


CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR



Lecture 10

Data Types (Part 1)

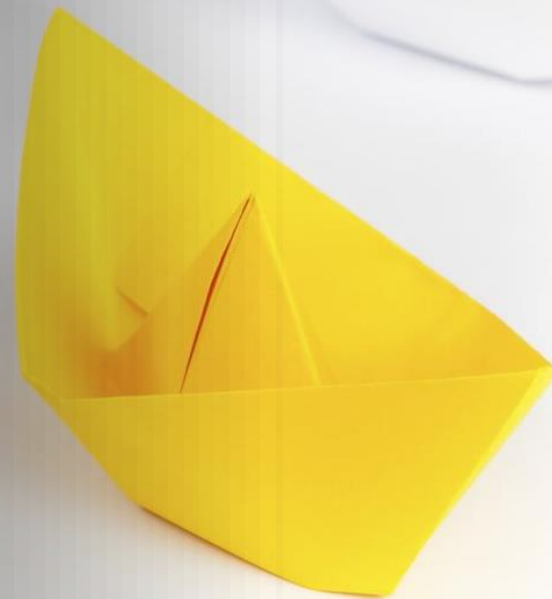


Lecture 10 Topics

- Introduction
- Primitive Data Types
 - Numeric Types (Integer, Floating-Point, Decimal, Complex)
 - Boolean Types
 - Character Types
- (Character) String Types
- Array Types
- Enumeration Types



Introduction



Data Types

Introduction

- A **data type** is the “information” that determines how a data will be kept in memory, how its value will be interpreted and what operations can be performed on the data.
- Data types in programming languages are generally examined in two groups:
 - **Primitive Data Types**
 - **Derived/Structured Data Types (Non-primitive)**: To specify the derived/structured types, the primitive data types of a language are used, along with one or more type constructors.



