PROGRAMMING

WEEK 2

MODULE 1: ARRAYS

Shankar Balachandran, IIT Madras

Arrays

- So far, we used datatypes provided by the language
  - *int, float, char*
- Aggregate Datatype
  - A logical collection of values
- Arrays
  - Aggregate datatype of same type of elements
  - Fixed size, sequentially indexed

Arrays

- Logical collection of values of the same type
  - List of marks of a student
  - Daily temperature over the last year
  - Matrices
  - List of students in a class
- Operations done on such collections
  - Find maximum, average, minimum…
  - Order the elements
  - Search for a particular element

Imagine this: Find Average Temperature of the Year

Code Segment:

```
float sum, average;

average = sum/365;
```

More Elegant: Arrays (Example 1)

Code Segment:

```
float temp[365];
float sum=0.0, average;
int i;

for(i=0; i<365; i++){
        scanf("%f",&temp[i]);
}

for(i=0; i<365; i++){
        sum += temp[i];
}
average = sum/365;
```

365 elements of the same type

Scan the elements one by one and store

Add the elements

Arrays

- Declaration
  - <type array name[number of elements]>
  - Examples:
    - int marks[7];
    - float temperature[365];
- int marks[7];
  - A contiguous group of 7 memory locations called “marks”
  - Elements
    - marks[0], marks[1], …, marks[6]
    - marks[i] $0 \leq i \leq 6$

Memory point of view

int marks[7] ;

- Each element can be thought of as a variable
- Just like individual variables, they start out uninitialized
- Values can be assigned to elements
  - marks[3] = 36;
- ‘&’ is used to get the location in the memory
  - Just like it was for a variable
  - &marks[1] would be 2735

Address: 2731	-	marks[0]
	-	marks[1]
	-	marks][2]
	36	marks[3]
	-	marks[4]
	-	marks[5]
Address: 2755	-	marks[6]

Revisit the Example

Code Segment:

```
float temp[365];
float sum=0.0, average=0.0;
int i;

for(i=0; i<365; i++){
        scanf("%f",&temp[i]);
}

for(i=0; i<365; i++){
        sum += temp[i];
}
average = sum/365;
```

365 elements of the type float

Read into memory location of temp[i]

Loop runs from i=0 to i=364, a total of 365 times

Initialization

- Arrays can be declared with initialization
  - int marks[7] = {22,15,75,56,10,33,45};
- New values can be assigned to elements
  - marks[3] = 36;

22	0
15	1
75	2
36	3
10	4
33	5
45	6

Few Fine Points

- Array indices start at 0
  - Not 1
- Very common mistake to assume index starting at 1
- int marks[7];
  - Valid entries are marks[0] to marks[6]
  - marks[7] is invalid
  - marks[-1] is also invalid

Example 2: Find the Hottest Day

Code Segment:

```
float temp[365], max;
int hottestDay, i;
/* NOT SHOWN : Code to read in temperature for the 365 days*/

for(i=1; i<365; i++){

}
printf("The hottest day was Day %d with temperature %f", hottestDay, max);
```

Assume Day 0 is the hottest and record the temperature.

If the i^{th} day is hotter than our current record, update

Multidimensional Arrays

- Arrays with two or more dimensions can be defined
  - Useful for matrices, 3D graphics etc.

int A[4][3];

	0	1	2
0			
1			
2			
3			

float B[2][4][3];

	0	1	2
0			
1			
2			
3			

	0	1	2
0			
1			
2			
3			

PROGRAMMING

WEEK 2

MODULE 2: WORKING WITH 1D ARRAYS

Shankar Balachandran, IIT Madras

Generic Programs


```
#include <stdio.h>
#define N  6
int main (void)
{
     int values[N];
     int i;
    for ( i = 0; i < N; i++ ) {
         printf("Enter value of element  number %d:\n",i);
         scanf("%d", &values[i]);
    }
    for ( i = 0; i < N; i++ )
         printf ("values[%d] = %d\n", i, values[i]);
}
```

Generic value; Change, recompile and run if N has to be different

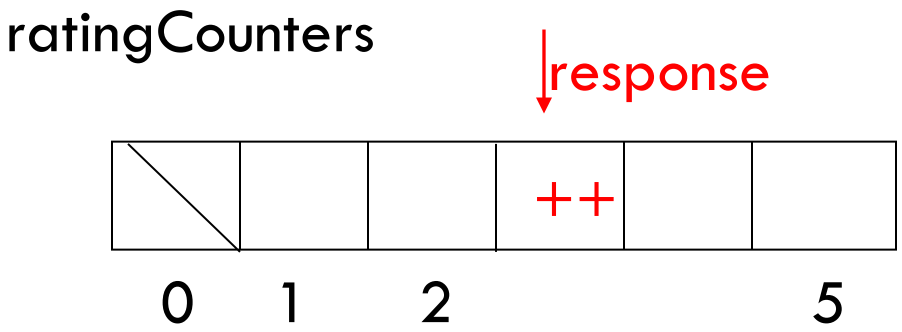
Ex 1: Generate First N Fibonacci numbers


```
#include <stdio.h>
#define N  10
int main (void)
{
     int Fib[N];
     int i;
     Fib[0] = 0;
     Fib[1] = 1;
    for ( i = 2; i < N; i++ ) {
         Fib[i] = Fib[i-1] + Fib[i-2];
    }
    for ( i = 0; i < N; i++ )
         printf (“Fib[%d] = %d\n", i, Fib[i]);
}
```

Ex 2: Using Arrays as Counters

- Let's say 1000 students took this course
- Each student rates the course from 1 to 5
- Find how the ratings are spread
  - Count the number of times each rating was given

Ex3: Array of counters



ratingCounters[i] = how many students rated the course as i

Ex3: Array of counters (contd.)

```
#include <stdio.h>
int main (void)  {
        int ratingCounters[6] = {0,0,0,0,0,0}, i, response;
        printf ("Enter your responses\n");
        for ( i = 1; i <= 1000; ++i ) {
                scanf ("%d", &response);



        }
        printf ("\n\nRating      Number of Responses\n");
        printf ("------ -------------------\n");
        for ( i = 1; i <= 5; ++i )
                printf ("%d     %d\n", i, ratingCounters[i]);
        return 0;
}
```

Record valid responses only. Ignore if invalid

Problem 3.1:Finding All Prime Numbers<=N

- Observations:
  - 2 is the only even prime number
  - 3 is the smallest odd prime number
  - To find out if a number p is prime or not, it is enough to check if p is divisible by all primes less than p
    - If some prime number i < p divides p, then p is composite
    - If there is no i such that i divides p fully, then p is prime

