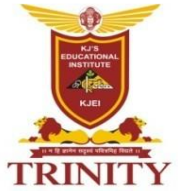


K J's EDUCATIONAL INSTITUTES
TRINITY COLLEGE OF ENGINEERING & RESEARCH
Approved by AICTE, Government of Maharashtra & Affiliated to Savitribai Phule Pune University

Accredited NAAC Grade 'A+'

Department of Computer Engineering

Course Name :Principles of Programming Languages

Course Code: 210255

Unit : 2 - Structuring the Data, Computations and Program

Course Coordinator: Mrs. Shaikh J. N.

A.Y : 2024-25

Semester: II

Course Objectives:

- **To learn** structuring the data and manipulation of data, computation and program structure.

Course Outcomes:

On completion of the course, learner will be able to–
CO2: **Develop** a program with Data representation and Computations.

Contents :

- **Elementary Data Types :** Primitive data Types, Character String types, User Defined Ordinal Types, Array types, Associative Arrays, Record Types, Union Types, Pointer and reference Type.
- **Expression** and **Assignment Statements:** Arithmetic expression, Overloaded Operators, Type conversions, Relational and Boolean Expressions, Short Circuit Evaluation, Assignment Statements, Mixed mode Assignment.
- **Statement level Control Statements:** Selection Statements, Iterative Statements, Unconditional Branching.
- **Subprograms:** Fundamentals of Sub Programs, Design Issues for Subprograms, Local referencing Environments, Parameter passing methods. Abstract Data Types and Encapsulation Construct: Design issues for Abstraction, Parameterized Abstract Data types, Encapsulation Constructs, Naming Encapsulations

Topic	Book To Refer
Elementary Data Types : Primitive data Types, Character String types, User Defined Ordinal Types, Array types, Associative Arrays, Record Types, Union Types, Pointer and reference Type.	**Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN-81-7808-161-X.** **Page No : 267 - 323**
Expression and Assignment Statements: Arithmetic expression, Overloaded Operators, Type conversions, Relational and Boolean Expressions, Short Circuit Evaluation, Assignment Statements, Mixed mode Assignment.	Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN-81-7808-161-X. Page No : 330-355
Statement level Control Statements: Selection Statements, Iterative Statements, Unconditional Branching.	**Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN-81-7808-161-X.** **Page No : 362-390**

Topic	Book To Refer
Subprograms: Fundamentals of Sub Programs, Design Issues for Subprograms, Local referencing Environments, Parameter passing methods.	**Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN-81-7808-161-X.** **Page No : 402-436**
Abstract Data Types and Encapsulation Construct: Design issues for Abstraction, Parameterized Abstract Data types, Encapsulation Constructs, Naming Encapsulations	**Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN-81-7808-161-X.** **Page No : 492, 508-521**

SPPU Questions

UNIT -2			
Q.1	Describe ordinal types: enumeration with 'C++' example.	CO2	U
Q.2	What are different parameters passing methods in programming languages with example.	CO2	U
Q.3	What are the different primitive data types? Explain with the examples of syntax, size and ranges.	CO2	R, U
Q.4	Explain following concepts with example: i) Overloaded unary operator ii) Short circuit evaluation	CO2	U
Q.5	What are subprograms? List and explain the design issues for subprograms .	CO2	U
Q.6	Write short note on: i) Mixed mode Assignment ii) Unconditional branching.	CO2	U
Q.7	What are the different control flow structures used in programming, explain with an example?	CO2	U
Q.8	What are functions and why are they important for code orgnization explain with an example?	CO2	U
Q.9	What is a mixed-mode-expression? Explain short circuit evaluation with an example.	CO2	U
Q.10	What are primitive data types? List the primitive data types in Java and their respective storage capacity?	CO2	R, U
Q.11	Why selection and iteration statements are used in programming languages. What is the general from of a two-way selector?	CO2	U
Q.12	Explain how pointers differ from references with example?	CO2	U

Elementary Data Types

Primitive Data Types	• Integer • Floating Point • Decimal
Character String Types	• Design Issues • String Operations • Implementation
User-Defined Ordinal Types	• Enumeration Types • Subrange Types • Implementation
Array Types	• Array and Indexes • Subscript bindings • Array Initialization
Associative Arrays	• Structure and Operation • Implementation
Record Types	• Definition • Operation • Evaluation and Implementation
Union Types	• Evaluation and Implementation
Pointer and Reference Types	• Design Issues • Pointe Operations • Evaluation and Implementation

Primitive Data Types

- Primitive data types are those **that are not defined in terms of other data types**
- **Early PL**s had **only numeric primitive types**, and still play a c**entral role among the collections of types** supported by contemporary languages.

Numeric Types

- Integer
- Floating Point
- Decimal

Boolean Types

Character Types

Numerical Data Types …Integer

Several size of integers and different **capabilities**

Java Uses: **byte, short, int, and long**

C++ and C# Include Unsigned Integers for Binary Data

An Signed integer is represented **by a string of bits, with the leftmost representing the sign bit.**

Most **computer use 2's complement to store negative numbers**

Numerical Data Types …Floating Point

Model real numbers but only as approximations

Languages **for scientific use** support at least two floating-point types; sometimes more.

Floating points **are stored in binary**

- Eg. 0.1 cannot be represented by finite number of digits using binary

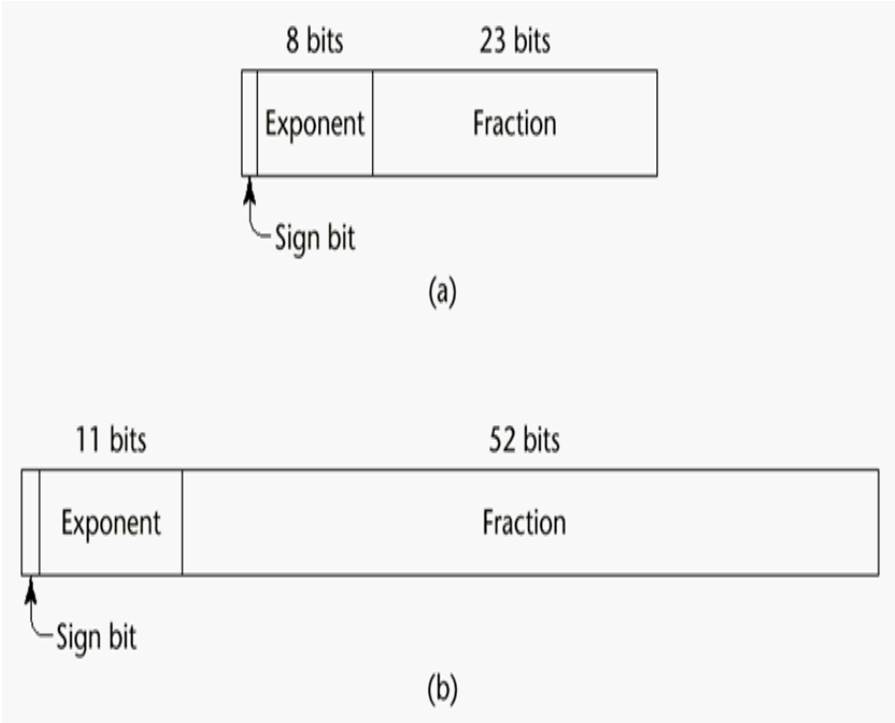
Loss of accuracy through aritmetic operations

