# CS19001: Programming and Data Structures Laboratory

## String, Pointers, Dynamic Memory Allocation

DRC, SD, SB; CSE, IIT Kharagpur

March 20, 2025

# Table of Contents

1 Tutorial: Characters

2 Tutorial: Strings

3 Tutorial: Pointers

4 Dynamic Memory Allocation

5 Assignments

# Characters

## Declaration and Initialization

```
char ch = ‘a’;   OR   char ch; ch = ‘a’;
```

## ASCII Values of Characters

Every character has an integer ASCII value and you can get that by printing it in integer format.

```
char ch = ‘a’;
printf(‘‘%d’’,ch);
   // prints ASCII value (97) of ‘a’
```

Let us not memorize the ASCII values (of a-z, A-Z and 0-9). It can easily be assigned to any integer and can be found/operated. Moreover, integers and characters are inter-operable.

```
int x = ‘A’;
printf(‘‘%c’’,x+3);
   // prints the character ‘D’
```

# Character manipulations

## Example: Simple text encryption

*Caesar cipher* is a simple technique of encryption of plain text byreplacing every character in the plain text by a character fixed number of positions down the list of the alphabet. The last characters are folded back to the beginning. The numerical digits and all other characters will remain unchanged.

| Shift: 5 | | Shift: 2 | |
|---|---|---|---|
| Original | Encrypted | Original | Encrypted |
| ‘A’ | ‘F’ | ‘a’ | ‘c’ |
| ‘B’ | ‘G’ | ‘b’ | ‘d’ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| ‘Y’ | ‘D’ | ‘y’ | ‘a’ |
| ‘Z’ | ‘E’ | ‘z’ | ‘b’ |

Let us program to read a text stream and will encrypt the English alphabets, [ A - Z ] and [ a - z ], using Caesar cipher. The value of shift should be within $1 - 10$ and will be decided by the `rand()` function.

# C-Program: Simple text encryption

```
#include <stdio.h>
#include <stdlib.h> // for rand()
#include <ctype.h> // for isalpha()
int main()
{
  char c, shift;
  // generating random shift
  shift = (char)(rand()%10 + 1);
  while((c = getchar()) != EOF) {
    if(isalpha(c)) { // checking for alphabets
      if(isupper(c)) // upper-case alphabet
        putchar((c-‘A’+shift)%26+‘A’);
      else // lower-case alphabet
        putchar((c-‘a’+shift)%26+‘a’);
    }
    else putchar(c); // other characters unchanged
  }
  putchar(‘\n’);
  return 0;
}
```

# Table of Contents

1 Tutorial: Characters

2 Tutorial: Strings

3 Tutorial: Pointers

4 Dynamic Memory Allocation

5 Assignments

# Strings

In C, a string is defined to be a null-terminated character array. The null character '\0’ is used to indicate the end of the string.

```
int main ()
{
  char greet[3]={'H','i','\0'};
  printf("Greeting message: %s\n",greet);
  return 0;
}
```

## Variation in initialization

```
char c[]="abcd";
char c[5]="abcd";
char c[]={'a','b','c','d','\0'};
char c[5]={'a','b','c','d','\0'};
```

# Reading string from terminal

```
#include <stdio.h>
int main(){
    char name[20];
    printf("Enter name: ");
    scanf("%s",name);
    printf("Your name is %s.",name);
    return 0;
}
```

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

- scanf() function takes only string before the white space.

# Reading a line of text

```
int main(){
    char name[30],ch;
    int i=0;
    printf("Enter name: ");
    while(ch!=’\n’)
    {// terminates if user hit enter
        ch=getchar();
        name[i]=ch;
        i++;
    }// inserting null character at end
    name[i]=’\0’;
    printf("Name: %s",name);
    return 0;
}
```

# Better method

```
int main(){
    char name[30];
    printf("Enter name: ");
    gets(name);
    //Function to read string from user.
    printf("Name: ");
    puts(name);
    //Function to display string.
    return 0;
}
```

```
Enter name: Dennis Ritchie
Name: Dennis Ritchie
```

