3.1 Strings

Objectives

- To understand design concepts for fixed-length and variable-length strings
- To understand the design implementation for C-language delimited strings
- To write programs that read, write, and manipulate strings.

Agenda

- Introduction
- Strings in C
    - Storing Strings
    - The String Delimiter
    - String Literals
    - Declaring Strings
    - Initializing Strings
    - Strings and the Assignment Operator
- String Input/Output Functions
    - Formatted String Input
        - String conversion code
        - Scan set conversion code
    - Formatted String Output
    - String-only Input
    - String-only Output

Recap -- data type ‘char’

- The domain of the data type char is the set of symbols that can be displayed on the screen or typed on the keyboard.
  - These symbols : the letters, digits, punctuation marks, spacebar, Return key, and so forth—are the building blocks for all text data.
- char is a scalar type and are stored as ASCII code,
- set of operations available for characters is the same as that for integers
  - Adding an integer to a character
  - Subtracting an integer from a character
  - Subtracting one character from another
  - Comparing two characters against each other

Introduction

- A string is a sequence of characters treated as a group
- What are the primitive operations that you might want to perform on strings?

  To begin with, you need to

  - Specify a string constant in a program
  - Read in a string form the user by using GetLine
  - Display a string on the screen by using print
  - Determine whether two strings are exactly equal by using StringEqual

Introduction(contd..)

- What else might you want to do?

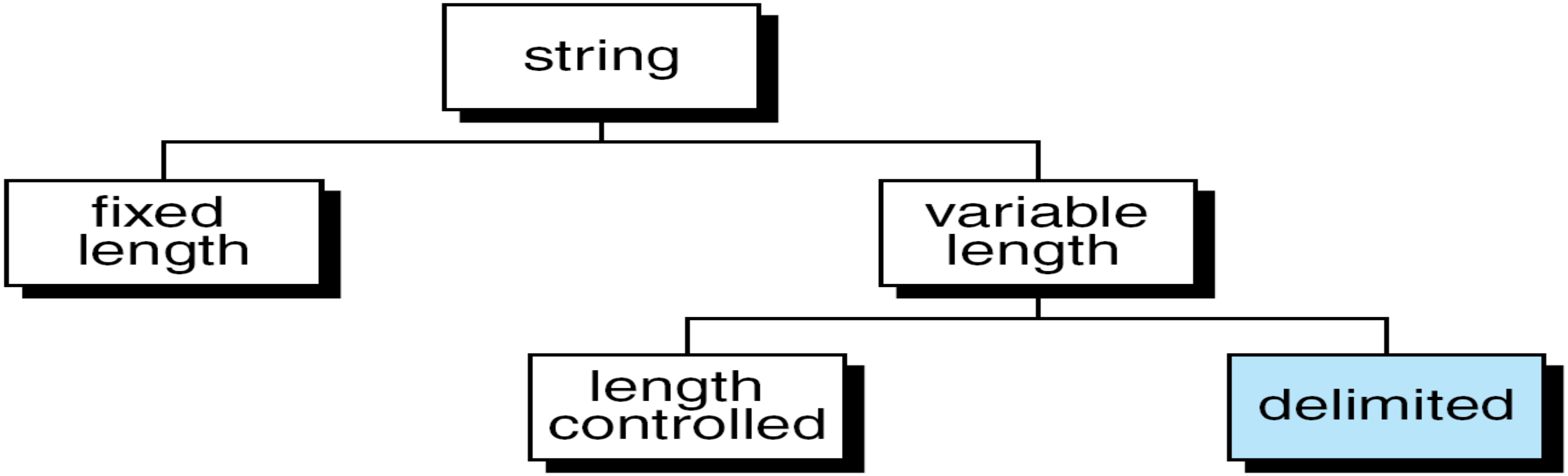
  When working with strings, you might, for example, want to perform any of the following operations:

  - Find out how long a string is
  - Select the first character—or, more generally, the i^{th} character—within a string.
  - Combine two strings to form a longer string
  - Convert a single character into a one-character string
  - Extract some piece of a string to form a shorter one
  - Compare two strings to see which comes first in alphabetical order
  - Determine whether a string contains a particular character or set of characters

Introduction(contd..)

