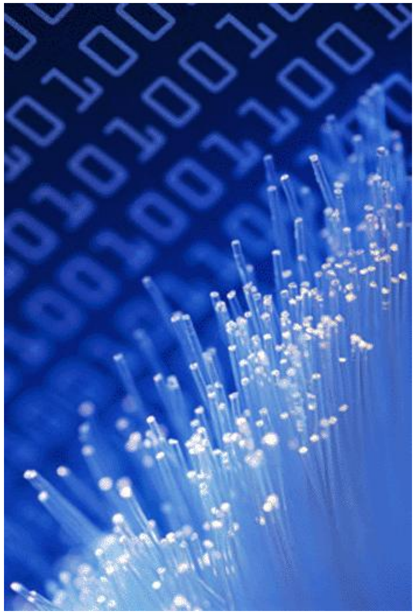
Chapter 5: Data Types

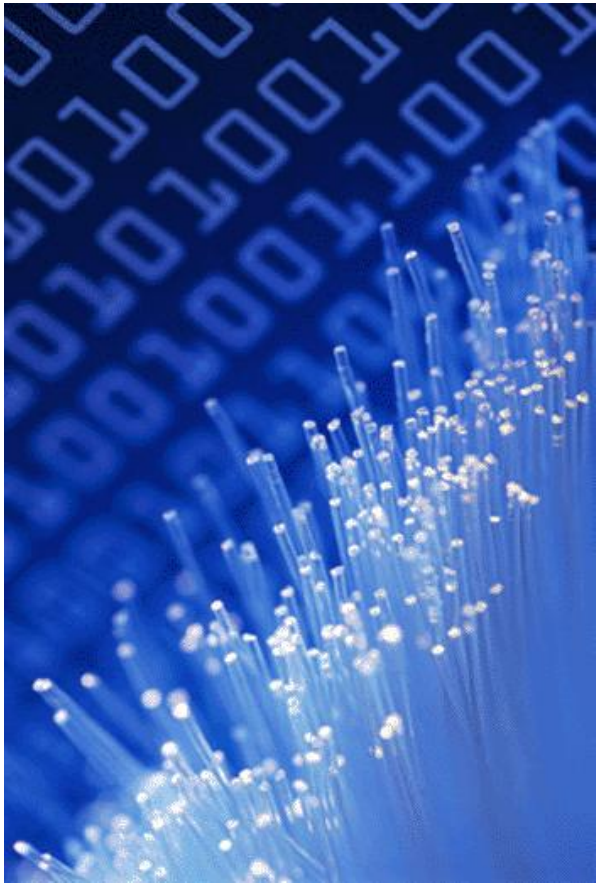
CMSC 124, 1st Semester, AY 2009-10



Discussion

Remember, we have 2 kinds of data types, which are:

- Primitive Data Types
- Composite Data Types

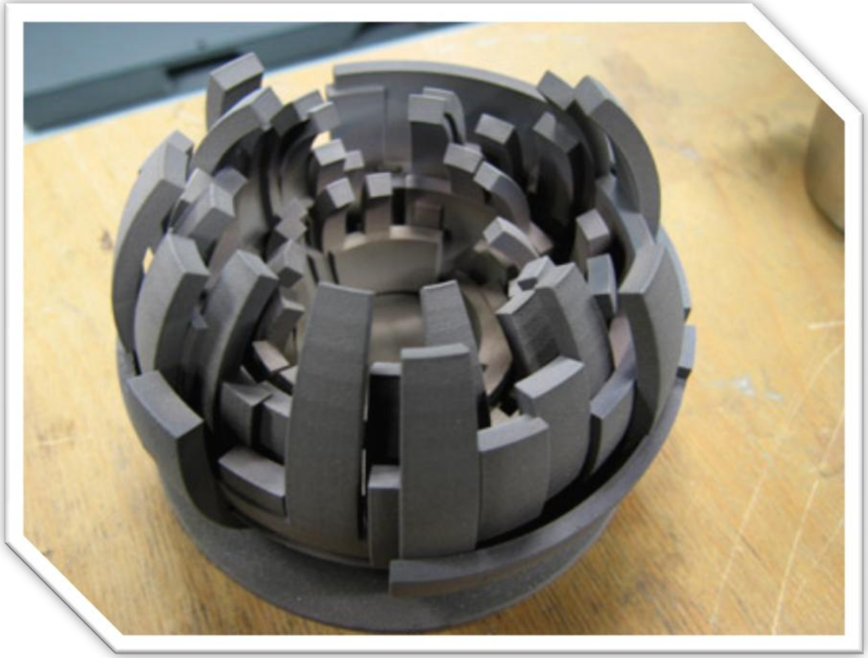


Chapter 5: Data Types

Data Objects

• Data objects represent container for data values.

• They are memory spaces where data values may be stored and later retrieved.



Two Types of Data Objects

1. **Programmer-Defined**
   - **Eg:** Variables, constants

2. **System-Defined**
   - **Eg:** Data objects maintained by the virtual computer.

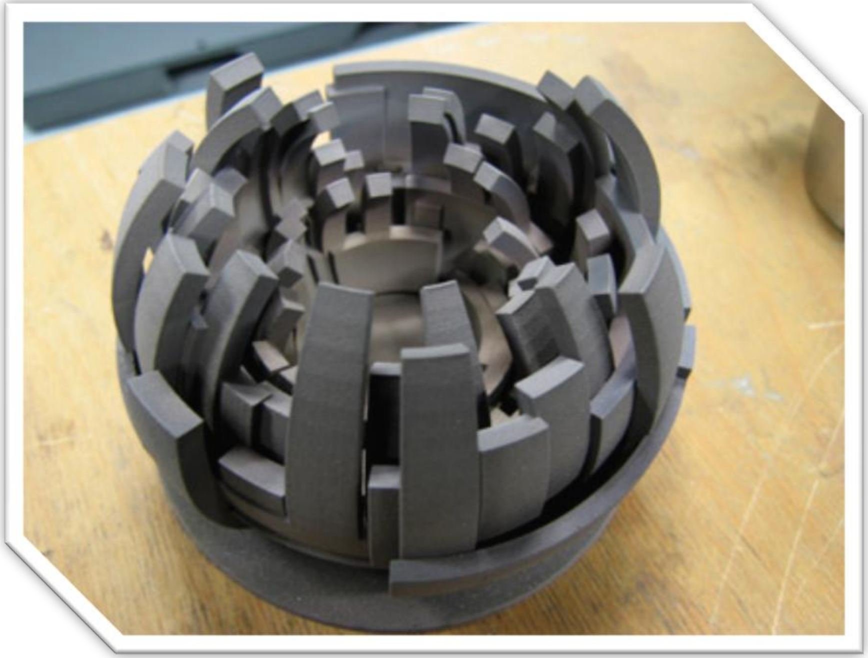
Chapter 5: Data Types

Attributes of Data Objects

1. **Value Type**
   - Types of values that the data object may contain.

2. **Size**
   - The amount of storage needed to store the values in the data object.

3. **Lifetime**
   - How long the data object exists.



Chapter 5: Data Types

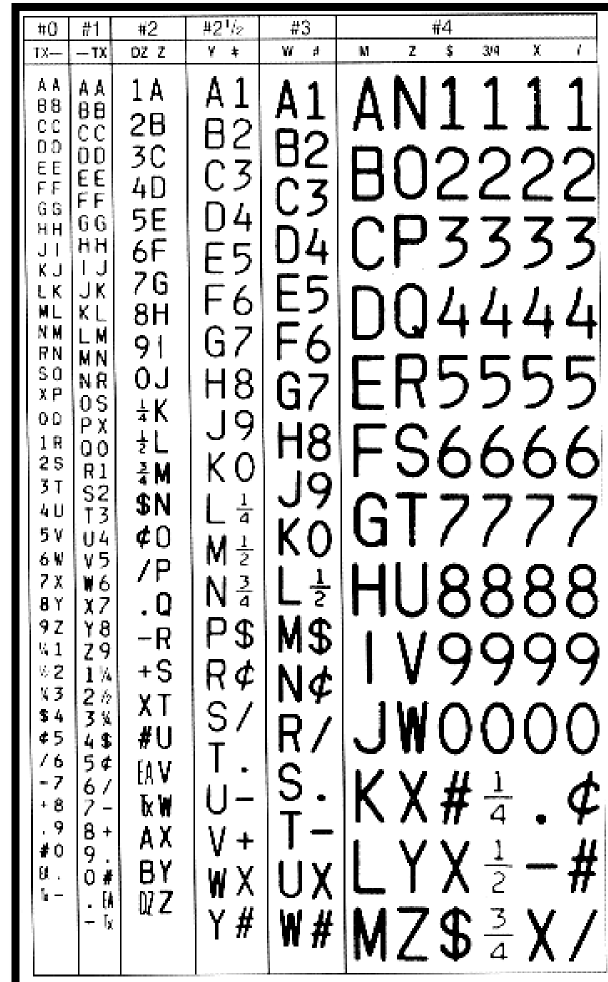
Data Values and Value Types

Value

- Anything that may be evaluated, stored, passed as a argument to a procedure, returned by a function, or may be a component of a data structure.

- The one that is placed on the data object.

- The values may be grouped into types.



Chapter 5: Data Types

Data Values and Value Types

Value Type

- A value type is just a set of values.

The Types of Values

1. Primitive Type

- One whose value is atomic and therefore cannot be decomposed.
- **Eg:** Characters, Integers, Booleans

2. Composite Type

- One whose values are composed or structured from simpler values.
- **Eg:** Records, Arrays, Sets, Strings*

3. Recursive Type

- It is defined in terms of itself.

Chapter 5: Data Types

Data Types

- Data types are class of data objects **PLUS** set of operations for creating and manipulating them.

- **Primitive data types** may be combined to form a composite data type.

- Certain PL's provide facilities for the programmer to define new composite data types.

Illustration on Data Objects

- **Data Object**
  A location in the computer memory with the name **A.**

A:

- **Data Value**
  A bit pattern used by the translator whenever the number 17 is used in the program.

10001

- **Bound Variable**
  Data object is bound to the value 17

A: **0000 0000 0001 0001**

Primitive Data Types

1. **Numeric Data Type**

- Includes integer and real number types.

- **In C:**

```
short int i
long int k
unsigned int u
float x
double y
```

- **In PL/I and ADA**, it allows the number of digits in the decimal representation to be specified.

```
DECLARE PAY FIXED DECIMAL
(7,2)
```

- **In COBOL:**

```
ROOT PICTURE 99999V99
```

Chapter 5a: Primitive Data Types

Primitive Data Types

2. Subrange Type

- This is introduced to save on storage and for better type checking.
- **Consider this:**
  A variable having a value in the subrange 1..100

  SOMETHING TO PONDER:
  - ✓ How do you save storage space?
  - ✓ How is type checking facilitated?

- **Eg:**
  age = 1..120, define as integer

  What can you say about?
  - ✓ `age = 130`
  - ✓ `age = 0`
  - ✓ `age = age - 1`

Chapter 5a: Primitive Data Types

Primitive Data Types

3. Enumeration Type

- It is common to have a variable that can take on one of a small set of values.

- **In Pascal: (Example)**

```
rank: (instructor,
assistant_professor,
associate_professor,
professor)

status: (single, married,
widowed)
```

- **In C:**

```
enum days {mon, tue,…sun}
week;
enum days week1, week2;
```

Chapter 5a: Primitive Data Types

Primitive Data Types

4. Boolean (or Truth Value) Type

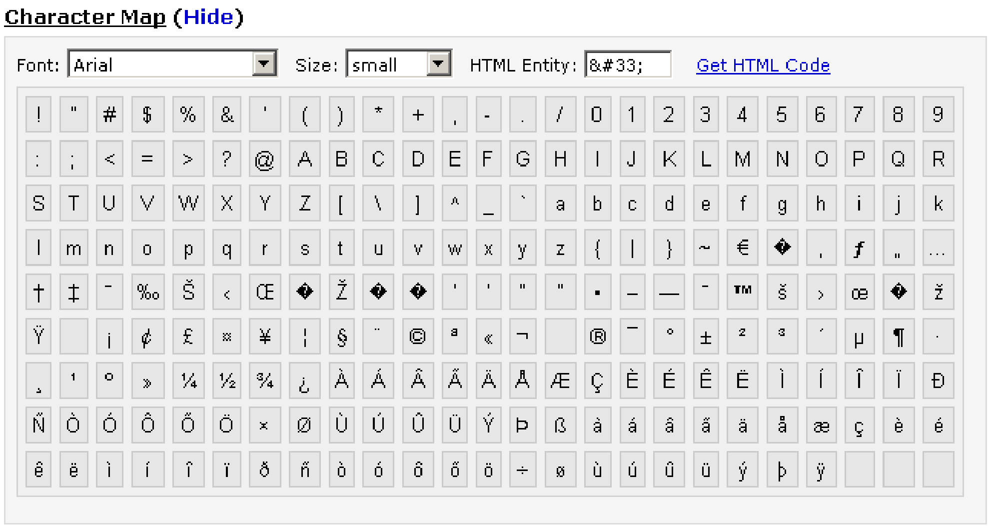
- A data type having one of the two possible values (true or false).