Code Segment


```
for ( p = 5; p <= N; p = p + 2 ) {//iterate over all odd numbers <= N
        isPrime = 1;//assume that it is prime
                for ( i = 1; i < primeIndex; ++i ){
                        //if p is divisible by some prime i, then p is not prime
                        if ( p % primes[i] == 0 )
                                isPrime = 0;
                }
                if ( isPrime == 1 ) {
                        primes[primeIndex] = p;
                        ++primeIndex;
                }
}
```

Exercise

- Can make the program more efficient
  - Check only till $\sqrt{N}$
  - If p is divisible by some prime number, the loop in Line 14 still runs till all prime numbers less than it are checked
- Each of these techniques will reduce number of steps
- Can also combine both

PROGRAMMING

WEEK 2

PRIME NUMBERS – DEMO OF DEBUGGING

Shankar Balachandran, IIT Madras

Problem 3.1:Finding All Prime Numbers<=N

- Observations:
  - 2 is the only even prime number
  - 3 is the smallest odd prime number
  - To find out if a number p is prime or not, it is enough to check if p is divisible by all primes less than p
    - If some prime number i < p divides p, then p is composite
    - If there is no i such that i divides p fully, then p is prime

PROGRAMMING

WEEK 2

MODULE 3: TWO DIMENSIONAL ARRAYS

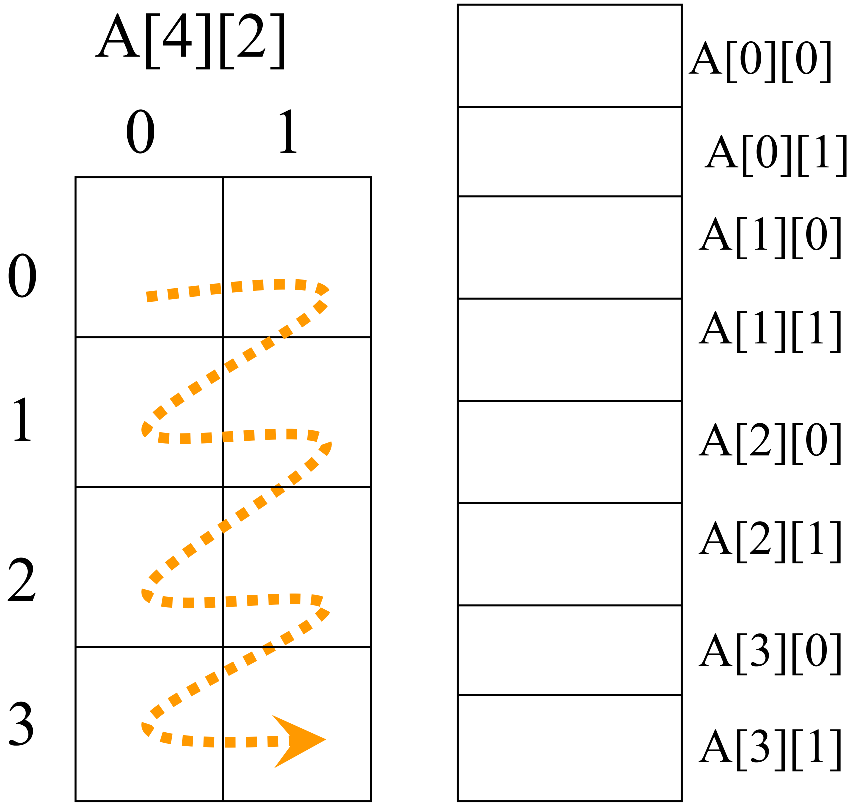
Shankar Balachandran, IIT Madras

Multidimensional Arrays

- Many times, data come in the form of tables
  - Spreadsheets etc.
- Matrices are common in several engineering disciplines
- Spreadsheets and matrices can be thought of as 2D arrays
- More than 2 dimensions are also common
  - Example: 3D graphics

Two Dimensional Arrays

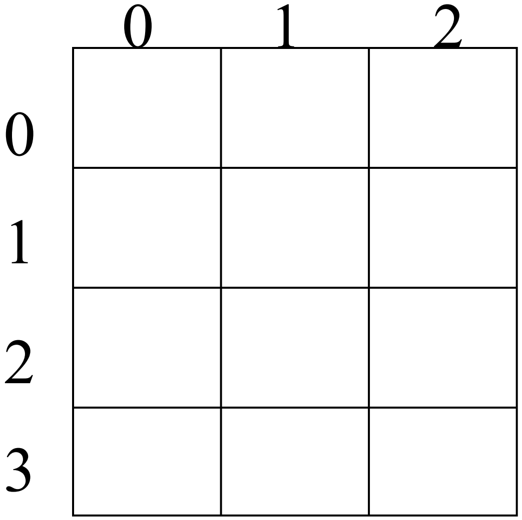
- int A[4][2];
  - 4 x 2 array; 4 rows, 2 columns
- Storage
  - Row major order
- Initialization
  - int B[2][3] = { {4, 5, 6},
                     {0, 3, 5}
                   };



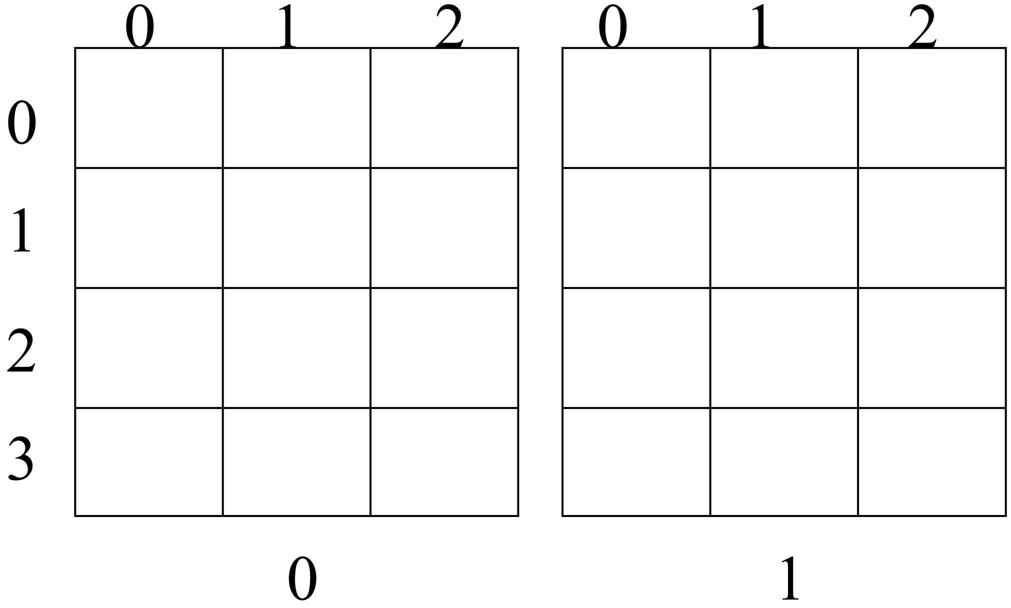
Multi-dimensional Arrays

Arrays with two or more dimensions can be defined

int A[4][3];



float B[2][4][3];