# Passing Strings to Functions

```
void Display(char ch[]);
int main(){
    char c[50];
    printf("Enter string: ");
    gets(c);
    Display(c);
    // Passing string c to function.
    return 0;
}
void Display(char ch[]){
    printf("String Output: ");
    puts(ch);
}
```

# Library functions

```c
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int  len ;
    strcpy(str3, str1);
    printf("strcpy(str3,str1): %s\n",str3);
    strcat( str1, str2);
    printf("strcat(str1,str2): %s\n",str1);
    len = strlen(str1);
    printf("strlen(str1) :  %d\n", len );
    return 0;
}
```

# Result

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

- do not forget to include string.h

# A bit more about string manipulation

- `int strcmp (char s[ ], char t[ ])`:
  Returns 0 if the two strings are identical, a negative value if s is lexicographically smaller than t (i.e., if s comes before t in the standard dictionary order), and a positive value if s is lexicographically larger than t. Comparison is done with respect to ASCII values (A - 65, a - 95)
- `int strlen (char s[ ])`:
  Returns the length (the number of characters before the first null character) of the string s.

# Table of Contents

# Pointers

VARIABLE that stores memory address

```
void main(){
  int i;
  int *ptr; //pointer to an int

  i = 4; /* store the value 4 into the
  memory location associated with i */
  ptr = &i; /* store the address of i
  into the memory location associated
  with ptr */
  *ptr = *ptr + 1;
  printf(   %d\ n   , i);  //i=5
}
```

# More examples

- argument for scanf()

```
scanf("%d %d", &data1, &data2);
```

Pass address of variable where you want result stored

- Declarations: (all have same meaning)

```
int* x, y;
int *x, y;
int *x; int y;
```

The $*$ operator binds to the variable name, not the type

# Relationship between Arrays and Pointers

An array name is essentially a pointer to the first element in the array

```
char data[10];
/* data = addr where first element
is located = &data[0] */
char *cptr;
cptr = data; /* points to data[0] */
```

# Pointers and Arrays

```
char data[10];
/* data = addr where first element
is located = &data[0] */
```

| | |
|---|---|
| data | &data[0] |
| (data + n) | &data[$n$] |
| ∗data | data[0] |
| ∗(data + n) | data[$n$] |

# Pointers and Arrays

```
int main(void) {
int a[N] = {84, 67, 24, ...};
/*
&a[0] = a+0 = D000
&a[1] = a+1 = D004
&a[2] = a+2 = D008

a[0] = *a     = 84
a[1] = *(a+1) = 67
a[2] = *(a+2) = 24
*/

return 0;
}
```

## Passing Pointers as Function Arguments

Alter variables outside a function's own scope

```
void swap(int *first, int *second);
int main(){
  int a = 4, b = 7;
  printf("pre-swap: a=%d, b=%d\n",a,b)
  swap(&a, &b);
  printf("post-swap: a=%d, b =%d\n",a,b)
  return 0;
}
void swap(int *first, int *second){
  int temp;
  temp = *first;
  *first = *second;
  *second = temp;
}
```

# Passing Pointers as Function Arguments

```
void swap(int *first, int *second);
int main(){
  int a = 4, b = 7;
  printf("pre-swap: a=%d, b=%d\n",a,b)
  swap(&a, &b);
  printf("post-swap: a=%d, b =%d\n",a,b)
  return 0;
}
void swap(int *first, int *second){
  int temp;
  temp = *first;
  *first = *second;
  *second = temp;
}
```

The address-of operator (&) is used to pass the address of the two variables rather than their values

# Passing Array as Function Argument

```
#define N 64
int average(int b[], int n) {
  int i, sum; // same as int *b
  //receives the value D000 from main
  for (i = 0; i < n; i++)
    sum += b[i];
  return sum / n;
}
int main(void) {
  int a[N] = {84, 67, 24, ..., 89, 90};
  printf("%d\n", average(a, N));
  return 0; //passes &a[0] = D000
}
```

# Advantage? working with subarray

```
#define N 64
int average(int b[], int n) {
  int i, sum; // same as int *b
  //receives the value D000 from main
  for (i = 0; i < n; i++)
    sum += b[i];
  return sum / n;
}
int main(void) {
  int a[N] = {84, 67, 24, ..., 89, 90};
  printf("%d\n", average(a+5, 10));
  return 0; //passes &a[0] = D000
}
```

