Languages for Informatics

5 – Arrays and Pointers

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Programming
  2. Variables, Data-types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. User defined datatype and data structure
  6. Input and Output
- Basic system programming in Linux (10h)

Overview

Array

- Array is a group of elements that share a common name, and that are different from one another by their positions within the array.
- The number of elements is prefixed.
- All elements have the same type.
- Example: keep in memory the age of 15 people, so that you will able to compute their average later on.

```
int age[15];
```

- Example: keep in memory the minimum temperature of the last 30 days, so that you will be able to compute the overall minimum.

```
double temp[30];
```

Declaration & Initialization

- *Declaration*: Memory is assigned but contents is **unknown** at init.

```
int age[15];
```

- *(Static) Initialization*: Contents is **known** at init.

```
int age1[] = {23, 24, 17, 27, 25, 24, 24}
int age2[15] = {23, 24, 17, 27, 25, 24, 24}
```

- What will be the result of age1[12] - age1[7] ?
- Demonstration ...

Array elements

- Access the *i*-th element: **`array[i]`** (*i* is called *index*)
- Example: assign a value

```
int array[30];
array[17] = 5;
```

- Example: read a value

```
int array[30];
int n;
...
n = array[17];
```

Note

In C, a *n*-dimensional array is indexed from 0 i.e., arr[0], arr[1], arr[2], arr[3], . . ., arr[n-1]. There is no element arr[n]!

Example

Average Age

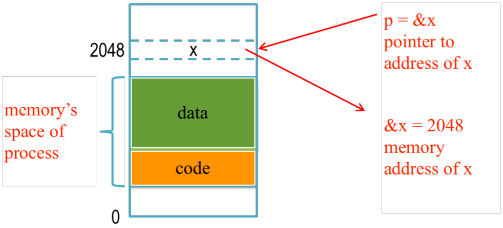
Scalar form:

```
int main(void){
  float average;
  int sum=0,age1=23;
  int age2=24,age3=17,age4=27;
  sum += age1;
  sum += age2;
  sum += age3;
  sum += age4;
  average = sum/4.0;
}
```

Vector form:

```
int main(void){
  float average;
  int i,n=4,sum=0;
  int age[]={23,24,17,27};
  for (i=0;i<n;i++) {
    sum += age[i];
  }
  average = (float) sum/n;
}
```

Introduction into Pointers



- In C, it is possible to know the address of the memory cell where a variable (or even a function!) is stored
  - The unary operator `&` returns the memory address of a variable, e.g. `&x`
- Pointer variable `*p` points to another variable in memory space of the process, e.g. `*p = x`.

A Scholarly Example

```
int a = 10; //a is an integer variable (init. to 10)
```

Variable	Address	Content
a	0x000064	10

A Scholarly Example

```
int a = 10; //a is an integer variable (init. to 10)
int *b; //Declare b a ptr to int variable
```

Variable	Address	Content
a	0x000064	10
b	0x000068	

A Scholarly Example

```
int a = 10; //a is an integer variable (init. to 10)
int *b; //Declare b a ptr to int variable
b = &a; //equiv. *b = a; b contains the address of a
```

Variable	Address	Content
a	0x000064	10
b	0x000068	0x000064

A Scholarly Example

```
int a = 10; //a is an integer variable (init. to 10)
int *b; //Declare b a ptr to int variable
b = &a; //equiv. *b = a; b contains the address of a
...
 //Using the memory address, it is possible to
manipulate to content of a variable
*b = *b - 2;
```

Variable	Address	Content
a	0x000064	10
b	0x000068	0x000064
Variable	**Address**	**Content**

Pointers

```
type * var
```

- Declares a pointer called `var`
- Its type is *address of variables* having type `type`
- The operator `&` is used to return the address of a variable
- The operator `*` is used to access the content of a memory address stored into a pointer (dereferencing)
- Indirection operator `*` is "inverse" to `&`.
- `p = &i; i = *p;` If we know the variable's address, we can access its data and vice versa.
- It is possible to declare pointers for any primitive type