Matrix Operations

An m-by-n matrix: M: m rows and n columns

Rows : 0, 1, … , m-1 and Columns : 0,1, … , n-1

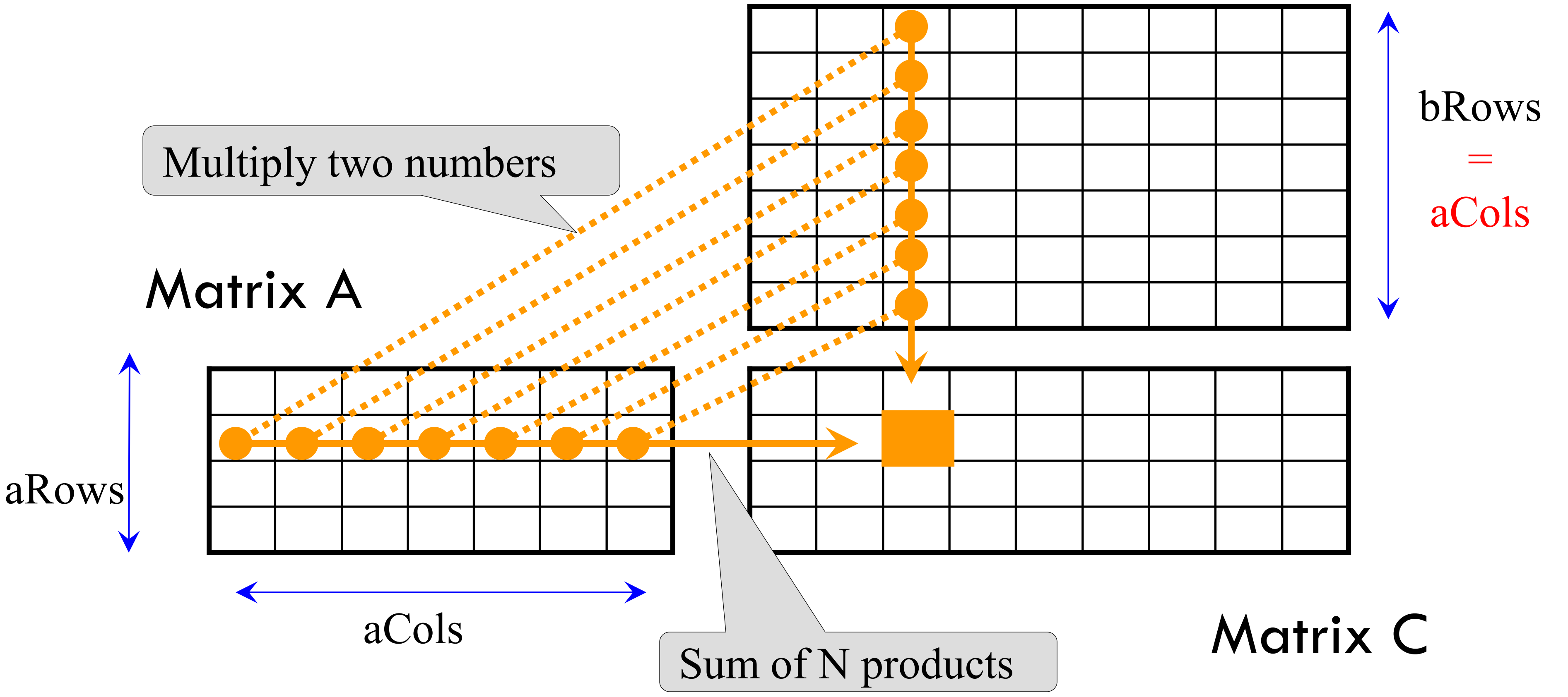
M[i][j] : element in i^{th} row, j^{th} column

$0 \le i < m, 0 \le j < n$

Filling Values Into and Printing a 2D Array

- Demo

Matrix multiplication



Using Matrix Operations

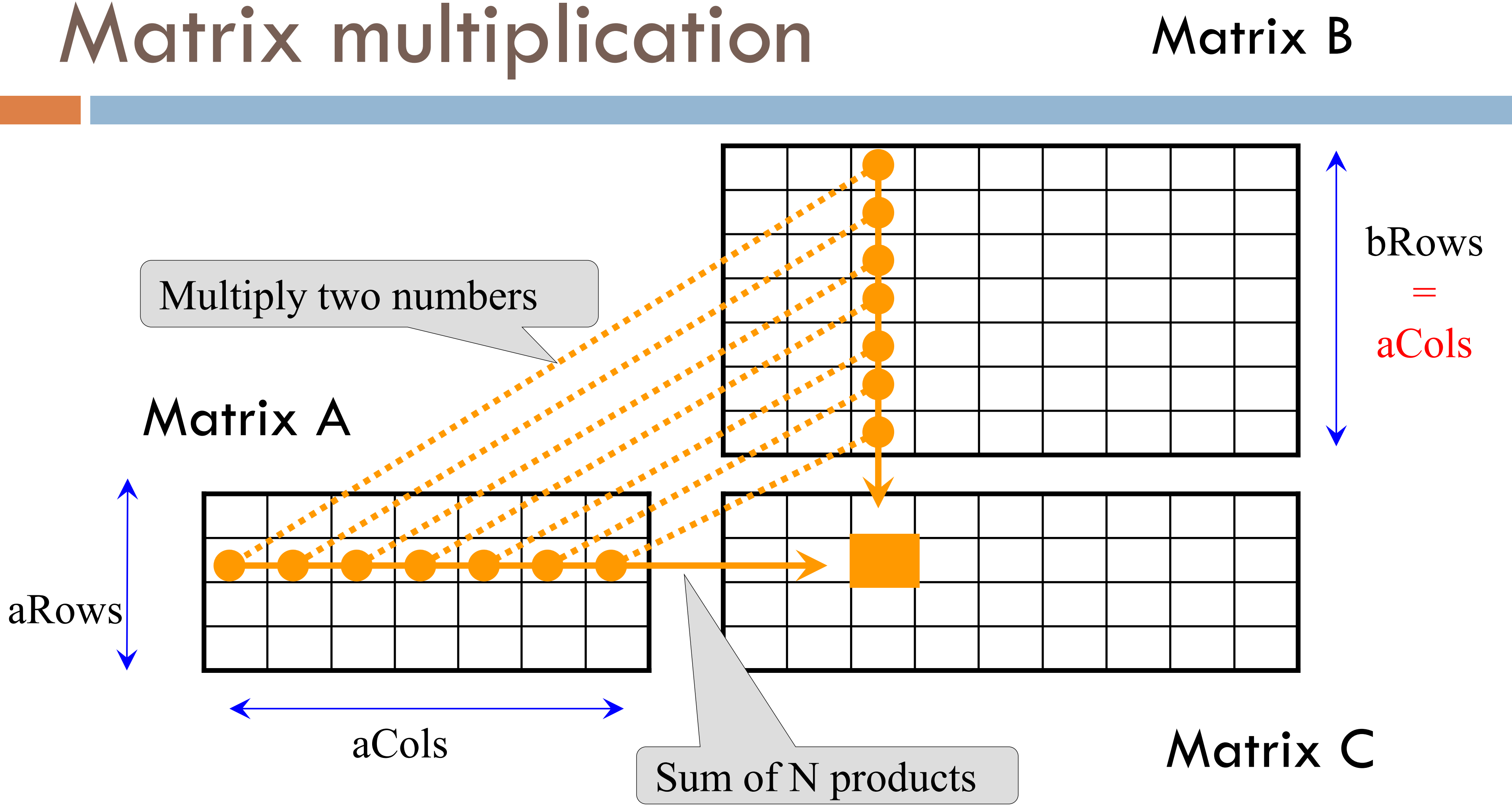
```
main(){
    int a[10][10], b[10][10], c[10][10];   / * max size 10 by 10 */
    int aRows, aCols, bRows, bCols, cRows, cCols;
    int i, j, k;
    scanf("%d%d", &aRows, &aCols);
    for(int i = 0; i < aRows; i++)
          for(int j = 0; j < aCols; j++)
                scanf("%d", &a[i][j]);
    scanf("%d%d", &bRows, &bCols);
    for(int i = 0; i < bRows; i++)
          for(int j = 0; j < bCols; j++)
                scanf("%d", &b[i][j]);
```