- **Eg:** Declaring a variable RESPONSE.

  ✓ **In PL/I:**
  ```
  LOGICAL RESPONSE
  ```
  ✓ **In Pascal:**
  ```
  RESPONSE: boolean;
  ```
  ✓ **In Java:**
  ```
  boolean RESPONSE;
  ```

5. Character Type

- Characters are stored as numeric coding, wherein the most popular scheme is ASCII.



6. String Type

- Natural extension of character types is the string type.

- Languages such as SNOBOL, PL/I, FORTRAN, BASIC allow a string to be manipulated as one unit.
  (Primitive Data Type)

- While others, like APL, Pascal, C, ADA, consider a string as a linear array of characters.
  (Composite Data Type)

- **Eg:**
  - ✓ **In PL/I:**
    ```
    DCL NAME CHAR(25)
    ```
  - ✓ **In Pascal:**
    ```
    name: array[1..25] of
    char;
    ```

SOMETHING TO PONDER:
What's the matter if a string is implemented as PDT or CDT?

Chapter 5a: Primitive Data Types

The Concept of Binding

Binding

- Association such as between an attribute and an entity, and an operation and a symbol.
- The time when binding takes place is called the **binding time**.
- **Eg:**
  - Variable to type
  - Variable to value
  - Symbol to operation.

Classes of Binding

1. **Execution Time (Runtime)**

a. **On entry to a procedure or block.**
   - Formal to actual parameters.
   - Formal parameters to particular storage locations.

b. **At arbitrary points during execution.**
   - Variables to values.

Classes of Binding

2. **Translation Time (Compile Time)**

a. **Chosen by the programmer.**
   - Variable types, names.
   - Statement structures.

b. **Chosen by the translator.**
   - Relative location of a data object.
   - How arrays are stored.

3. **Language Design (Definition) Time**

- Bindings set by the designer of the programmer.
- Program structures, possible statement forms, data structure types.
- **Eg:** Possible types of a variable in every programming language.

Chapter 5a: Primitive Data Types

Classes of Binding

4. Language Implementation Time

- Brought about by the differences in hardware where PL's are implemented.
- Possible range of values may be dictated by the PL.
- Details associated with the representation of numbers and arithmetic operations.

Chapter 5a: Primitive Data Types

Illustration on Binding

```
int count;
…
count = count + 926;
```

Considerations

- Set of possible type for **`count`**
- Type of **`count`**
- Set of possible values of **`count`**
- Value of **`count`**
- Set of possible meanings for the operator symbol **+**
- Meaning of the operator symbol **+**
- Internal representation of the literal **`926`**

Binding Types

Before a variable can be referenced in a program, it must be bound to a data type.

1. **Static Binding**
- Types are specified through declarations.
- **Explicit Declaration**
  In C: int j;
- **Implicit Declaration**
  In FORTRAN, variables with names that start with I,J,K,L,M,N are said to be integer types.

2. **Dynamic Binding**
- The type is not specified by the declaration statement.
- Variable is bound to a type when it is assigned a value in an assignment statement.
- **Eg:**
  $grade = "mataas"
  $grade = 100;

Chapter 5a: Primitive Data Types

Declaration

Declaration is the part of the program where the programmer communicates to the language translator information on the numbers and types of data objects needed during program execution.

Purposes of a Declaration

1. Choice of storage representation
2. Storage management
3. Generic operations
4. Type checking

Type Checking

- Type checking is done to ensure that an operation is provided with the correct types.
- Consider this: **A:=B+C;**
  - The types for operands B and C must be those allowed for the operands of addition.
- It is done almost everywhere in a program.

Type Checking

1. **Static Type Checking**
- Type checking is done during compilation.
- A lot of information is needed in order to do checks at compile time.
  - This is the reason why most data objects are declared.

2. **Dynamic Type Checking**
- Type checking is done during execution time.
- Performed immediately before the execution of a particular operation.
- Usually slows down the execution of the program.

Type Checking

Type checking is dependent on whether the language is:

1. **Statically Typed**
   - Every variable and parameter has a fixed type that is chosen by the programmer.
   - The type of each expression can be deduced and each operation can be type-checked at compile time.

2. **Dynamically Typed**
   - Only the values have fixed types.
   - A variable may have no designated type and may take on values of different types at different stages of execution.

Chapter 5a: Primitive Data Types

Strong Typing

A PL is **strongly typed** if type errors are always detected whether at compile time or at run time.

NOT STRONGLY TYPED

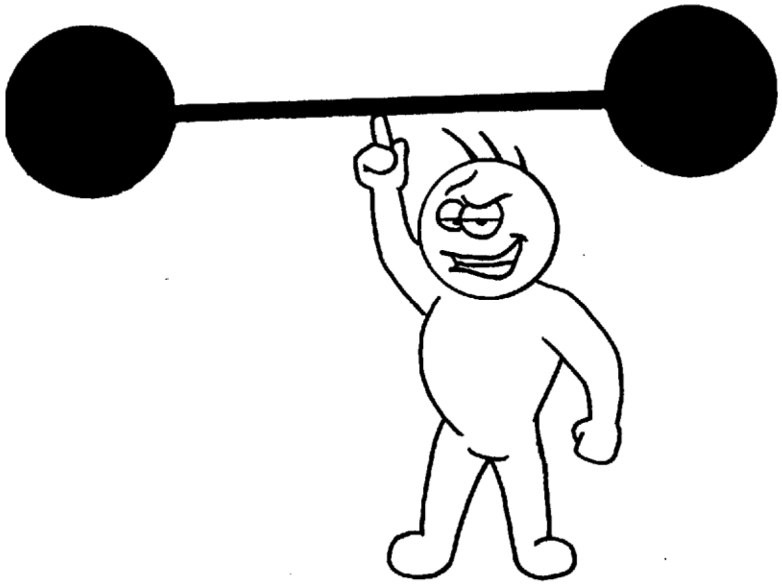
- FORTRAN-77
- MODULA 2

NEARLY STRONGLY TYPED

- ADA
- Pascal

PURE STRONGLY TYPED

- ML
- Miranda



Chapter 5a: Primitive Data Types

Type Equivalence

- When two data objects are involved in one operation, this issue arises.
- When two data objects are identical in types, then there is no problem.
- But, we often encounter operations involving non-identical data objects but closely related.
- **"Are they compatible?"**

Type Equivalence

1. **Structural Equivalence**

- They have the same logical structure.
- However, it is not always easy to decide the logical equivalence of 2 data types.

Pascal Declaration

```
type rangetype1 = 1..120;
type rangetype2 = 1..120;
var  age1:rangetype1;
var  age2:rangetype2;
```

Assignment Statement

```
age1 := age2;
- LEGAL
```

Type Equivalence

2. Named Equivalence

- Look at the example. ☺

ADA Declaration

```
TYPE meter IS NEW integer;
TYPE yard IS NEW integer;
m, n: meter;
y: yard;
```

Assignment Statements

```
m := n;
- LEGAL
y := m;
- ILLEGAL
```

Chapter 5a: Primitive Data Types

Type Conversion and Coercion

- When the operands of an operation have types that are not exactly the same, either a type mismatch error occurs or some type conversion has to be done.
- Type conversion is an operation that converts a data object of one type and produces the corresponding data object in another type.
- It has two types:

1. **Narrowing Conversion**
   - Converts the object type with a certain storage reqm’t to the one with lesser storage reqm’t.
   - **Eg:** real -> int

2. **Widening Conversion**
   - Storage reqm’t of the original type is lesser than the converted type.
   - **Eg:** int -> real

Type Conversion and Coercion

- Type conversion may be done implicitly (coercion) or explicitly.
- **Coercion**
  - Way by which a data object of a certain type is changed to the correct type.
  - Usually carried out by compiler or virtual computer.

- **Eg:** Consider the following Pascal expression.

```
x := n * 26;
```

✓ Assume x is real and n is integer.
✓ The type n is coerced to be real.
✓ This illustrates implicit integer-to-real conversion in Pascal.

Chapter 5a: Primitive Data Types

Initialization

- Uninitialized variables may contain "garbage" and when used, may cause errors.

- Most PLs provide for initialization immediately after declaration (allocation of storage).

Composite Data Type

- Data type whose values are composed or structured from simpler values.

MATHEMATICAL CONCEPT	COMPOSITE DATA TYPE
Cartesian Product	Records, Tuples
Disjoint Unions	Variants, Unions
Mappings	Arrays, Functions
Power Sets	Sets
Recursive Types	Dynamic Data Structure

Cartesian Product

- The **Cartesian product** of two sets **S** and **T**, denoted by **S x T**.
  - *"The set of all ordered pairs, such that the first value of the pair is chosen from the set S and the second value from the other set T."*

- **In General:**
  - The Cartesian product $\mathbf{S_1 \times S_2 \times \ldots \times S_n}$ stands for the set of n-tuples, such that the first component of the n-tuple comes from $\mathbf{S_1}$, the second $\mathbf{S_2}$, so on…

Chapter 5b: Composite Data Types

Cartesian Product

- **Eg: Pascal**

```
type date = record
     d: 1..31;
     m: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep,
     oct, Nov, Dec);
      y: 1900..3000
end
```

- **How do we define date using Cartesian product notation?**
  - ✓ `date = {1, 2,… 31} x {Jan, Feb, … Dec} x {1900, 1901, … 3000}`

Cartesian Product

- **Another Eg: C**

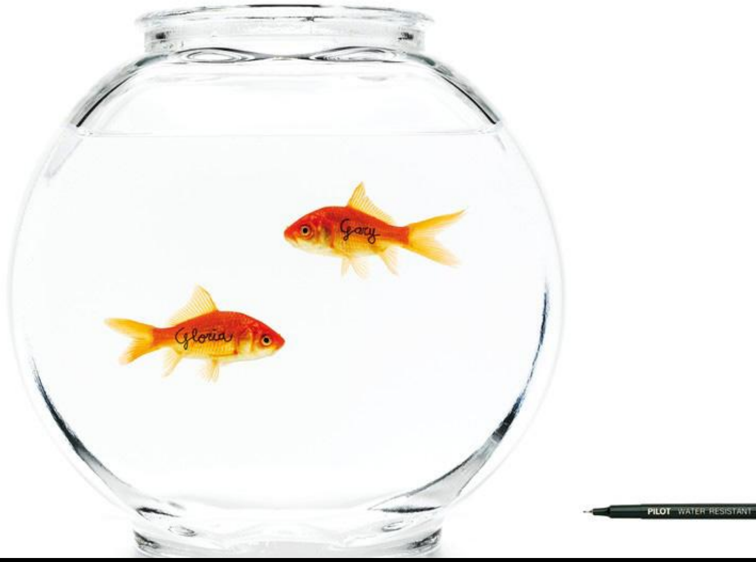
```
struct date {
    int day;
    int month;
    int year;
    char dow;
} today
```

- **How do we define date using Cartesian product notation?**
  - ✓ `date = int x int x int x char`

Disjoint Unions

- The **disjoint unions** of two sets **S** and **T**, denoted by **S + T**.
  - *"The set of values in which each value is chosen from either set S or set T."*

- **In General:**
  - $S_1 + S_2 + \ldots + S_n$ stands for the set in which each value is chosen from one of $S_1, S_2, \ldots,$ or S_n.