General - String taxonomy

- Strings in Pascal is different from strings in C

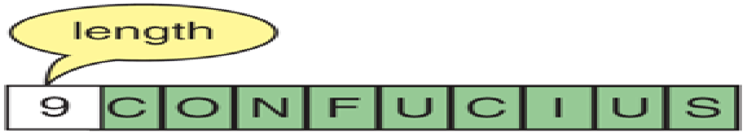


Pascal Strings

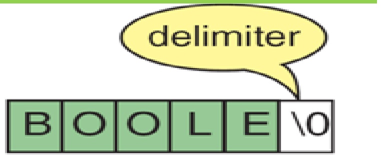
C Strings

Length-controlled Strings: stores the number of characters in the string

Delimited Strings: adds a delimiter at the end of string



Length-controlled string



Delimited string

Strings in C

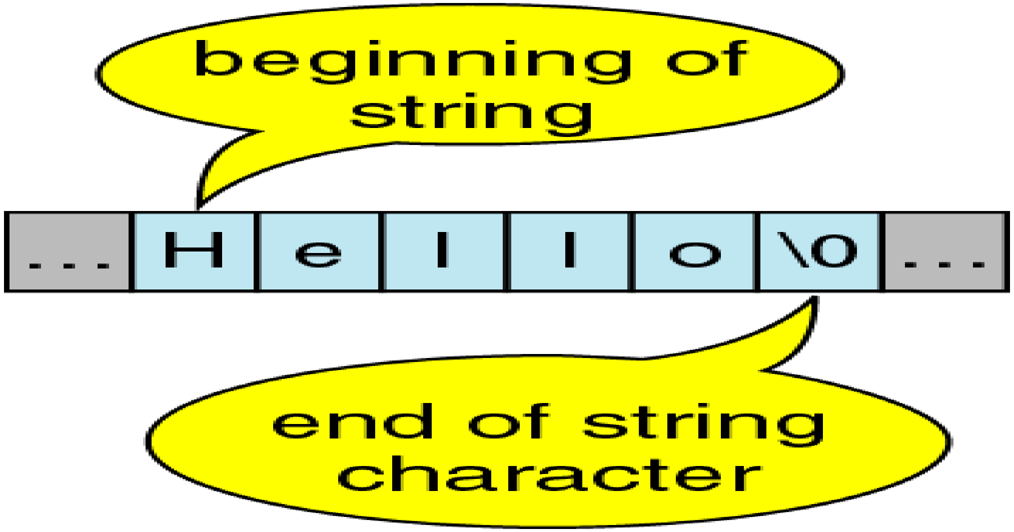
C uses variable-length, delimited strings.

- String is not an explicit type, instead strings are maintained as arrays of characters
- Representing strings in C
  - stored in arrays of characters
  - array can be of any length
  - end of string is indicated by a *delimiter*, the zero character '\0'

Strings in C - -(contd..)

- Strings is a arrays of characters delimited by null character ('\0').

“Hello”



Strings in C (contd..)

Storing strings:

- A character, in single quotes:

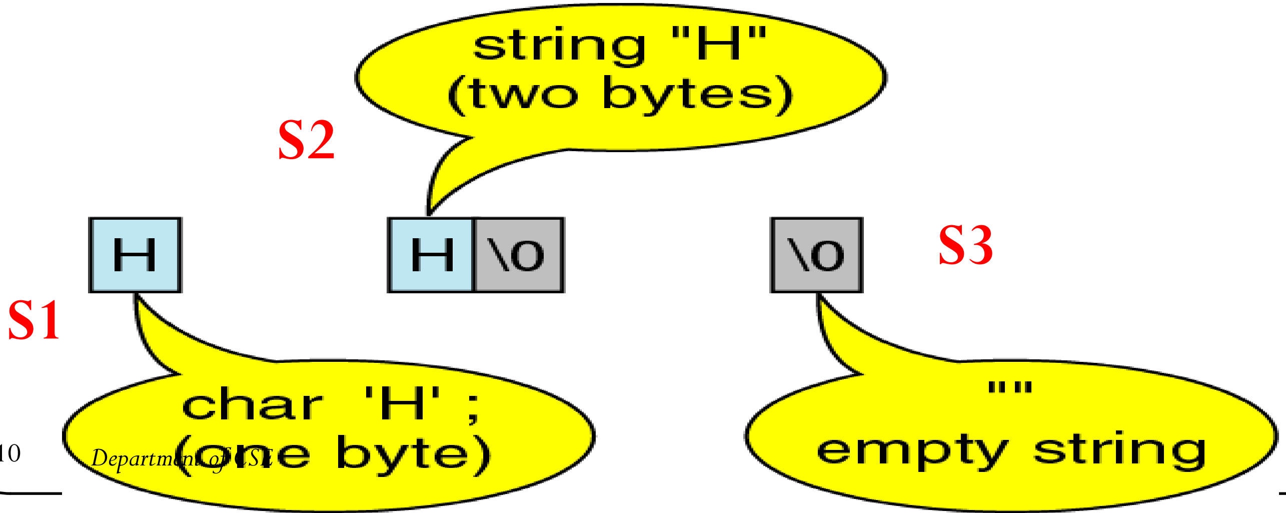
```
char s1= `h`; //Takes only one byte of storage.
```