The operator *

Used in the declaration of a pointer variable

```
int *a;
```

Used in statements to obtain dereferentiation

- Within an expression, it gives access to the content of memory cell it is pointing at
  - `if (*a > 10) { ... } else { ... }`
  - `*a = 10;`

Caution

Illegal expressions

```
&i = p; /* addresses allocated by declaration */
p = &10;
p = &(i+j); /* const. & expressions don't have
addresses */
```

Constants and pointers

These two declarations are equivalent, that is pointers to integer constants

```
const int *a;
int const *a;
```

How about these?

```
const int *a;  //Pointer to integer constant
int *const a; //?? and this ??
```

Constants and pointers

These two declarations are equivalent, that is pointers to integer constants

```
const int *a;
int const *a;
```

How about these?

```
const int *a;   //Pointer to integer constant
int *const a;  //Constant pointer to integer
```

Note

They are not equivalent! In the second case you cannot modify the content of `a` (i.e., the address contained in `a`) but you can modify the content of the variable pointed by `a`, i.e. `*a`.

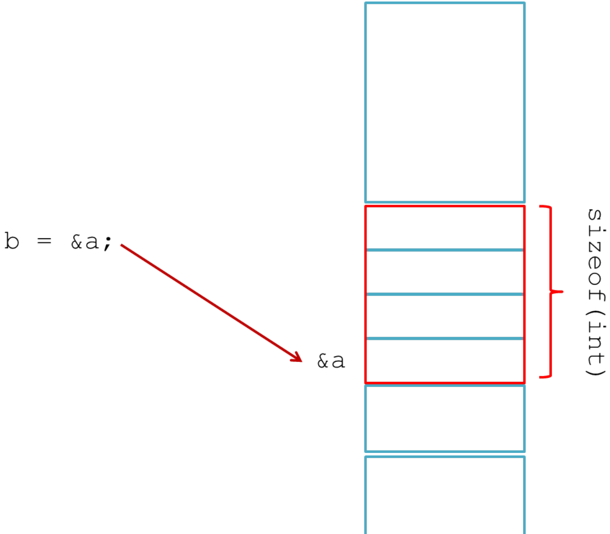
Casting Pointers

It is possible to cast one type of pointer to another type

```
int a = 8;
int *b;   //Pointer to integer
double *c; //Pointer to double
....
b = &a;
c = (double*) b;
```

What do we have by dereferencing `c`?

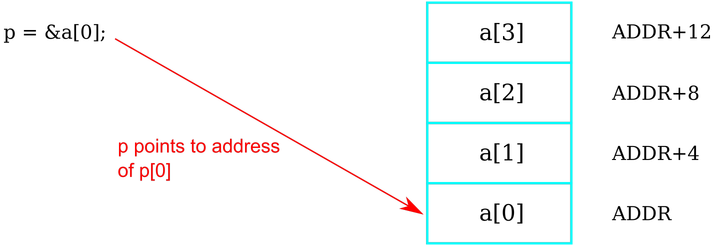
Casting Pointers



Address arithmetic

- A pointer in C is an address which is a numerical value.
- It is possible to use the arithmetic operators $+, -, ++, --$ and
- the comparison operators $<, <=, >, >=, ==, !=$ to write expressions with pointers

```
int a[4], *p; //Declare an array of integers and a pointer to integer
```

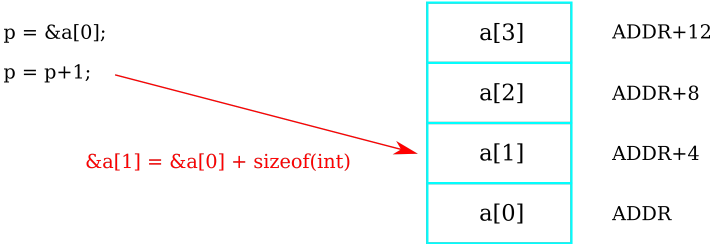


Address arithmetic

Increment/Decrement a Pointer

- A pointer in C is an address which is a numerical value.
- It is possible to use the arithmetic operators $+, -, ++, --$ and
- the comparison operators $<, <=, >, >=, ==, !=$ to write expressions with pointers

```
int a[4], *p; // Declare an array of integers and a
   pointer to integer
```