Disjoint Unions

- **Eg: Variant records in Pascal**

```
type paytype = (salaried, hourly);
type employeetype = record
   id:   integer;
   dept: char;
   age:  integer;
    case payclass: paytype of
            salaried: (monthlyrate: real; startdate:
                       integer);
            hourly:   (rateperhour: real; regularhours:
                       integer;overtime: integer);
    end
end
```

- **How do we define employeetype using disjoint unions notation?**
  - ✓ `employeetype = integer x char x integer x` **`( (real x integer) + (real x integer x integer) )`**

Disjoint Unions

- **Eg: Unions in C**
  - ✓ A **union** is a variable which may hold (at diff. times) objects of different sizes and types

```
union number {
    short shortnumber;
    long longnumber;
    double floatnumber;
} anumber;
```

  - ✓ Defines a union called **`number`** and an instance **`anumber`**
  - ✓ To access: **`anumber.longnumber`**

Chapter 5b: Composite Data Types

Mappings

- A mapping **f** maps every value x in set S to a value y in set T: **f:S->T**
- **Examples:**
  - ❖ **Array**
    - ✓ mapping: (index set) -> (component set)
  - ❖ **Function Abstraction**
    - ✓ mapping: (parameter) -> (returned value)

Mappings

- **Eg: Pascal**
  `amikind = array[1..161] of boolean`

- **How do we define amikind using mappings notation?**
  `amikind = {1, 2, 3, …, 161} -> {true, false}`

Mappings: More Examples

```
function odd (n: integer):boolean;
 begin
     odd := (n mod 2=1)
 end;
```

- **Set S:** integer
- **Set T:** boolean
- **Notation:** `odd = integer -> boolean`

```
function gcd (m, n: integer):integer;
 begin
     if n = 0 then gcd := m
    else gcd := gcd (n, m mod n)
 end;
```

- **Set S:** integer x integer
- **Set T:** integer
- **Notation:** `gcd = integer x integer -> integer`

Powersets

- The set of all subsets of **S** is called the **powerset of S**.
- **Operations:** (defined in set theory)
  - Membership test
  - Inclusion test
  - Union
  - Intersection
- Pascal and ML support set type.
- **Eg: Pascal**

```
type color = red, yellow, blue;
mix = set of color;
```

// mix = powerset of color

Recursive Types

- A **recursive type** is one whose values are composed from values of the same type.
  - ❖ Defined in terms of itself.

- **In General:**
  - ❖ The set of values of a recursive type **T**, will be defined by a recursive equation of the form **T = … T …**

Recursive Types

- **Eg: List Type in ML**

  Notation:

  ```
  Integer-list = Unit + (Integer x Integer-list)
  ```

  In ML:

  ```
  datatype intlist = nil | cons of int * intlist
  ```

  Sample Values:

  ```
  nil
  cons(11, nil)
  cons(1, cons(2, cons(3, cons (4, nil))))
  ```

Chapter 5b: Composite Data Types

QUIZ Number N(?) – Get 1/4

- Show the set of values of each of the following Pascal types using **formal notations**.

```
type cmscsubj = (CMSC 124, CMSC 150, CMSC 100, CMSC 128)
type numgrade = (1.0, 1.25, 1.5, …., 3.0, 5.0)

type studrec = record
        subject: cmscsubj;
        enjoyed: boolean;
        grade: numgrade;
end;

sreinrecord = array[1..161] of studrec
```

Chapter 5b: Composite Data Types

QUIZ Number N(?) – Get ¼ - Answer

Formal Set Notation

cmscsubj = {CMSC 124, CMSC 150, CMSC 100, CMSC 128}
numgrade = {1.0, 1.25, 1.5, …., 3.0, 5.0}

studrec = {CMSC 124, CMSC 150, CMSC 100, CMSC 128} x boolean x {1.0, 1.25, 1.5, …., 3.0, 5.0}

OR

{CMSC 124, CMSC 150, CMSC 100, CMSC 128} x {true, false} x {1.0, 1.25, 1.5, …., 3.0, 5.0}

sreinrecord = {1, 2, … 161} -> {CMSC 124, CMSC 150, CMSC 100, CMSC 128} x {true, false} x {1.0, 1.25, 1.5, …., 3.0, 5.0}

Specification of Composite Data Types

The Properties

1. **Number of Components**
   - **Consider:** Fixed size or variable size?
   - **Fixed size** signifies invariant number of components.
     - ✓ **Eg:** Pascal arrays and records
   - **Variable size** signifies dynamically changing number of components.
     - ✓ **Eg:** ML's list types, Java's array list

The Properties

2. **Type of Each Component**
   - **Consider:** Homogenous or heterogenous?
   - **Homogenous** requires all components are of the same type.
     - ✓ **Eg:** Arrays, Strings*
   - **Heterogeneous** allows components of different types.
     - ✓ **Eg:** Records, Variant records

Specification of Composite Data Types

The Properties

3. **Name to be Used in Selecting the Component**
   - Dependent on the syntax adopted by the language.

4. **Maximum Number of Components**
   - Dictated whether dynamic or static?
   - For **dynamic structures**: Usually no limitation on size.
   - For **static structures**: Programmer usually sets the size.

5. **Organization of Components**
   - Dependent on the type of each component.

Operations on Composite Data Types

1. **Selection Operation**
   - Operation to access components of the composite structure.
   - **Issue:** Random versus Sequential?

2. **Whole Structure Operations**
   - Take the whole structure as argument.
   - **Eg:** Equality test of strings, Union, Intersection of sets

3. **Insertion/Deletion of Components**
   - Actually relevant only to dynamic composite structures.
   - **Ponder:** How about static composite structures?

4. **Creation/Destruction of Data Objects**
   - Supported by most PL's
   - Again, dependent whether structure is static or dynamic.

Chapter 5b: Composite Data Types

Review: Given these Pascal types, use formal set notations to show the set of values of each.

```
{define a gendertype and civiltype}
type gendertype = (male, female);
type civiltype = (single, married);

{define another type called person, which is a record
   containing ...}
type person = record
   name: array[1..3] of string;
   gender: gendertype;
   age: integer;

   case civilstat: civiltype of
      single: (num_fiance: integer);
      married: (spouse: string;
                num_offspring: integer);
end;
```

Chapter 5b: Composite Data Types

Review: Given these Pascal types, use formal set notations to show the set of values of each.

Answers:

gendertype = {male, female}
civiltype = {single, married}

$S_{1\ (name)}$ = {1,2,3} -> string
$S_{2\ (gender)}$ = {male, female}
$S_{3\ (age)}$ = integer
$S_{4\ (civilstat)}$ = integer + (string x integer)

person = ({1,2,3}->string) x {male, female} x integer x (integer + (string x integer))

Chapter 5b: Composite Data Types

Review: Given these Pascal types, use formal set notations to show the set of values of each.

```
{function used in checking if a person is a zombie ☺}
function is_a_zombie(p: person, x:integer):
   boolean;
begin
     if p.age > 70 then
          is_a_zombie := true
     else
          is_a_zombie := false;
end;
```

Chapter 5b: Composite Data Types

Review: Given these Pascal types, use formal set notations to show the set of values of each.

Answers:

S = person x integer
= ({1,2,3}->string) x {male, female} x integer x (integer + (string x integer)) x integer

T = boolean or {true, false}

is_a_zombie = (({1,2,3}->string) x {male, female} x integer x (integer + (string x integer)) x integer) -> {true, false}

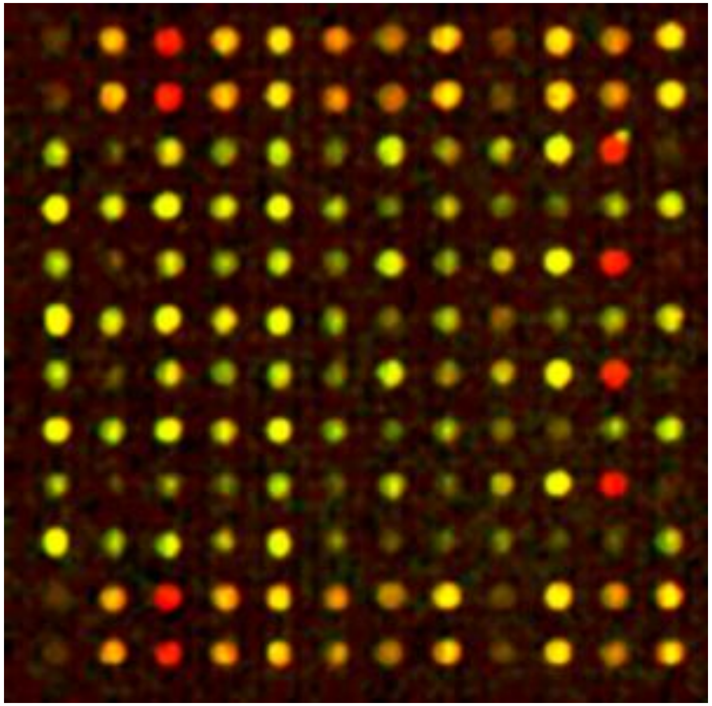
Implementation: Arrays

An **array** is an ordered sequence of identical objects.

✓ **Static arrays**
Index set is fixed at compile time.

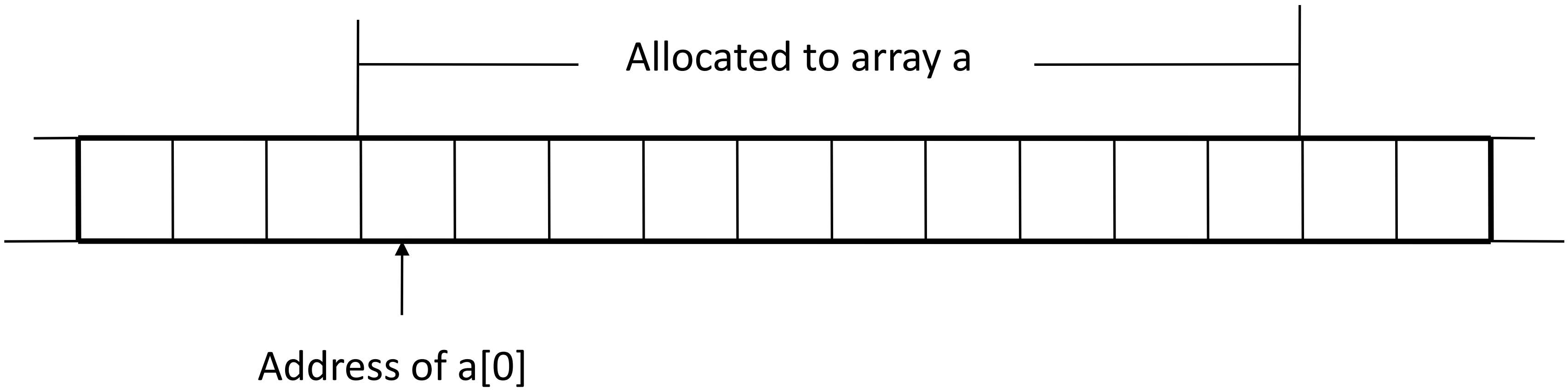
✓ **Dynamic arrays**
Index set is fixed on creation of the array during execution.

✓ **Flexible arrays**
Index set is not fixed at all.