Can **be represented using fractions and exponents.**

Most languages support two floating point types:

- Float: standard size, 4 bytes of memory
- Double :larger fraction parts are needed, twice memory compared t0 float

Newer Machines Use IEEE (The Institute of Electrical and Electronics Engineers) floating-point formats: (a) Single precision, (b) Double precision



Numerical Data Types …Decimal

For **business applications (money)**

Store a fixed number of decimal digits (coded)

Stored **like character string, binary codes** for decimal digits

One digit per byte or two digits per byte

Four bits to code decimal digit, **six digit** decimal number needs **24 bits of memory**

- **Advantage**
  - Accuracy
- **Disadvantages:**
  - Limited range
  - Wastes memory

Boolean Types

The range of values has only two elements **TRUE or FALSE**

Introduced in **ALGOL 60**

Booleans types are often used to represent **switches or flags in programs**

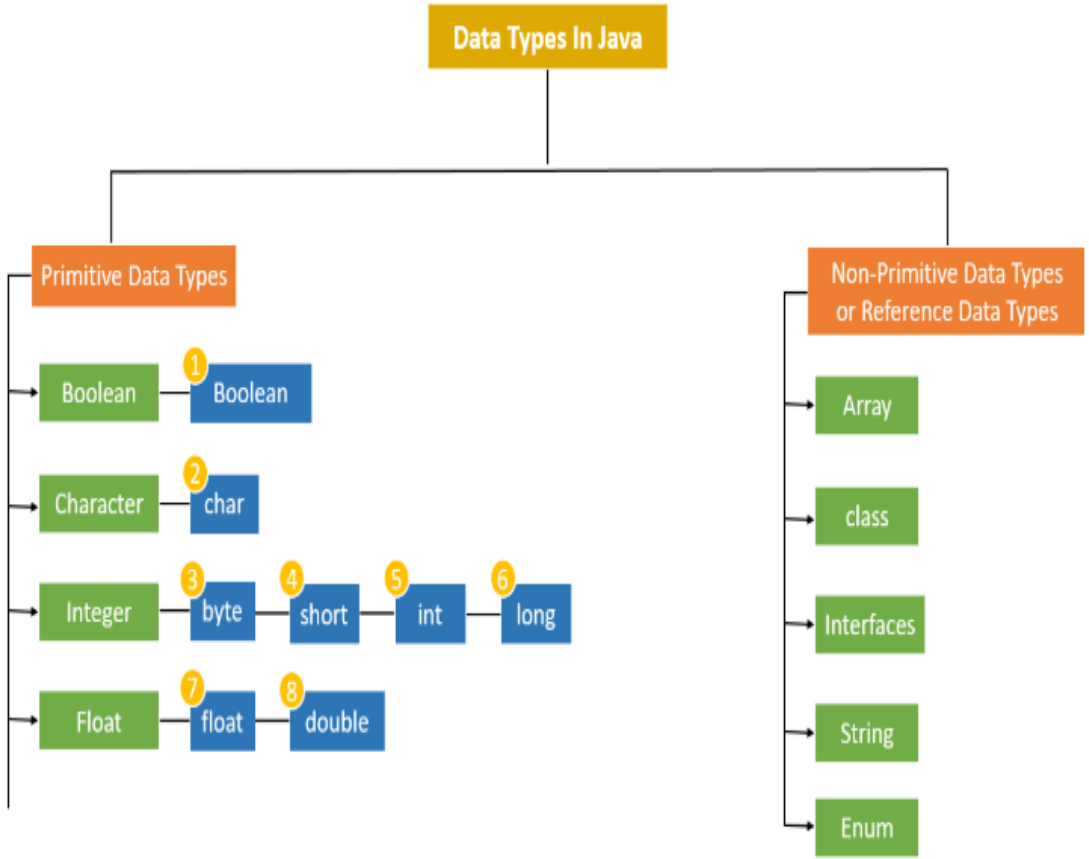
Zero is Considered as **False** and All **Non-Zero are True**

Represented using s**ingle bit, but it is difficult to access on many machine**

Often stored in the Smallest efficiently a**ddressable cell of memory i.e byte**

Advantage:
Readability

Primitive Data types in Java



Type	Size (in bits)	Range
byte	8	-128 to 127
short	16	-32,768 to 32,767
int	32	-2^{31} to $2^{31}-1$
long	64	-2^{63} to $2^{63}-1$
float	32	1.4e-045 to 3.4e+038
double	64	4.9e-324 to 1.8e+308
char	16	0 to 65,535
boolean	1	true or false

Character Types

Stored as **numeric codings**

Usually **ASCII** but **Unicode** has appeared as an alternative

A new 16-bit character set named Unicode had been developed as an alternative.

Unicode Includes Characters from most of the world's natural language like **Thai digits, Cyrillic albhaets of Serbia**

Java is the first to use Unicode

Character String Types

Character string type is one in which the values **consist of sequences of characters**

Design issues with the string types

- **Should strings be simply a special kind of character array or a primitive type?**
- **Should strings have static or dynamic length?**

Character String Types .... String Operations

Assignment

- (Java: str1 = str2;) (C: strcpy(pstr1, pstr2);

Comparison (=, >, etc.)

- BASIC: str1 < str2

Concatenation

- C: strcat (str1,str2) (Java : str2 + str3;)

Substring reference

- **Also** known as **Slices** i.e **reference to substring**

Pattern matching

Character String Types.. String Length Options

Static length

- Set when string is created

Limited dynamic length

- Store number of characters between 0 and maximum

Dynamic length

- **Varying Length** with no maximum.

Character String Types

- **C and C++**
  - not primitive
  - use char arrays and a library of functions that provide operations
- **Java** : String class (not arrays of char)
  - objects are immutable
  - StringBuffer is a class for changeable string objects
  - String length options
- **Static** – Python, **Java's String class**, C++ standard class library, **Ruby's** built-in **String class**, and the .NET class library in C# and F#.
- **Limited dynamic length** – C and C++ (up to a max length indicated by a null character)
- **Dynamic** –Perl, JavaScript

Character String Types ...Implementation

- **Static length - Compile-time descriptor**
- **Limited dynamic length** - may need a **run-time descriptor for length** (but not in C and C++ because the end of a string is marked with the null character)
- d**ynamic length - need simpler run-time descriptor**; allocation/deallocation is the biggest implementation problem

Static string
Length
Address

Limited dynamic string
Maximum length
Current length
Address

Compile – time descriptor for static strings

Run-time descriptor for limited dynamic strings

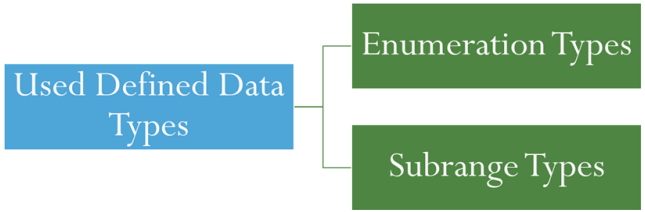
Dynamic length allocation by three approaches:

Dynamic length allocation by three approaches:

- **First approach :**
  - **Using linked list**
  - **Disadvantage - Extra storage for links** and **complexity of operations**
- **Second approach :**
  - Store **as array of pointers** to individual characters allocated in a heap.
  - **Disadvantage-** Still uses **extra memory**
- **Third approach:**
  - To store complete strings in **adjacent storage cells**
  - When **a string grows** and adjacent storage is not available, **a new area of memory is found** that can **store the complete new string** and the **old part is moved to this area,** and the memory cells f**or the old string are deallocated.**
  - This results in **faster string operations** and requires **less storage**
  - **Disadvantage : = Allocation / deallocation process is slower.**

Used Defined Ordinal Types

An ordinal type is one in which t**he range of possible values can be easily associated with the set of positive integers**



Enumeration Types

- **A**ll possible values, which are **named constants**, are provided in the definition
- C# example

  enum days {mon, tue, wed, thu, fri, sat, sun};

- The enumeration constants are typically i**mplicitly assigned the integer values, 0, 1, …**, but can be **explicitly assigned any integer literal in the type's definition**

Design issues

- Is an enumeration c**onstant allowed to appear in more than one type definition**, and if so, how is the **type of an occurrence of that constant checked?**
- Are enumeration values **coerced to integer?**
- **Any other type coerced** to an enumeration type?