Remember bRows=aCols; Validate user input if you desire to

Using Matrix Operations

```
    cRows = aRows; cCols = bCols;
    for(int i = 0; i < cRows; i++)
            for(int j = 0; j < cCols; j++) {
                c[i][j]=0;
                for(int k = 0; k <  aCols; k++)
                        c[i][j] += a[i][k]*b[k][j];
            }
    for(int i = 0; i < cRows; i++){
            for(int j = 0; j < cCols;  j++)     /* print a row */
                 printf("%d  ", c[i][j]);    /* notice missing \n */
            printf("\n");                    /* print a newline at the end a row */
    }
}
```

Matrix multiplication



End of Module

PROGRAMMING

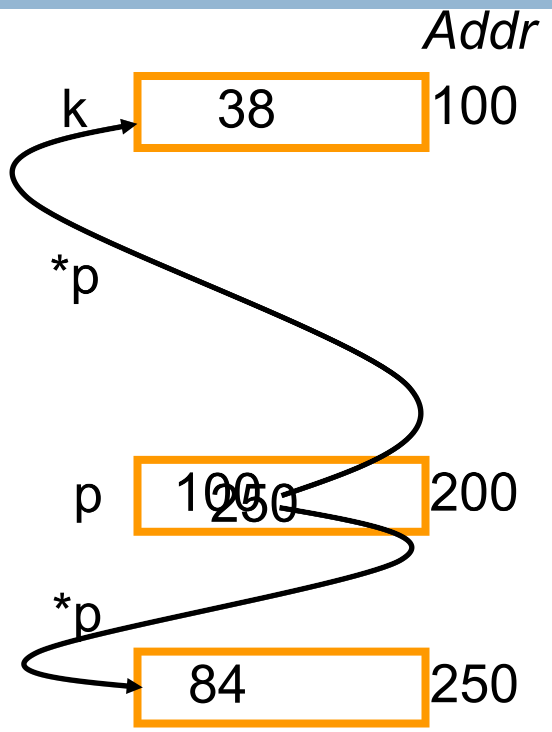
WEEK 2

MODULE 4: POINTERS

Shankar Balachandran, IIT Madras

What is a pointer?

- *Recap:* a variable `int k`
  - Names a memory location that can hold one value at a time
- A pointer variable: **`int *p`**
  - Contains the address of a memory location that contains the actual value
  - Can only hold one address at a time
    - Because it is a variable
  - Can point to different addresses at different times



l-value and r-value

- Given a variable k
- Its l-value refers to the *address* of the memory location
  - l-value is used on the left side of an assignment

    k = expression
- its r-value refers to the *value* stored in the memory location
  - r-value is used in the right hand side of an assignment

    var = k + ...
- pointers allow one to manipulate the l-value!

Pointers

- Pointers are themselves *variables* that store the *address* of a memory location
- The memory required by a pointer depends upon the size of the memory in the machine
  - one byte could address a memory of 256 locations
  - two bytes can address a memory of 64K locations
  - four bytes can address a memory of 4G locations
  - modern machines have RAM of 1GB or more…
- The task of allocating this memory is best left to the system

Declaring Pointers

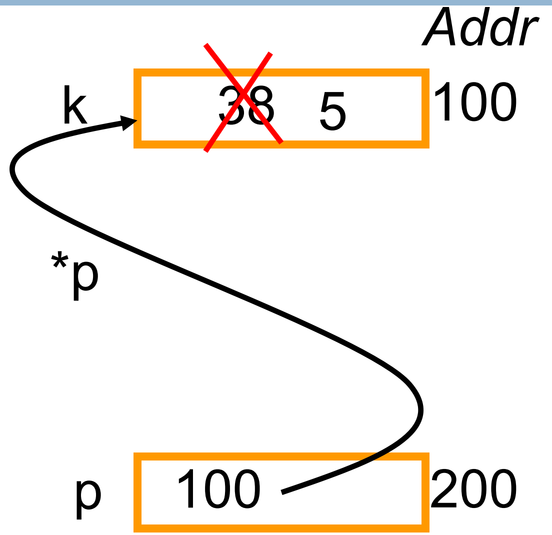
- Pointer variable – precede its name with an asterisk
- Pointer type - the type of data stored at the address we will be storing in our pointer. For example,

```
int *p;
```

- **p** is the *name* of the variable
  - The '*' informs the compiler that we want a *pointer variable*
  - Sets aside however many bytes is required to *store an address* in memory.
- The **int** says that we intend to use our pointer to point to an integer value