One Dimensional Arrays

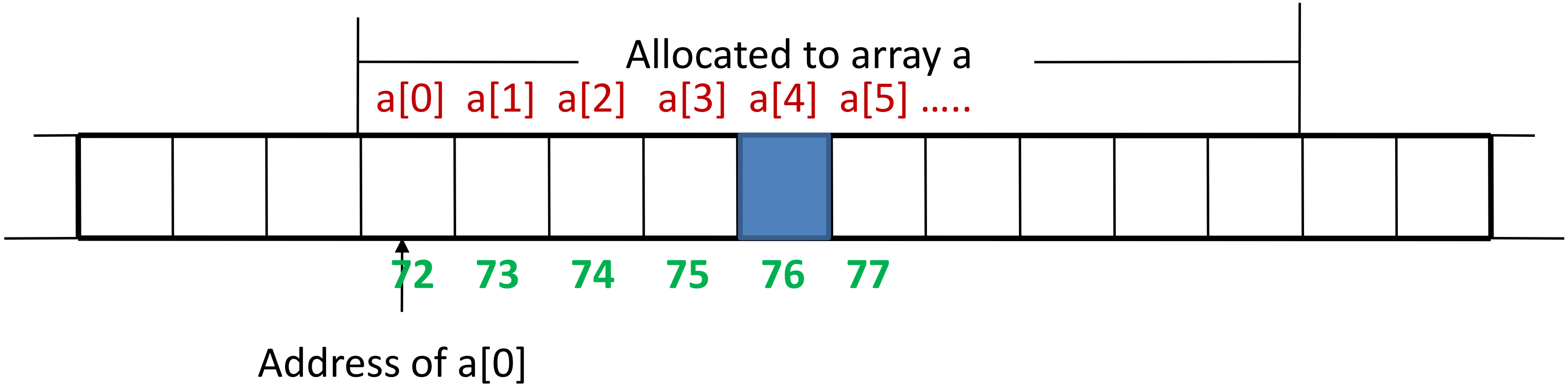
- **Storage Representation**



- **Accessing Elements (a[i])**
  1. Load address of **a[0]** to register **r**
  2. Add i to register **r**
  3. Get the content of memory whose address is in **r**

One Dimensional Arrays (Plain)

- **Eg: Access a[4]**

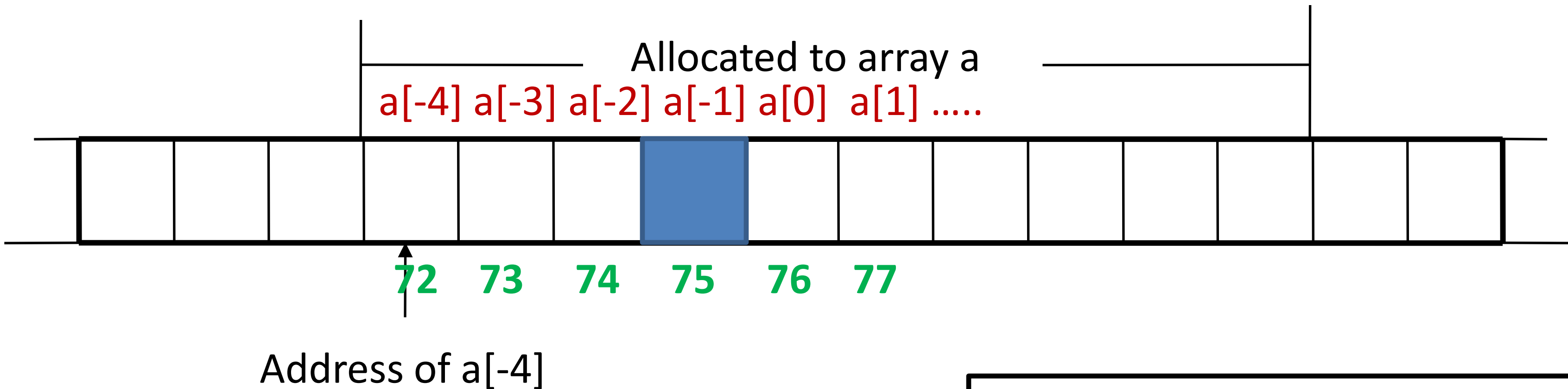


- **Accessing Elements (a[i])**
  1. Load address of **a[0]** to register **r.**
  2. Add i to register **r.**
  3. Get the content of memory whose address is in **r.**

```
i = 4

1. r = 72
2. r = 72 + 4 = 76
3. get_content(at 76)
```

One Dimensional Arrays (without Subscript Checking)

- **Eg: Access a[-1]**

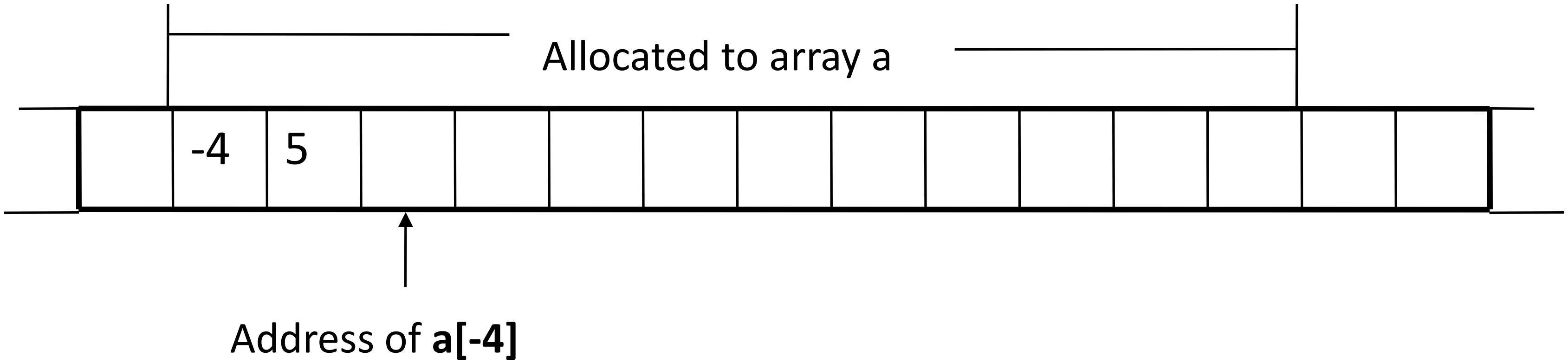


- **Accessing Elements (a[i])**
  1. Load address of **a[-4]** to register **r**
  2. Add i to register **r**
  3. Subtract **-4** from register **r**
  4. Get the content of memory whose address is in **r**

```
i = -1

1. r = 72
2. r = 72 + -1 = 71
3. r = 71 - (-4) = 75
4. get_content(at 75)
```

One Dimensional Arrays (with Subscript Checking)

➢ **Storage Representation**

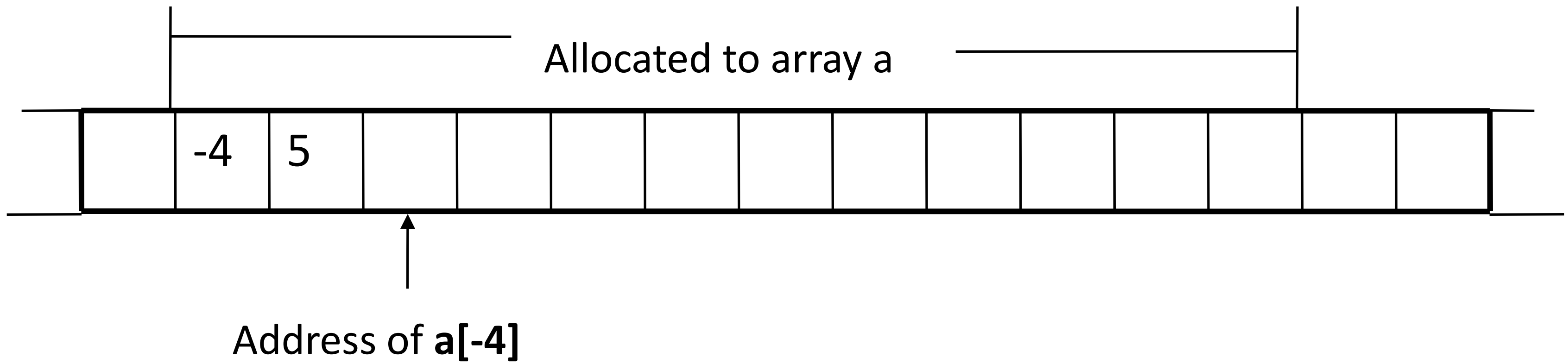


• The lower and upper bounds of the indices of the array should be made available during runtime.

• One way is to store these bound in the memory, together with the components of the array.

• Above example shows **a:array[-4..5] of integer**.

One Dimensional Arrays (with Subscript Checking)



- **Accessing Elements** (element a[i])
  1. Load address of **a[-4]** to register **r1**
  2. Load i to register **r2**
  3. Is **r2** < the content of **r1-2** then error subscript out of range
  4. Is **r2** > the content of **r1-1** then error subscript out of range
  5. Add **r2** to **r1**
  6. Subtract **-4** from register **r1**

Two Ways

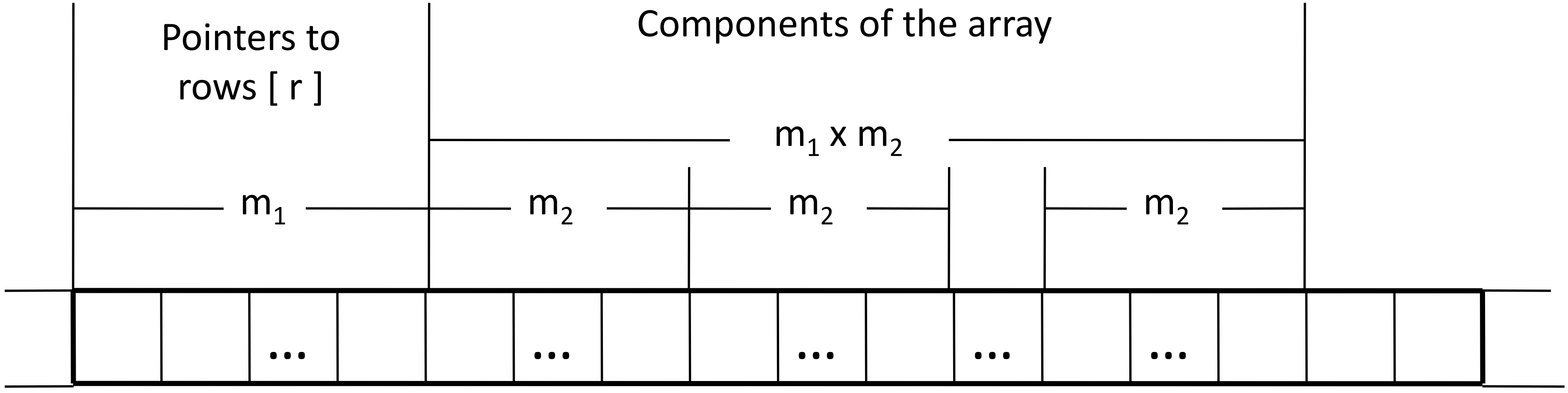
- Indirect Access via Pre-Calculated Vectors of Addresses
- Multiplicative Subscript Calculation

Indirect Access via Pre-Calculated Vectors of Addresses

Two Dimensional Arrays

- **Storage Allocation**

Array m_1 x m_2 will require $m_1 * m_2$ storage locations for the components and m_1 storage locations for the pointers.



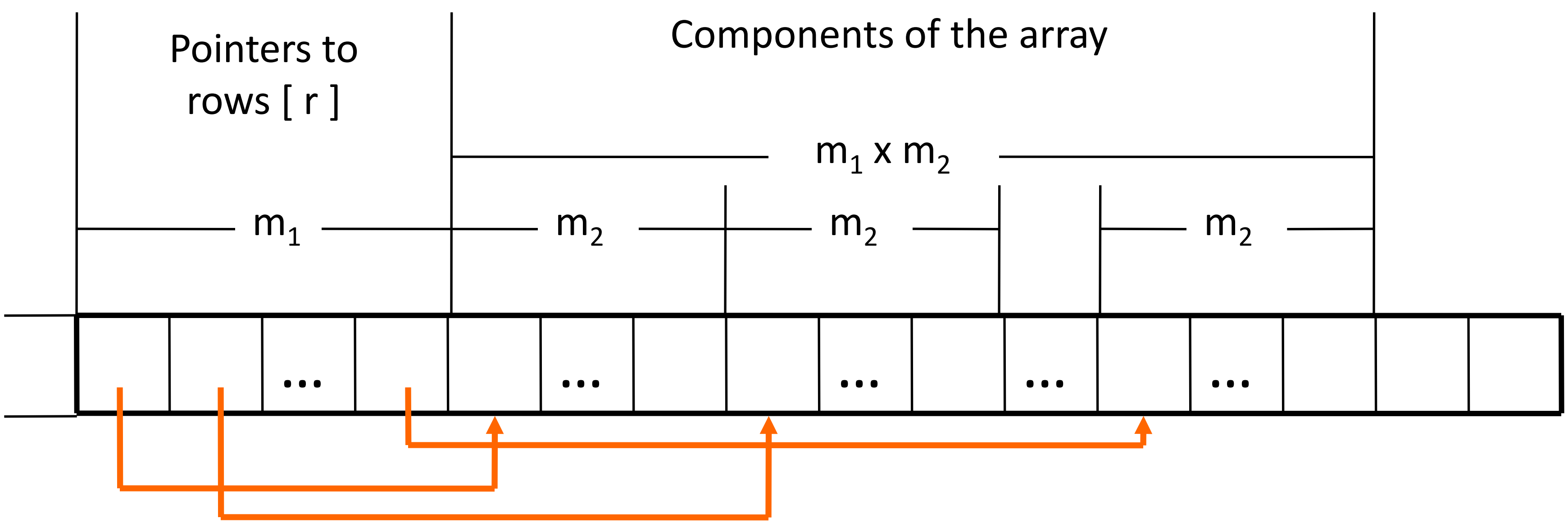
Indirect Access: Two Dimensional Arrays

- **Pointer Initialization**

  Let **b** – address of the first storage location of the array

  $\mathbf{v_1}$ – address of the first storage location of the first row

for i:=0 to m_1-1 do r[i]:= v_1 + m_2*i;



