

# C Character and Strings

Characters, Strings, Null Terminator, Manipulating Strings



**SoftUni Team**  
**Technical Trainers**  
**Software University**  
<http://softuni.bg>



# Table of Contents

1. Strings in C
2. Processing Characters
3. Parsing Strings to Numeric Types
4. String Processing Functions



# Strings in C

# The String Data Type

- The **string data type** represents a sequence of characters
  - Declared as a **char** array
  - Size should be **string length + 1**
  - Should always end with null terminating character (**'\0'**)

```
char name[6] = "Pesho";
```

| 0   | 1   | 2   | 3   | 4   | 5    |
|-----|-----|-----|-----|-----|------|
| 80  | 101 | 115 | 104 | 111 | 0    |
| 'P' | 'e' | 's' | 'h' | 'o' | '\0' |

Null terminating character



# Strings Literals

- Rules when initializing strings in C:
  - Size should be **string length + 1**
  - Last character is reserved for **null terminator** character '**\0**' representing end of string
- Strings can be initialized in several ways:
  1. **char** array declaration:

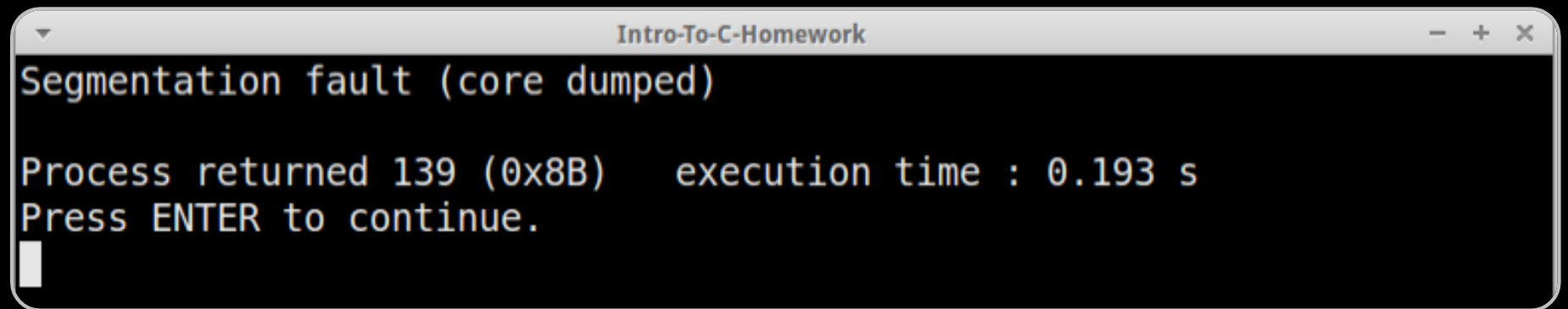
```
char firstName[6] = "Petar";  
char lastName[] = "Tsutsumkov";
```

# String Literals (2)

## 2. Constant declaration:

```
char *name = "Pesho";  
name[0] = 'G'; // Will cause segmentation fault
```

- **Warning:** Modifying string will cause segmentation fault because it is located in read-only memory



```
Intro-To-C-Homework  
Segmentation fault (core dumped)  
Process returned 139 (0x8B)   execution time : 0.193 s  
Press ENTER to continue.  
█
```

# String Literals (3)

## 3. Heap initialization:

```
char *firstName = malloc(6);  
if (firstName != NULL) {  
    strcpy(firstName, "Pesho");  
    // ...  
    free(firstName);  
}
```

NULL is returned when there is not enough memory

Frees the occupied memory

- Requests 6 bytes from the operating system (OS)
- If the OS finds 6 free bytes, it returns a **pointer** to their address
- Copies "Pesho" to the address where the pointer points to

# Declaring Strings

## Live Demo



# Character Processing Functions

# Processing Characters

- The character-handling library `<ctype.h>` includes several functions for manipulating characters
  - `int isblank(int c)` – returns whether c is a space character
  - `int isdigit(int c)` – returns whether c is a digit (0-9)
  - `int isalpha(int c)` – returns whether c is a letter (a-z, A-Z)

```
int isBlank = isblank(' '); // 1
int isDigit = isdigit('5'); // 1
int isAlpha = isalpha('y'); // 1
```

# Processing Characters (2)

- `int islower(int c) / int isupper(int c)` – checks if the character is lowercase/uppercase
- `int tolower(int c) / int toupper(int c)` – returns the character as lowercase/uppercase

```
int isLower = islower('B'); // 0  
char upper = toupper('a'); // A
```

- More: [http://www.tutorialspoint.com/c\\_standard\\_library/ctype\\_h.htm](http://www.tutorialspoint.com/c_standard_library/ctype_h.htm)

# Character Processing Functions

## Live Demo



# Parsing Strings to Numeric Types

# Parsing Strings

- Strings can be **parsed** (converted) to numeric types
  - The **<stdlib.h>** library provides parsing functions

```
char *text = "3 minutes";  
char *remainder;  
  
long num = strtol(text, &remainder, 10);  
printf("%ld\n", num);           // 3  
printf("%s\n", remainder);      // minutes
```