- In languages that **do not have enumeration types**, programmers **usually simulate them with integer values**.
- E.g. Fortran 77, use 0 to represent blue and 1 to represent red:

```
INTEGER RED, BLUE

DATA RED, BLUE/0,1/
```

- Problem:

  There is **no type checking when they are used**.

  It would be legal to add two together. Or they can be assigned any integer value thus **destroying the relationship with the colors.**

Enumeration Types-Design

- In C++, we could have

  enum colors {red, blue, green, yellow, black};

  - colors myColor = blue, yourColor = red;
- The enumeration **values are coerced to int when they are put in integer context**.

  E.g. **myColor++** would assign green to **myColor.**
- In Java, all enumeration types are **implicitly subclasses of the predefined class Enum**. They can have instance **data fields, constructors and methods**

Java Example

```
Enumeration days;
Vector dayNames = new Vector();
dayNames.add("Monday");
…
dayNames.add("Friday");
days = dayNames.elements();
while (days.hasMoreElements())
        System.out.println(days.nextElement());
```

- **C# enumeration types are like those of C++** except that t**hey are never coerced to integer.**
- **Operations are restricted to those that make sense**.
- The range of values is **restricted to that of the particular enumeration type.**

Enumeration Types- Evaluation

- **Aid to readability,** e.g., no need to code a color as a number
- **Aid to reliability,** e.g., compiler can check: operations (don't allow colors to be added)
- **No enumeration variable can be assigned a value outside its defined range**,
  - e.g. if the colors **type has 10 enumeration constants** and uses **0 .. 9** as its internal values, **no number greater than 9 can be assigned to a colors type variable**.
- **Ada, C#, and Java 5.0 provide better support** for enumeration **than C++ because enumeration type variables in these languages are not coerced into integer types**
- **C treats enumeration variables like integer variables**; it does not provide either of the two advantages.
- **C++ is better.** Numeric values **can be assigned to enumeration type variables only if they are cast to the type of the assigned variable.** Numeric values are checked to determine in they are in the range of the internal values. However **if the user uses a wide range of explicitly assigned values, this checking is not effective.**
  - E.g. enum colors {red = 1, blue = 100, green = 100000}
  - A value assigned to a variable of colors type will only be checked to determine whether it is in the range of 1..100000.
- **Java 5.0, C# and Ada are better**, as variables **are never coerced to integer types**

Subrange Type

An ordered contiguous subsequence of an ordinal type

- Not a new type, but a restricted existing type

  Example: 12..18 is a subrange of integer type

Introduced by Pascal and included in Ada

Ada's design

type Days **is** (Mon, Tue, Wed, Thu, Fri, Sat, Sun);

subtype Weekdays **is** Days **range** Mon..Fri;

subtype Index **is** Integer **range** 1..100;

Day1: Days;

Day2: Weekday;

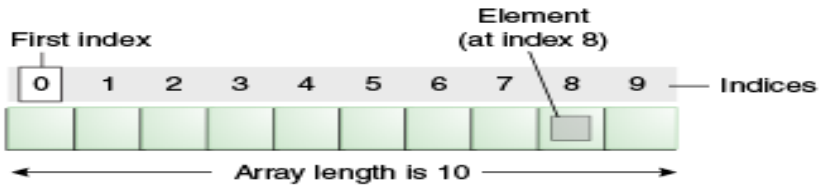
Day2 := Day1; **//legal if Day1 it not set to Sat or Sun**

Subrange Type -Evaluation

- **Aid to readability**
  - Make it clear to the readers **that variables of subrange can store only certain range of values**
- **Reliability**
  - Assigning a value **to a subrange variable that is outside the specified range is detected as an error.**

Array Type

An array is **an aggregate of homogeneous data elements i**n which **an individual element is identified by its position in the aggregate, relative to the first element.**




```
int [ ] num = new int [ 10 ];
```

- type of each element
- name of array
- subscript (integer or constant expression for number of elements.)

Array Type- Design Issues

- What **types are legal** for subscripts?
- Are subscripting expressions **in element references range checked**?
- When **are subscript ranges bound**?
- When **does allocation take place**?
- Are **ragged or rectangular multi dimensioned arrays** allowed, or both?
- Can arrays be initialized when **they have their storage allocated?**
- Are any **kind of slices allowed?**

Array and Indexes

- **Indexing (or subscripting) is a mapping from indices to element**
  - array_name (index_value_list) -> element
- **Index Syntax**
  - **FORTRAN, PL/I, Ada use parentheses** , Sum-:= Sum +**B(I)**;
  - **Ada explicitly uses parentheses** to show uniformity between array references and function calls because both are mappings
  - **Most other languages use brackets**
  - E.g. **List(27)** direct reference to the List array's element with the subscript 27

Language	Index Type
FORTRAN , C, Java	**Integer**
PASCAL	**Any Ordinal Type** (Integer, Boolean, Enumeration, Character)
Ada	**Integer Or Enumeration**

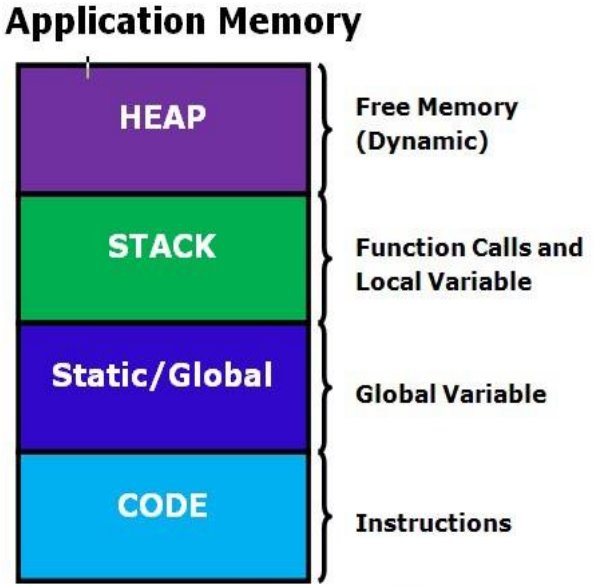
Array and Indexes

- **C, C++, Perl, and Fortran do not specify range checking**
- **Java, ML, C# specify range checking**