- On the other hand, the character string:

```
char s2[2]=“H"; //Takes two bytes of storage.
```

- An empty string

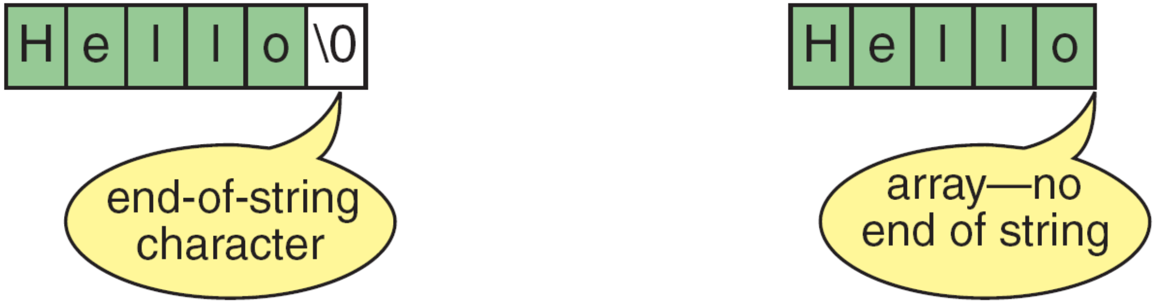
```
char s3[]= “”; //Takes only one byte of
 Storage storage.
```



Strings in C (contd..)

String Delimiter:

- Strings as we know are data type.
  - It uses physical structure as arrays
  - So, it needs a logical end within the physical structure to indicate variable length
  - Therefore, null character('\0') is used as delimiter



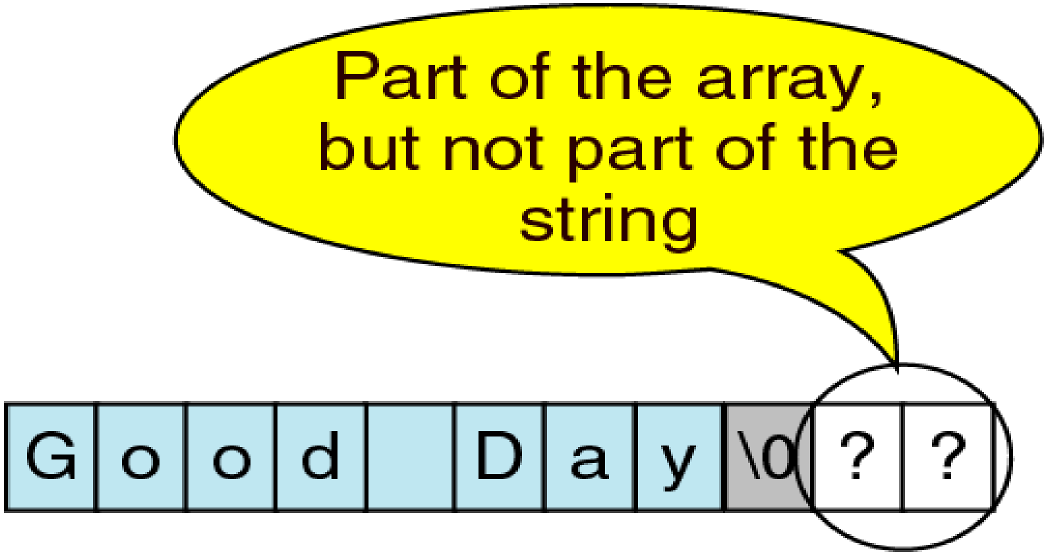
Difference between string and character array is shown in the figure

Strings in C (contd..)

