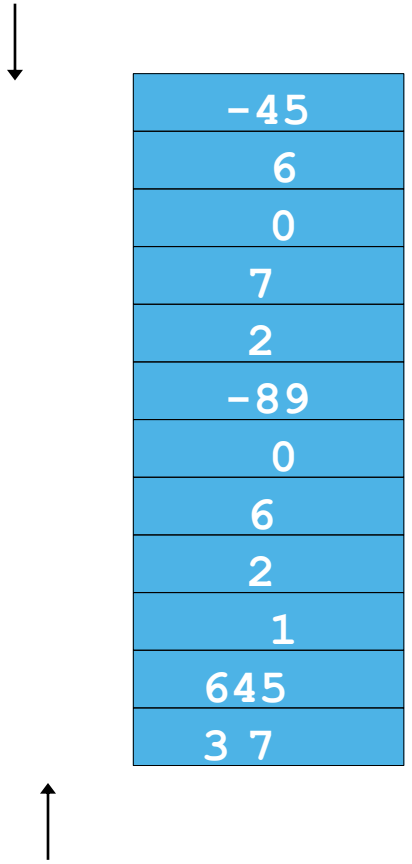
Programming with C - Arrays & Pointers

P S V S Sridhar

Assistant Professor, Centre for Information Technology, University of Petroleum & Energy Studies, Dehradun

- A collection of objects of the *same type* stored contiguously in memory under one name
  - Group of consecutive memory locations
  - Same name and type
- To refer to an element, specify
  - Array name
  - Position number
- Format:

  arrayname **[** *position number* **]**
  - First element at position **0**
  - **n** element array named **c**:
    - **c[0], c[1]...c[n – 1]**



of the element within array **c**

Examples

- **`int A[10]`**
  - An array of ten integers
  - **`A[0]`**, **`A[1]`**, …, **`A[9]`**
- **`double B[20]`**
  - An array of twenty long floating point numbers
  - **`B[0]`**, **`B[1]`**, …, **`B[19]`**
- Arrays of **`structs`**, **`unions`**, **`pointers`**, etc., are also allowed
- Array indexes *always* start at zero in *C*

Examples (continued)

- **`int C[]`**
  - An array of an unknown number of integers (allowable in a parameter of a function)
  - `C[0]`, `C[1]`, …, `C[`*max-1*`]`
- **`int D[10][20]`**
  - An array of ten rows, each of which is an array of twenty integers
  - `D[0][0]`, `D[0][1]`, …, `D[1][0]`, `D[1][1]`, …, `D[9][19]`

Array Element

- May be used wherever a variable of the same type may be used
  - In an expression (including arguments)
  - On left side of assignment

- Examples:–

```
A[3] = x + y;
x = y - A[3];
z = sin(A[i]) + cos(B[j]);
```

Array Elements (continued)

- Generic form:–
  - *ArrayName*[*integer-expression*]
  - *ArrayName*[*integer-expression*] [*integer-expression*]
    - Same type as the underlying type of the array
- Definition:– *Array Index* – the expression between the square brackets

Array Elements (continued)

- Array elements are commonly used in loops
- E.g.,

```
for(i=0; i < max; i++)
  A[i] = i*i;

sum = 0; for(j=0; j < max; j++)
  sum += B[j];
```

Declaring Arrays

- Static or automatic
- Array size determined explicitly or implicitly
- Array size may be determined at run-time
  - Automatic only

Declaring Arrays (continued)

- Outside of any function – always static

```
int A[13];


#define CLASS_SIZE 73
double B[CLASS_SIZE];


const int nElements = 25
float C[nElements];
```