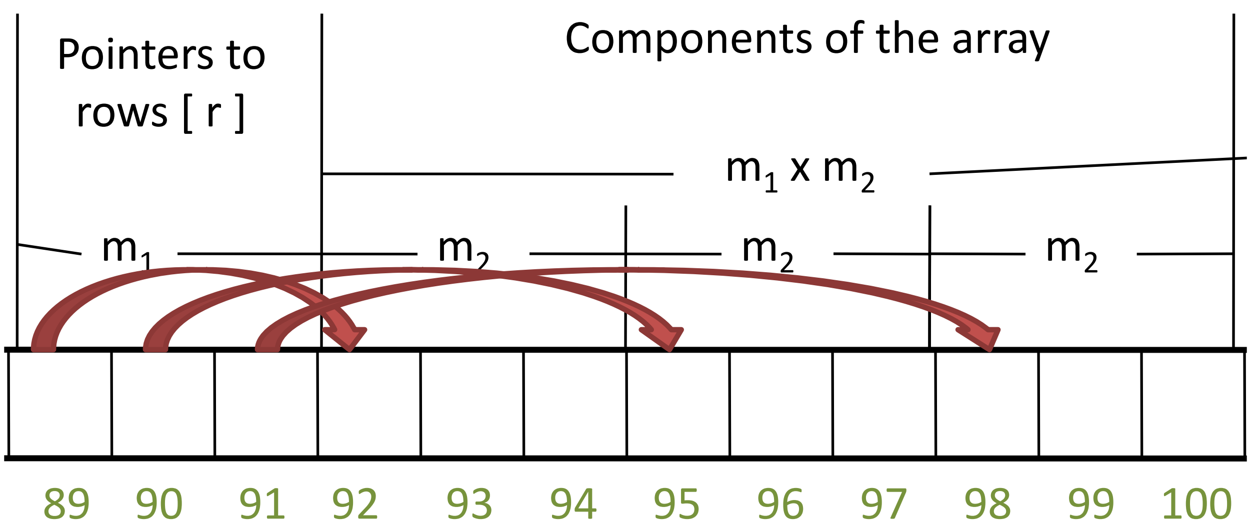
Indirect Access: Two Dimensional Arrays

- **Eg: Initialize 3 x 3 array**

  Let **b** – address of the first storage location of the array

  $\mathbf{v_1}$ – address of the first storage location of the first row

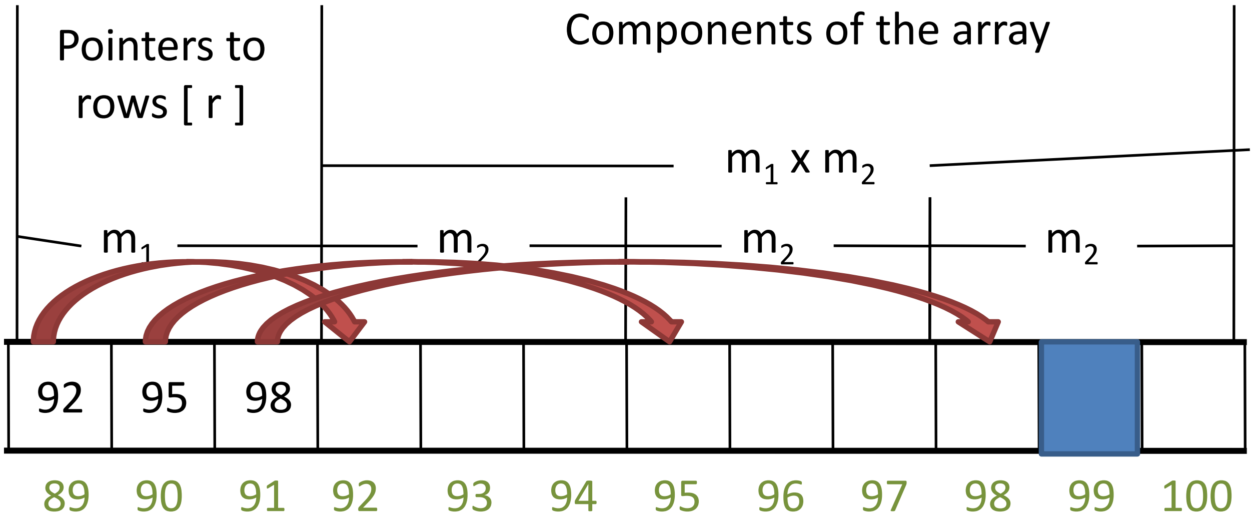
```
for i:=0 to m1-1 do r[i]:= v1 + m2*i;
```




```
b  = 89
v1 = 92
m1 = 3
m2 = 3

r[0] = 92+3*0
     = 92

r[1] = 92+3*1
     = 95
```

Indirect Access: Two Dimensional Arrays

- **Example: Access a[2,1]**




```
b = 89
i = 2
j = 1

1. r = b = 89
2. r = 89+2
     = 91
3. r =
   get_content
   (at 91)
     = 98
4. r = 98+1
5. get_content
   (at 99)
```

- **Accessing Elements (element a[i, j])**
  1. Load address of the first memory allocated to the array to register **r** --- **b**
  2. Add i to register **r**
  3. Get the content of memory address is in **r** and store it to **r**
  4. Add **j** to **r**
  5. Get the content of the memory whose address is in **r**

Indirect Access via Pre-Calculated Vectors of Addresses

N - Dimensional Arrays

- Consider an n-dimensional array

$$\mathbf{a[\ lo_1..hi_1,\ lo_2..hi_2,\ \ldots,\ lo_n..hi_n]}$$

where

$\mathbf{m_i = hi_i - lo_i + 1,\ 1 \le i \le n}$

$\mathbf{lo_i = 0,\ 1 \le i \le n}$

b = address of the first storage location allocated for array **a**

N - Dimensional Arrays

- **Storage Allocation**

SPACES FOR THE VECTOR OF ADDRESSES	SPACES FOR THE COMPONENTS OF ARRAYS
m_1 $m_1 * m_2$ $m_1 * m_2 * m_3$ .... $m_1 * m_2 * m_3 * \ldots m_{n-1}$	$m_1 * m_2 * m_3 * \ldots m_n$

TRY THIS!

Compute total number of spaces for a 3 x 3 x 3 array.

- The total number of cells needed for an n-dimensional array is

$$\mathbf{m_1 + (m_1 * m_2) + (m_1 * m_2 * m_3) + \ldots + (m_1 * m_2 * m_3 * .. * m_n)}$$

Indirect Access: N-Dimensional Arrays

- **Pointer Initialization**

$v_1 = b + m_1$

$v_2 = b + m_1 + (m_1 * m_2)$

$v_3 = b + m_1 + (m_1 * m_2) + (m_1 * m_2 * m_3)$

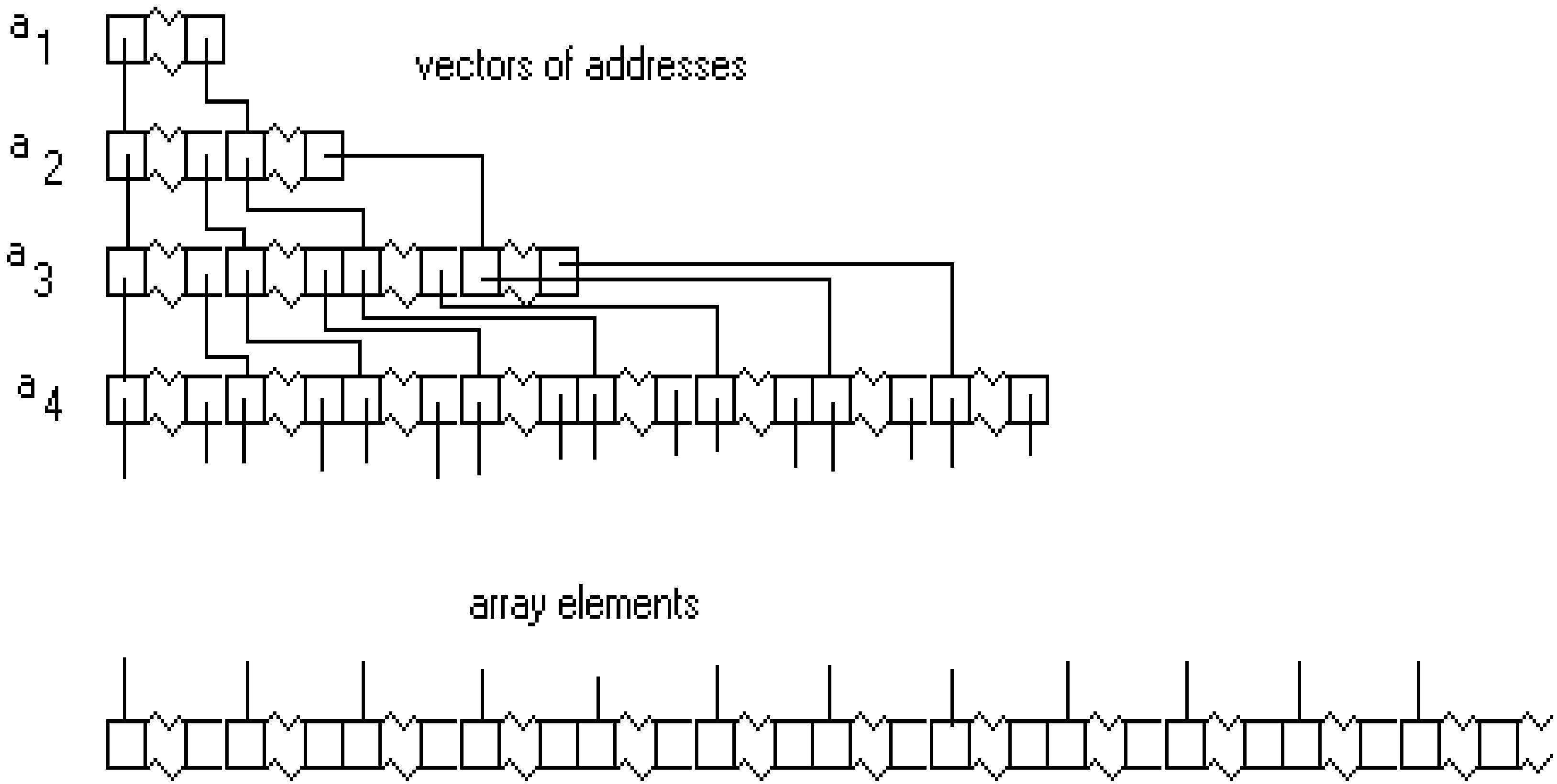
…

$v_{n-1} = b + m_1 + (m_1 * m_2) + (m_1 * m_2 * m_3) + (m_1 * m_2 * m_3 * .. * m_{n-1})$

```
for i:=0 to m1-1 do r1[i] := v1 + m2 * i;
for i:=0 to m1*m2-1 do r2[i]:= v2 + m3 * i;
for i:=0 to m1*m2*m3-1 do r3[i]:= v3 + m4 * i;
…
for i:=0 to m1*m2*m3*…*mn do rn-1[i]:= vn-1 + mn * i;
```

Chapter 5b: Composite Data Types

Indirect Access: N-Dimensional Arrays



Indirect Access: N-Dimensional Arrays

- **Accessing Elements**

  For simplicity, we will access an element in a 3-dimensional array which is **a[i, j, k].**

  1. Get the first memory allocated to the arrays –- **b**
  2. Go to storage location **b+i**
  3. Get the contents of **b+i** and add **j** to this
  4. Go to the resulting address and get its content and add **k**
  5. Get the content of this last address. The content of this is the value of **a[i, j, k]**

Chapter 5b: Composite Data Types

Indirect Access: Analysis

✓ Fast component access

✓ Large memory requirement

✓ Overhead of the initialization process

✓ Ideal for machines with high memory capacity

✓ In pass by value, the whole array including its vectors of addresses have to be copied to the called procedure. The initialization process have to be redone.