Address arithmetic

Increment/Decrement a Pointer

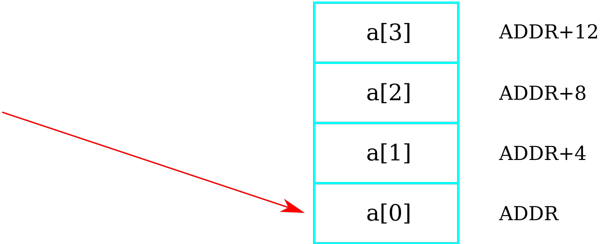
- A pointer in C is an address which is a numerical value.
- It is possible to use the arithmetic operators $+, -, ++, --$ and
- the comparison operators $<, <=, >, >=, ==, !=$ to write expressions with pointers

```
int a[4], *p; //Declare an array of integers and a
  pointer to integer
```

p = &a[0];

p = p+1;

p = --p;



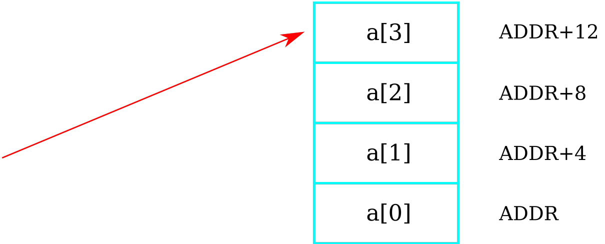
Address arithmetic

Increment/Decrement a Pointer

- A pointer in C is an address which is a numerical value.
- It is possible to use the arithmetic operators $+, -, ++, --$ and
- the comparison operators $<, <=, >, >=, ==, !=$ to write expressions with pointers

```
int a[4], *p; //Declare an array of integers and a
   pointer to integer
```

```
p = &a[0];
p = p+1;
p = --p;
p +=3;
```



Address arithmetic

Pointer Comparison

The following code snippet increments the variable pointer and assigns a value to it so long as the the address to which it points is either less than or equal to the address of the last element of the array.

```
int *ptr = a;   /* a is an integer array filled with some values
     */
int i=0;
...
while (ptr < &a[MAX]) {
  printf("Address of a[%d] = %p \t", i, ptr );
  printf("Value of a[%d] = %d \n",i,*ptr);

  ptr++;   /* point to next address */
  i++;
}
```

Pointer arithmetic

Pointer Comparison

Result

```
Address of a[0] = 61fdfc   Value of a[0] = 1
Address of a[1] = 61fe00   Value of a[1] = 2
Address of a[1] = 61fe04   Value of a[1] = 3
```

Arrays and pointers (I)

Consider the following scenario:

```
int a[3], *p, tmp;
p = a;  //Pointer to the (first element of the) array
tmp = a[2];  /* The 2nd index of a is equal */
tmp = p[2];  /* to the second index of p */
```

`p = &tmp`? `p` points to the memory address of `tmp`.

`a = &tmp`?

Arrays and pointers (II)

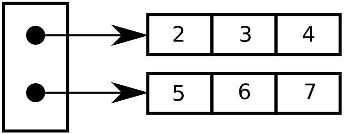
gcc says

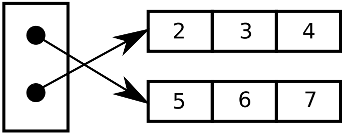
```
$ error: assignment to expression with array type
```

`int a[3]` declares a constant pointer to integers (int *const)
$\Longrightarrow$ We cannot modify the memory cell where `a` points to!