Array Initialization

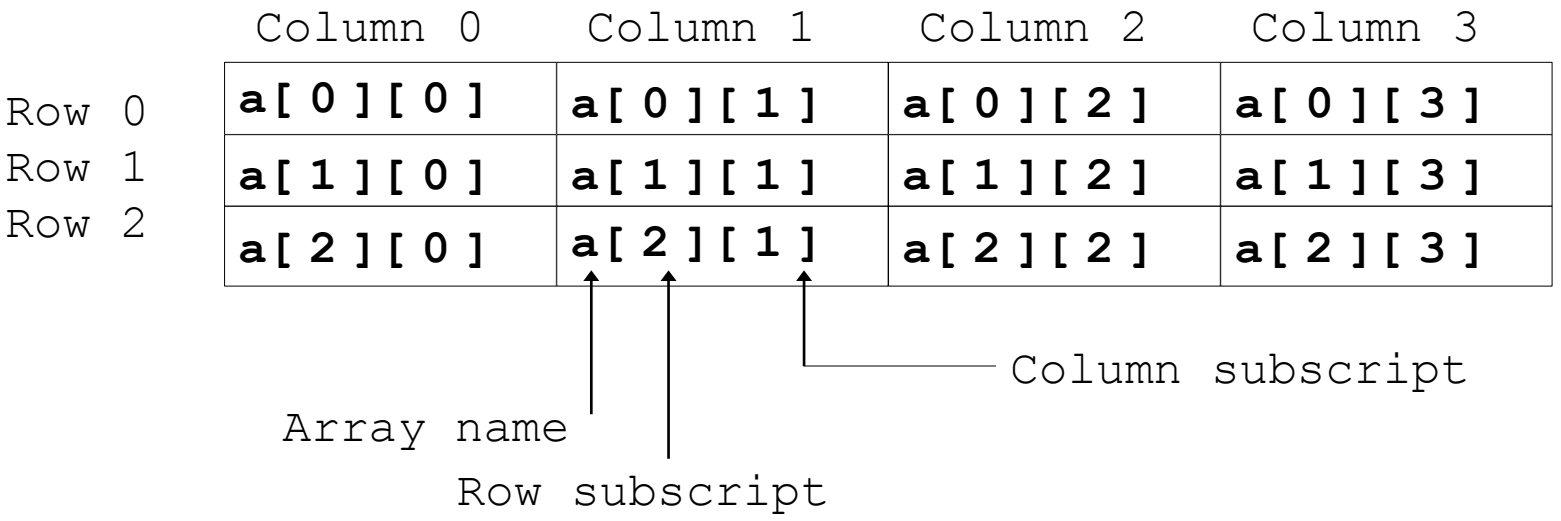
- `int A[5] = {2, 4, 8, 16, 32};`
  - Static or automatic
- `int B[20] = {2, 4, 8, 16, 32};`
  - Unspecified elements are guaranteed to be zero
- `int C[4] = {2, 4, 8, 16, 32};`
  - Error — compiler detects too many initial values
- `int D[5] = {2*n, 4*n, 8*n, 16*n, 32*n};`
  - Automatically only; array initialized to expressions
- `int E[n] = {1};`
  - Dynamically allocated array (automatic only). Zeroth element initialized to *1*; all other elements initialized to *0*

Implicit Array Size Determination

- `int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};`
  - Array is created with as many elements as initial values
    - In this case, 12 elements

Multiple-Subscripted Arrays

- Multiple subscripted arrays
  - Tables with rows and columns (**m** by **n** array)
  - Like matrices: specify row, then column



6.9 Multiple-Subscripted Arrays

- Initialization

```
1  2
3  4
```

  - `int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`
  - Initializers grouped by row in braces
  - If not enough, unspecified elements set to zero

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

```
1  0
3  4
```

- Referencing elements
  - Specify row, then column

```
printf( "%d", b[ 0 ][ 1 ] );
```

Character Arrays

When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant.

Char string[3] = “xyz”;

- Character arrays
  - String “**first**” is really a static array of characters
  - Character arrays can be initialized using string literals

```
char string1[] = "first";
```

    - Null character '\0' terminates strings
    - **string1** actually has 6 elements
      - It is equivalent to

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

  - Can access individual characters

string1[3] is character ‘s’

Reading Strings from Terminal

- **Using Scanf**

```
char address[10];
scanf("%s", address);
```

Terminates its input on the first white space

```
scanf("%ws", name);        // w is width of  word

scanf("%[^\n]",line);      // accept white spaces also

char ch;
ch = getchar();        // accept single character
```

Reading Strings from Terminal

- main()
- char line[81], character;
- int c=0;
- do
- {
- character = getchar();
- line[c] = character;
- c++;
- } while (character != ‘\n’);

Reading Strings from Terminal

- gets(str); //accept string until a new line character is encountered
- Eg. char line[80];
- gets (line);
- printf(“%s”,line);

What is a pointer variable?