Chapter 5b: Composite Data Types

Multiplicative Subscript Calculation

✓ Transforms an $\mathbf{m_1 \times m_2 \times m_3 \times \ldots \times m_n}$ array into an equivalent one dimensional array with $\mathbf{m_1 * m_2 * m_3 * \ldots * m_n}$ elements.

✓ Instead of using n indices to access an element, they are used to calculate the one index that tells the position of the element to be accessed in the equivalent one-dimensional array.

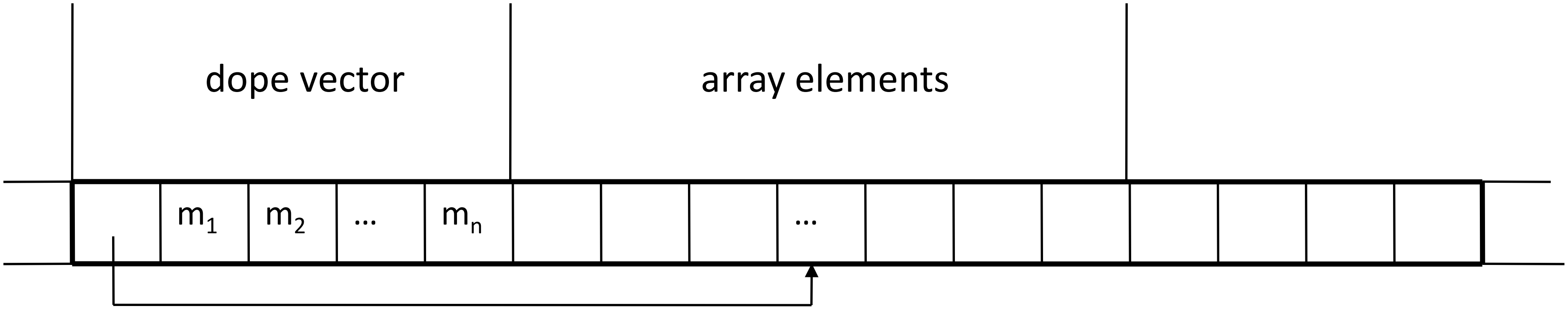
$$\text{address of } a[i_1, i_2, i_3, \ldots, i_n] = \text{address of } a[0, 0, 0, \ldots, 0] + (\ldots(((((i_1*m_2) + i_2)*m_3) + i_3)*m_4)+i_4\ldots*m_n)+i_n$$

✓ A requirement in the use of the formula is to know the values of these terms at runtime.

- **address of a[0,0,0…,0] = b**
- m_i, $2 <= i <= n$

Multiplicative Subscript Calculation

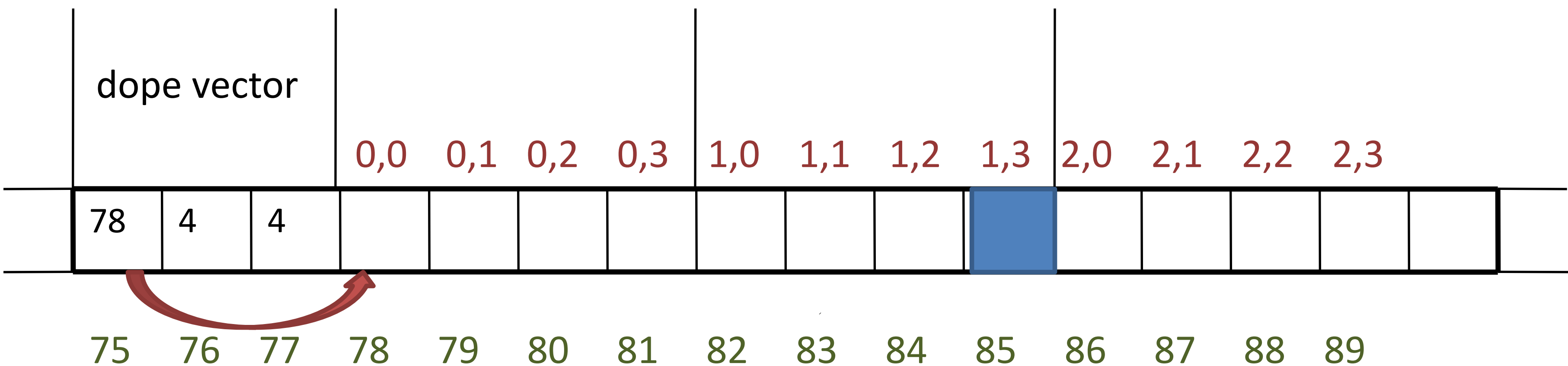
✓ We use here a storage called a **dope vector**.



Multiplicative Subscript Calculation

2 - Dimensional Arrays

- **Eg.** Consider a 4 x 4 array. Access a[1,3]
- **Storage Allocation**



- **Accessing element a[i, j]**
  $b + (i * m_2) + j$
- **Accessing element a[1, 3]**
  $78 + (1 * 4) + 3 = 85$

Chapter 5b: Composite Data Types

Multiplicative Subscript Calculation

N - Dimensional Arrays

- Consider an n-dimensional array $\mathbf{a[lo_1..hi_1,lo_2..hi_2,\ldots,lo_n..hi_n]}$ where

  $\mathbf{m_i = hi_i - lo_i + 1,\ 1 <= i <= n}$

  $\mathbf{lo_i = 0,\ 1 <= i <= n}$ **(assumption)**

- **Storage Allocation**

  Allocating for the offset a[0,0,…,0], dope vector and the components of the array

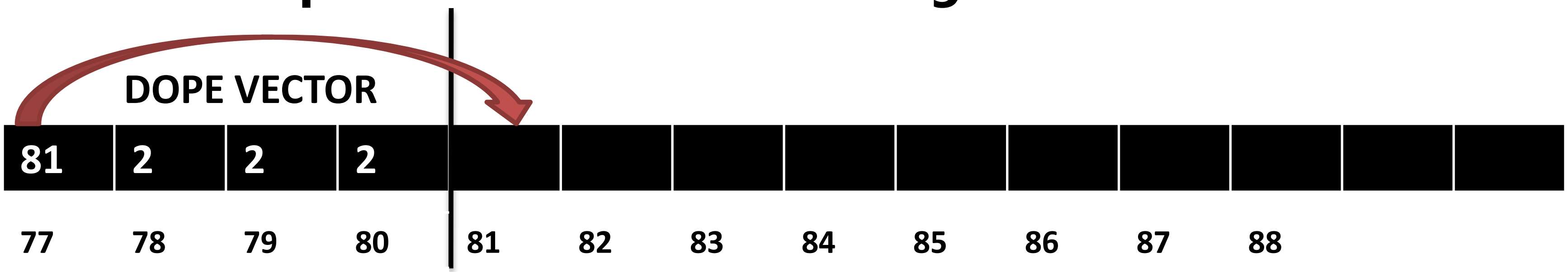
- **Accessing Elements**

  $\texttt{address of } a[i_1, i_2, i_3,\ldots,i_n] = \texttt{address of } a[0, 0, 0, \ldots, 0]$
  $+ (\ldots(((((i_1 * m_2) + i_2) * m_3) + i_3) * m_4) + i_4 \ldots * m_n) + i_n$

Chapter 5b: Composite Data Types

Multiplicative Subscript Calculation: N-Dimensional Arrays

- **Given this particular allocated storage**



1. Get ¼.
2. What are the dimensions of the array? (**m_1, m_2, m_3**)
3. What is **b**?
4. Show the complete computation to access **a[0,1,1]**.

$$\texttt{address of } a[i_1, i_2, i_3, \ldots, i_n] = \texttt{address of } a[0, 0, 0, \ldots, 0] + (\ldots(((((i_1 * m_2) + i_2) * m_3) + i_3) * m_4) + i_4 \ldots * m_n) + i_n$$

Chapter 5b: Composite Data Types

Multiplicative Subscript Calculation: N-Dimensional Arrays

- **Given this particular allocated storage**

				0,0,0	0,0,1	0,1,0	0,1,1	1,0,0	1,0,1	1,1,0	1,1,1		
81	2	2	2										
77	78	79	80	81	82	83	84	85	86	87	88		

Access a[0, 1, 1]

$\mathbf{i_1} = 0$

$\mathbf{i_2} = 1$

$\mathbf{i_3} = 1$

$\mathbf{m_1} = 2$

$\mathbf{m_2} = 2$

$\mathbf{m_3} = 2$

$\mathbf{n} = 3$

$\mathbf{b} = 81$

address of a[0, 1, 1] = b + (((i1 * m2) + i2) * m3)+ i3
= 81 + (((0 * 2) + 1) * 2) + 1
= 84

Multiplicative Subscript Calculation: Analysis

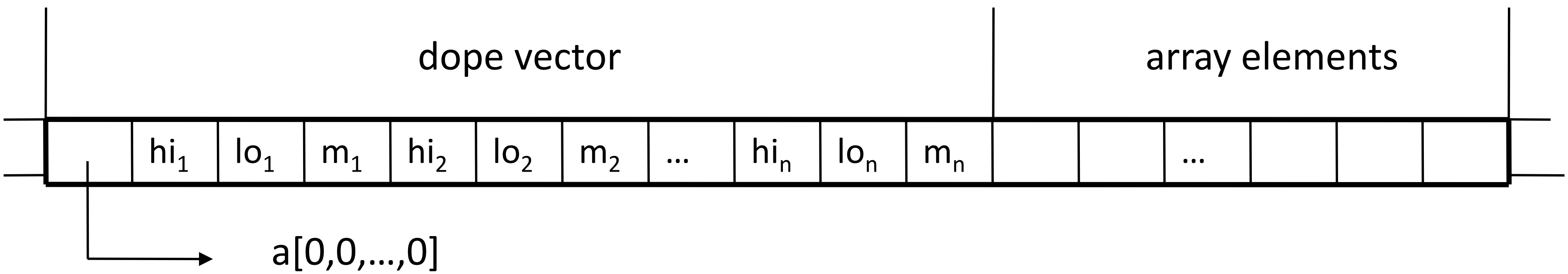
- ✓ Requires much less storage than indirect access via pre-calculated addresses.
- ✓ Initialization does not require expensive loops.
- ✓ But requires extensive computation of the address of an element at run time.

Chapter 5b: Composite Data Types

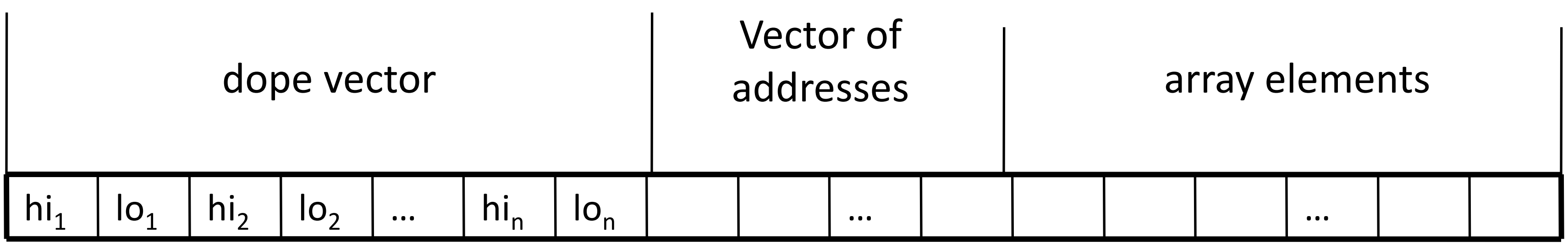
Subscript Checking

- ✓ Usually carried out at run-time.
- ✓ Usual procedure for doing this is through the dope vector.
- ✓ It is expensive to perform subscript checking.
- ✓ Most languages allow subscript checking for programmers to save hours from debugging errors in arrays.

Subscript Checking



Dope vector for multiplicative subscript calculation



Dope vector for indirect access via pre-calculated addresses

Chapter 5b: Composite Data Types

Implementation: Records

✓ A record structure is composed of a fixed number of components that can be of different types.

✓ Indices in records, called **record offsets**, are of fixed size.

