Introduction to Computer Science – D684

Section 1

Lesson 1

* 1. **– Data Types**

**Data** = basic values or facts

**Information** = organized and/or processed data – useful in solving problems

|  |  |
| --- | --- |
| **Data** | |
| **Analog** | **Digital** |
| Continuous and variable form | Represented by binary values  1s and 0s  Bit- 1 or 0  Bit Combination = (2n) |
|  |  |

**Radix point** = decimal point

**Floating point** = number of digits is fixed but radix floats (exponents)

**Overflow** = calculation exceeds max value represented w/ # of bits

|  |  |  |
| --- | --- | --- |
| **Representing Negative Values - Binary** | | |
| **Feature** | Signed Magnitude | 2’s Complement |
| **Definition** | Uses the most significant bit (MSB) as a sign bit (0 for positive, 1 for negative), while the remaining bits represent the magnitude. | Uses the MSB as a sign bit, but negative numbers are represented by inverting all bits of the positive number and adding 1. |
| **Zero Representation** | Two representations: 0000 (+0) and 1000 (-0) | Single representation: 0000 (0) |
| **Addition/Subtraction** | More complex due to separate handling of signs | Easier because addition and subtraction work naturally |
| **Overflow Handling** | More prone to inconsistencies | More systematic and efficient |
| **Overflow Example** | Adding +5 (0101) and +4 (0100):  - Expected: +9 (out of range)  - Incorrect result: 1001 (which incorrectly represents -1) | Adding +7 (0111) and +3 (0011):  - Expected: +10 (out of range)  - Incorrect result: 1010 (which represents -6 due to overflow) |
| [**Video Explanation**](https://youtu.be/sJXTo3EZoxM?si=qYFqXQH_d6vFXCvH) | | |

**Character set** = list of characters and the codes used to represent each one (ASCII or Unicode)

|  |  |  |
| --- | --- | --- |
| **Feature** | **ASCII (American Standard Code for Information Interchange)** | **Unicode** |
| **Definition** | A character encoding standard that assigns numeric values to characters using 7 or 8 bits | A universal character encoding standard that supports a wide range of scripts and symbols using multiple encoding formats |
| **Bit size** | 7-bit (original ASCII)  8-bit (extended ASCII) | Variable: 8-bit (UTF-8)  16-bit (UTF-16) or  32-bit (UTF-32) |
| **Character Support** | 128 characters (7-bit) or  256 characters (8-bit) | Over 143,000 characters from multiple language and symbol sets |
| **Languages Supported** | English  Basic symbols | Almost all languages |

**Compression** = reduces amount of space needed to store data

**Compression ratio** =

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Keyword** | **Run-Length** | **Huffman** |
| **Definition** | Replaces frequently used words or characters with shorter symbols or codes | Replaces consecutive repeating characters with a single character followed by a count | Assigns shorter binary codes to more frequent characters and longer codes to less frequent ones |
| **Best for** | Text data with common words/phrases | Data with long sequences of repeated characters | Data where some characters appear more frequently than others |
| **Compression Efficiency** | Moderate (depends on word frequency) | High if there are long runs of repeated characters | Very high if data has a skewed frequency distribution of characters |
| **Example Input** | “this is a test. this is only a test.” | "AAAABBBCCDAA" | "ABRACADABRA" |
| **Example Encoding** | Replace "this" → #, "is" → \*, "test" → @  Encoded: "# \* a @. # \* only a @." | "A4B3C2D1A2" (compresses consecutive characters) | Assigns variable-length codes:  A=10, B=00, R=01, C=110, D=111  Encoded: "1000011010110100" |
| **Lossless?** | Yes | Yes | Yes |

|  |  |
| --- | --- |
| **Lossy** | **Lossless** |
| Data compression  Information is lost  Reduces file size by permanently eliminating some information  Affects quality | Data compression  No information is lost  Original data can be perfectly reconstructed |