Subscript Bindings and Array Categories

- Binding is **usually static**
- Subscript **value ranges are dynamically bound**
- Lower bound are **implicit**
- In **C based languages- lower bound of all index ranges is fixed at 0;**
- Fortran **95 defaults to 1**
- In some other languages, ranges **must be completely specified by the programmer**
- **Array occurs in 5 categories**

Flip ClassRoom Activity Memory Allocation : Stack and Heap Gap Analysis



Static Allocation	Dynamic Allocation
Performed at static or compile time	Performed at dynamic or run time
Assigned to stack	Assigned to heap
Size must be know at compile time	Size may be unknown at compile time
First in last out	No particular order of assignment
It is best if required size of memory known in advance	It is best if we don't have idea about how much memory require

Categories of Arrays

- Static
- Fixed Stack – Dynamic Array
- Stack- Dynamic Array
- Fixed Heap – Dynamic Array
- Heap- Dynamic Array

1. Static

- subscript ranges are **statically bound** and **storage allocation is static** (before run- time)
- Advantage: **efficiency** (no dynamic allocation/deallocation required)
- Example: **In C and C++ arrays** that include the static modifier are static
- **static int myarray[3] = {2, 3, 4};**
- **int static_array[7];**

2. Fixed stack-dynamic

- **subscript** ranges are **statically bound,** but the **allocation is done at declaration time**
- Advantage: **space efficiency**
- Example: arrays **without static modifier are fixed stack-dynamic**
- **int array[3] = {2, 3, 4};**
- **E.g.**

```
void foo()
{   int fixed_stack_dynamic_array[7];
  /* ... */
}
```

3. Stack-dynamic

- **Stack-dynamic: subscript ranges are dynamically bound** and t**he storage allocation is dynamic** (done at run- time)
- Advantage: **flexibility** (the size of an array need not be known until the array is to be used)
- **Example:** In Ada, you can use stack-dynamic arrays as

```
Get(List_Len);
 declare List: array (1..List_Len) of Integer
```

4. Fixed heap_dynamic_array

- similar to fixed stack- dynamic: **storage binding is dynamic but fixed after allocation (i.e., binding is done when** requested & storage is allocated from heap, not stack)
- Example: **In C/C++, using malloc/free to allocate/deallocate** memory from the heap
- Java has fixed **heap dynamic arrays**
- C# includes a **second array class ArrayList that**

provides fixed heap-dynamic

5. Heap_dynamic_array

- Binding of subscript ranges and **storage allocation is dynamic and can change any number of times**
- **Advantage:** flexibility (arrays can grow or shrink during program execution)
- **Examples: Perl, JavaScript, Python, and Ruby** support heap-dynamic arrays
- Perl: @states = (“Idaho",“Washington",“Oregon");
- Python: a = [1.25, 233, 3.141519, 0, -1]

Static	**int static_array[7];**
fixed stack-dynamic	void foo() **{ int fixed_stack_dynamic_array[7]; }**
Stack-dynamic	void foo(int n) **{ int stack_dynamic_array[n];}**
fixed heap_dynamic_array	**int * fixed_heap_dynamic_array = malloc(7 * sizeof(int));**
heap_dynamic_array	void foo(int n) **{ int * heap_dynamic_array = malloc(n * sizeof(int)); }**

Subscript Bindings and Array Categories

Categories of Array	Range of subscripts	Storage Bindings	Advantages	Example
1. Static Array	Statically bound	Static (done before runtime)	Execution efficiency	Fortran-77, C, C++
2. Fixed Stack Dynamic Array	Statically bound	Done during execution	Storage space can be reused	C , C++
3. Stack Dynamic Array	Dynamic	Dynamic	Flexibility	Ada
4. Fixed Heap Dynamic Array	Fixed after storage is allocated -done when user program requests during execution	Fixed after storage is allocated -allocated from the heap	Flexibility	Fortran 95, C, C++, Java, C#
5. Heap Dynamic Array	Dynamic	Dynamic	Flexibility	C#, Java, JavaScript, Python, Ruby

Array Initialization

Some language allow initialization at the time of storage allocation

- **Fortran**

  List (3) Data List /0, 5, 5/ **// List is initialized to the values**

- **C, C++, Java, C# example**

  int list [] = {4, 5, 7, 83}

- **Character strings in C and C++**

  char name [] = “freddie”; // **eight elements, including last element as null character**

- **Arrays of strings in C and C++**

  char *names [] = {“Bob”, “Jake”, “Joe”];

- **Java initialization of String objects**

  String[] names = {“Bob”, “Jake”, “Joe”};

- **Ada positions for the values can be specified:**

  List : array (1..5) of Integer := (1, 3, 5, 7, 9);

  Bunch : array (1..5) of Integer:= (1 => 3, 3 => 4, others => 0);

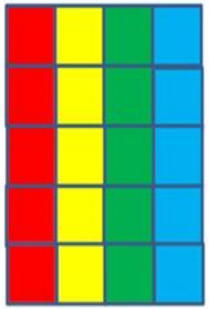
  Note: the array value is (3, 0, 4, 0, 0)

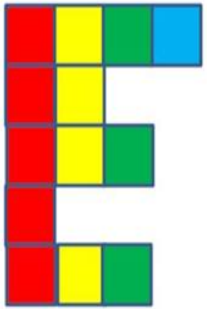
  Concatenation. One array can be appended to another array using the & operator provided that they are of the same type.

Array Operations

Language	Array Operation
C-based	No Operations , only through methods
C#, Perl	Array Assignments
Ada	Assignment, Concatenation(&), Comparison
Python	Concatenation(+), Element Membership(in), Comparison(==, is)
FORTRAN	Matrix Multiplication, Transpose
APL(Most powerful)	+.* ,

Rectangular and Jagged Arrays





Language supported:

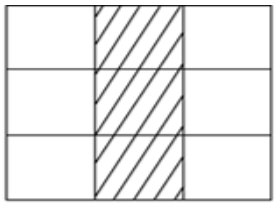
- Fortran
- Ada
- C#

Language supported:

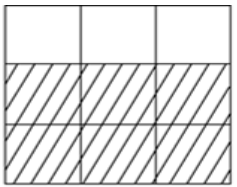
- Java
- C
- C++
- C#

Slice

- A slice is some substructure of an array; nothing more than a referencing mechanism
- Slices are only useful in languages that have array operations
- Fortran Declaration:
  - Vector(3:6) Four element from third to sixth
  - Mat(1:3,2) Second Column of Mat
  - Mat(3, 1:3) Third row of Mat
  - Vector(2:10:2) Complex size , second, fourth ,sixth, eight, tenth
  - Vector(/3,2,1,8/)



MAT (1:3, 2)



MAT (2:3, 1:3)

Implementation of array Types

Actual Address of the **1st** element of the array is known as **Base Address (B)**

Here it is 1100

Memory space acquired by every element in the Array is called **Width (W)**

Here it is 4 bytes

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Lower Limit/Bound of Subscript **(LB)**

Address Calculation in One Dimensional Array:

Address of **A [I] = B + W * (I – LB)**

Where,
B = Base address
W = Storage Size of one element stored in the array (in byte)
I = Subscript of element whose address is to be found
LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Address of **A [I] = B + W * (I – LB)**

- I=3 , B=1100 W=4 LB=0
- A[3]= 1100+4*(3-0)
- A[3]= 1100+12 =1112

Actual Address in the Memory	1100	1104	1108	1112	1116	1120
Elements	**15**	**7**	**11**	**44**	**93**	**20**
Address with respect to the Array (Subscript)	0	1	2	3	4	5