Primitive Data Types



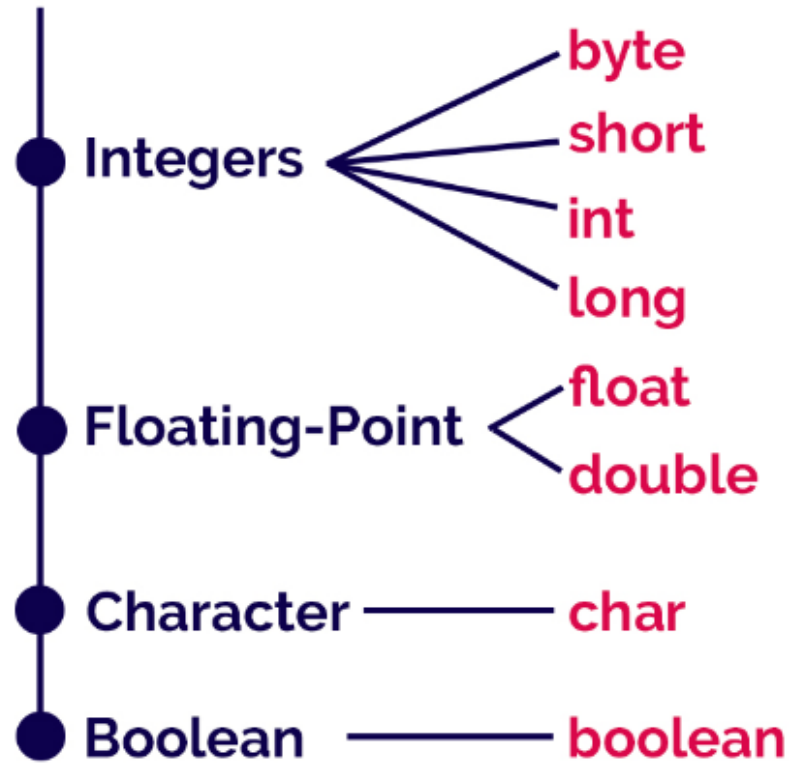


Primitive Data Types

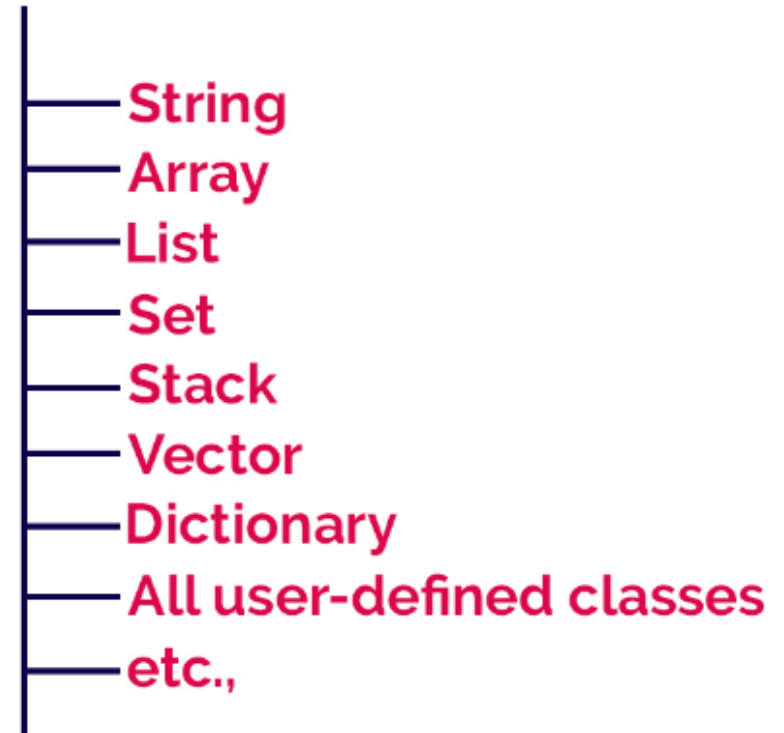
- Data types that are not defined in terms of other types are called **primitive data types**.
- Nearly all programming languages provide a set of primitive data types. For example, **Java supports eight primitive data types**.
- Primitive Data Types Groups:
 - *Numeric Types*
 - Integer (**byte, short, int, long**)
 - Floating-Point (**float, double**)
 - Decimal
 - Complex
 - *Boolean Types* (**boolean**)
 - *Character Types* (**char**)

Data Types in java

Primitive Data Types



Non-primitive Data Types



Java Primitive Types

Type	Size in bits	Values	Standard
boolean		true or false	
[Note: A boolean's representation is specific to the Java Virtual Machine on each platform.]			
char	16	'\u0000' to '\uFFFF' (0 to 65535)	(ISO Unicode character set)
byte	8	-128 to +127 (-2^7 to $2^7 - 1$)	
short	16	-32,768 to +32,767 (-2^{15} to $2^{15} - 1$)	
int	32	-2,147,483,648 to +2,147,483,647 (-2^{31} to $2^{31} - 1$)	
long	64	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (-2^{63} to $2^{63} - 1$)	
float	32	Negative range: -3.4028234663852886E+38 to -1.40129846432481707e-45 Positive range: 1.40129846432481707e-45 to 3.4028234663852886E+38	(IEEE 754 floating point)
double	64	Negative range: -1.7976931348623157E+308 to -4.94065645841246544e-324 Positive range: 4.94065645841246544e-324 to 1.7976931348623157E+308	(IEEE 754 floating point)

Primitive Data Types (Numeric): **Integer**

- **Integer types** are almost always an exact reflection of the hardware, so the mapping is trivial.
- Java's signed integer sizes: **byte, short, int, long**.
- A **signed integer** value is represented in a computer by a string of bits, with one of the bits (typically the leftmost) representing the sign.
- Most computers now use a notation called **twos complement** to store negative integers, which is convenient for addition and subtraction.
- In twos-complement notation, the representation of a negative integer is formed by taking the logical complement of the positive version of the number and adding one.