* 1. **– Variables**

|  |  |  |
| --- | --- | --- |
| Broad Type | Data Type | Definition |
| Numeric | Integer (int) | Whole numbers (positive, negative, or zero) with no decimal point. Example: -5, 0, 42 |
| Floating-point (float, real numbers) | Numbers that include a fractional part (decimal values). Example: 3.14, -0.001, 2.0 |
| Double (double-precision float) | Similar to float but with higher precision for larger or more precise decimal values. Example: 3.1415926535 |
| Long (long int) | An integer with a larger storage capacity than a standard int, allowing for bigger whole numbers. Example: 9223372036854775807 |
| Boolean (bool) | A binary data type representing true (1) or false (0). Example: true, false |
| String | Character (char) | A single letter, digit, or symbol enclosed in quotes. Example: ‘A’, ‘7’, ‘#’ |
| String (str, text) | A sequence of characters (letters, numbers, symbols). Example: “Hello, World!” |

**Boolean expressions** = true or false logical statements

* Uses operators AND, OR, and NOT

**Data types** = define what kind of value

* Integers = whole numbers
* Floats = numbers with decimal place
* Strings = usually words but technically anything contained in quotation marks (ex “blue” “3.14”)

**Declaration** = statement that specifies name and type of variable or function without assigning a value. Provides: data type, identifier, (optionally- initial value)

**Initializing a variable** = declaring a starting value to a variable (eg int x=10)

**Right-to-left associativity** = assignment operator operates from right to left, meaning that the value of the expression to the right of the assignment operator is evaluated first and then the result is assigned to the operand on the left

**Identifier** = name given to variable, function, or other item in code to identify it (eg myNumber, calculateSum)

* Can refer to variables, functions, classes, etc

**Keyword** = a reserved word that has specific meaning and cannot be used as an identifier (eg if, while, class)

Data type (declares sum as integer)

Variable Name (identifier storing the result)

Operands (values used in the operation)

Binary Operator (performs addition)

Assignment Operator

(assigns value to sum)

int sum = 5 + 3;

**Assignment operator** = symbol used to assign value to a variable (typically =)

**Assignment statement** = line of code that assigns value to a variable (eg x=5)

**Binary operator** = operator that takes 2 operands (+, -, \*, /)

**Operand** = value or variable on which operator acts (3 and 5 in example)

**lvalue** = a memory location which can appear on the left side of an assignment (eg x in x=5)

**Variable** = named memory locations whose contents can vary or differ over time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Category** | **Numeric Variable** | **Numeric Constant** | **String Variable** | **String Constant** |
| **Definition** | A named storage location that holds a numeric value | A fixed numeric value | A named storage location that holds a string value | A fixed string value |
| **Example declaration** | int age; | const int  MAX\_AGE = 100; | string name; | const string GREETING = “Hello!”; |
| **Example assignment** | age = 25 | // Value is fixed at declaration | name = “Alice” | // Value is fixed at declaration |
| **Mutability (can be changed?)** | Can | Cannot | Can | Cannot |
| **Use Case** | Storing user input, calculations, or temporary values | Defining fixed numerical values (eg PI, max limits) | Storing dynamic text values such as names, messages, or file paths | Defining fixed messages, such as prompts or labels |

**See Chart Below**

* Type safety prevents type-related errors by enforcing proper data type usage.
* Strongly typed languages require explicit type conversions, reducing the risk of unintended behavior.
* A language can be type-safe but not strongly typed, and vice versa (though they often overlap).

|  |  |  |
| --- | --- | --- |
| **Concept** | **Type Safety** | **Strongly Typed Languages** |
| **Definition** | Ensures that operations use compatible data types to prevent type errors | Enforces strict data type rules; does not allow implicit type conversions between unrelated types |
| **Key Feature** | Prevents type mismatches (eg trying to add a string and a number) | Requires explicit type conversions to avoid unintended behavior |
| **Relationship** | A type-safe language enforces type rules to avoid unexpected behavior | Strongly typed languages tend to be type-safe, but weakly typed languages still have some type safety mechanisms |
| **Example** | In Java: int x = “hello” -> Error!  (string assigned to an integer) | In Python: 5 + “10” -> Error!  (requires explicit conversion: 5 +I int(“10”)) |

|  |  |  |
| --- | --- | --- |
| **Variable Naming Conventions** | | |
| **Naming convention** | **Definition** | **Example** |
| **Camel Casing** | 1st word is lowercase and each subsequent word starts with an uppercase letter | myVariableName |
| **Kebab Case** | All words are lowercase and separated by hyphens | my-variable-name |
| **Hungarian Notation** | Prefix indicates the variable type followed by a descriptive name in camel case | strUserName – (string)  intCounter – (integer) |
| **Mixed case with underscores** | Combination of camel case and underscores | my\_Variable\_Name |
| **Pascal Casing** | 1st letter of each word is capitalized | MyVariableName |
| **Snake Casing** | All words lowercase and separated by underscores | my\_variable\_name |