- We pass a pointer to **\*remainder** (double pointer)
  - **strtol()** assigns to it the remaining text after "3"

string



long

# Parsing String to Long

- `long strtol(const char *str, char **remainder, int base)` – parses `*str` to a long integer in the given `base`
  - `base` is the number of digits in a numeral system (e.g. 10)
  - Example:

```
char text[] = "101";  
char *remainder;  
  
long dec = strtol(text, &remainder, 10);    // 101  
long hex = strtol(text, &remainder, 16);    // 257  
long binary = strtol(text, &remainder, 2);  // 5
```

# Handling Format Error

- The **\*remainder** can be compared to the original string pointer
  - If they point to the same string, then parsing failed → **format error**

```
char *text = "asd";  
char *remainder;  
  
long num = strtol(text, &remainder, 10);  
if (remainder == text)  
    printf("Format error");  
else  
    printf("%ld\n", num);
```



# Handling Out of Range Error

- **strtol()** sets **errno** to **ERANGE** if an overflow occurs

```
#include <errno.h>
#include <limits.h>
int main()
{
    char *text = "853185325718243124921421421";
    char *remainder;

    errno = 0;
    long lNum = strtol(text, &remainder, 10);
    if (errno == ERANGE || (lNum < INT_MIN || lNum > INT_MAX))
        printf("Int must be in range [%d..%d]\n",
               INT_MIN, INT_MAX);

    ...
}
```

# Safely Parsing Integer – Example

```
#include <stdlib.h>
#include <limits.h>
#include <errno.h>

int main()
{
    char text[30];
    scanf("%29s", text);
    char *remainder;

    errno = 0;
```

Reads up to 29 characters  
from the standard input

Sets the global error  
object to **0** (i.e. no error)

# Safely Parsing Integer – Example (2)

```
long num = strtol(text, &remainder, 10);
if (errno == ERANGE || (num < INT_MIN || num > INT_MAX))
{
    printf("Int should be in range [%d..%d]\n", INT_MIN, INT_MAX);
}
else if (text == remainder)
{
    printf("Invalid format\n");
}
else
{
    printf("Num: %d\n", num);
}

return 0;
}
```

Compares if pointers are the same (e.g. nothing was parsed)

# Parsing String to Double

- **double strtod(const char \*nPtr, char \*\*endPtr)** – parses the floating-point number from **\*nPtr**
  - Assigns the remaining string to **\*endPtr**

```
char *text = "4.20 blaze it friend";  
char *remainder;  
  
double num = strtod(text, &remainder);  
printf("%f\n", num); // 4.20000  
printf("%s\n", remainder); // blaze it friend
```



# Safely Parsing Integer

## Live Demo

# Processing Strings

# strlen

- **size\_t strlen(const char \*s)** – returns the length of the string as number of characters before null terminator

```
size_t size = 6;  
char *street = "Tintyava 15-17";  
char uniName[] = "SoftUni";  
char founder[size];  
  
strncpy(founder, "Nakov", size - 1);  
founder[size - 1] = '\0';  
  
printf("%d\n", strlen(street)); // 14  
printf("%d\n", strlen(uniName)); // 7  
printf("%d\n", strlen(founder)); // 5
```

# sprintf

- **sprintf(char \*str, const char \*format, ...)** – writes the formatted string to **\*str**
  - **format** contains standard C format specifiers (e.g. **%d**, **%s**, etc.)
  - Writes the null terminator too
  - Similar to **printf()**, but destination is a string

```
char text[128];  
sprintf(text, "%d + %d = 20", 15, 5);  
  
printf("%s!", text);
```



# strcpy

- **strcpy(char \*dest, const char \*src)** – copies the bytes from **\*src** (until null terminator is reached) to **\*dest**
  - Writes null terminator in the end

```
char city[128];  
strcpy(city, "Sofia");  
  
printf("%s!\n", city); // Sofia!
```



- Use **strncpy()** to limit copy size

# strncpy

- **strncpy(char \*dest, const char \*src, size\_t n)** – copies **n** bytes from **\*src** to **\*dest**

```
#define SIZE 20

int main()
{
    char buffer[SIZE];
    strncpy(buffer, "Blagoevgrad ice cold beer", SIZE);
    buffer[SIZE - 1] = '\0';

    printf("%s\n", buffer);
    return 0;
}
```

Set null terminating character by hand

# strdup

- **char\* strdup(const char \*str)** – returns a string copy
  - Allocated on the heap, should be eventually **free()**'d

```
const char *str = "C# is awesome";  
char *copy = strdup(str);  
copy[1] = ' ';  
  
printf("%s\n", copy); // C  is awesome  
free(copy);
```

- Note: **strdup()** is a POSIX function

# strncat

- **strncat(char \*dest, const char \*src, size\_t n)** – appends at most **n** bytes from **\*src** to the end of **\*dest**
  - Overwrites **'\0'** at the end of **\*dest**

```
size_t size = 10;
char *name = malloc(size);
if (!name) return 1;

strncpy(name, "Pesho", size);
strncat(name, " Kitaeca", size - strlen(name) - 1);
printf("%s\n", name); // Pesho Kit
free(name);
```