String Delimiter:

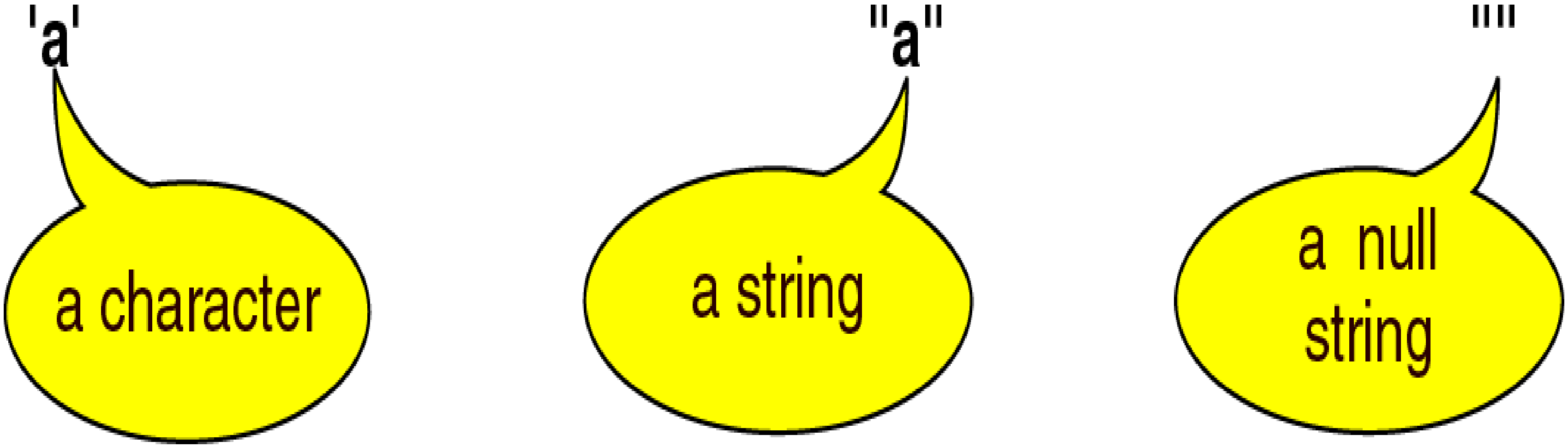
- **Important note:**
  - on declaring array take care to leave one byte for delimiter.
  - String ignores anything that follows null character.

```
char str[11];
```



Strings in C – (contd..)

String literals : A string constant or literal is enclosed in double quotes.



'a'

a character

"a"

a string

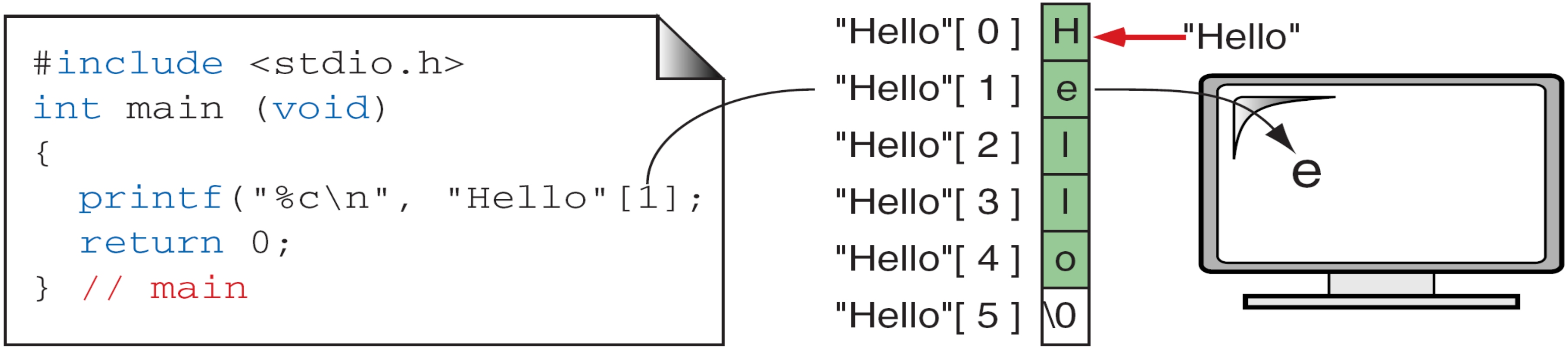
""

a null string

Strings in C (contd..)

String Literals:

- String literal has an address in memory
- String literal is an array of characters, it is a pointer constant to the first element of the string.
  - Hence, the entire string is referenced using this as shown below..