**3 Rules for Variable Names:**

1. **Must be one word**
2. **Must start with a letter**
3. **Should have appropriate meaning**
   1. **-Constructs**

**Algorithm**- plan for a solution

**Pseudocode**- a language that allows us to express algorithms in a clearer form, shorthand-like

**Functionality:**

**Variables**- names that refer to places in memory where values are stored

**Assignment**- putting a value into a variable

**Input/Output**- read for input, write for output

**Selection**- construct that allows a choice, perform action or skip it

-if-then

-if-then-else

**Repetition**- instructions are allowed to repeat

**Example:**

Set limit to number of values to sum

WHILE (counter<limit)

Read num

Set sum to sum + num

Set counter to counter + 1

* 1. **-Operators**

**Operators include (precedence low -> high)**

**= assignment**

**+ addition**

**- subtraction**

**\* multiplication**

**/ division**

**Rules of precedence (aka order of operations)**

1. Expressions within parentheses evaluated first
   1. Innermost evaluated first if multiple sets
2. Multiplication and division (left to right)
3. Addition and subtraction (left to right)

Lesson 2

**2.1-Data Structures**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Array** | **Record** |
| **Definition** | Collection of elements of same data type stored in a contiguous memory location | Data structure that holds related fields, typically with named properties (key-value pairs) |
| **Data Organizaiton** | Indexed collection  Ordered list | Key-value pairs  Unordered, structured |
| **Access method** | Via index  (eg array[0]) | Via field names  (eg record.name) |
| **Homogenous or Heterogeneous?** | Homogenous | Heterogeneous |
| **Example (in Python)** | arr = [1, 2, 3] | record = {‘name’ : ‘Alice”, ‘age’ : 30} |

**Container-** holds collection of elements, provides methods to add, remove, and access items within it

**Abstract data type (ADT)-** container whose properties (data and operations) are specified independently of any particular implementation

**3 Levels of How We View Data:**

1. **Application level-** view of data within particular problem
2. **Logical (abstract) level-** abstract view of data values (the domain) and the operations that manipulate them
3. **Implementation level-** specific representation of the structure that holds the data items and the coding of the operations in a programming language
   1. Sees properties represented as specific data fields and subprograms
   2. Concerned with data structures

|  |  |  |
| --- | --- | --- |
| **Feature** | **Stack** | **Queue** |
| **Definition** | Follows LIFO (Last In, First Out) principle | Follows FIFO (First In, First Out) principle |
| **Insertion** | Performed at the top | Performed at the rear (end) |
| **Access Order** | Most recently added element is removed first | Oldest added element is removed first |
| **Terminology/Operations** | Push= adding item  Pop= removing item | No standard terminology  Enter and Insert= insertion operation  Delete and Remove= deletion operation |
| **Think of:** | Cafeteria plates | Waiting in line at bank |

|  |  |
| --- | --- |
| **Feature** | **Lists** |
| **Definition** | a collection of items arranged in a linear sequence, allowing for easy access to and the insertion and deletion of elements |
| **3 properties** | 1. Homogenous 2. Linear    1. Each item except 1st has a unique component that comes before it, and each item except the last has a unique component that comes after it 3. Have varying lengths |
| **Operations** | Insert= add item  Delete= delete item  IsThere= checks if item is there  GetLength= reports number of items in list  Reset, GetNext, MoreItems= allows user to see each item in sequence |
| **Types** | Array-based: uses fixed or dynamically resizable array  Linked: uses nodes with pointers to next elements  Sorted: elements are in sorted order (makes search operations faster but insertions slower)  Unsorted: elements appear in any order (makes fast insertions but slower search) |