Contents of Pointer Variables

```
int k=38;
int *p;
p = &k;
*p = 5;
```



Example: pointers

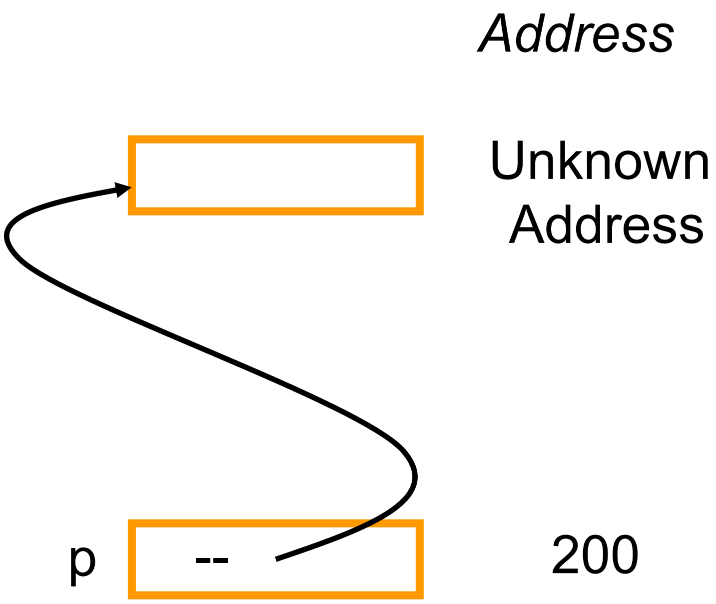
```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
        int a = 10, b = 5;
        int *ip;
        ip = &a;
        printf ("a = %d, ip = %p, *ip = %d\n", a, ip, *ip);
        *ip=4;
        printf ("a = %d, ip = %p, *ip = %d\n", a, ip, *ip);

        ip = &b;
        printf ("b = %d, ip = %p, *ip = %d\n", b, ip, *ip);
        return 0;
}
```

Memory Allocated to Values

- Declaring a pointer does not allocate memory for the value.

```
int *p;
*p = 4;
```

- & p = 200
- p is unknown
- *p is illegal



End of Module

PROGRAMMING

WEEK 2

MODULE 5: MORE ON POINTERS

Shankar Balachandran, IIT Madras

Contents of Pointer Variables

```
int k=38;
int *p;
p = &k;
*p = 5;
```

- * is used for two things (besides multiplication)
  - When you declare a pointer
  - To "dereference" the pointer
    - To get the rvalue
- p is said to "point to" k

Dereferencing operator

- The "dereferencing operator" is the asterisk and it is used as follows:

```
*p= 7;
```

  - will copy 7 to the address pointed to by **p**. Thus if **p** "points to" **k**, the above statement will set the *value of **k** to 7*.
- Using '*' is a way of referring to the value of that which **p** is pointing to, not the value of the pointer itself.
- `printf("%d\n",*p);`
  - prints the number 7

NULL pointers

- Values of a pointer variable:
  - Usually the value of a pointer variable is a pointer to some other variable
- A *null pointer* is a special pointer value that is known not to point anywhere.
- No other valid pointer, to any other variable, will ever compare equal to a null pointer !

NULL Pointers

- Predefined constant NULL, defined in <stdio.h>
- Good practice: test for a null pointer before inspecting the value pointed !

```
#include <stdio.h>

int *ip = NULL;

ip = …

if(ip != NULL)  printf("%d\n", *ip);
```

or

```
if(ip )  printf("%d\n", *ip);
```

Pointer types

- C provides for a pointer of type void. We can declare such a pointer by writing:

  ```
  void *vptr;
  ```

- A void pointer is a *generic* pointer. For example, a pointer to *any* type can be compared to a void pointer
- Typecasts can be used to convert from one type of pointer to another under the proper circumstances

Trying out pointers

```
#include <stdio.h>
int main(void) {
  int m=1, k=2, *ptr;
  ptr = &k;
  printf("\n");
  printf("m has the value %d and is stored at %p\n", m, (void *)&m);
  printf("k has the value %d and is stored at %p\n", k, (void *)&k);
  printf("ptr has the value %p stored at %p\n", ptr, (void *)&ptr);
  printf("The value of the integer pointed to by ptr is %d\n", *ptr);
  return 0;
}
```

Generic address of j

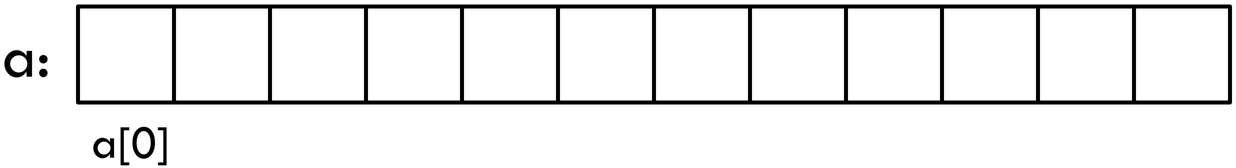
Dereferencing – will print r-value of k

Pointers and arrays

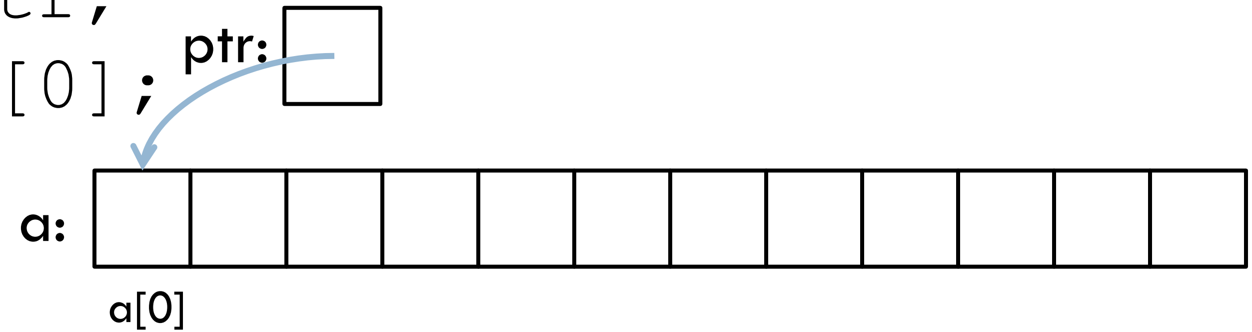
- In C, there is a strong relationship between pointers and arrays
- Any operation that can be achieved by array subscripting can also be done with pointers

