CS 332/532 Systems Programming

Lecture 6

- Pointers, Char, String-

Professor : Mahmut Unan – UAB CS

Agenda

- Char
- Strings
- scanf
- gets
- puts
- Memory Blocks
- Memory operations

Passing Pointers to functions

```c
#include <stdio.h>
void test(int a);
int main(void)
{
    void (*ptr)(int a);
    ptr = test;
    (*ptr)(10);
    return 0;
}
void test(int a)
{
    printf("%d\n", 2*a);
}
```

```
20
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr[1](i, j);
    printf("Result  : %d\n", result);
    return 0;
}
int addTwoNumbers(int a, int b)
{   return a+b;  }
int subtractTwoNumbers(int a, int b)
{   return a-b;   }
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr
    printf("Result
    return 0;
}
int addTwoNumbers(int
{   return a+b;  }
int subtractTwoNumber
{   return a-b;   }
```

```
Enter two integer numbers: 20 30
Result   : 50
```

```
Enter two integer numbers: 45 20
Result   : 25
```

The `char` Type

- Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the **char** type to store its value.
- Once a character is stored into a variable, it is the character's ASCII value that is actually stored.

```
char ch;
ch = 'c';
```

  - the value of `ch` becomes equal to the ASCII value of the character 'c'.
  - Therefore,
    - the statements `ch = 'c';` and `ch = 99;` are equivalent.
    - Of course, 'c' is preferable than 99; not only it is easier to read, but also your program won't depend on the character set as well.

```c
#include <stdio.h>
int main(void)
{
    char ch;
    ch = 'a';
    printf("Char = %c and its ASCII code is %d\n", ch, ch);
    return 0;
}

```

```
Char = a and its ASCII code is 97
```

- Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c’;
int i; ch++; /* ch becomes 'd'. */
ch = 68; /* ch becomes 'D'. */
i = ch-3; /* i becomes 'A',that is 65 */
```

Example

upperLower.c > ...

```c
#include<stdio.h>
#include<string.h>
int main()
{
  char s[100];
  printf("Enter a  smaple string:   ");
  scanf("%[^\n]",s);

  printf("Output in uppercase:\n");
  puts(strupr(s));

  return 0;
}
```

Example

upperLower.c > ...

```c
#include<stdio.h>
#include<string.h>
int main()
{
  char s[100];
  printf("Enter a  smaple string:   ");
  scanf("%[^\n]",s);

  printf("Output in uppercase:\n");
  puts(strupr(s));
```

```
PS C:\Users\unan\Desktop\c_coding> gcc upperLower.c -o upperLower
PS C:\Users\unan\Desktop\c_coding> .\upperLower.exe
Enter a  smaple string:   mahmut unan
Output in uppercase:
MAHMUT UNAN
```

Without using string library

upperLower2.c > main()

```c
#include<stdio.h>
int main()
{
    char s[100];
    int i = 0;

   printf("Enter a  smaple string:   ");
    scanf("%[^\n]",s);

    while( s[i] != '\0' )
    {

       if( s[i] >= 97 && s[i] <= 122 )
       // or  if( s[i] >= "a" && s[i] <= "z" )
       {
          s[i] = s[i] - 32;
       }
       i++;
    }

    printf("Output in uppercase:\n");
    puts(s);

    return 0;
}
```

Without using string library

C upperLower2.c > main()

```c
#include<stdio.h>
int main()
{
    char s[100];
    int i = 0;

   printf("Enter a  smaple string:   ");
    scanf("%[^\n]",s);

    while( s[i] != '\0' )
    {

       if( s[i] >= 97 && s[i] <= 122 )
       // or  if( s[i] >= "a" && s[i] <= "z" )
       {
          s[i] = s[i] - 32;
       }
       i++;
    }

}
```

```
PS C:\Users\unan\Desktop\c_coding> gcc upperLower2.c -o upperLower2
PS C:\Users\unan\Desktop\c_coding> .\upperLower2.exe
Enter a  smaple string:   mahmut unan
Output in uppercase:
MAHMUT UNAN
```

`getchar()` and `putchar()`

- The `getchar()` function is used to read a character from stdin.
- The `putchar()` function writes a character in stdout, for example, `putchar('a')`

Strings

- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = {'m', 'e', 's',
's', 'a', 'g', 'e', '\0'};
```

puts()

```c
#include <stdio.h>
int main(void)
{
    char str[] = "UAB CS 330 Course";
    puts(str);
    puts(str);
    str[4] = '\0';
    printf("%s\n", str);
    return 0;
}
```

```
UAB CS 330 Course
UAB CS 330 Course
UAB
```

scanf()

- `scanf()` takes as an argument a pointer to the array that will hold the input string.