Address Calculation in Double (Two) Dimensional Array

Row Major Order

Int num[3][3];

	[0]	[1]	[2]
[0]	2	6	10
[1]	3	5	8
[2]	12	1	7

2 6 10 3 5 8 12 1 7

Language : C, Pascal, Java

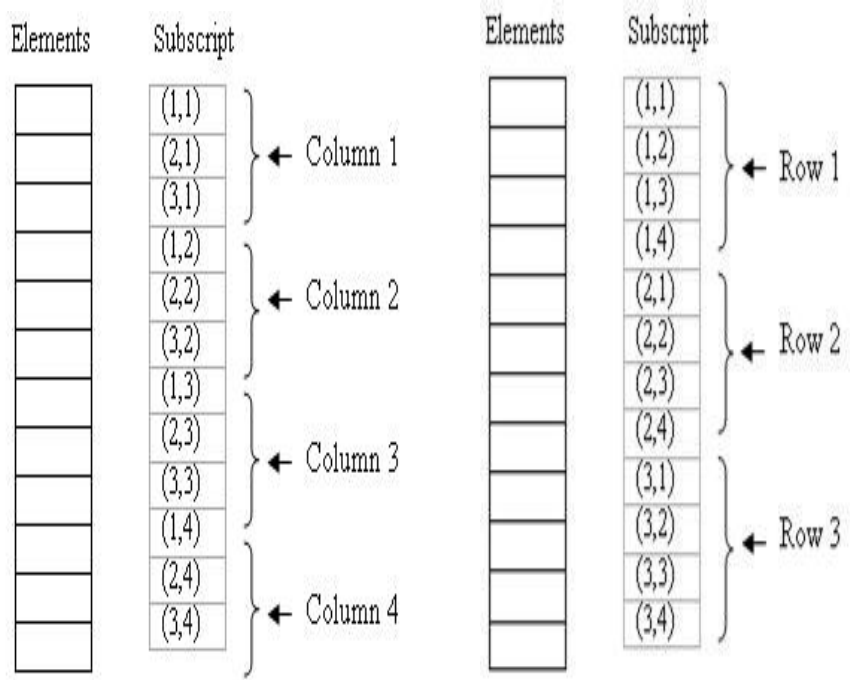
Column Major Order

Int num[3][3];

	[1]	[2]	[3]
[1]	2	6	10
[2]	3	5	8
[3]	12	1	7

2 3 12 6 5 1 10 8 7

Language : Fortran



- B = Base address
- I = Row subscript of element whose address is to be found
- J = Column subscript of element whose address is to be found
- W = Storage Size of one element stored in the array (in byte)
- Lr = Lower limit of row/start row index of matrix, if not given assume 0 (zero)
- Lc = Lower limit of column/start column index of matrix, if not given assume 0 (zero)
- M = Number of row of the given matrix
- N = Number of column of the given matrix

Row Major System:

Address of A [I][J] = B + W * [N * (I – Lr) + (J – Lc)]

Column Major System:

Address of A [I][J] Column Major Wise = B + W * [(I – Lr) + M * (J – Lc)]

Usually number of rows and columns of a matrix are given (**like A[20][30] or A[40][60]**) but

if it is given as A[Lr- – – – – Ur, Lc- – – – – Uc]. In this case number of rows and columns are calculated using the following methods:

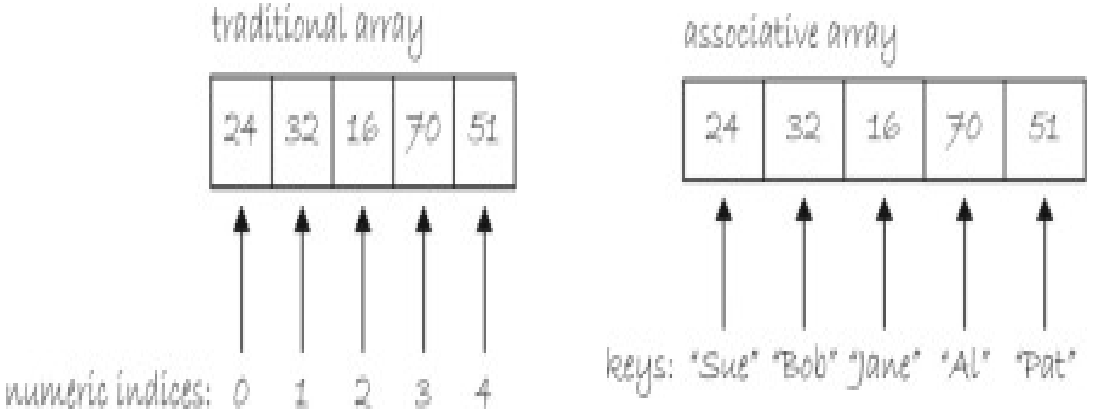
Number of rows (M) will be calculated as = (Ur – Lr) + 1
Number of columns (N) will be calculated as = (Uc – Lc) + 1

An array X [-15…….10, 15……40] requires one byte of storage.

If beginning location is 1500 determine the location of X [15][20]

Associative Array

- An associative array is **an unordered collection of data** elements that are indexed by an equal number of values called keys
- Also known as Hash tables
  - Index **by key (part of data) rather than value**
  - **Store both key and value** (take more space)
  - Best when **access is by data** r**ather than index**



Design Issues

- What is the **form of references to elements?**
- Is the size **static or dynamic?**

Associative Array Example in Java

```
Map<String, String> map = new HashMap<String, String>();

// method to add the key,value pair in hashmap
map.put("geeks", "course");
map.put("name", "geeks");


// method to get the value
map.get("name"); // returns "geeks"
```

Steps

1. First **initialize** the map

```
Map<String ,String> map = new HashMap<>();
```

2. Then Put the **Key, Value** to the **map** using put method

```
map.put("geeks","Course");
```

3. After putting all Key Value to the map Convert the map to set using the **entrySet()** Method

```
Set<Map.Entry<String ,String> > set = map.entrySet();
```

4. Then Now convert the set to the List Array using the function ;

```
List<Map.Entry<String ,String>> list=new ArrayList<>(set);
```

Record Type

What is record?

Record is an aggregate of data elements in which the individual elements are identified by names and accessed through offset from the beginning of the structure.

Record	Array
Fields are referenced by using their names	**Fields** are referenced by indices
Use when the collection of data objects is heterogeneous and fields are not processed in same way.	Use when all the data elements have same type and are processed in same way.
Fields of records are not processed in any particular sequential order.	Processing of array elements is usually done by in a sequential order

In object-oriented programming, an object is a record that contains state and method fields.