Pointers and Arrays


```
int a[12];
```

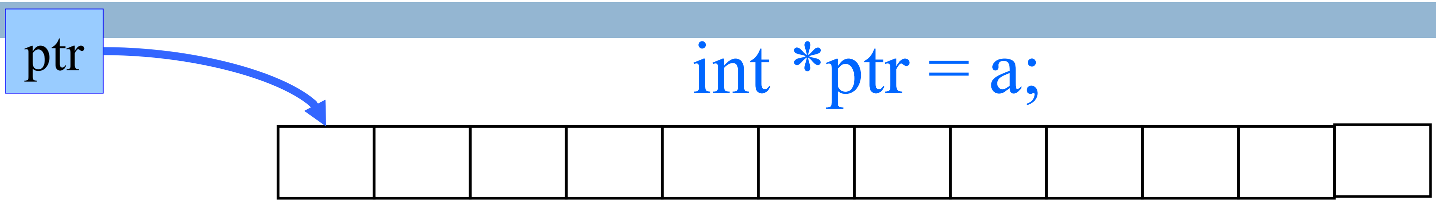



```
int *ptr;
ptr=&a[0];
Or
ptr=a;
```

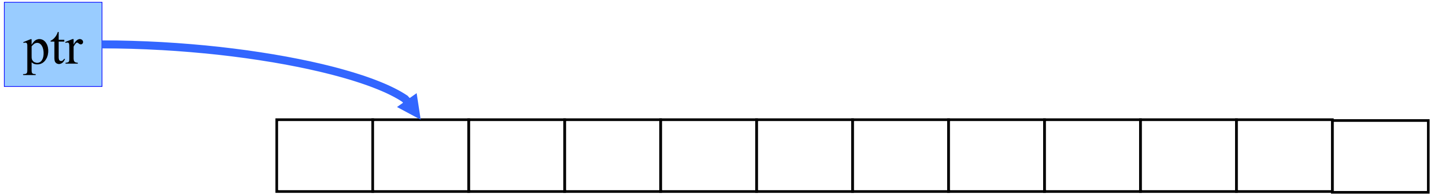


The name of an array is a synonym for the address of the 0^{th} element.

Pointer arithmetic



- ptr = ptr +1;
  - says to point to the *next* data item after this one



*ptr = *ptr + 1;

What does this do?

Arrays are **constant** pointers

```
int a[10];
int *pa;

pa=a;

pa++;
```

```
int a[10];
int *pa;

a=pa;

a++;
```

OK. Pointers are variables that can be assigned or incremented

Error!!!

The name of an array is a CONSTANT with the value as the location of the first element.
You cannot change the address where the array is stored !
An array's name is equivalent to a ***constant*** pointer

Accessing Arrays with Pointers

```
#include <stdio.h>
int main(void)
{
     int myArray[] = {1,23,17,4,-5,100}, *ptr,  i;
     ptr = &myArray[0];     /* point our pointer to the first element of the array */
    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
        printf("myArray[%d] = %d ", i, myArray[i]);     /*<-- A */
        printf("Contents in address ptr + %d = %d\n", i, *(ptr + i));     /*<-- B */
    }
  return 0;
}
```

ptr++ and ++ptr

- **++ptr** and **ptr++** are both equivalent to **ptr + 1**
  - though they are "incremented" at different times
- Change line B to read:

```
printf("ptr + %d = %d\n",i, *ptr++);
```

- and run it again... then change it to:

```
printf("ptr + %d = %d\n",i, *(++ptr));
```

sizeof( ) operator

```
#include <stdio.h>
main()
{ int int_size;
  int chr_size;
  int flt_size;
  int_size = sizeof(int); chr_size =sizeof(char);
  flt_size = sizeof(float);
  printf(“int, char, and float use %d %d and %d bytes\n”,
    int_size, chr_size; flt_size);
}
```

Pointer Arithmetic

- *Valid pointer operations:*
  - Assignment between pointers of the same type
  - Addition/ subtraction between a pointer and an integer
  - Comparison between two pointers that point to elements of the same array
  - Subtraction between two pointers that point to elements of the same array
  - Assignment or comparison with zero (NULL)

Pointer Arithmetic – Increment/Decrement

- **Increment/decrement:** if *p is a pointer to type T, p++ increases the value of p* by *sizeof(T) (sizeof(T) is the amount of storage needed for an object of type T)*. Similarly, *p--* decreases *p* by *sizeof(T);*

```
T tab[N];
T * p;
int i=4;
p=&tab[i];
p++;   // p contains the address of tab[i+1];
```

Add/Subtract Integers from Pointers

- **Addition/subtraction with an integer:** if *p is a pointer to type T* and *n* an integer, *p+n increases the value of p* by *n*sizeof(T)*.
- Similarly, *p-n* decreases *p* by *n*sizeof(T);*

```
T tab[100];
T *p;
p=tab;
p=p+5;    // p contains the address of tab[5].
```

Comparing Pointers

- ***If p and q point to members of the same array***, then relations like ==, !=, <, >=, etc., work properly.
  - For example, p < q is true if p points to an earlier element of the array than q does.
- Any pointer can be meaningfully compared for equality or inequality with zero.

Example: Pointer Subtraction

```
/* strlen: return length of string s */
int strlen(char *s)
{
   char *p = s;
   while (*p != '\0')
      p++;
   return p - s;
}
```