**2.2-Subprograms**

**Value Parameter Example:**

def change\_value(x):

x = 10 *# Changes only a copy*

num = 5

change\_value(num)

print(num) *# Output: 5 (unchanged)*

*Key ideas: x inside the function is just a copy*

**Reference Parameter Example:**

def change\_first(lst):

lst[0] = 10 *# Modifies the original list*

numbers = [5, 6, 7]

change\_first(numbers)

print(numbers) *# Output: [10, 6, 7] (changed)*

*Key ideas: changes inside the function affect the original list*

|  |  |  |
| --- | --- | --- |
| **Term** | **Definition** | **Example** |
| **Subprogram (Function)** | Reusable block of code that performs a specific task (aka function or procedure) | def greet():  print(“Hello”) |
| **Argument** | **A**ctual value or data passed to a function when calling it | add\_numbers(5,3)   * 5 and 3 are arguments |
| **Parameter** | **P**laceholder variable in a function definition that receives and argument | def add\_numbers(a,b)   * a and b are parameters |
| **Parameter list** | A set of parameters in a function definition specifying the number and types of inputs | def multiply(x,y,z):   * parameter list = x, y, z |
| **Reference Parameter** | Parameter that allows a function to modify the original variable passed to it  -accesses the contents of the *address* | See example below |
| **Value Parameter** | Parameter that passes a copy of the argument’s value, changes inside the function do not affect the original variable  -accesses the *content* | See example below |

Lesson 3

**3.1-Object-Oriented Programming Basics**

**OOP** is a programming paradigm that organizes code into objects, which are instances of classes. It helps in code reuse, organization, and scalability.

-main purpose is to organize code into reusable components called objects

|  |  |  |
| --- | --- | --- |
| Term | Definition | How they work |
| **Class** | A blueprint or template for creating objects. Defines properties (fields) and behaviors (methods) | Define the structure and behavior of objects |
| **Object** | An instance of a class. Has its own values for the fields and can use the class’s methods | Created using the class and store individual data |
| **Field (Attribute)** | A variable inside a class that stores data related to an object | Hold values specific to each object |
| **Method** | A function defined inside a class that performs actions on the object | Perform actions related to the object |

Example (Python)

class Car: *# Class*

def \_\_init\_\_(self, brand, color):

self.brand = brand *# Field*

self.color = color *# Field*

def drive(self): *# Method*

print(f"The {self.color} {self.brand} is driving!")

my\_car = Car("Toyota", "red") *# Object*

my\_car.drive() *# Output: The red Toyota is driving!*

OOP languages are executed in two main ways: **Compilation** and **Interpretation**.

|  |  |  |
| --- | --- | --- |
| Term | Definition | Language Examples |
| Compiler | Translates the entire code from a high-level language into a machine code or bytecode before execution | C++  Java  FORTRAN  COBOL  ALGOL |
| Interpreter | Translates and executes code line by line instead of all at once | Python  JavaScript  Lisp  SNOBOL4  APL |

**Bytecode**

1. is an **intermediate** representation of a program, between source code and machine code. It is not directly executed by the CPU but instead runs on a **virtual machine** (such as the Java Virtual Machine, or JVM)
2. Bytecode itself is **neither** a compiler nor an interpreter. However, it is the result of **compilation**, and it is executed by an **interpreter**. Here's how it works in Java:
   1. **Compilation Phase**:
      1. A **compiler** (e.g., javac for Java) converts source code (.java files) into bytecode (.class files).
   2. **Execution Phase**:
      1. A **bytecode interpreter** (like the Java Virtual Machine, JVM) reads and executes the bytecode, translating it into machine code at runtime.
3. Bytecode runs on different operating systems without modification, as long as the appropriate virtual machine (JVM, PVM, CLR) is available
4. The virtual machine acts as a layer of protection, preventing direct execution of harmful machine code.

**Programming Paradigms** = style or way of programming that defines how code is structured and executed

|  |  |  |
| --- | --- | --- |
| Paradigm | Key Idea | Language Examples |
| Imperative  -procedural | Step-by-step execution, modifying program state directly  Procedural: Uses functions/procedures to structure code for reuse  Emphasizes **code reuse** and modularity using procedures | FORTRAN  BASIC  C  Pascal  C++ |
| Declarative  -functional  -logic | Focuses on what should be done rather than how  Functional: uses pure mathematical functions with no side effects  Logic: uses logic rules and facts to infer results | Functional:  Haskell  Lisp (Scheme)  Logic:  Prolog  Datalog |
| Object-Oriented (OOP) | Organizes code into objects combining data and behavior  Structures code into **classes and objects** to model real-world entities | C++  Java  Python  C#  Kotlin |