Definition of Records in COBOL

COBOL uses level numbers to show nested records; others use recursive definition

```
01 EMP-REC.
        02 EMP-NAME.
                05 FIRST   PIC X(20).
                05 MID                PIC X(10).
                05 LAST               PIC X(20).
        02 HOURLY-RATE PIC 99V99.
```

References to Records

- **Record field references**

  1. COBOL : field_name OF record_name_1 OF ... OF record_name_n

  2. Others (dot notation) :record_name_1.record_name_2. ... record_name_n.field_name

- **Fully qualified references** must include all record names
- **Elliptical references** allow leaving out record names as long as the reference is unambiguous,

for example in COBOL

FIRST, FIRST OF EMP-NAME, and FIRST of EMP-REC are elliptical references to the employee's first name

```
struct {              int age;
        struct {
                       char *first;
                       char *last;
                      } name;
 } person;
```

f**ully qualified reference**: lists all intermediate names as in

person.name.first

elliptical: may omit some of them if unambiguous (COBOL,PL/1) as in

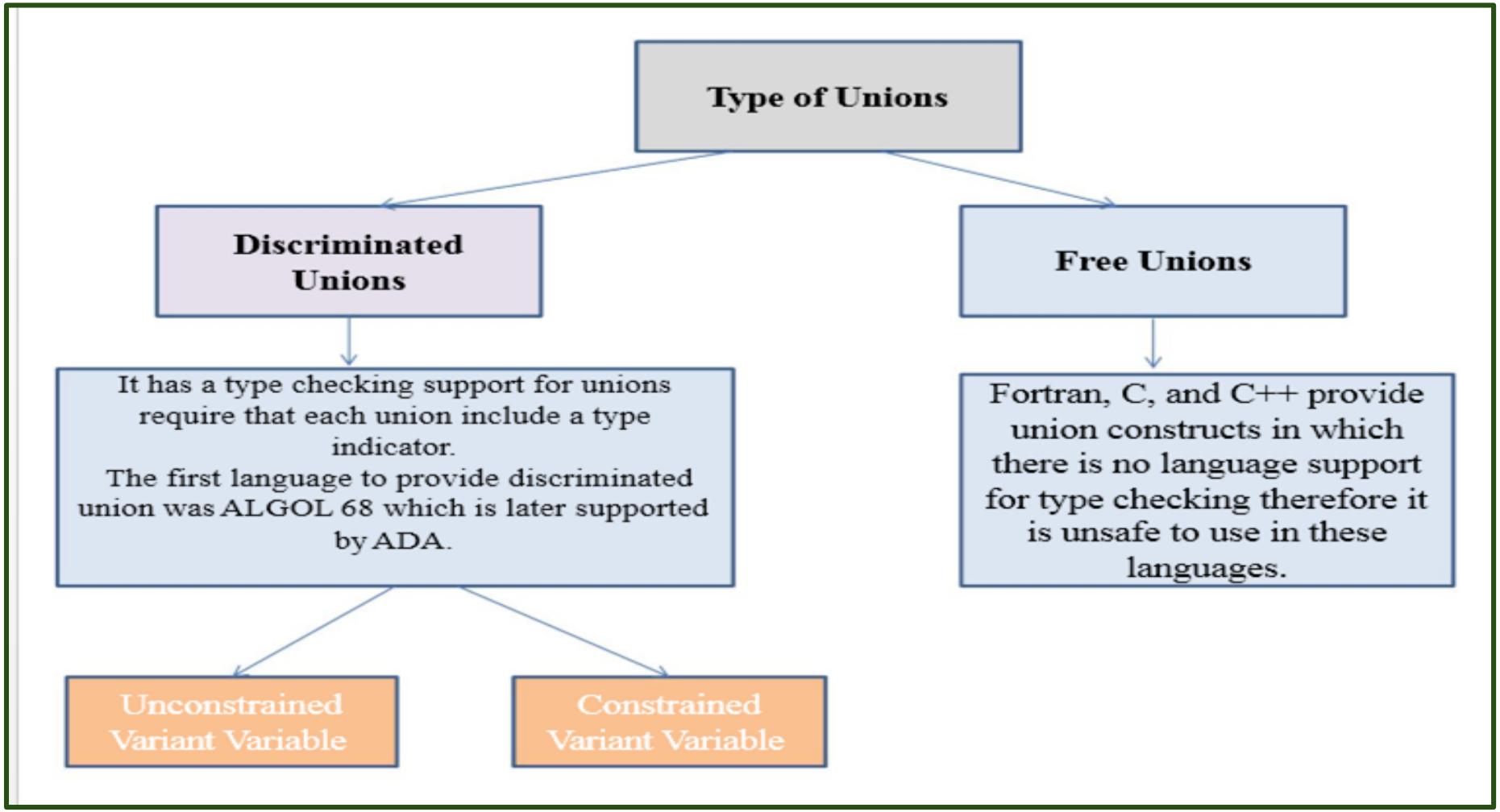
first of person

Union

- A union is a type w**hose variables are allowed to store different type values at different times during execution**
- **Design issues**
  - Should type checking be required?
  - Should unions be embedded in records?

Discriminated vs. Free Unions

- **Fortran, C, and C++ provide union constructs in which there is no language support for type checking**; the union in these languages is called **free union**
- **Type checking of unions require that each union include a type indicator called a discriminant**
- Supported by **Ada**


Type of Unions
Discriminated Unions
Free Unions
It has a type checking support for unions require that each union include a type indicator.
The first language to provide discriminated union was ALGOL 68 which is later supported by ADA.
Fortran, C, and C++ provide union constructs in which there is no language support for type checking therefore it is unsafe to use in these languages.
Unconstrained Variant Variable
Constrained Variant Variable

How does union implemented in programming languages?

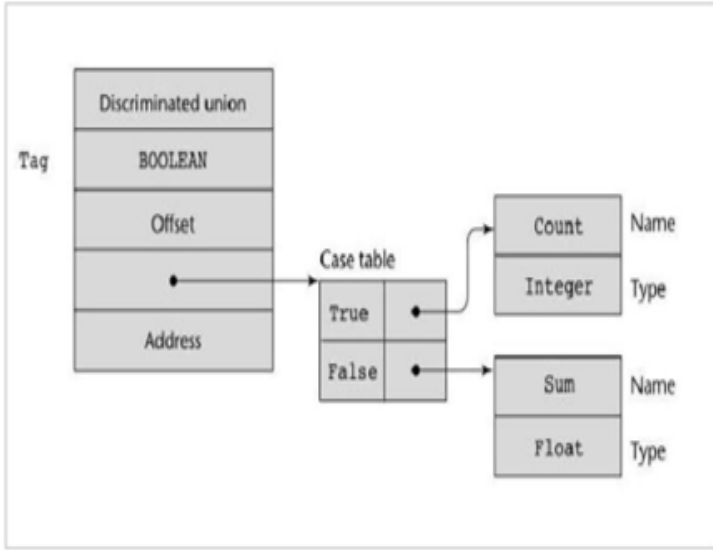
Use the same address for every variant.

At compile time, the complete description of each variant must be stored by creating a case table.

Sufficient storage for the largest variant is allocated.