End of Module

PROGRAMMING

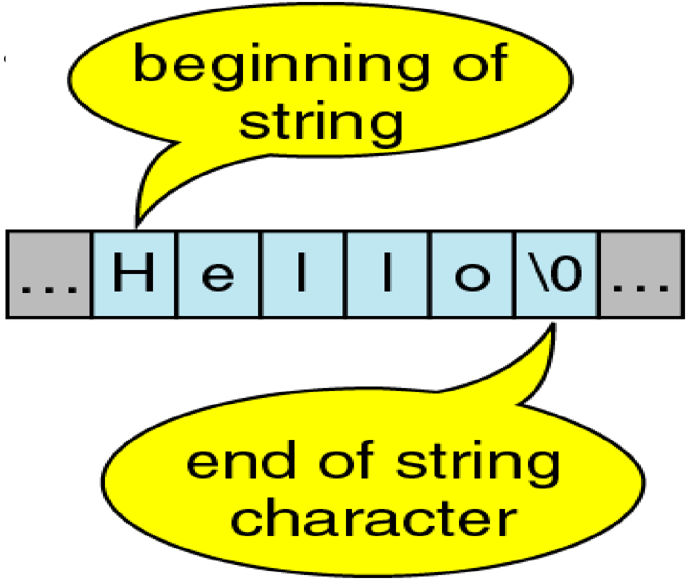
WEEK 2

MODULE 6: STRINGS

Shankar Balachandran, IIT Madras

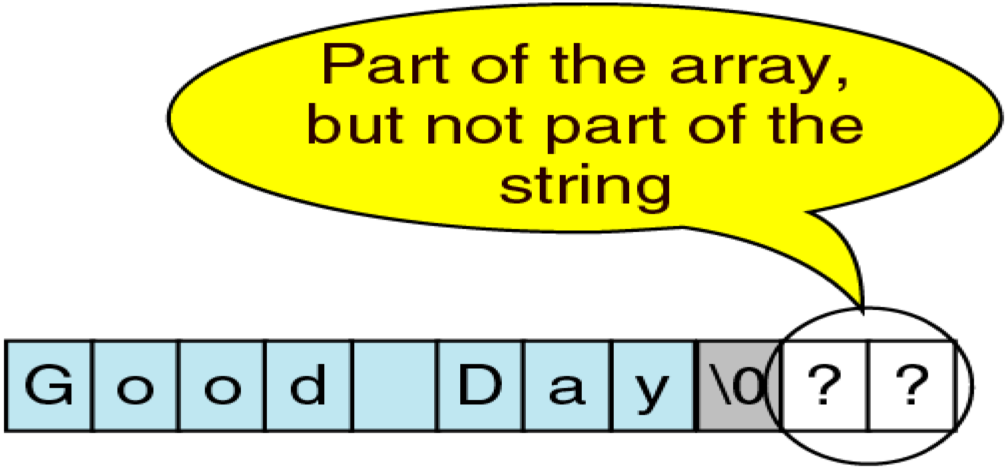
Strings

- A sequence of characters is often referred to as a character "string".
- A string is stored in an array of type `char` ending with the null character '\0 '.

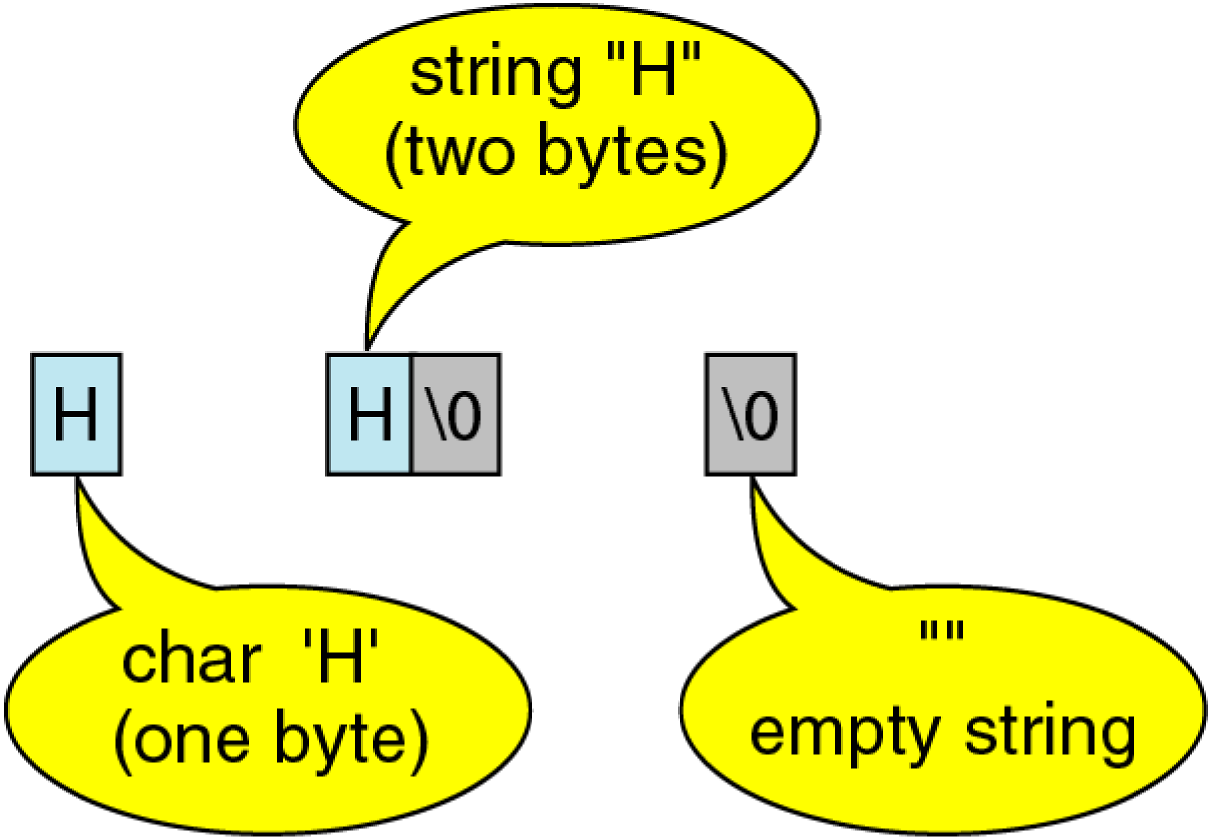


Strings

`char str[11];`



Character vs. Strings



Character vs. String

- A string constant is a sequence of characters enclosed in double quotes.
  - For example, the character string:

```
char s1[2]="a"; //Takes two bytes of storage.
```

s1:

a	\0

  - On the other hand, the character, in single quotes:

```
char s2= 'a'; //Takes only one byte of storage.
```

s2:

a

Example 1

```
char message1[12] = "Hello world";
```

message1:

H	e	l	l	o		w	o	r	l	d	\0

```
char message2[12];
scanf(“%s”,messsage2);    // type "Hello” as input
```

message2:

H	e	l	l	o	\0	?	?	?	?	?	?

Initializing Strings

```
char *message3 = "Hello world";
printf (“%s”, message3);
```

H	e	l	l	o		w	o	r	l	d	\0

- message3 is a pointer to an array of characters
- Contents of message3 should not be changed
  - message3 points to a sequence of locations that are “read-only” portion of the program that is executing
  - message3[1] = ‘a’; //undefined behavior

Sample Code


```
int main() {
        char *a1 = "Hello World";
        char a2[] = "Hello World";
        char a3[6] = "World";
        printf("%d %d\n", sizeof(a1), sizeof(a2) );
        a1[1] = 'u'; //undefined behavior
        a1 = a2;
        printf("%s",a1);
        a2 = a3;//error
}
```

Pointer to constant string

Constant pointer to string

Reading Strings

- scanf (“%s”, pointer_to_char_array);

```
char A_string[80], E_string[80];
printf("Enter some words in a string:\n“);
scanf(“%s%s”,A_string, E_string);
printf(“%s%s”,A_string, E_string);
```