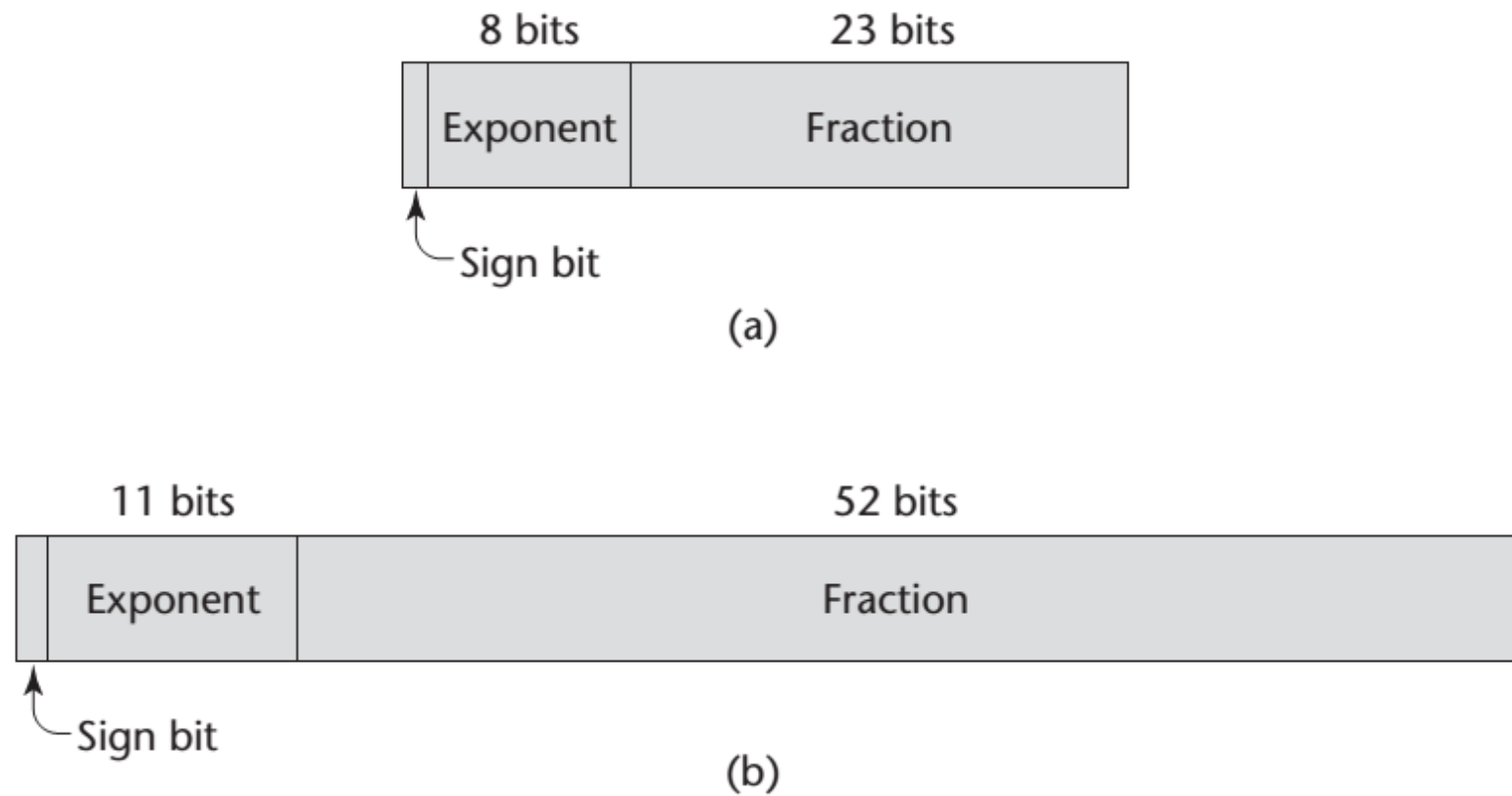
Primitive Data Types (Numeric): Floating Point

- **Floating-point** data types model **real numbers**, but the representations are only approximations for many real values.
- For example, neither of the fundamental numbers “ π ” or “e” (the base for the natural logarithms) can be correctly represented in floating-point notation.
- Most languages support at least two floating-point types (e.g., **float** and **double**), sometimes more.
- The collection of values that can be represented by a floating-point type is defined in terms of **precision** and **range**.
- Precision is the “accuracy” of the fractional part of a value, measured as the number of bits. Range is a combination of the range of fractions and, more important, the range of exponents.