**\*\*Below Material also partially covered in Lesson 1.2 – Variables\*\***

A screenshot of a computer

AI-generated content may be incorrect.**Boolean expression** = a sequence of identifiers, separated by compatible operators, that evaluates to either true or false (see table below from Dale & Lewis (2023, p. 298))

**Special Operators for Boolean operators:**

1. AND – returns true if both expressions are true, and false otherwise
2. OR – returns false if both expressions are false and true otherwise
3. NOT – changes the value of the expression

**Common Data Types**

|  |  |  |
| --- | --- | --- |
| Data Type | Description | Example |
| Integer  (int) | Whole numbers  Positive or negative | 5, -42, 1000 |
| Real  (float/double) | Numbers with decimal points  Floating-point numbers | 3.14, -0.001, 2.5 |
| Character  (char) | A single letter, number, or symbol enclosed in quotes | ‘A’, ‘z’, ‘5’, ‘\*’ |
| Boolean  (bool) | Logical values that represent true or false | True, False |
| String  (str) | A sequence of characters | “Hello”, “123ABC”, “!” |

**Declaration** = statement in a programming language that defines a variable, specifying its **name** and **data type** before use. Some languages require explicit declaration, while others infer it

Explicit = programmer must specify the variables type before using it

Implicit = variable type is inferred by the language based on the assigned value

**Input/Output (I/O) Structures** = allow a program to interact with users by receiving input and displaying output

Example I/O:

name = input("Enter your name: ") *# Input*

print("Hello,", name) *# Output*

**Control Structures** = manage flow of execution in program

|  |  |  |
| --- | --- | --- |
| Type | Description | Example |
| **Sequence** | Executes statements in order | print(“Hello”) then print(“World”) |
| **Selection**  **(decision making)** | Uses conditions to decide execution | if age >=18: print(“Adult”) |
| **Iteration**  **(loops)** | Repeats execution of a block | for i in range(5): print(i) |

**Keywords in Iteration Control Structures**

|  |  |  |
| --- | --- | --- |
| **Loop type** | **Common keywords** | **Description** |
| For Loop | for | Repeats a block of code a fixed number of times |
| While Loop | while | Continues execution as long as condition is true |
| Do-While Loop | do, while | Executes the loop body **at least once** before checking the condition |
| Foreach Loop | foreach,  for…in | Iterates over elements in an array, list, or collection |
| Loop Controlled Statements | break,  continue | break exits a loop early; continue skips the current iteration and moves to the next one |

**Reserved Word (Keyword) =** Special words that have a predefined meaning in a language (e.g., int, if, while). These cannot be used as variable names.

**Case Sensitivity =** Some languages differentiate between uppercase and lowercase (e.g., Name and name are different in Python and Java, but not in VBScript).

**Nested Logic** = occurs when control structures (like if-statements or loops) are placed inside one another

Outer if statement checks if user is 18 or older

Inner if statement determines if they qualify as a senior citizen

Example Nested If-Else:

age = int(input("Enter age: "))

if age >= 18:

if age >= 65:

print("Senior Citizen")

else:

print("Adult")

else:

print("Minor")

**Asynchronous processing** = allows program to execute tasks without waiting for previous task to finish

* AKA **event-driven** processing
* Useful for multitasking (ex handling user input while loading data)
* Let tasks run in parallel instead of waiting for each other

**3 Pillars of OOP:**

1. **Encapsulation**
   1. Bundling data (fields/variables) and methods (functions) together inside a class
   2. Practice of keeping data and methods that manipulate the data in a single unit
   3. Restricts direct access to some data, ensuring controlled interaction via methods
   4. **Abstraction** is the concept of hiding implementation details while exposing only necessary functionalities
      1. Closely related to encapsulation
      2. Some consider it another pillar, but others consider it an extension of encapsulation
2. **Inheritance**
   1. Allows one class (**child/subclass**) to inherit properties and behaviors from another class (**parent/superclass**)
   2. Promotes **code reuse** and **hierarchical relationships**
   3. **Inheritance hierarchy =** organization of classes into a tree structure based on inheritance relationships
3. **Polymorphism**
   1. Allows objects of different classes to be treated as objects of a common superclass
   2. Enables methods or functions to take **multiple forms** based on input or object type
   3. Two types:
      1. **Method Overloading** (same method name, different parameters)
      2. **Method Overriding** (subclass modifies superclass method)