Pointer Arrays (1)

- Pointers are variables themselves.
- Pointers can be stored in arrays as other variables can.
- When two out-of-order lines have to be exchanged, the **pointers in the pointer array are exchanged, not the text lines themselves.**





Pointer Arrays (2)

- To maintain an array that stores pointers to `int`,

```
int *ptr[MAX];
```

  declaring `ptr` as an array of MAX integer pointers. Each element in `ptr`, holds a pointer to an `int` value.

- Consider the `int` array

```
int a[MAX] = {1,2,3};
```

- For each array index, the pointer **`ptr`** has to point to the corresponding address of the integer array:

```
for (int i = 0;i<MAX;i++) {
    ptr[i] = &a[i];
  }
```

Pointer Arrays (3)

To print the **addresses** of the respective integers,

```
for (int i = 0;i<3;i++) {
  printf("ptr[%d] = %p \t",i,ptr+i);  // ptr + i ===
    ptr[i]
}
```

To print the **values** of the respective integers,

```
for (int i = 0;i<MAX;i++) {
  printf("*ptr[%d] = %d \t",i,*ptr[i]);
}
```

Result

```
Addresses
p[0] = 6422000 p[1] = 6422008 p[2] = 6422016
Values
*ptr[0] = 1   *ptr[1] = 2   *ptr[2] = 3
```

Example

Let us return to our averaging function. This time, in pointer form.

```
double average(int *age, int n)  // argument: pointer to an array of int
{
    int *p;
    double res;
    res = 0.0;
    for (p=age;p<&age[n];++p)  //start: points to 1st address in age
       res += *p;              //stop: points to the last address
    return (res/n);            //contents of p is added to res.
}

int main(void){
  float result;
  int n=4, age[]={23,24,17,27};
  result = average( age, n);
  printf("average = %2.2f",result);
  return 0;
}
```

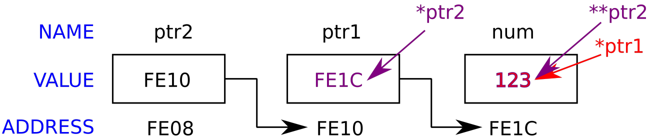
Result

```
average = 22.75
```

Pointers to Pointers

- A pointer to a pointer is a chain of pointers.
- Many practical applications in C: pointer arrays, string arrays.
- The first pointer contains the address of a variable.
- The second pointer points to the location that contains the actual value as shown below

```
int num = 123;       // an integer
int *ptr1, **ptr2;
ptr1 = &num;   // ptr to the address of num
ptr2 = &ptr1; // ptr to the address of the 1st
    pointer
```



Pointers to Pointers

```
int main() {
int num = 123;
int *ptr1, **ptr2;
ptr1 = &num;
ptr2 = &ptr1;
printf("\n Adr. of num = %p",&num);
printf("\n Value of num = %d",num);
printf("\n Adr. of ptr 1 = %p",&ptr1);
printf("\n ptr 1 = %p",ptr1);
printf("\n Value of *ptr1 = %d",*ptr1);
printf("\n Adr. of ptr 2 = %p",&ptr2);
printf("\n ptr 2 = %p",ptr2);
printf("\n Value of *ptr2 = %p",*ptr2);
printf("\n Value of **ptr2 = %d",**ptr2);
return 0;
}
```

Result

```
Adr.  of num = 0x7ffd562b5394
Value of num = 123
Adr.  of ptr 1 = 0x7ffd562b5398
ptr 1 = 0x7ffd562b5394
Value of *ptr1 = 123
Adr.  of ptr 2 = 0x7ffd562b53a0
ptr 2 = 0x7ffd562b5398
Value of *ptr2 = 0x7ffd562b5394
Value of **ptr2 = 123
```