IEEE Floating-Point Standard 754 format for single and double-precision representation

Figure

IEEE floating-point formats: (a) single precision, (b) double precision



Primitive Data Types (Numeric): **Decimal**

- **Decimals** are the primary data types for business data processing and are therefore essential to COBOL.
- C# and F# also have decimal data types.
- Decimal types have the advantage of being able to precisely store decimal values, at least those within a restricted range, which cannot be done with floating-point. (Advantage: *accuracy*)
- Decimal types are stored very much like (character) strings, using binary codes for the decimal digits. These representations are called **binary coded decimal (BCD)**.
- The disadvantages of decimal types are that the range of values is restricted because no exponents are allowed, and their representation in memory is mildly wasteful. (Disadvantages: *limited range, wastes memory, slow*)

Decimal(BCD) Example

◆ Representation of 93.2 (Decimal / BCD)

Step 1 – Separate the digits:

The number 93.2 consists of:

- 9
- 3
- 2 (from the fractional part)

Each digit is stored **individually**: 9, 3, 2

◆ BCD Encoding:

- 9 → 1001
- 3 → 0011
- 2 → 0010

So, in BCD, this would be stored in memory as:

1001 0011 0010

Total: 12 bits

Decimal(BCD) Comparison with Float/Double

Feature	Decimal (BCD)	Float / Double (IEEE 754)
Precision	Exact representation of decimal digits	May introduce rounding errors in decimal values
Representation	Each digit is stored separately in binary	Stored as sign, exponent, and mantissa
Performance	Slower (requires more processing)	Faster for computations
Memory Usage	May require more bits	Fixed size (32/64 bits)
Use Case	Financial applications (precision critical)	Scientific and statistical calculations

Primitive Data Types (Numeric): **Complex**

- Some programming languages support a **complex** data type—for example, Fortran and Python.
- Complex values are represented as ordered pairs of floating-point values (i.e., each value consists of two floats, the real part and the imaginary part).
- In Python, the imaginary part of a complex literal is specified by following it with a “j” or “J”—for example,
$$(7 + 3j) \rightarrow 7 \text{ is the real part and } 3 \text{ is the imaginary part.}$$
- Languages that support a complex type include operations for arithmetic on complex values.

Primitive Data Types: Boolean

- **Boolean** types are perhaps the simplest of all types.
- Their range of values has only two elements: one for **true** and one for **false**.
- In some languages (like C) numeric expressions are used a conditionals, all operands with nonzero values are considered true, and zero is considered false (but this decreases readability).
- A boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

Primitive Data Types: Character

- **Character** data are stored in computers as numeric codings.
- Traditionally, the most commonly used coding was the 8-bit code **ASCII** (American Standard Code for Information Interchange), which uses the values 0 to 127 to code 128 different characters.
- The Unicode Consortium published the UCS-2 standard, a 16-bit character set. This character code is often called **Unicode**.
- Unicode includes the characters from most of the world's natural languages (For example ç, ş, ü, vs. for Turkish).
- The first 128 characters of Unicode are identical to those of ASCII.
- Java was the first widely used language to use the Unicode character set. Since then, it has found its way into JavaScript, Python, Perl, C#, and F#.

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	n1	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