Output:

```
Enter some words in a string:
This is a test.
Thisis
```

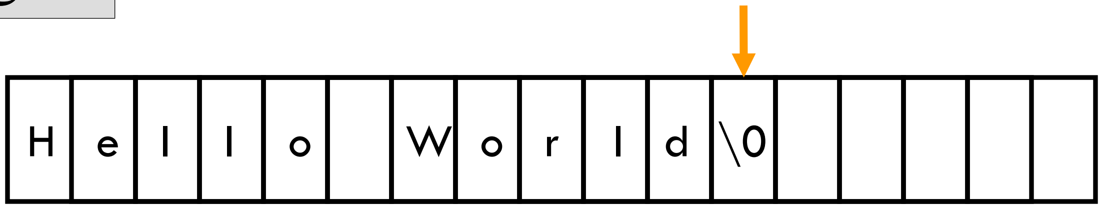
Functions to Handle Strings

- String is a non-basic data type
  - Constructed data type
  - Requires functions to handle
  - Regular datatypes have operators in C directly
- Typical functions on strings are:
  - Length of a string
  - Are two strings equal?
  - Does a given pattern occur as a substring?
  - Concatenate two strings and return the result
- These functions are provided as a library
  - string.h
  - We will see how to write some of these functions

String Length

Find length of string A

A

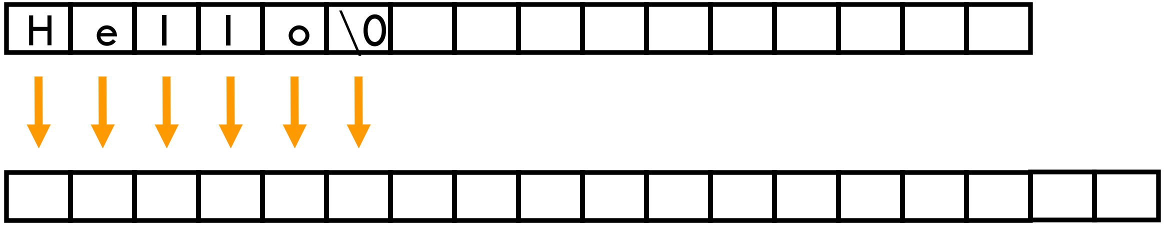



```
int stringLength(char *A) {
        int i = 0;
        while (A[i] != '\0')  i++;
        return i;
}
```

String Copy

Copy array A to B

A

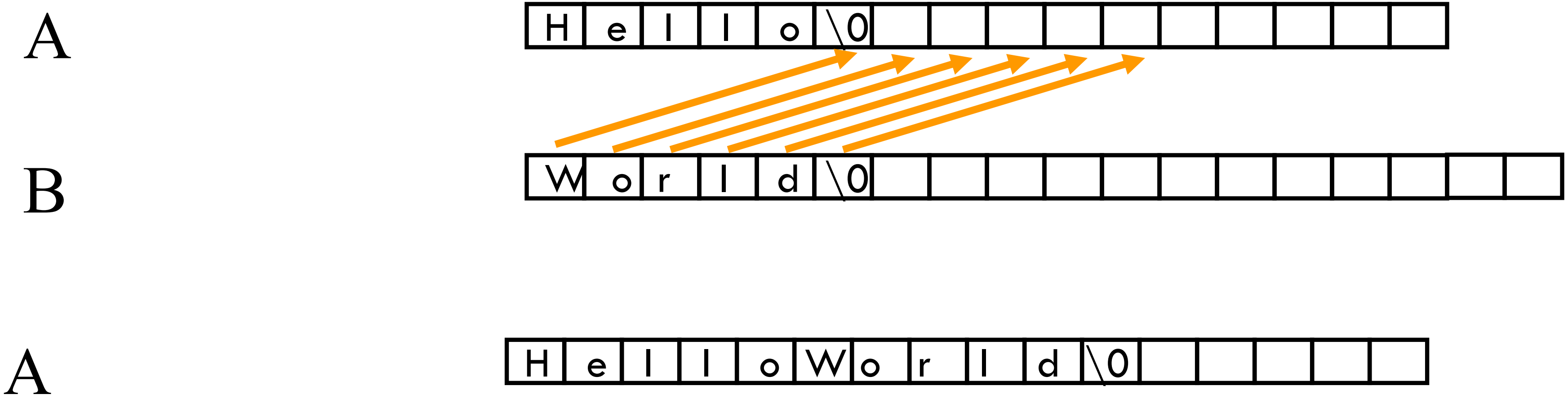


B

```
void stringCopy (char *A, char *B) {
  int N1  = stringLength(A);
  for (int k=0; k<=N1; k++)
                    B[k] = A[k];
}
```

String Concatenation

Concatenate array B to A



Exercise for you!

Lexicographic Ordering

- Badri < Devendra
- Janak < Janaki
- Shiva < Shivendra
- Seeta < Sita
- Badri < badri
- Bad < Badri

upper case before lower case

- Based on the ordering of characters

A < B ... < Y < Z < a < b < c < . . . < y < z

Lexicographic ordering

- What about blanks?
  - “Bill Clinton” < “Bill Gates”
  - “Ram Subramanian” < “Ram Subramanium”
  - “Ram Subramanian” < “Rama Awasthi”
- In ASCII the blank (code = 32) comes before all other characters. The above cases are taken care of automatically.

String Comparison

```
int strCompare(char *A, char *B, int N1, int N2) {
        int k=0;
        while ((A[k] == B [k]) && k<N1  && k<N2)
                k++;
        if(N1  == N2 && k == N1) printf("A = B");
        else if (A[k] == '\0') printf("A < B");
        else if (B[k] == '\0') printf("A > B");
        else if (A[k] < B[k]) printf("A < B");
         else printf("A > B");
}
```

Examples

A = "Hello" B = "Hello"

A = "Hell" B = "Hello"

A = "Hello" B = "Hell"

A = "Bell" B = "Bull"

A = "Hull" B = "Hello"

Built-in string functions

- #include <string.h>
- ***int strlen(const char* s)*** - strlen returns the length of a string, excluding the NUL character.
- ***char* strcpy(char* dest, char* src)*** - strcpy copies one string to another. The destination must be large enough to accept the contents of the source string.
- ***char* strcat(char* dest, char* src)*** - strcat combines two strings and returns a pointer to the destination string. In order for this function to work, you **must** have enough room in the destination to accommodate both strings.

Built-in string comparison

- int strcmp(const char *s1, const char *s2);
- int str**n**cmp(const char *s1, const char *s2, size_t n);

Compares first n characters only

The return values are

- 0 if both strings are equal.
- 1 if first string is lexicographically greater than second.
- -1 if second string is lexicographically greater than first.

End of Week 2