Chapter 5b: Composite Data Types

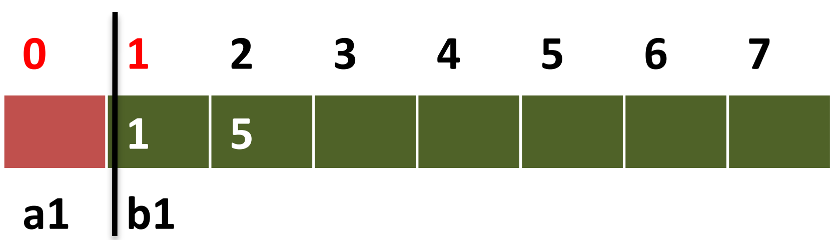
Implementation: Records

In Pascal,

type **r1 = record**   **OFFSET**
      a1: integer;   **0**
      b1: array[1..5] of char;   **1**
end;
type **r2 = record**
      a2: char;
      b2: array[-1..1] of real;
      c2: r1;
end;

r3 = record
      a3: r2;
      z1: integer;
end

var **id**: r3;



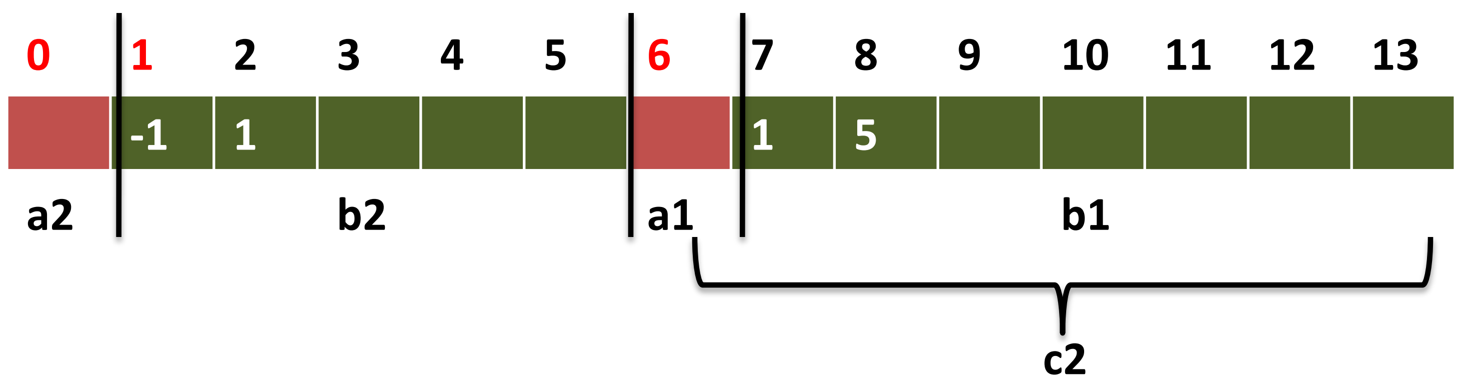
Chapter 5b: Composite Data Types

Implementation: Records

In Pascal,

```
type  r1 = record
        a1: integer;
        b1: array[1..5] of char;
end;
type r2 = record                    OFFSET
       a2: char;                      0
       b2: array[-1..1] of real;      1
       c2: r1;                        6
end;
```

```
r3 = record
     a3: r2;
     z1: integer;
end

var id: r3;
```



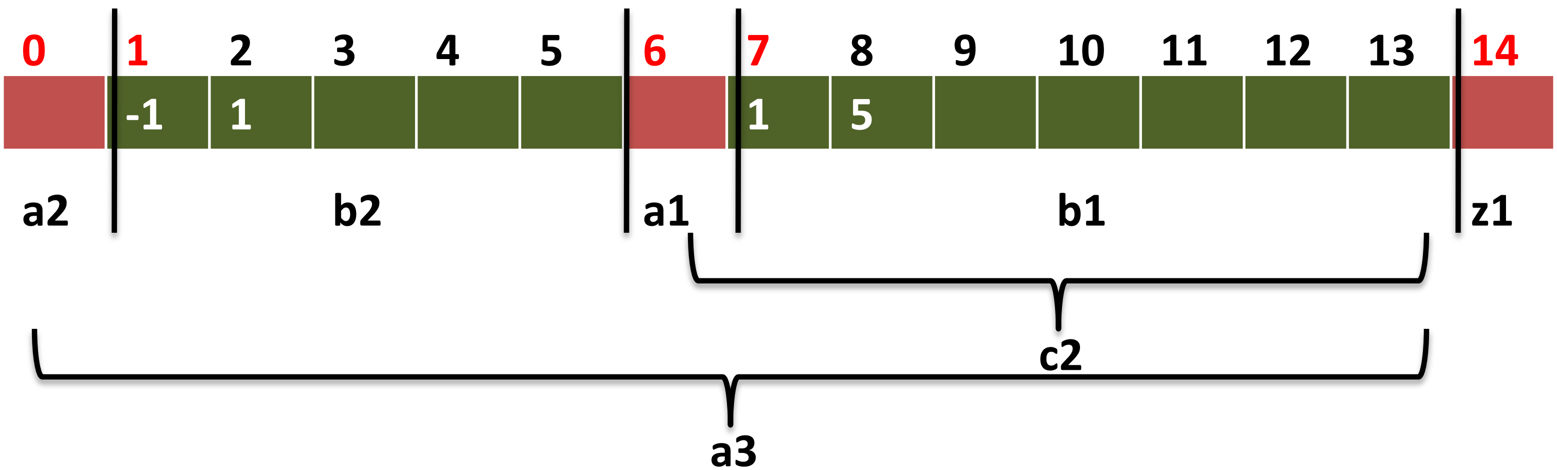
Chapter 5b: Composite Data Types

Implementation: Records

In Pascal,

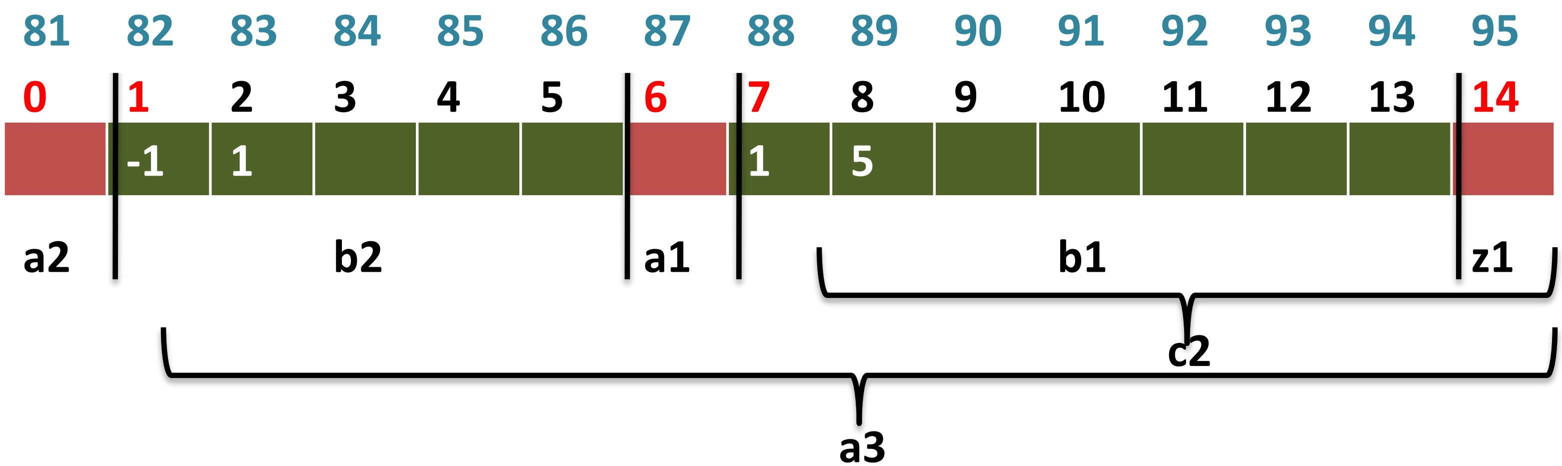
```
type  r1 = record
        a1: integer;
        b1: array[1..5] of char;
end;
type r2 = record
        a2: char;
        b2: array[-1..1] of real;
        c2: r1;
end;

r3 = record
    a3: r2;
    z1: integer;
end

var id: r3;
```

	OFFSET
a3	0
z1	14



Chapter 5b: Composite Data Types

Implementation: Records



Accepting Elements

1. Load the address of first memory allocated to id to register r.
2. Add offset of a3 to r.
3. Add offset of c2 to r.
4. Get the content of the memory whose address is in r.

Eg: id.a3.c2

1. r = 81
2. r = 81 + 0 = 81
3. r = 81 + 7 = 88
4. get_content(at 88)

Take Note:

- ✓**r** = Address of the first storage location allocated to the record.
- ✓r = r + component offset
- ✓If component is an array
  - r = r + offset of the array
  - The rest is done like in arrays

Chapter 5b: Composite Data Types

Implementation: Variant Records

✓ Variant records are records with one or more variants.

✓ Part of a record may assume different number of components and different types of components.

✓ **Storage Representation**
- w/ dynamic type checking
- w/o dynamic type checking

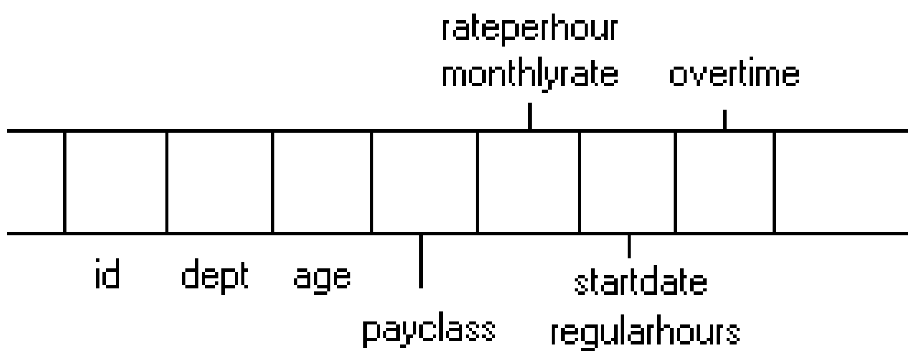
Implementation: Variant Records

Declaration	**RECORD OFFSET**
type paytype = (salaried, hourly)	
type employeetype = record	
id: integer;	**0**
dept: char;	**1**
age: integer;	**2**
case payclass: paytype of	**3**
salaried: (monthlyrate: real;	**4**
startdate: integer);	**5**
hourly: (rateperhour: real;	**4**
regularhours: integer;	**5**
overtime: integer);	**6**
end	
end	
var employee: employeetype;	

Chapter 5b: Composite Data Types

Implementation: Variant Records (with Dynamic Type Checking)

Storage Representation



Accessing Elements
Eg. employee.startdate

1. Load address of the first
cell allocated to record employee (r).
2. Make a copy of that address.
3. Load value of tag field.
4. = salaried.
5. True, jump to @1.
6. False.
7. Call error handler.
8. Quit.
9. @1:Load employee.startdate.

Chapter 5b: Composite Data Types

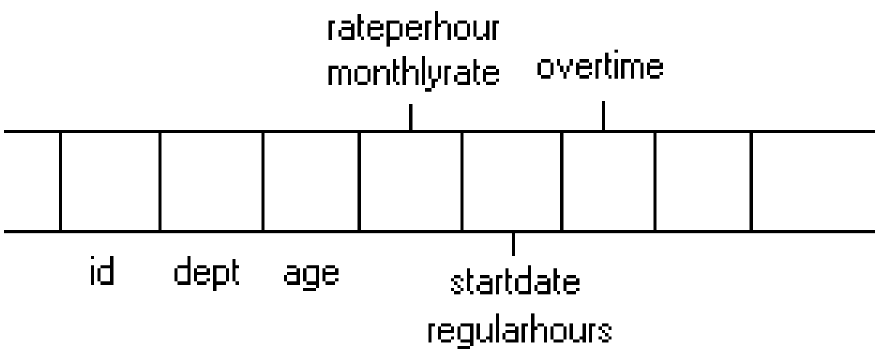
Implementation: Variant Records (without Dynamic Type Checking)

Declaration	RECORD OFFSET
type paytype = (salaried, hourly)	
type employeetype = record	
id: integer;	**0**
dept: char;	**1**
age: integer;	**2**
case payclass: paytype of	
salaried: (monthlyrate: real;	**3**
startdate: integer);	**4**
hourly: (rateperhour: real;	**3**
regularhours: integer;	**4**
overtime: integer);	**5**
end	
end	
var employee: employeetype;	

Chapter 5b: Composite Data Types

Implementation: Variant Records (without Dynamic Type Checking)

Storage Representation



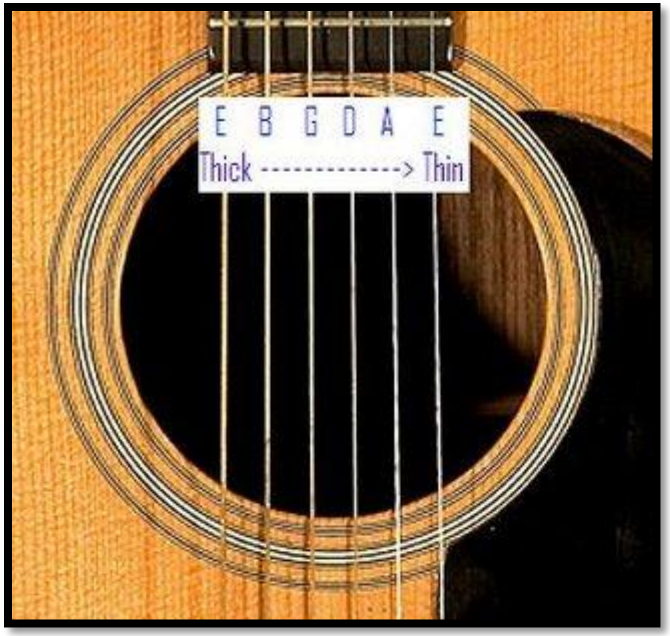
Accessing Elements
Eg. employee.startdate

1. Load the address of the first cell allocated to employee to register r.
2. Add offset of startdate (that is 4) to r.
3. Get the content of the memory whose address is in r.