*** The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

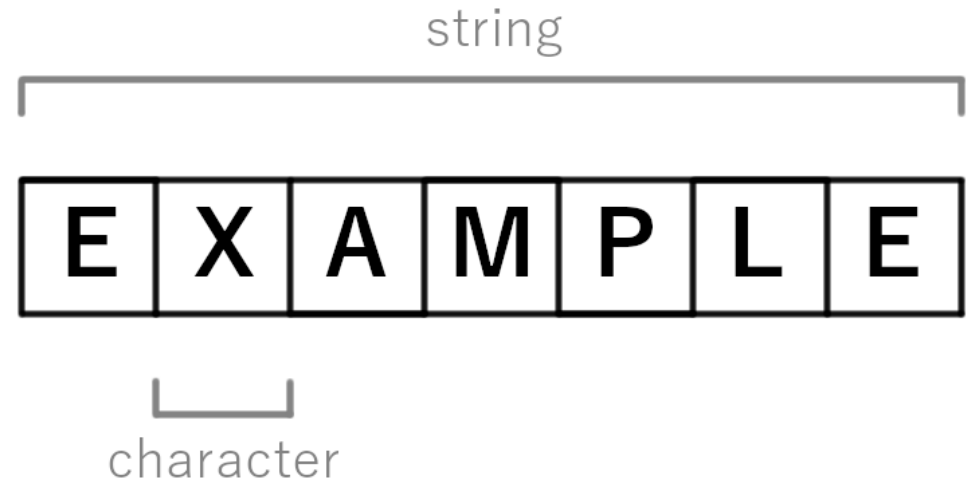


(Character) String Types



(Character) String Types

- A **(character) string type** is one in which the values consist of sequences of characters.
- The two most important design issues that are specific to (character) string types are the following:
 - Should strings be a special kind of character array or a primitive type?
 - Should strings have static or dynamic length? (String Length Options)



(Character) String Type in Certain Languages

- C and C++
 - Not primitive.
 - Use **char** arrays and a library of functions that provide operations.
- Java
 - Primitive (!) via the `String` class.
- Perl, JavaScript, Ruby, and PHP
 - Provide built-in **pattern matching**, using “regular expressions”.

String Types Operations

- **Assignment and copying**
- **Concatenation:** Concatenation is the process of appending one string to the end of another string. (Concatenation can be done with '+' operator in Java).
- **Comparison** (==, >, etc.)

String Comparison Example (Java)

```
//    ***    Checking string equality    ***
String str3 = new String("Example String");
String str4 = new String("Example String");

//incorrect usage
if(str3 == str4)
    System.out.println("str3 and str4 are equal");
else
    System.out.println("str3 and str4 are not equal");

//correct usage
if(str3.equals(str4))
    System.out.println("str3 and str4 are equal");
else
    System.out.println("str3 and str4 are not equal");
```

str3 and str4 are not equal

str3 and str4 are equal

String Types Operations: Substring Reference

- **Substring reference:** A substring reference is a reference to a substring of a given string. Substring references are sometimes called **slices**.

Example (Java)

```
//    ***    "contains" method    ***  
str1 = "You will never walk alone";  
System.out.println(str1.contains("abc"));           false  
System.out.println(str1.contains("walk"));          true
```

String Types Operations: Pattern Matching

- **Pattern matching:** In some languages, pattern matching is supported directly in the language (Perl, JavaScript, Ruby, PHP, etc.).
- The pattern-matching expressions are often called **regular expressions**.
- In other languages, it is provided by a function or class library (C++, Java, Python, C#, and F#).

history note

SNOBOL 4 was the first widely known language to support pattern matching.