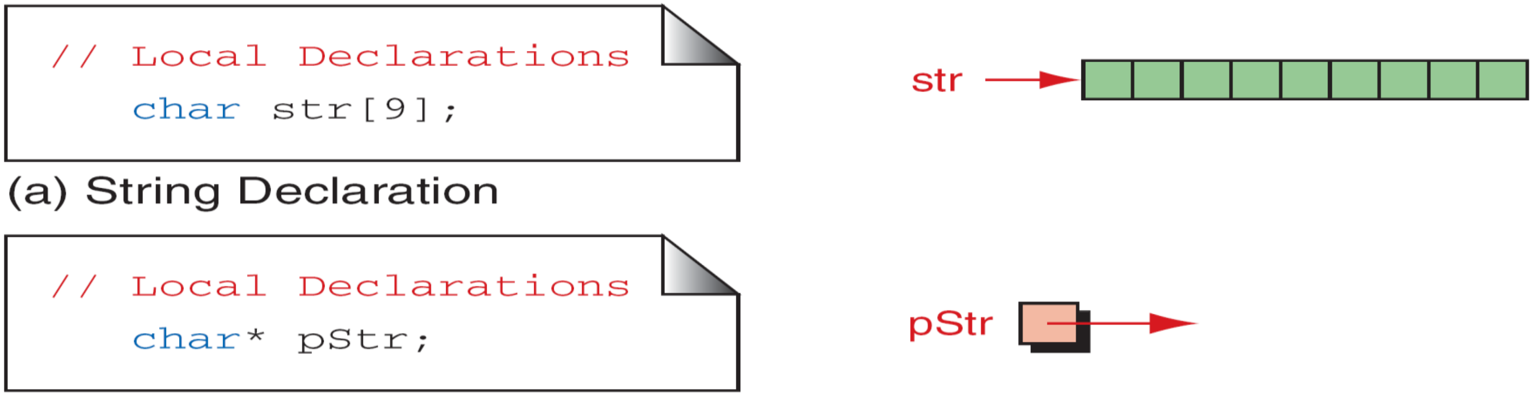


```
#include <stdio.h>
int main (void)
{
  printf("%c\n", "Hello"[1];
  return 0;
} // main
```

Strings in C(contd..)

Declaring Strings:

- Case (a) has the ceiling of 8-characters plus a delimiter
- However, case (b) allows length to be defined before usage.



```
// Local Declarations
   char str[9];
```

(a) String Declaration

```
// Local Declarations
   char* pStr;
```

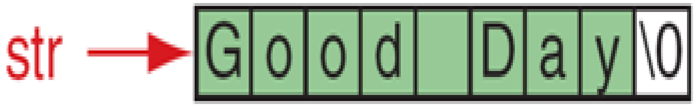
(b) String Pointer Declaration

Memory for strings must be allocated before the string can be used.

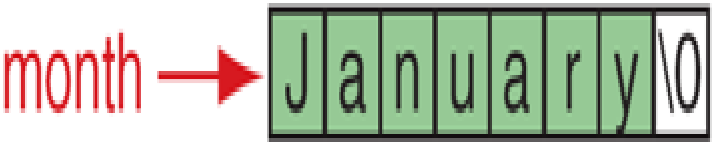
Strings in C(contd..)

Initializing Strings:

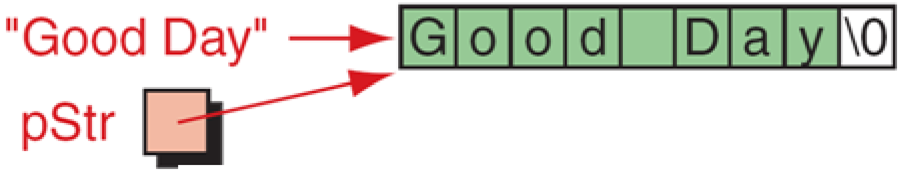
- char str[9]= “Good Day”; or char str[9]= {‘G’,’o’,’o’,’d’,’ ‘, ‘D’,’a’,’y’,’\0’}



- char month[]= “January”;



- char *pStr =“Good Day”;



Illustration

```
#include <stdio.h>
 int main()
{
     char *pstr;
     int  length;
      printf("Enter the length of the string : ");
     scanf (“%d”, &length);
      pstr = (char *)malloc((length+1)*sizeof(char));    //remember to allocate (length + 1) space
      printf("Enter a string : ");
     printf("You entered: %s", pstr);
     return(0);
}
```

On execution:

```
Enter the length of the string: 17
Enter a string : tutorialspoint.com
You entered: tutorialspoint.com
```

Strings in C(contd..)

