CS 332/532 Systems Programming

Lecture 6

- String, Dynamic Memory Allocation-

Professor : Mahmut Unan – UAB CS

Agenda

- gets
- puts
- Memory Blocks
- Memory operations
- Struct and Union

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()`

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{   int *ptr,n,i;
    /* the number of array elements */
    printf("How many elements?:\n");
    scanf("%d",&n);

    ptr = (int*)malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation was NOT successful.\n");
        exit(0);
    }
    else {
        printf("Memory  allocation was successful.\n");
        for (i = 0; i < n; i++)
            ptr[i] = (i+1) * 10;

        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free(ptr);
    printf("\nMemory deallocation was successful.\n");
    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{   int *ptr,n,i;
    /* the number of array elements */
    printf("How many elements?:\n");
    scanf("%d",&n);

    ptr = (int*)malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation was NOT successful.\n");
```

```
How many elements?:
8
Memory  allocation was successful.
10, 20, 30, 40, 50, 60, 70, 80,
Memory deallocation was successful.
```

```c
    return 0;
}
```

Structures & Unions

```
struct structure_tag {
 member_list;
} structure_variable_list;
```

A **struct** declaration defines a type. Although the structure_tag is optional, we prefer to name the structures we declare and use that name later to declare variables.

```c
struct company
{
    char name[50];
    int start_year;
    int field;
    int tax_num;

    int num_empl;
    char addr[50];
    float balance;
};
```

sizeof()

```c
#include <stdio.h>
struct date
{
        int day;
        int month;
        int year;
};
int main(void)
{
        struct date d;
        printf("%u\n", sizeof(d));
        return 0;
}
```

sizeof()

```
struct test1
{
        char c;
        double d;
        short s;
};
struct test2
{
        double d;
        short s;
        char c;
};
```

```c
#include <stdio.h>
struct student
{
    int code;
    float grd;
};
int main(void)
{
    struct student s1, s2;
    s1.code = 1234;
    s1.grd = 6.7;
    s2 = s1; /* Copy structure. */
    printf("C:%d G:%.2f\n", s2.code, s2.grd);
    return 0;
}
```

Dot notation vs Arrow notation

- In C, both dot (.) and arrow (->) operators are used to access members of a structure, but they are used in different situations.
- **Dot Notation (.):**This is used when you have a structure variable, not a pointer to a structure. It directly accesses members of the structure.
- **Arrow Notation (->):**This is used when you have a pointer to a structure. It dereferences the pointer and then accesses the member.

Dot notation

```c
#include <stdio.h>
#include <string.h>

struct Person {
    char name[50];
    int age;
};

int main() {
    struct Person person1;
    // Using dot notation to access members
    person1.age = 30;
    strcpy(person1.name, "Alice");

    printf("Name: %s\n", person1.name);
    printf("Age: %d\n", person1.age);
    return 0;
}
```

```
Name: Alice
Age: 30
```

Arrow operator

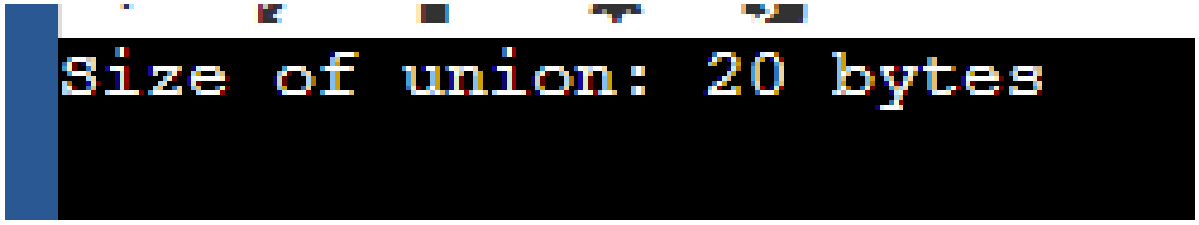
```c
#include <stdio.h>
#include <string.h>

struct Person {
    char name[50];
    int age;
};

int main() {
    struct Person person2;
    struct Person *ptr = &person2;  // Pointer to structure

    // Using arrow notation to access members
    ptr->age = 25;
    strcpy(ptr->name, "Bob");

    printf("Name: %s\n", ptr->name);
    printf("Age: %d\n", ptr->age);
    return 0;
}
```

```
Name: Bob
Age: 25
```

Unions

- Like a structure, a union contains one or more members, which may be of different types. The properties of unions are almost identical to the properties of structures; the same operations are allowed as on structures.
- Their difference is that the members of a structure are stored at *different* addresses, while the members of a union are stored at the *same* address.

```c

#include <stdio.h>

union Data {
    int i;          // 4 bytes
    float f;        // 4 bytes
    char str[20]; // 20 bytes
};

int main() {
    union Data data;  // Declare a union variable

    // Use sizeof to determine the size of the union
    printf("Size of union: %lu bytes\n", sizeof(data));
    return 0;
}

```



Sample Union usage

```c
#include <stdio.h>
#include <string.h>

union SensorData {
    int temperature;     // Temperature sensor in degrees Celsius
    float humidity;      // Humidity sensor in percentage
    char status[10];     // Status message like "OK", "ERROR"
};

int main() {
    union SensorData data;

    // Case 1: Using temperature sensor data
    data.temperature = 25;
    printf("Temperature: %d°C\n", data.temperature);

    // Case 2: Using humidity sensor data (overwrites temperature)
    data.humidity = 65.5;
    printf("Humidity: %.1f%%\n", data.humidity);

    // Case 3: Using status message (overwrites humidity)
    strcpy(data.status, "OK");
    printf("Status: %s\n", data.status);

    return 0;
}
```

```
Temperature: 25°C
Humidity: 65.5%
Status: OK
```

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/