String Types Operations: Pattern Matching

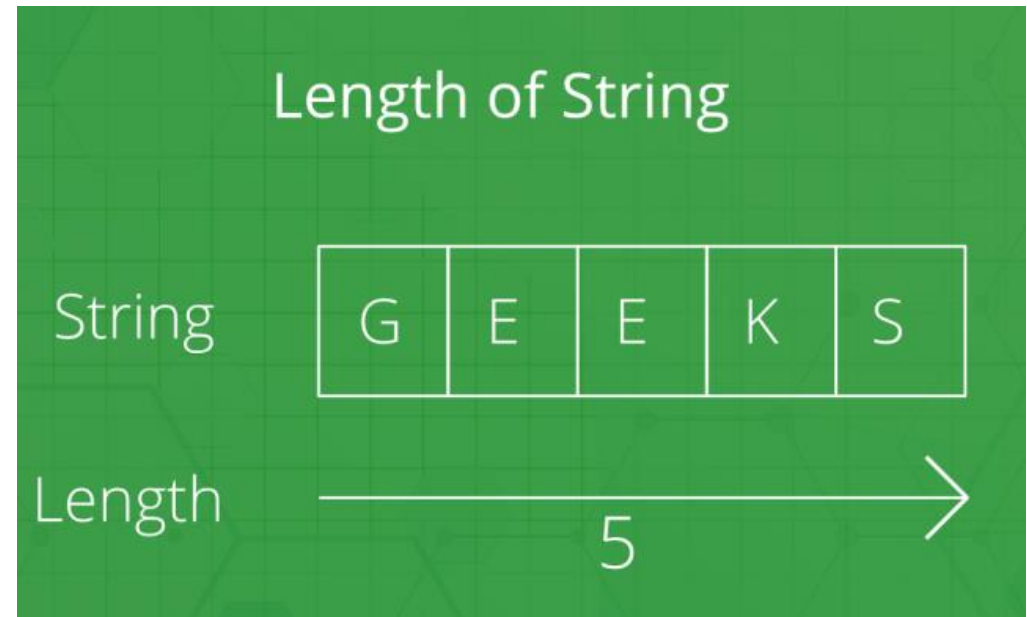
- Consider the following pattern (regular) expression:

`/\d+\.?\d*|\.\d+/'`

- This pattern matches numeric literals.
- The `\.` specifies a literal decimal point (The period must be “escaped” with the backslash because period has special meaning in a regular expression).
- The “question mark” quantifies what it follows to have zero or one appearance.
- The vertical bar (`|`) separates two alternatives in the whole pattern.
- The first alternative matches strings of one or more digits, possibly followed by a decimal point, followed by zero or more digits; the second alternative matches strings that begin with a decimal point, followed by one or more digits.

String Length Options

- There are several design choices regarding the length of string values.
 - Static Length Strings
 - Limited Dynamic Length Strings
 - Dynamic Length Strings





Static Length Strings

- The length of a string can be **static** and set when the string is created.
- Such a string is called a **static length string**.
- COBOLs, Pythons, Rubys strings are static length.
- This is also the choice for the immutable objects of Java's String class.

Example (Java)

```
String str1 = "Example String";  
System.out.println("The length of the string1 : " + str1.length());
```

```
String str2 = str1;
```

```
str1 = str1 + " emre";
```

```
System.out.println("The length of the string1 : " + str1.length());
```

```
System.out.println("The length of the string2 : " + str2.length());
```

```
The length of the string1 : 14
```

```
The length of the string1 : 19
```

```
The length of the string2 : 14
```

Example (Java)

```
String str1 = "Example String";  
System.out.println("Original String : " + str1);  
  
str1.toUpperCase();  
System.out.println("String after toUpperCase method : " + str1);  
  
String str2 = str1.toUpperCase();  
System.out.println("Other String after toUpperCase method : " + str2);
```

Original String : Example String

String after toUpperCase method : Example String

Other String after toUpperCase method : EXAMPLE STRING



Limited Dynamic Length Strings

- The second option is to allow strings to have varying length up to a declared and fixed maximum set by the variable's definition, as exemplified by the strings in C.
- These are called **limited dynamic length strings**.
- Such string variables can store any number of characters between zero and the maximum. Recall that strings in C use a special character (`'\\0'`) to indicate the end of the string's characters, rather than maintaining the string length.

Example (C)

```
int main()  
{  
    char myString[100];  
  
    printf("Enter a string : ");  
    scanf("%s", myString);  
  
    printf("The length of the string : %d", strlen(myString));  
  
    return 0;  
}
```

```
Enter a string : emre  
The length of the string : 4
```

Example (C)

```
int main()
{
    char myString1[] = {'e', 'm', 'r', 'e', '\0'};
    char myString2[] = "emre";

    printf("The length of the string1 : %d\n", strlen(myString1));
    printf("The length of the string2 : %d\n", strlen(myString2));

    return 0;
}
```

```
The length of the string1 : 4
The length of the string2 : 4
```



Dynamic Length Strings

- The third option is to allow strings to have varying length with no maximum, as in JavaScript and Perl.
- These are called **dynamic length strings**.
- This option requires the overhead of dynamic storage allocation and deallocation but provides maximum flexibility.