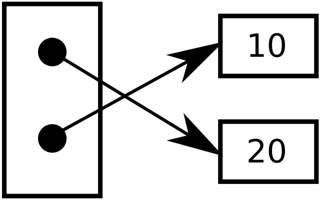
Pointers to Pointers

Example

Swap two pointers

```
void swap(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int main () {

int a = 10;
int b = 20;
printf("a=%d, b=%d \n", a, b);
swap(&a,&b);     //swap pointers
printf("a=%d, b=%d \n", a, b);
...
}
```



Result

```
a=10, b=20
a=20, b=10
```

Pointers to Pointers

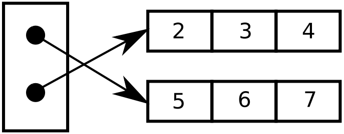
Example

What is this function doing ?

```
void swap(int** a, int** b)
{
    int* tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Swapping pointers to an array.

```
int main () {
  int c[3] = {2,3,4}, d[3] = {5,6,7};
  int *cptr = c, *dptr = d;
  for (int i=0; i<3; i++) {
   printf("c[%d]=%d, d[%d]=%d \n", i,cptr[i], i,
      dptr[i]); //2,3,4; 5,6,7
  }
  swap(&c, &d);
  for (int i=0; i<3; i++) {
   printf("c[%d]=%d, d[%d]=%d \n", i,cptr[i], i,
      dptr[i]); //5,6,7; 2,3,4
  }
}
```



Multi-dimensional arrays (1)

- Structure

Scalar variable	a
Vector variable (1D)	$a_0, a_1, a_2, \ldots$
Matrix variable (2D)	$a_{00}, a_{01}, a_{02}, \ldots$ $a_{10}, a_{11}, a_{12}, \ldots$ $a_{20}, a_{21}, a_{22}, \ldots$ $\ldots$

- C also permits multidimensional arrays specified by the bracket $[\cdot]$ operator.
  - rectangular form
  - fixed dimensions

Multi-dimensional arrays (2)

- Declaration:

```
int L = 10, M = 10;
int age[L][M];     // L-rows and M-columns
```

or

```
#define L 10
#define M 10
...
int age[L][M];
```

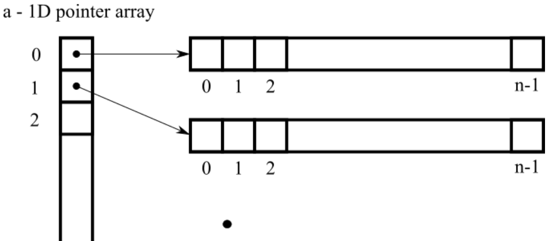
- Initialization:

```
int age[2][2] = {23, 24, 17, 27}; // row-wise init.
int age[2][2] = {{23, 24}, {17, 27}}; // row-by-row
int age[0][1] = 24;     // element-wise init.
```

Multi-dimensional arrays (3)

A few differences to vector arrays.

- The variable `*age` points to base address of `&age[0][0]` (rather than its value `age[0][0]`).
- Hence, `**age` is the value of `age[0][0]`).
- `*(age+i)` points to the address of the i-th row `&age[i][0]`.
- `*(age+i)+j` is the address of `&age[i][j]`.
- `*(*(age+i)+j)` is the element of `age[i][j]`.



Multi-dimensional arrays (4)

- Higher dimensions are possible:

```
double bigmatrix [12][3][5][35];  // dimension = 4
```

- Multidimensional arrays are **rectangular**.
- Pointer arrays can be arbitrary shaped.

Example

Function that computes the trace of a square matrix

$$tr(A) = \sum_{i=0}^{n-1} a_{ii}$$

```
double trace(double a[][COLS], int rows) {
   double sum = 0.0;
   for (int i=0;i<rows;i++)
     sum += a[i][i];
   return sum;
}
```

Example

```
#include <stdio.h>
#define ROWS 3
#define COLS 3
double trace (double a[][COLS], int rows) {
  double sum = 0.0;
  for (int i = 0; i<rows; i++)
    sum += a[i][i];
  return sum;
}
int main() {
  double A[ROWS][COLS];
  A[0][0] = 0.1;
  A[1][1] = 1.1;
  A[2][2] = 2.2;
  printf("trace = %lf \n",trace(A,ROWS));
  return 0;
}
```

Result