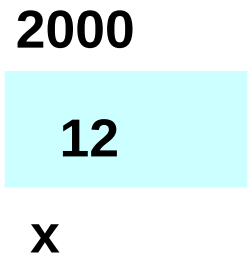
- A pointer variable is a variable whose value is the address of a location in memory.
- To declare a pointer variable, you must specify the type of value that the pointer will point to, for example,

```
int*   ptr; // ptr will hold the address of an int

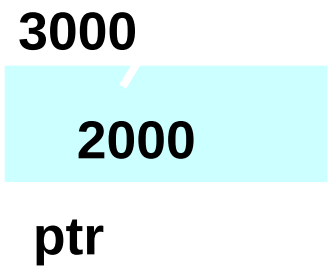
char*  q;   // q will hold the address of a char
```

Using a Pointer Variable

```
int  x;
x = 12;
```



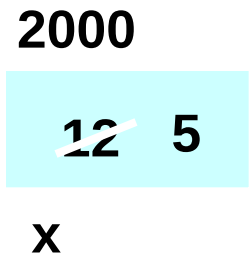

```
int*  ptr;
ptr = &x;
```



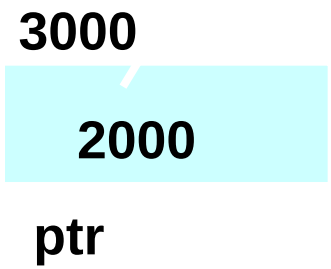
NOTE: Because ptr holds the address of x, we say that ptr "points to" x

Using the Dereference Operator

```
int  x;
x = 12;
```




```
int*  ptr;
ptr = &x;
```




```
*ptr = 5;
```

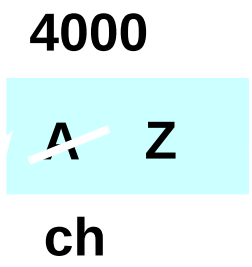
```
// changes the value at the
   address ptr points to 5
```

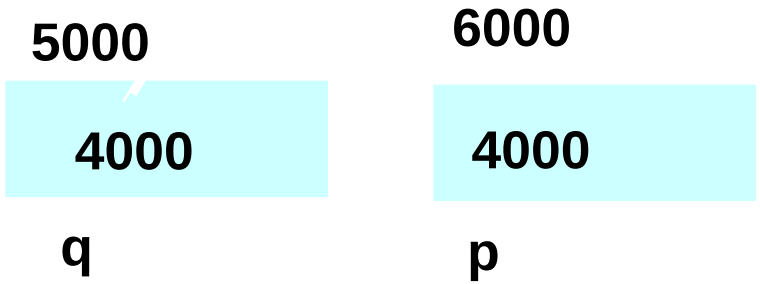
Self –Test on Pointers

```
char   ch;
ch =   'A';

char*   q;
q   = &ch;

*q = 'Z';
char*   p;
p = q;
```



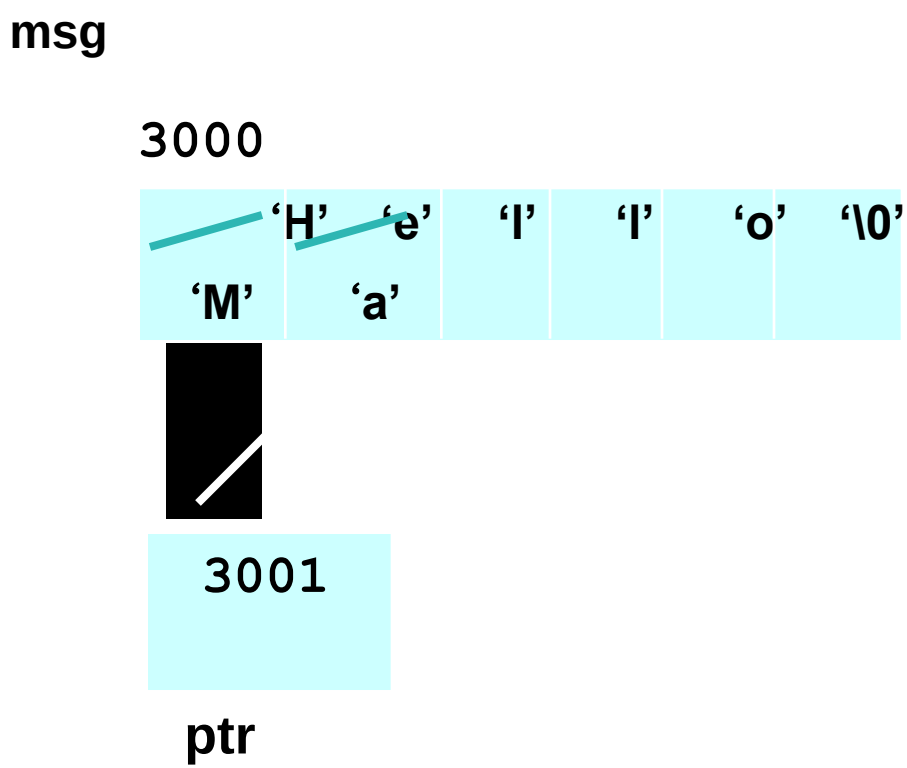



```
// the rhs has value 4000

// now p and q both point to ch
```