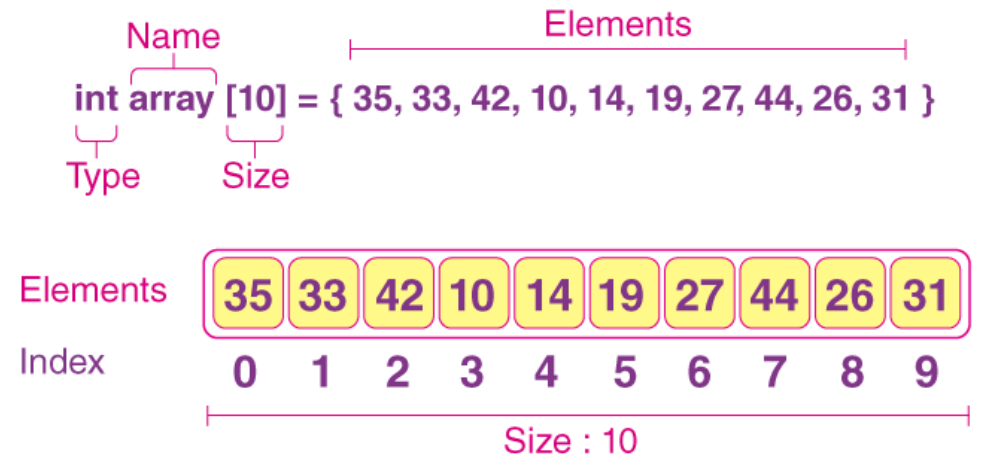


Array Types



Array Types

- An **array** is a “homogeneous” aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- The individual data elements of an array are of the same type.
- References to individual array elements are specified using subscript expressions (indexing).



Arrays and Indices

- Indexing (or subscripting) is a mapping from indices to elements.

`array_name(subscript_value_list) → element`

- **Index Syntax**
- Fortran and Ada use parentheses.
 - A problem with using parentheses to enclose subscript expressions is that they often are also used to enclose the parameters in subprogram calls; this use makes references to arrays appear exactly like those calls. This results in reduced readability.
- Most other languages use brackets (“[index]”).

Array Initialization

- Some languages provide the means to initialize arrays at the time their storage is allocated.

- C and C++ example

```
int numbers[] = {4, 5, 7, 83};
```

- (Character) strings in C and C++

```
char name[] = "freddie";
```

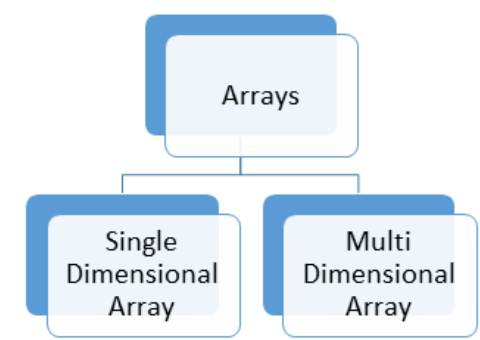
- Arrays of strings in C and C++

```
char *names[] = {"Bob", "Jake", "Joe"};
```

- Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Multidimensional Arrays



- A **multi-dimensional array** can be termed as an array of arrays that stores homogeneous data in tabular form.
- 2-dimensions → rectangular tables / **matrices**.
- In C-based languages (C, C++, Java, etc.), a reference to an element of a multidimensioned array uses a separate pair of brackets for each dimension. For example,

```
myArray[3][7];
```


Implementation of Single-dimensioned Arrays

- A single-dimensioned array is implemented as a list of adjacent memory cells.
- Suppose the array `list` is defined to have a subscript range lower bound of 0. The access function for `list` is often of the form:

$$\text{address}(a[i]) = \text{address}(a[0]) + i * \text{element_size}$$

Implementation of Multi-dimensional Arrays

- There are two ways in which multidimensional arrays can be mapped to one dimension:
 - **Row major order** – used in most languages.
 - **Column major order** – used in Fortran.
- For example, if the matrix had the values