- compute average of a[5] through a[14]

# Table of Contents

1 Tutorial: Characters

2 Tutorial: Strings

3 Tutorial: Pointers

4 Dynamic Memory Allocation

5 Assignments

# Dynamic memory allocation

```
#include<stdio.h>
#include<stdlib.h>
int max(int a[], int c, int *b);
int main(){
 int i, j, m, *a;
 printf("enter number of elements\n");
 scanf("%d",&i);
 a=(int *)malloc(i * sizeof(int));
 for(j=0;j<i;j++){
   printf("enter element no. %d:",j);
   scanf("%d", &a[j]);
  }
 m=max(a, i, &j); // next slide
 printf("Max value is %d stored in a[%d]\n",m,j);
 return 0;
}
```

# Returning multiple values

```
int max(int *a, int i, int *j)
{
    int k, max=-32767;
    for (k=0; k<i; k++)
    {
            if (a[k]>max)
            {
                max=a[k];
                *j=k;
            }
    }
    return(max);
}
```

# Table of Contents

1. Tutorial: Characters
2. Tutorial: Strings
3. Tutorial: Pointers
4. Dynamic Memory Allocation
5. Assignments

# Programming Assignments

Complete and submit during lab

# Assignment 1: [Variable-sized Strings]

Write a program that achieves the following functionality:

- Write a function – `storeString`, that reads a string of length at most 100 from the keyboard, and stores it in a dynamically allocated array consuming the minimum required memory for that string.
- Create an array of 100 pointers that can be used to store the strings created by the above function and use it to store and print names of $n$ persons. Here, $n < 100$ is the first input to the program.

The second functionality should be implemented in the main function. Note that all the $n$ strings should be first entered by the user and then printed together.

# Assignment 2: [Name Database (contd. from assign. 1)]

In continuation from assignment 1, implement the following functionality:

- Write a function `int search(char *a[],char *q, int n)` that searches for string pointed to by $q$ in the array of $n$ strings $a$. It should return the index $i$ such that $a[i]$ points to the returned string that contains $q$. For example “IIT Kharagpur” contains “T Khar”.
- In the main function, after entering a set of names as in assignment 1, allow the user to interactively search for a given name and delete it from the list. After deleting the searched name, print the list of remaining names. Note that, you have to move pointers all the strings after the removed string by one position.

# Assignment 3: [Alliteration Detection]

Write a program that, given a poem as input, detects if there is an alliteration, and prints the alliterative words. An example is:

*”The fair breeze blew, the white foam flew,*
*The furrow followed free;”*

You can use the following definition of alliteration:

- The first letter of at least two consecutive words should be the same.
- After meeting the above criteria, all words starting with the same character are part of the alliteration. In the example, *fair foam flew furrow followed free* are all part of the alliteration.
- If there are more than one candidates satisfying the above two requirements, report the one with the maximum number of total words. In the above example *breeze blew* is not reported.

# Bonus Assignment: [Alliterative Rhyme Detection]

Write a function that can detect whether two input words rhyme in the alphabetic sense. For example, the words *light* and *fight* rhyme in the alphabetic sense, while the words *fight* and *kite* don't rhyme in the alphabetic sense, but do rhyme in the phonetic sense. **Hint**: portion of the word from the last vowel should be identical.

Using the above function, detect whether an input poem paragraph is couplet-rhyming, end-rhyming, or not rhyming. A couplet consists of two successive lines that rhyme. For example:

"The sky is bright, the stars do shine, (A)
Their golden glow is so divine." (A)
"The breeze is soft, the night is cool, (B)
As ripples dance upon the pool." (B)

# Bonus Assignment: [Alliterative Rhyme Detection] (Contd.)

End-rhyming poems have other types of rhyming lines at regular intervals (e.g. ABAB or ABCB). For example:

"The waves crash hard upon the shore, (A)
The seagulls soar so high above, (B)
The tide will rise, then fall once more, (A)
A rhythm set by those we love." (B)

If there are no rhyming words in the end, then the poem will be called non-rhyming. Write a main function that takes lines of poems as input and categorizes them into the above three categories.

# Thank You