```
trace =
3.400000
```

Dynamic Memory Allocation for Arrays

1-dim pointer arrays

The task is to

- Dynamically declared arrays at runtime are more flexible.
- declare an array of `<TYPE>` (`int`, `double`, etc...) pointers
- allocate and initialize memory for each element

```
#include <stdio.h>
#include <stdlib.h>  // lib for dyn memory allocation.
#define n 10 // dimension
void main(){
  int *A;
  A = malloc (sizeof(int) * n); //allocate memory and
    return pointer to it
  for (i = 0; i < n; i ++)
    A[i] = 0;  // example allocation
  ...
  free(A);
}
```

Dynamic Memory Allocation for Arrays

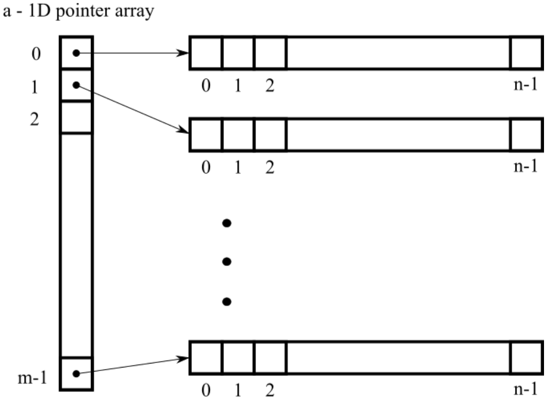
1-dim pointer arrays (2)

- Method `malloc(byte-size)` declares a single large block in the heap segment of the memory, that is initialized with default garbage value.
- To free the space, use the library call `free(A)`.
- Method `calloc(n, element-size)` does the same as a malloc but initializes each block with the default value `NULL`. Two arguments are required.
- Method `realloc(ptr, newSize)` dynamically change the memory size of a previously allocated memory. Already present value do remain.

Dynamic Memory Allocation for Arrays

2-dim pointer arrays

We need to initialize the array of pointers to pointers and then initialize each 1d array in a loop.



Dynamic Memory Allocation for Arrays

2-dim pointer arrays (2)

In computer memory, the $m \times n$-matrix has the form

```
#include <stdio.h>
#include <stdlib.h>
#define M 2     // rows
#define N 3     // columns

void main() {
double **A;
A = calloc(M, sizeof(double *));   //array of pointer to
    double to rows
for (int i = 0; i < M; i++)
  A[i] = calloc(N, sizeof(double));   //init cols.
...
free(A);
}
```

Dynamic Memory Allocation for Jagged Arrays (1)

- Pointer arrays can be arbitrary shaped.
- Consider a jagged array with $M = 3$ rows and $N = N[m]$ columns:

Columns

Rows	0	1	2	3
0	0.0	0.1	0.2	0.3
1	1.0	1.1		
2	2.0	2.1	2.2	

Dynamic Memory Allocation for Jagged Arrays (2)

```
#include <stdio.h>
#include <stdlib.h>
#define M 3

void main() {

  double **A;
  int N[M] = {4,2,3};

  A = calloc(M,sizeof(double *));
  for (int i = 0; i < M; i++)
    A[i] = calloc(N[i],sizeof(double));
  ...
  free A;
}
```

We have created a matrix with **variable-length rows.**

Example (1)

Computing the trace of a matrix, revisited in pointer notation

```
#include <stdio.h>
#include <stdlib.h>
#define ROWS 3
#define COLS 3

double trace (double **a, int rows);

int main() {

double **A = calloc(ROWS, sizeof(double*));
for (int i=0;i<ROWS;i++)
   A[i] = calloc(COLS, sizeof(double));