3 4 9 1

6 2 5 2

1 7 8 3

- it would be stored in row major order as

3, 4, 9, 1, 6, 2, 5, 2, 1, 7, 8, 3

* The address of an element calculated as (n is the size of a row):

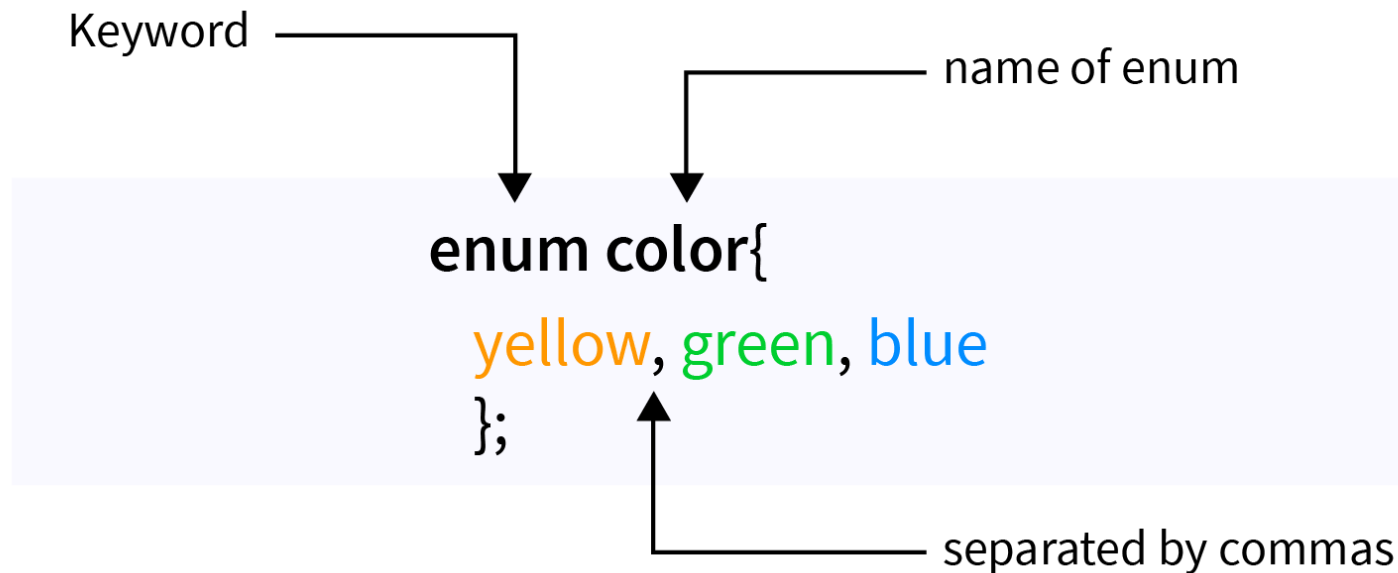
$\text{address}(a[i][j]) = \text{address}(a[0][0]) + ((i * n) + j) * \text{element_size}$



Enumeration Types



Enumeration Types



- An **enumeration type** is one in which all of the possible values, which are **named constants**, are provided, or enumerated, in the definition.
- Enumeration types provide a way of defining and grouping collections of named constants, which are called “enumeration constants”.
- The definition of a typical enumeration type is shown in the following C# example:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

Enumeration Types – Java Example

```
public class EnumeratedTypesExample
{
    private enum Status
    {
        WON, LOST, CONTINUE;
    }

    public static void main(String[] args)
    {
        Status gameStatus = Status.LOST;

        switch(gameStatus)
        {
            case WON:
                System.out.println("You won the game !");
                break;
            case LOST:
                System.out.println("You lost the game !");
                break;
            case CONTINUE:
                System.out.println("The game continues !");
                break;
        }
    }
}
```

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color, day etc. as a number.
- Aid to reliability, e.g., compiler can check:
 - Operations (don't allow colors, days to be added)
 - No enumeration variable can be assigned a value outside its defined range.