Pointers into Arrays

```
char   msg[ ]
   =“Hello”;
char*  ptr;
ptr  =  msg;
*ptr  = ‘M’ ;
ptr++;
*ptr = ‘a’;
```



Pointers and Constants

- char s[] = “Hello”;
- const char* pc = s; // pointers to constant
- pc[3] = ‘g’; // error
- pc = p; // ok

Declaring Pointers in *C*

- **`int *p;`** — a pointer to an **`int`**
- **`double *q;`** — a pointer to a **`double`**
- **`char **r;`** — a pointer to a pointer to a **`char`**
- *type* **`*s;`** — a pointer to an object of type *type*

Pointer Arithmetic

- `int *p, *q;`
  `q = p + 1;`
  - Construct a pointer to the next *integer* after `*p` and assign it to `q`
- `long int *p, *q;`
  `p++; q--;`
  - Increment `p` to point to the next `long int`; decrement `q` to point to the previous `long int`
- `float *p, *q;`
  `int n;`
  `n = p - q;`
  - `n` is the number of floats between `*p` and `*q`; i.e., what would be added to `q` to get `p`

Pointers & arrays

- Arrays and pointers are *closely related* in *C*
  - In fact, they are essentially the same thing!
  - Esp. when used as parameters of functions
- `int A[10];`
  `int *p;`
  - *Type* of `A` is `int *`
  - `p = A;` and `A = p;` are legal assignments
  - `*p` refers to `A[0]`
    `*(p + n)` refers to `A[n]`
  - `p = &A[5];` is the same as `p = A + 5;`

Arrays and Pointers (continued)

- **`double A[10];`** *vs.* **`double *A;`**
- *Only* difference:–
  - **`double A[10]`** sets aside *ten* units of memory, each large enough to hold a **`double`**
  - **`double *A`** sets aside *one* pointer-sized unit of memory
    - You are expected to come up with the memory elsewhere!
  - Note:– all pointer variables are the same size in any given machine architecture
    - Regardless of what types they point to

Note

- *C* does *not* assign arrays to each other
- *E.g,*
  - ```
    double A[10];
    double B[10];

    A = B;
    ```
    - assigns the pointer value **B** to the pointer value **A**
    - Contents of array **A** are untouched
    ```

This method called as call by reference

```
#include<stdio.h> /*The function calls is Call by Reference*/
#define pi 3.14
void area_perimeter(float, float *, float *);
int main( )
{
  float r, a, p;
  printf("Enter the radius\n");
  scanf(""%f",&r);
  area_perimeter(r,&a,&p);
  printf("The area = %8.2f, \n The Perimeter = %8.2f", a, p);
  return 0;
}
```

Pointer & Functions

```
void area_perimeter(float x, float *aptr, float
  *pptr)
{
  *aptr = pi*x*x;
  *pptr = 2.0*pi*x;
}
```

Passing an Arrays to Function

- When an array is passed to a function what is actually passed is its initial elements location in memory. i.e., the address of an initial element

```
main()
{
    char str[20];
    int len;
    gets(str);
    len = stringlen(str);
    printf("%d", len);
}
int  stringlen(char *s)
{   char *p = s;
    while( *p != '\0')
                p++;
    return p-s;
}
```

Dynamic Memory Allocation (Malloc, Calloc, Sizeof, Free)

- C provide the concept of dynamic memory allocation to allocate memory space for variable at run time.
- C allows users to dynamically allocate memory by malloc() and calloc() functions
- sizeof() which determines how much memory a specified variable
- free() de-allocates the memory assigned to a variable

sizeof()

```
struct date
{
int hour;
int minute;
int second;
};
int x;
x = sizeof(struct date);
```

x now contains the information required by calloc()

Malloc()

- This function is used to allocate storage to a variable while the program is running.
- This function takes an argument that specifies the size of each element in bytes.
- The function returns a character pointer to the allocated storage, which is initialized to zero

struct date *date_pointer;

date_pointer = (struct date *) malloc sizeof(struct date);

The (struct date *) is a type cast operator which converts the pointer returned from malloc to a character pointer to a structure of type date.

Calloc()

- This function is also used to allocate storage to a variable while the program is running.
- This function takes two arguments that specify the number of elements to be reserved.
- The size of each element obtained from sizeof()
- The function returns a character pointer to the allocated storage, which is initialized to zero

struct date *date_pointer;

date_pointer = (struct date *) calloc(20, sizeof(struct date);

The above function call will allocate size for twenty such structures, and date_pointer will point to first in the chain.

Free()

- When variables are no longer required, the memory will be released by free()
- free(date_pointer);

End