided:

- **Arithmetic Operators** are used to perform mathematical calculations.
- **Assignment Operators** are used to assign a value to a property or variable. Assignment Operators can be numeric, date, system, time, or text.
- **Comparison Operators** are used to perform comparisons.
- **Ternary Operators** are used to combine strings.

- **Logical Operators** are used to perform logical operations and include AND, OR, or NOT.
- **Boolean Operators** include AND, OR, XOR, or NOT and can have one of two values, true or false.

Arithmetic Operators

The C language supports all the basic arithmetic operators such as addition, subtraction, multiplication, division, etc.
The following table shows all the basic arithmetic operators along with their descriptions.

Operator	Description	Exampler
+	adds two operands (values)	a+b
-	subtract second operands from first	a-b
*	multiply two operands	a*b
/	divide numerator by the denominator, i.e. divide the operand on the left side with the operand on the right side	a/b
%	This is the modulus operator, it returns the remainder of the division of two operands as the result	a%b
++	This is the Increment operator - increases integer value by one. This operator needs only a single operand.	a++ or ++a

--	This is the Decrement operator - decreases integer value by one. This operator needs only a single operand.	--b or b--

Relational operators

The relational operators (or comparison operators) are used to check the relationship between two operands. It checks whether two operands are equal or not equal or less than or greater than, etc.
It returns 1 if the relationship checks pass, otherwise, it returns 0.
For example, if we have two numbers 14 and 7, if we say 14 is greater than 7, this is true, hence this check will return 1 as the result with relationship operators. On the other hand, if we say 14 is less than 7, this is false, hence it will return 0.
The following table shows all relational operators supported in the C language

Operator	**Description**	**Example** **(a and b, where a = 10 and b = 11)**
==	Check if two operands are equal	a == b, returns 0
!=	Check if two operands are not equal.	a != b, returns 1 because a is not equal to b

>	Check if the operand on the left is greater than the operand on the right	a > b, returns 0
<	Check operand on the left is smaller than the right operand	a < b, returns 1
>=	check left operand is greater than or equal to the right operand	a >= b, returns 0
<=	Check if the operand on left is smaller than or equal to the right operand	a <= b, returns 1

Logical Operator

Operator	**Description**	**Example** **(a and b, where a = 1 and b = 0)**
&&	Logical AND	a && b, returns 0
||	Logical OR	a || b, returns 1
!	Logical NOT	!a, returns 0

Bitwise Operators

Bitwise operators perform manipulations of data at the bit level. These operators also perform the shifting of bits from right to left. Bitwise operators are not applied to float or double, long double, void, etc.

Operator	**Description**
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR (XOR)
~	One's complement (NOT)
>>	Shift right
<<	Shift left

Assignment Operators

The assignment operators are used to assign value to a variable. For example, if we want to assign a value 10 to a variable x then we can do this by using the assignment operator like: x = 10; Here, = (equal to) operator is used to assign the value.

Operator	Description	Example (a and b are two variables, with where a=10 and b=5)
=	assigns values from right side operand to left side operand	a=b, a gets value 5
+=	adds right operand to the left operand and assign the result to left operand	a+=b, is same as a=a+b, value of a becomes 15
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b, is same as a=a-b, value of a becomes 5
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b, is same as a=a*b, value of a becomes 50
/=	divides left operand with the right operand and assign the result to left operand	a/=b, is same as a=a/b, value of a becomes 2
%=	calculate modulus using two operands and assign the result to left operand	a%=b, is same as a=a%b, value of a becomes 0

CONTROL STRUCTURE

Control Structures can be considered as the building blocks of computer programs. They are commands that enable a program to “take decisions”, following one path or another. A program is usually not limited to a linear sequence of instructions since during its process it may bifurcate, repeat code or bypass sections. Control Structures are the blocks that analyze variables and choose directions in which to go based on given parameters. There are three basic types of logic, or flow of control, known as:

1. Sequence logic, or sequential flow
2. Selection logic, or conditional flow
3. Iteration logic, or repetitive flow

Sequential Logic (Sequential Flow)
Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer. Unless new instructions are given, the modules are executed in the obvious sequence. The sequences may be given, by means of numbered steps explicitly. Also, implicitly follows the order in which modules are written. Most of the processing, even some complex problems, will generally follow this elementary flow pattern.

- Sequence is the easiest control structure to use and make sense of
- This control structure runs through the code **sequentially**
- This means that steps are followed in order from line 1 of the program to the end no matter what conditions are met

Selection Logic (Conditional Flow)
Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules. The structures which use these type of logic are known as Conditional Structures.

- The selection control structure allows a program to test conditions based off certain criterion
- Selection follows IF [this] THEN [that] logic
- It can be seen as a boolean condition as the result is going to be true or false
- Selection structures also rely on an ELSE statement
- When the specified or expected criterion isn't met, the program will follow the ELSE instructions

Iteration Logic (Repetitive Flow)

- Iteration is another term for **looping**
- A function or program utilizing looping will run a specific block of code until certain criterion is met
- Three main types of iteration exist; **while**, **do-while** and **for**

WHILE Loops (Test-first)

- A while loop can be seen as a repeating IF statement
- At the start of the statement, a boolean condition is given
- While this statement is true, the code within the WHILE loop will execute
- Once this statement is false, the compiler or interpreter will move on to the next block of code
- This is known as test-first because the expression is always considered before running the loop, meaning that this block of code has the option to *never run* if the condition isn't met
- Example:
- int i=0;

```
while (i >= 2){
    // Do something
}
```

FOR Loops (Fixed)

- FOR loops run until a particular condition is satisfied

- A boolean expression is given at the start of the control structure
- There is an incremental counter in the for loop expression that tells the function when to stop
- This is known as a fixed control structure as there's a finite number of times the loop will run (as per determined by the expression)

- Example:

```
for (int i = 1; i <= 200; i++){
    // Do something
}
```

DO-WHILE Loops (Test-last)

- A DO-WHILE loop will run a condition regardless of the previous outcomes
- This is because the boolean expression is evaluated at the end of the control structure
- This means that a DO-WHILE loop is guaranteed to run *at least* once

- Example:

```
do {
    // Do something
    i++;
} while (i < 4);
```