# Joining Strings

## Live Demo



# strcmp

- `int strcmp(const char *s1, const char *s2)` – compares the bytes of the two strings

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

int main(void)
{
    printf("%d\n", strcmp("A", "A")); // 0
    printf("%d\n", strcmp("A", "B")); // -1
    printf("%d\n", strcmp("B", "A")); // 1
    return 0;
}
```

# Reading Commands

## Live Demo

# strchr / strrchr

- **char \*strchr(const char \*str, char ch)** – returns a pointer to the first occurrence of **ch** in **\*str**

```
const char *url = "https://softuni.bg/forum";  
char *firstMatch = strchr(url, '/');  
if (firstMatch)  
    printf("%s\n", firstMatch); // //softuni.bg/forum
```

- **char \*strrchr(const char \*str, char ch)** – returns a pointer to the last occurrence of **ch**

```
char *lastMatch = strrchr(url, '/'); // /forum
```

# strstr

- **char \*strstr(const char \*str, const char \*search)** – returns a pointer to the first occurrence of **\*search** in **\*str**

```
const char *sentence = "The quick brown fox jumped over the  
lazy dog";  
  
char *substr = strstr(sentence, "fox");  
if (substr)  
{  
    printf("Index: %lu\n", substr - sentence); // Index: 16  
    printf("Substring: %s\n", substr); // Substring: fox ...  
}
```

# Validate XML

## Live Demo



# strtok

- **char \*strtok(char \*src, const char \*delimiter)** – a sequence of calls to this function return tokens split by the passed **delimiter**

```
char sentence[] = "He entered the room, despite her warning.";
char *token = strtok(sentence, " ,."); // He
```

- Multiple calls must be made to return all tokens

```
while (token != NULL) {
    printf("(%s)\n", token);
    token = strtok(NULL, " ,.");
}
```

Pass **NULL** to continue from previous index

# strtok (2)

- The consequent calls should pass **NULL** as string source
  - **strtok()** globally keeps the last token's index
  - Passing **NULL** tells it to start from the last token (not the beginning)
  - In multi-threaded applications, always use **strtok\_r()** instead
- Note: **strtok()** modifies the source string
  - Make a copy of the string if you must before calling the function

# Splitting String using strtok() – Example

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char sentence[] = "He entered the room, despite her warning.";
    char *token = strtok(sentence, " ,.");

    while (token != NULL)
    {
        printf("(%s)\n", token);
        token = strtok(NULL, " ,.");
    }

    printf("%s\n", sentence); // He
    return 0;
}
```

Source string has  
been modified

# strtok\_r

- `char *strtok_r(char *str, const char *delim, char **savePtr)` – splits `*str` by `*delim`, returning a token on each function call
  - Thread-safe version of `strtok()`
  - `**savePtr` is used to keep track of the splitting
  - Consequent calls should take `NULL` as `*str` (to continue from the last split in `**savePtr`)
  - The original string is again modified

# Splitting String using strtok\_r() – Example

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

int main(int argc, char** argv)
{
    char text[] = "(300)(3.14)(Word)";
    char *savePtr;
    char *token = strtok_r(text, "()", &savePtr);
    while (token) {
        printf("%s ", token);
        token = strtok_r(NULL, "()", &savePtr);
    }

    return (EXIT_SUCCESS);
}
```



# Splitting Strings

## Live Demo

# The errno Integer

- **errno** is a integer variable from the **<errno.h>** header
  - Set to **0** on program start
  - Modified by many system calls in case of failure, e.g.:
    - **strtol()** sets **errno** to **ERANGE** if overflow occurs
    - **malloc()** sets **errno** to **ENOMEM** if no memory is available
    - Full list: <http://www.virtsync.com/c-error-codes-include-errno>
  - Thread-safe (each thread has its own thread-specific **errno**)
  - Can be used to check if a function encountered an error

# strerror

- **char \*strerror(int errornum)** – maps an error code to a string message
  - Error code definitions can be accessed from the **errno.h** header
  - Example:

```
errno = 0;
char *ptr = malloc(2L << 32);
if (!ptr)
{
    fprintf(stderr, "%s", strerror(errno));
    exit(1);
}
```

Get a user-friendly message by error code

# C Programming – Characters and Strings



## Questions?





# License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license



- Attribution: this work may contain portions from
  - "Programming Basics" course by Software University under CC-BY-SA license



# Free Trainings @ Software University

- Software University Foundation – [softuni.org](http://softuni.org)
- Software University – High-Quality Education, Profession and Job for Software Developers
  - [softuni.bg](http://softuni.bg)
- Software University @ Facebook
  - [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)
- Software University @ YouTube
  - [youtube.com/SoftwareUniversity](https://youtube.com/SoftwareUniversity)
- Software University Forums – [forum.softuni.bg](http://forum.softuni.bg)