Assignment Operator:

- The name of the string is a pointer constant.
- Pointer constant can be used only as rvalue
  - hence it cannot be used as left operand of assignment.

char str1[6]=“Hello”;

~~str1=str2;~~ gives compilation error

- Copying strings is done either character-by-character using loops or using library function

String Input/Output Functions

- C provides two basic ways to read and write strings.
  - First, we can read and write strings with the **formatted input/output functions**, scanf/fscanf and printf/fprintf.
  - Second, we can use a special set of **string-only functions**, get string (gets/fgets) and put string (puts/fputs).

String Input/Output Functions (contd..)

Formatted String Input:

- Strings can be read using *scanf* from console and using *fscanf* from files.
- Two Conversion codes are possible for reading strings
  - String conversion code --- “s”
  - Scan set conversion code ---- [...]
- Three optional conversion modifiers are possible preceding the conversion code:

% [*] [*maximum-field-width]* *[size]* Letter

String Input/Output Functions (contd…)

Formatted String Input:

Conversion Modifier	Description
Flag (*)	Assignment Supression. This modifier causes the corresponding input to be matched and converted, but not assigned (no matching argument is needed). Eg. int anInt; scanf("%*s %i", &anInt); Matching Input---- Age: 29 Result ---- anInt==29
Maximum-field-width	This is the maximum number of character to read from the input. Any remaining input is left unread. (**Always** use this with "%s" and "%[…]" in **all** production quality code! (No exceptions!) You should use one less than the size of the array used to hold the result.) *example discussed in following slides.*
Size	Read normal 8-bit ASCII characters if not specified. Otherwise, with option l (note it is letter ell) reads wide characters like UCS and Unicode

String Input/Output Functions(contd…)

Formatted String Input – String Conversion Code:

- Use %s field specification in scanf to read string
  - ignores leading white space
  - reads characters until next white space encountered
  - C stores null (\0) char after last non-white space char
- Example:

```
char Name[11];
scanf("%s", Name); /* Note:   need not use & before Name */
```

- Problem: no limit on number of characters read (need one for delimiter), if too many characters for array, problems may occur

The string conversion code(s) skips whitespace.

String Input/Output Functions (contd…)

Formatted String Input – String Conversion Code:

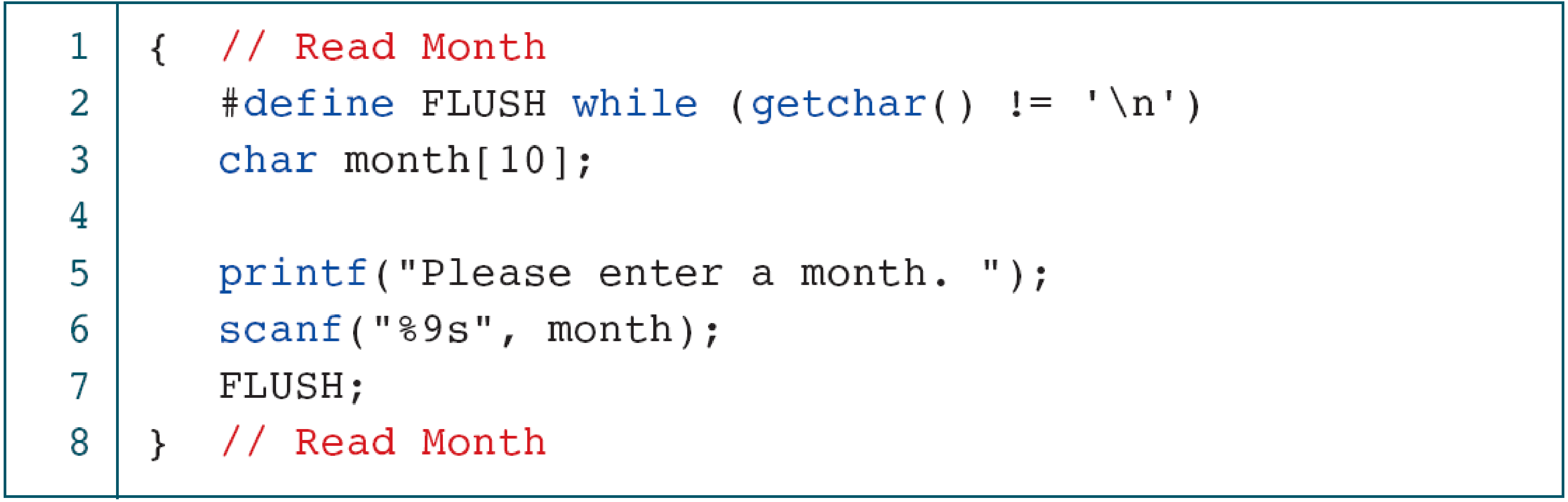
- Can use the width value in the field specification to limit the number of characters read:

```
char month[10];
scanf("%9s",month);
```