In Ada language, the exact amount of storage can be used because there is no variation.

```
type Node (Tag : Boolean) is
  record
  case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
  end case;
 end record;
```



Pointer and reference type

- A pointer type variable has a **range of values that consists of memory addresses and a special value, nil**
- Provide **the power of indirect addressing**
- Provide **a way to manage dynamic memory**
- A pointer can be used **to access a location in the area where storage is dynamically created** (usually called a heap)

Design Issues of Pointers

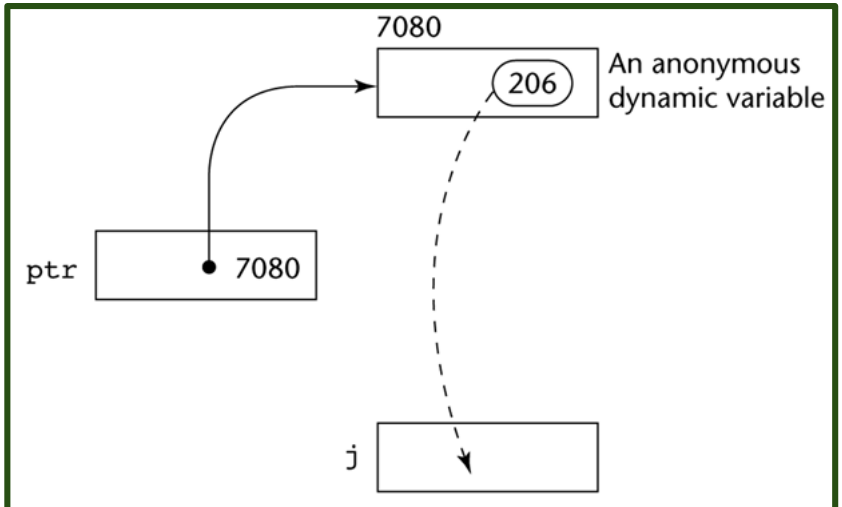
- What are the sc**ope of and lifetime of a pointer variable?**
- What **is the lifetime of a heap-dynamic variable?**
- Are pointers restricted **as to the type of value to which they can point?**
- Are pointers **used for dynamic storage management, indirect addressing, or both?**
- Should the language support **pointer types, reference types, or both?**

Pointer Operations

- Two fundamental operations: **assignment and dereferencing**
- **Assignment** is used **to set a pointer variable's value to some useful address**
- **Dereferencing yields the value stored at the location represented by the pointer's value**
- Dereferencing can be **explicit or implicit**
- C++ uses an explicit operation via *

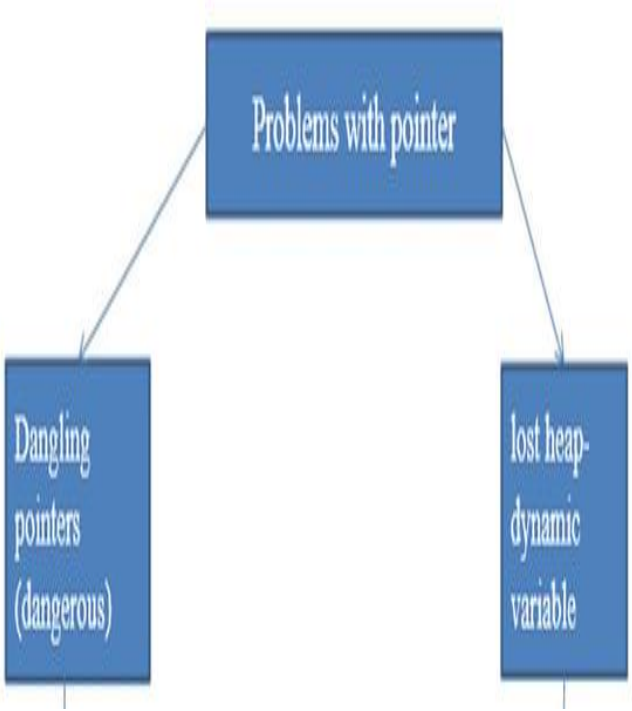
j = *ptr

sets j to the value located at ptr



The assignment operation

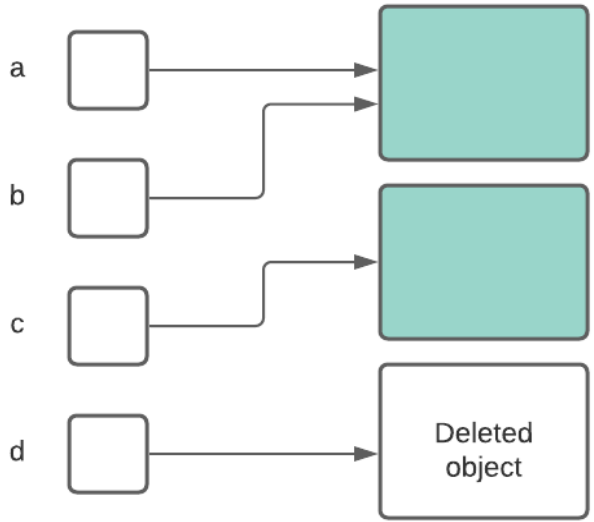
j = *ptr



Dangling Pointer

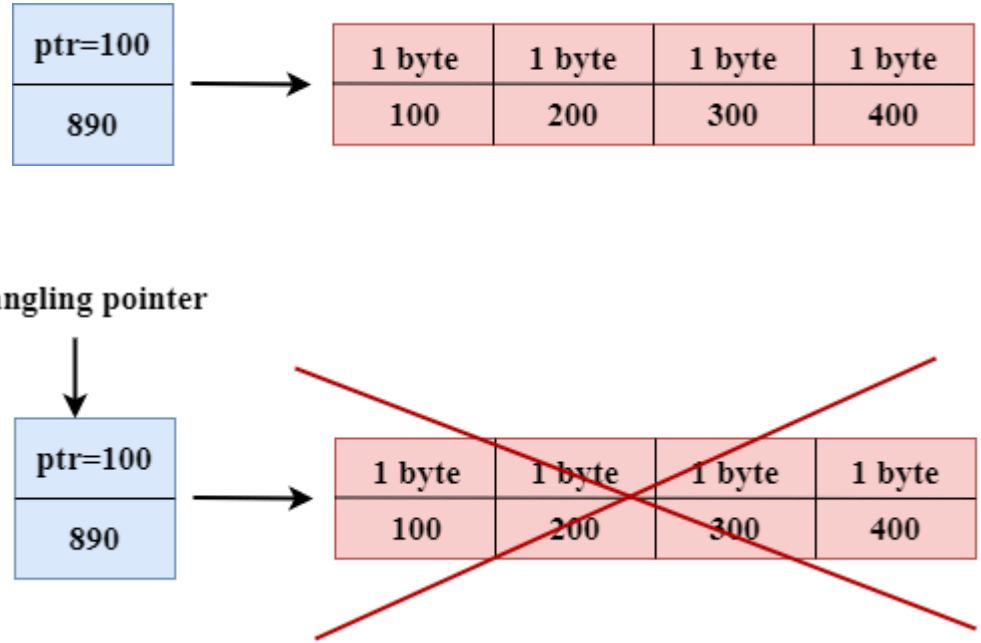
Dangling pointers (dangerous)

A pointer pointing to data that does not exist anymore is called a dangling pointer.



d is a dangling pointer.

```
#include <stdio.h>
int main()
{
   int *ptr=(int *)malloc(sizeof(int));
   int a=560;
   ptr=&a;
   free(ptr);
   return 0;
}
```