- Since we're using the name of the array as a pointer, we don't add the address operator **&** before its name.

- Because `scanf()` stops reading once it encounters the space character, only the word this is stored into str. Therefore, the program outputs this.

- To force `scanf()` to read multiple words, we can use a more complex form such as `scanf("%[^\n]", str);`

gets() fgets()

```
char *gets(char *str);
```

`gets()` is not safe, don't use it

```c
#include <stdio.h>
int main(void)
{
    char str[100];
    int num;
    printf("Enter number: ");
    scanf("%d", &num);
    printf("Enter text: ");
    fgets(str, sizeof(str), stdin);
    printf("%d %s\n", num, str);
    return 0;
}
```

```
Enter number: 1223132312312312312323123123231231231231232131231231231231231231231231231231231231
Enter text: -1
```

The `strlen()` Function

```
size_t strlen(const char *str);
```

The size_t type is defined in the C library as an unsigned integer type (usually as **unsigned int**).

`strlen()` returns the number of characters in the string pointed to by str, not counting the null character.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[100], str2[100];
    printf("Enter text: ");
    fgets(str2, sizeof(str2), stdin);
    strcpy(str1, str2);
    printf("Copied text: %s\n", str1);
    return 0;
}
```

```
Enter text: Hello CS330
Copied text: Hello CS330
```

Search the following functions

strcat()

strcmp()

Functions →Array as Arguments

- When a parameter of a function is a one-dimensional array, we write the name of the array followed by a pair of brackets.

- The length of the array can be omitted; in fact, this is the common practice.

- For example:

```
void test(int arr[]);
```

- When passing an array to a function, we write only its name, without brackets. For example:

```
test(arr);
```

Memory Blocks

- Code
- Data
- Stack
- Heap

```c
#include <stdio.h>
#include <stdlib.h>
void test(void);
int global;
int main(void)
{
    int *ptr;
    int i;
    static int st;

    /* Allocate memory from the heap. */
    ptr = (int*) malloc(sizeof(int));

    if(ptr != NULL)
    {
        printf("Code seg: %p\n", test);
        printf("Data seg: %p %p\n", &global, &st);
        printf("Stack seg: %p\n", &i);
        printf("Heap: %p\n", ptr);
        free(ptr);
    }
    return 0;
}
void test(void)
{ /* Do something. */
}
```

```
Code seg: 0x106e1ff30
Data seg: 0x106e21024 0x106e21020
Stack seg: 0x7ffee8de091c
Heap: 0x7f8a55405840
```

Static Memory Allocation

- In static allocation, the memory is allocated from the stack.
- The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution.
- For example, with the statement:

```
float grades[1000];
```

Dynamic Memory Allocation

- In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified.
- Furthermore, this size may dynamically shrink or grow according to the program's needs.
- Typically, the default stack size is not very large, the size of the heap is usually much larger than the stack size.

malloc()

```
void *malloc(size_t size);
```

The `size_t` type is usually a synonym of the **unsigned int** type.

The size parameter declares the number of bytes to be allocated.

If the memory is allocated successfully;

`malloc()` returns a pointer to that memory, NULL otherwise.

Check the following functions

`realloc()`

`calloc()`

`free()`

`memcpy()`

`memmove()`

`memcmp()`

References

- C From Theory to Practice - 2nd edition, Nikolaos D. Tselikas and George S. Tselikis
- https://www.tutorialspoint.com/cprogramming/c_pointers.htm
- https://www.programiz.com/c-programming/c-pointers-arrays
- https://www.geeksforgeeks.org/function-pointer-in-c/