- Remember, you need one space for the \0
  - width should be one less than size of array
- Strings shorter than the field specification are read normally, but C always stops after reading 9 characters
  - The remaining part string that is not read upto newline can be/ has to be flushed as shown in the program as follows:

String Input/Output Functions (contd…)

Formatted String Input – String Conversion Code:



```
{   // Read Month
    #define FLUSH while (getchar() != '\n')
    char month[10];

    printf("Please enter a month. ");
    scanf("%9s", month);
    FLUSH;
}   // Read Month
```

String Input/Output Functions (contd…)

Formatted String Input – Scan Set Conversion Code:

- Edit set input %[*ListofChars*]
  - ListofChars specifies set of characters (called scan set)
  - Characters read as long as character falls in scan set
  - Stops when first non scan set character encountered
  - Note, does not ignored leading white space
  - Any character may be specified except]
  - Putting ^ at the start to negate the set (any character BUT list is allowed)
- Examples:
  - scanf("%10[-+0123456789]",Number);
  - scanf("%81[^\n]",Line); /* read until newline char */
  - scanf("%15[^~!@#$%^&*()_+]", str) ; /*reads characters other than specified*/
  - scanf("%15 [ ] [0123456789]",str); /* reads square bracket and num */

String Input/Output Functions (contd…)

Formatted String Input – Scan Set Conversion Code:

- Note:

The edit set does not skip whitespace.

Always use a width in the field specification when reading strings.

String Input/Output Functions (contd…)

Formatted String Output:

- Strings can be write using *printf* to console and using *fprintf* to files.
- Conversion codes are possible for writing strings is "s"
- Four optional conversion modifiers are possible preceding the conversion code:

% [*Justification Flag*] [*minimum-field-width]* *[precision]* [*size*] s

The maximum number of characters to be printed is specified by the precision in the format string of the field specification.

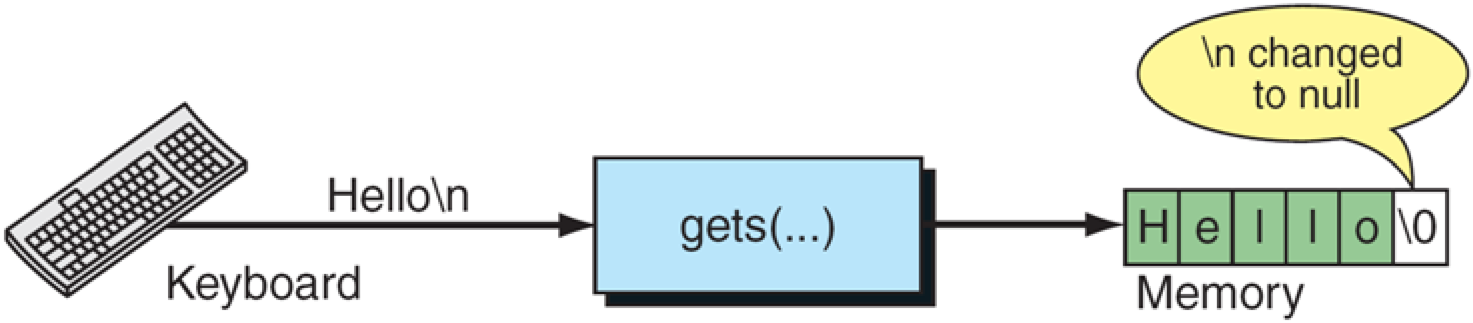
String Input/Output Functions (contd…)