A[0][0] = 0.1;
A[1][1] = 1.1;
A[2][2] = 2.2;
printf("trace = %lf \n",trace(A,ROWS));
return 0;
}
```

Result

```
trace =
3.400000
```

Example (2)

Multiplication of matrices (1)

Let us write a function that multiplies two input matrices and **returns a matrix inline.**

$$[\boldsymbol{C}]_{i,j} = \sum_{k=1}^{n} [\boldsymbol{A}]_{i,k}[\boldsymbol{B}]_{k,j}$$

```
void mat_mult(double **A, double **B, double **C, int
    dim) {
  for (int k = 0; k < dim; k++){
    for (int i = 0; i < dim; i++) {
      for (int j = 0; j < dim; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Example (2)

Multiplication of matrices (2)

Calling by the main function, it follows

```
#include <stdio.h>
#include <stdlib.h>
#define M 2

void mat_mult(double **A, double **B, double **C, int dim);
int main() {
double **A, **C;

A = calloc(M,sizeof(double *));
for (int i = 0; i < M; i++)
  A[i] = calloc(M,sizeof(double));
C = calloc(M,sizeof(double *));
for (int i = 0; i < M; i++)
  C[i] = calloc(M,sizeof(double));
A[0][0] = 1.0;
A[0][1] = 1.1;
A[1][0] = 2.0;
A[1][1] = 2.1;

mat_mult(A,A,C,M);
...
free(A);
free(C);
}
```

Result

```
C[0][0] = 3.200000
C[0][1] = 3.410000
C[1][0] = 6.200000
C[1][1] = 6.610000
```

Example (3)

Multiplying a matrix with a vector (1)

Let us write a function that multiplies a matrices with a vector and **returns a pointer** to the result.

$$[\boldsymbol{x}]_i = \sum_{k=1}^{n} [\boldsymbol{A}]_{i,k}[\boldsymbol{b}]_k$$

```
double * matvec_mult(double **A, double *b, int dim) {
  double *x = calloc(dim,sizeof(double));
  for (int k = 0; k < dim; k++){
    for (int i = 0; i < dim; i++) {
      x[i] += A[i][k] * b[k];
    }
  }
  return x;
}
```

Example (3)

Multiplying a matrix with a vector (2)

Calling by the main function, it follows

```
#include <stdio.h>
#include <stdlib.h>
#define M 2
double * matvec_mult(double **A, double *b, int dim);
int main() {
double **A, *b, *x;
A = calloc(M,sizeof(double *));
for (int i = 0; i < M; i++)
  A[i] = calloc(M,sizeof(double));
b = calloc(M,sizeof(double *));
A[0][0] = 1.0; A[0][1] = 1.1; A[1][0] = 2.0; A[1][1] = 2.1;
b[0] = 0.8; b[1] = 1.3;

x = matvec_mult(A,b,M);
for (int i = 0;i<M;i++)
    printf("x[%d] = %.2lf \n",i,x[i]);
free(A);
free(b);
free(x);
}
```

Result

```
x[0]= 2.23
x[1] = 4.33
```

Quiz

Consider the following program snippset:

```
int main (void){
int n = 4, i, *A;
void *ptr;
A = (int *) malloc(sizeof(int) *n);
for (i=0;i<4;i++)   A[i] = i;
ptr = A;
i = 2;
```

Based on the above code, mark all of the following expressions that access *A*[*i*]

1. `*(A+i)`
2. `*(ptr+i)`
3. `*(int *)(ptr + i)`
4. `*((int *)ptr + i)`
5. `*(ptr + sizeof(int)*i)`

Quiz (2)

Consider the following program snippset:

```
void init(<YOUR TASK>)) {<YOUR TASK>}

int main (void){
int a;
double b;
char c;
init(&a, &b, &c);
printf("a = %d, b = %lf , c = %c", a, b, c);
return(EXIT_SUCCESS);
}
```

```
$ ./myexample
a=1, b = 2.0, c = P;
```