1st Solution to Dangling Pointer

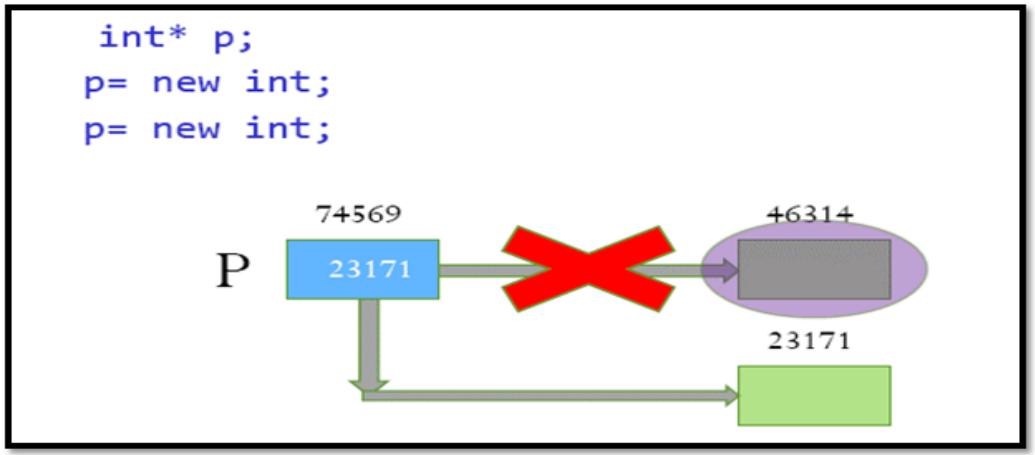
If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:

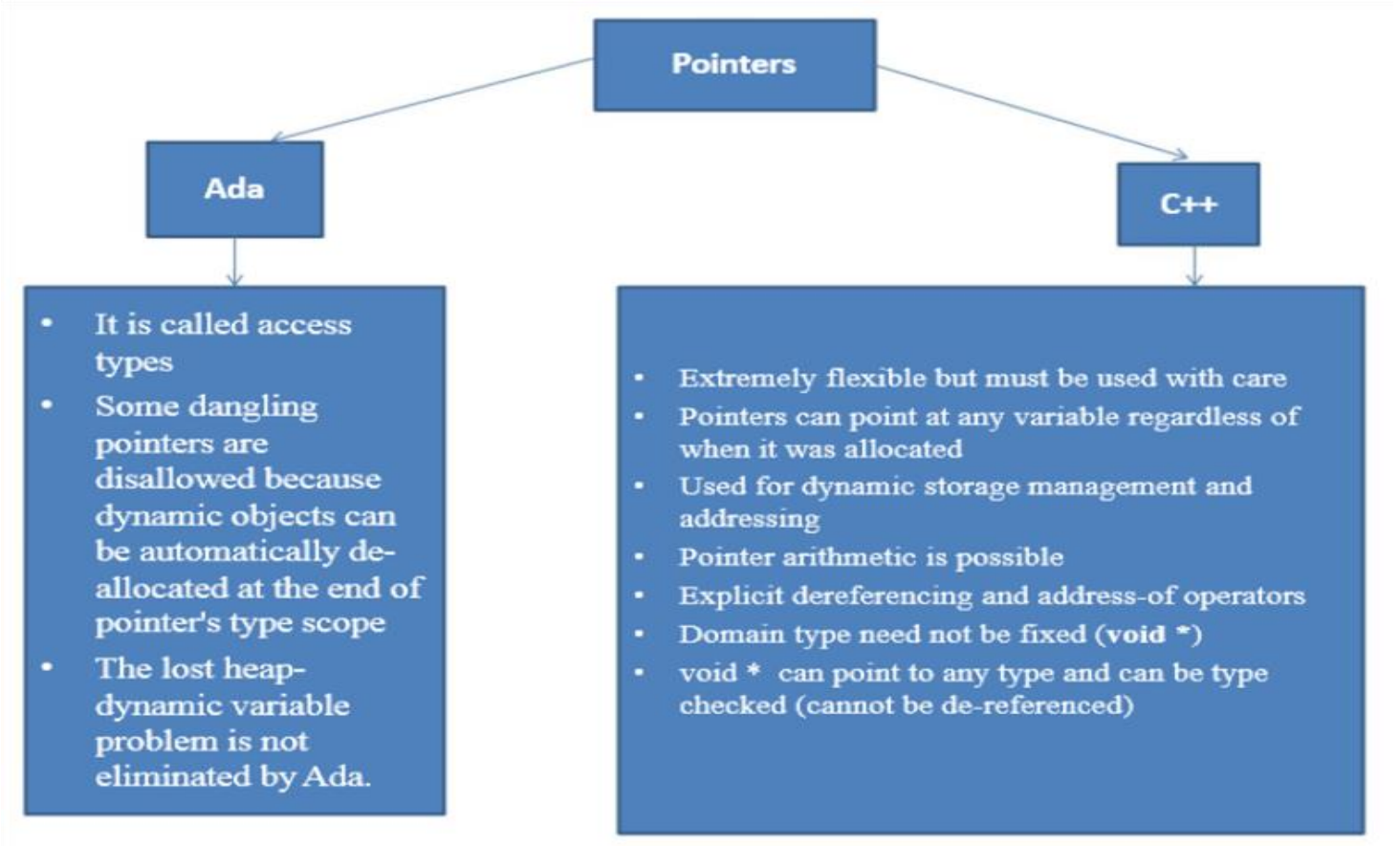
Lost Heap dynamic Variable

- An allocated heap-dynamic variable that **is no longer accessible to the user program** (often called garbage)
- Pointer **p1 is set to point to a newly created heap-dynamic** variable
- Pointer **p1 is later set to point to another newly created** heap-dynamic variable
- The process **of losing heap-dynamic variables is called memory leakage**

```
int * Ptr1 = new int(10);
Ptr1 = new int (20);
```

The first heap-dynamic variable is now inaccessible, or lost.




Pointers
Ada
C++
It is called access types
Some dangling pointers are disallowed because dynamic objects can be automatically de-allocated at the end of pointer's type scope
The lost heap-dynamic variable problem is not eliminated by Ada.
Extremely flexible but must be used with care
Pointers can point at any variable regardless of when it was allocated
Used for dynamic storage management and addressing
Pointer arithmetic is possible
Explicit dereferencing and address-of operators
Domain type need not be fixed (void *)
void * can point to any type and can be type checked (cannot be de-referenced)

Reference Type

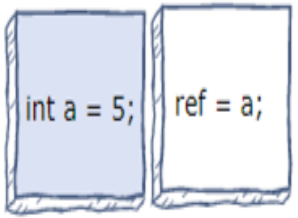
- **C++ includes** a special kind of pointer type called a r**eference type** that is used **primarily for formal parameters**
  - Advantages of **both pass-by-reference and pass-by-value**
- **Java extends C++'s reference variables and allows them to replace pointers entirely**
  - References are **references to objects, rather than being addresses**
- **C# includes both the references of Java and the pointers of C++**
- **Dangling pointers and dangling objects are problems as is heap management**
- Pointers **are like goto's--they widen the range of cells** that can be accessed by a variable
- Pointers or references are necessary for **dynamic data structures--so we can't design a language without them**

```
int result = 0;
int &ref_result = result;
. . .
ref_result = 100;
Cout<<result; //100
```

result and **ref_result** are **aliases**.

Pointer Vs Reference

The pointer variable **p** stores the address of the variable **a**; it "points" to the memory location of **a**.



Creating a reference to **a** just makes an alias for it; it does not "point" to **a** by storing its address in a separate memory location

When to use What : Pointer Vs Reference

Use references

- In function **parameters and return types.**

Use pointers:

- Use pointers **if pointer arithmetic or passing NULL-pointer is needed**.

  For example for arrays (Note that array access is implemented using pointer arithmetic).

- To implement **data structures like linked list, tree, etc and their algorithms because to point different cell, we have to use the concept of pointers.**


Any
questions


Thank you