Conversion Modifier	Description
Justification Flag	left-justify within the field. Eg. char Name[10] = “Rich”; printf(“|%-10s|”,Name); /* Outputs: |Rich | */
Minimum-field-width	After converting any value to a string, the field width represents the minimum number of characters in the resulting string. If the converted value has fewer characters, then the resulting string is *padded* with spaces (or zeros) on the left (or right) by default (or if the appropriate flag is used.) Eg. printf("|%5s|", "ABC"); /* outputs |··ABC| */
Precision	specifies the maximum number of bytes written. If the string is too long it will be truncated. Eg. printf("|%-5.3s|", "ABCD"); /* outputs |ABC··| */
Size	Read normal 8-bit ASCII characters if not specified. Otherwise, with option l (note it is letter ell) reads wide characters like UCS and Unicode

String Input/Output Functions (contd…)

String-only Input :

- Read without reformatting any data is provided by the function *gets*
- *gets* converts line to string
- char *gets(char *str)
  - reads the next line (up to the next newline) from keyboard and stores it in the array of chars pointed to by str
  - returns str if string read or NULL if problem/end-of-file
  - not limited in how many chars read (may read too many for array)
  - newline included in string read



String Input/Output Functions (contd…)

Illustration:

```
#include <stdio.h>
 int main()
{
    char str[50];
     printf("Enter a string : ");
    gets(str);
    printf("You entered: %s", str);
    return(0);
}
```

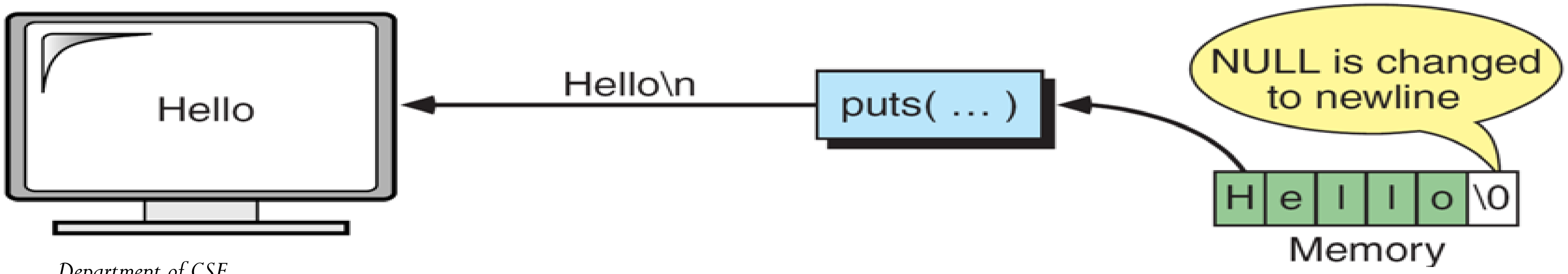
On execution:

```
Enter a string : tutorialspoint.com
You entered: tutorialspoint.com
```

String Input/Output Functions (contd…)

String-only Input:

- write without reformatting any data is provided by the function *puts*
- *puts* converts String to line
- int puts(char *str)
  - prints the string pointed to by str to the screen
  - prints until delimiter reached (string better have a \0)
  - returns EOF if the puts fails
  - outputs newline if \n encountered (for strings read with gets or fgets)



Try it Yourself

- Predict the output

```
#include<stdio.h>
void main() {
char *str="CQUESTIONBANK";
clrscr();
printf(str+9);
getch();   }
```

What will output when you compile and run the above code?

Answer: BANK

Try it Yourself

- Predict the output

What will be output when you will execute following c code?

```
#include<stdio.h>
void main(){
    char arr[7]="Network";
    printf("%s",arr);
}
```

Answer: garbage value (Reason: as the string "Network" is of length 7 so the string is not null terminated)

Try it Yourself

- Predict the output

What will be output when you will execute following c code?

```
#include<stdio.h>
#define var 3
void main(){
   char *cricket[var+~0]={"clarke","kallis"};
   char *ptr=cricket[1+~0];
   printf("%c",*++ptr);
}
```

Answer: l

Reason:

- In the expression of size of an array can have micro constant.   var +~0 = 3 + ~0 = 3 + (-1) = 2
- Therefore circket[2] = {pointer to c, pinter to k}
- ++ptr --- ptr+1 is equal to the location following c in clarke, hence th answer is l(letter ell).

Summary

- The string storage schemes in old styled programming language vs new languages were discussed.
- Declaring and initializing strings was discussed.
- Raw vs formatted read and write of strings was dealt .