Chapter 5b: Composite Data Types

Implementation: Strings

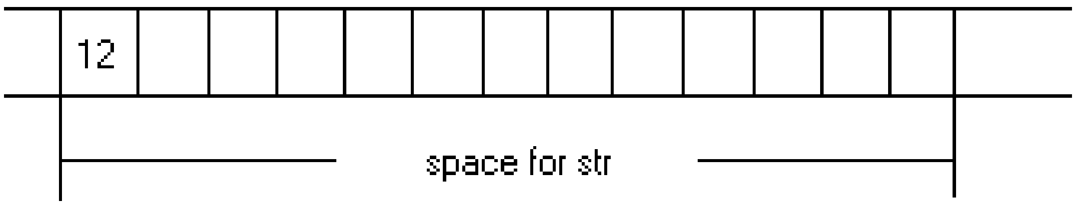
- ✓ A string is a sequence of characters.
- ✓ Can be treated as:
  - Primitive data type with the built-in functions
  - Array of chars
  - List of chars



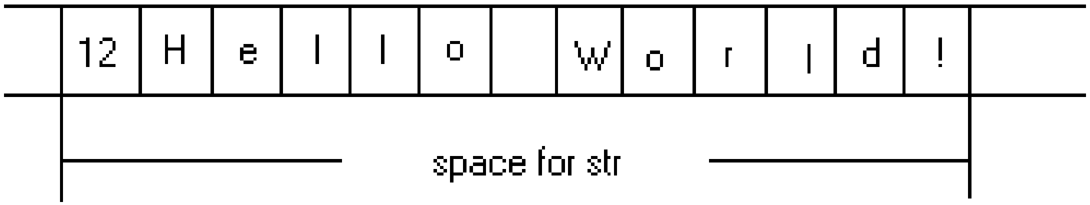
Chapter 5b: Composite Data Types

Implementation: Strings

- ✓ In ADA, a string is simply a string of characters.
- ✓ To declare an array in ADA:
  - **`str :String(1..12);`**



  - **`str: String := "Hello World!" -> str: String(12)`**



Implementation: Sets

- ✓ Defines the set of values that is a powerset of the base type.
- ✓ **In Pascal,**

  type <identifier> = set of <base type>
  where <base type> must be a scalar type or a subrange type.

```
type onedigitset = set of 0..9;
     colorset = set of (blue, yellow, red);

var  d: onedigitset;
     c: colorset;
```

Implementation: Sets

- **The Key:** Characteristic Function **c(s)**.
  - An array of logical values (0 or 1) whose **ith** component specifies the presence or absence of the **ith** value in the set.

- **Eg:**

```
var d : onedigitset;
```

The representation of **d = [1,3,5,7,9]** is

c(s) = [0,1,0,1,0,1,0,1,0,1]

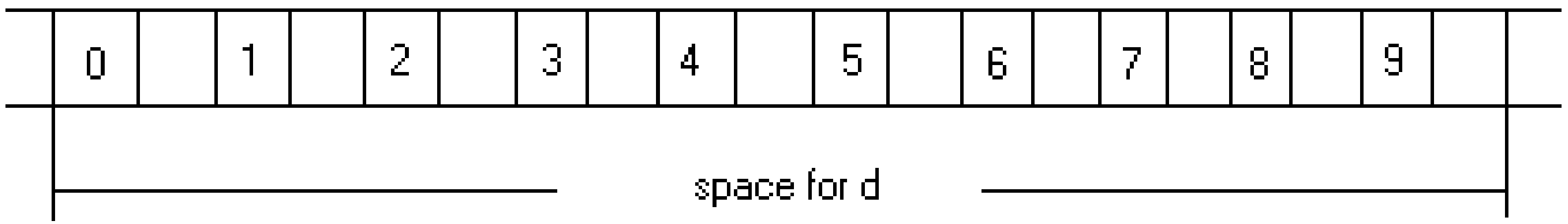
REMEMBER!

```
type onedigitset = set of 0..9;
```

Chapter 5b: Composite Data Types

Implementation: Sets

- **Implementation**
  - Individual values of the set must be known at run time.
  - Hence, there is a need to store the individual values of the base type.
- **Storage Representation** of **var d : onedigitset**



- **Accessing Elements**
  - Similar to accessing a one-dimensional array.

Chapter 5b: Composite Data Types

Practice: Show the storage representation of a

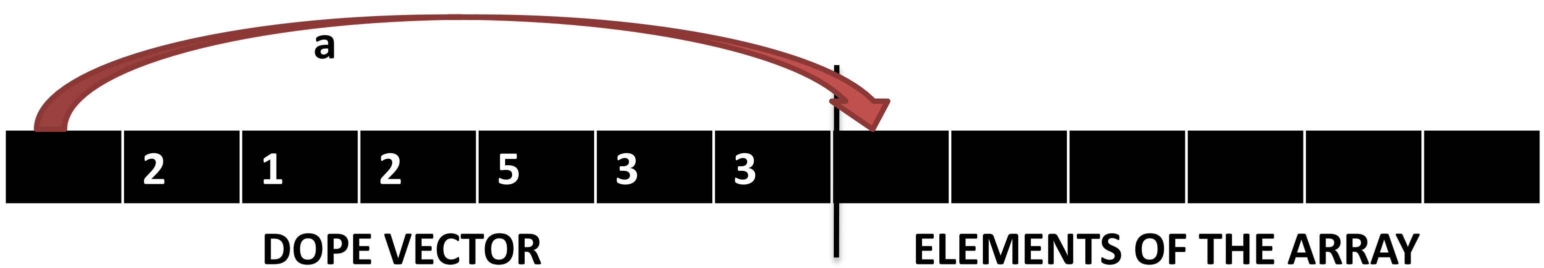
Sample 1. Use multiplicative subscript calculation (without subscript checking)

```
program arrayaccess;
var a: array[1..2, 3..5] of integer
begin
     a[1,3] := 100;
end.
```

Practice: Show the storage representation of a

Sample 2. Use multiplicative subscript calculation (with subscript checking)

```
program arrayaccess;
var a: array[1..2, 3..5] of integer
begin
     a[1,3] := 100;
end.
```



Chapter 5b: Composite Data Types

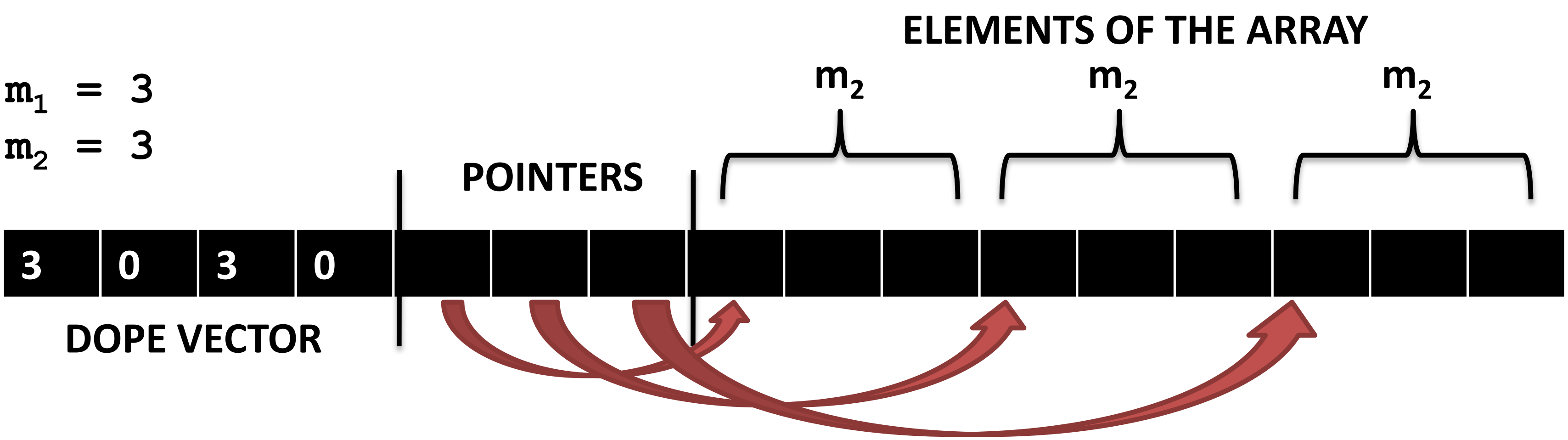
Practice: Show the storage representation of a

Sample 3. Use indirect access (with subscript checking)

```
program arrayaccess;
var a: array[0..2, 0..2] of integer
begin
     a[1,1] := a[2,2];
end.
```

REMEMBER!
There are m_1 m_2's

$m_1 = 3$
$m_2 = 3$



Practice: Show the storage representation of r

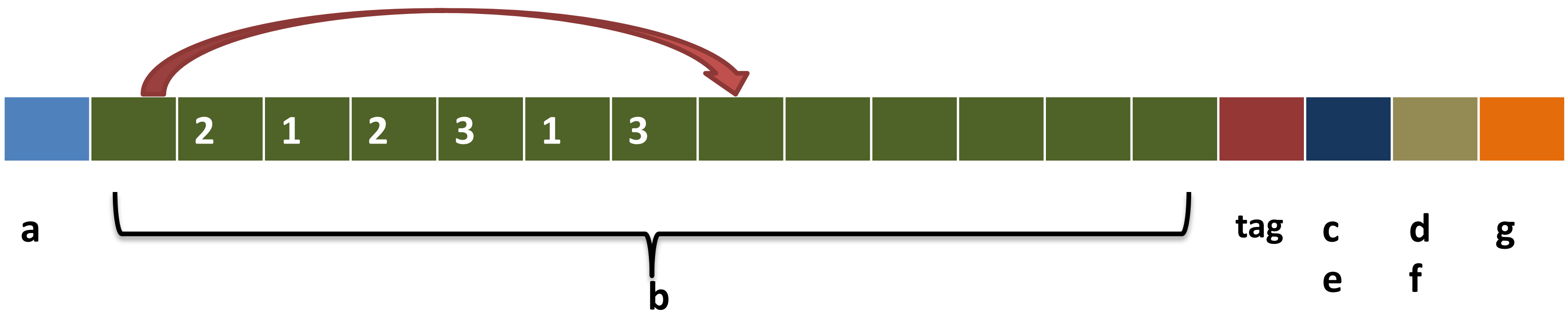
Sample 4. Use MSC and dynamic TC.

```
program recordaccess;
type rtype = record
       a: boolean;
       b: array[1..2,1..3] of integer;
       case tag: (x,y) of
              x: (c: boolean; d: real);
              y: (e: integer; f: real; g: char);
       end;
end;
var r: rtype;
```

Chapter 5b: Composite Data Types

Practice: Show the storage representation of r

Sample 4. Use MSC and dynamic TC.



EXERCISES @ Home:

- Now, use **indirect access (with subscript checking)** and still dynamic type checking to show r's storage representation.
- Btw, try experimenting in defining your own structures and showing their corresponding storage representations; and tracing in accessing elements. ☺