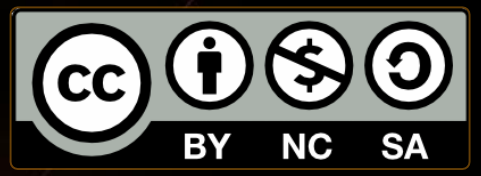
C Functions

Writing and using functions, function arguments, return types



SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents

Table of Contents (2)

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

SOFTWARE UNIVERSITY FOUNDATION

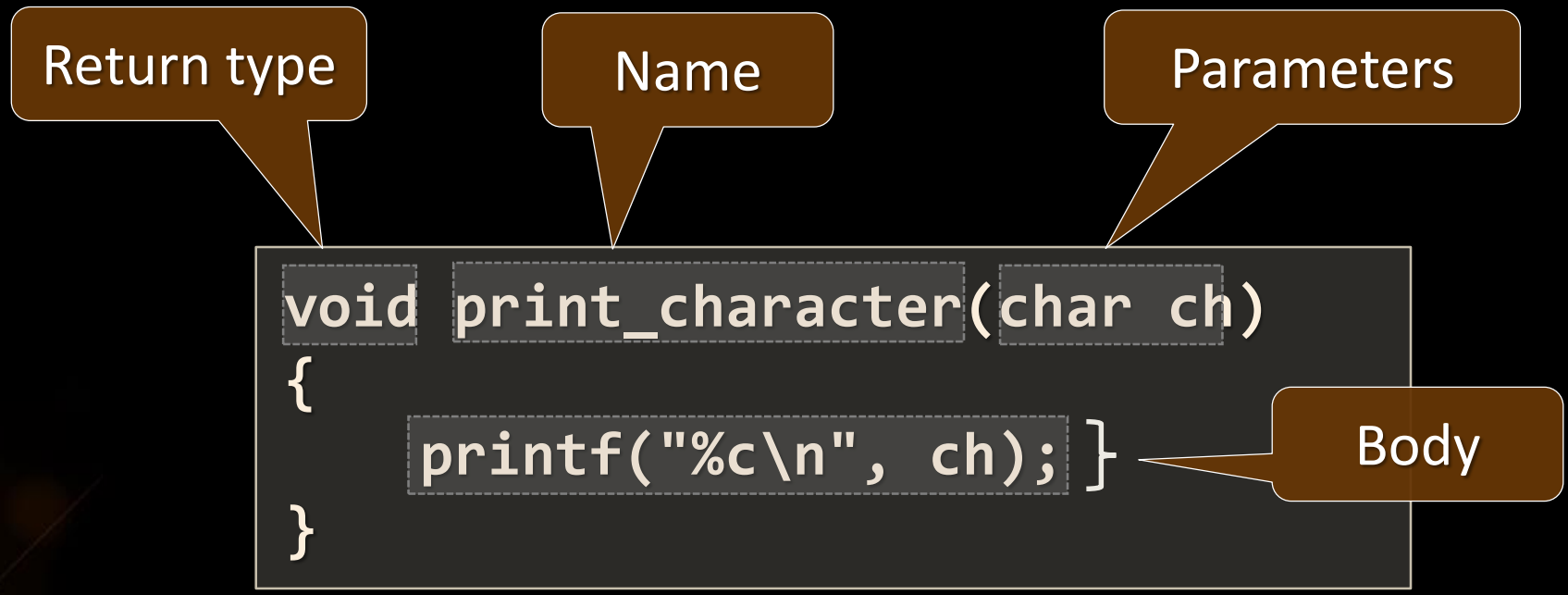
```
void print_hyphens(int count)
{
    char buffer[count + 1];
    for (int i = 0; i < count; i++)
        buffer[i] = '-';
    buffer[count] = '\0';
    printf("%s", buffer);
}

int main()
{
    print_hyphens(10);
    return 0;
}
```

Functions

Declaring and Invoking Functions

Declaring Functions

- A function is a named piece of code
- Each functions has:



Invoking Functions

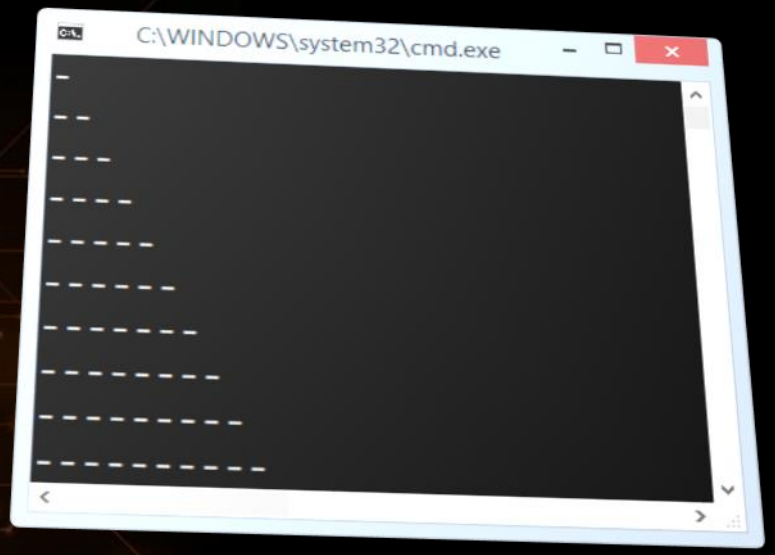
```
void print_hyphens(int count)
{
    char buffer[count + 1];
    for (int i = 0; i < count; i++)
        buffer[i] = '-';
    buffer[count] = '\0';
    printf("%s", buffer);
}

int main()
{
    print_hyphens(10);
    return 0;
}
```

Function body always surrounded by { }

Function called by name

Functions

Live Demo




```
void print_hyphens(int count)
{
    char buffer[count + 1];
    for (int i = 0; i < count; i++)
        buffer[i] = '-';
    buffer[count] = '\0';
    printf("%s", buffer);
}

int main()
{
    print_hyphens(10);
    return 0;
}
```

Function Parameters

- Function parameters can be of any type
  - Separated by comma

```
void print_numbers(int start, int end)
{
    for (int i = start; i <= end; i++)
    {
        printf("%d ", i);
    }
}

int main()
{
    print_numbers(5, 10);
    return 0;
}
```

Declares the use of `int` start and `int` end

Passed concrete values when called

Function Parameters – Example

```
void print_sign(int number)
{
    if (number > 0)
        printf("The number %d is positive.", number);
    else if (number < 0)
        printf("The number %d is negative.", number);
    else
        printf("The number %d is zero.", number);
}

int main(void)
{
    print_sign(5);
    print_sign(-3);
    print_sign(0);

    return 0;
}
```

Declares `main()` does not take any arguments

SOFTWARE UNIVERSITY FOUNDATION

Function Parameters

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Exercises in Class

Printing Triangle - Exercise

- Write a function for printing triangles as shown below:

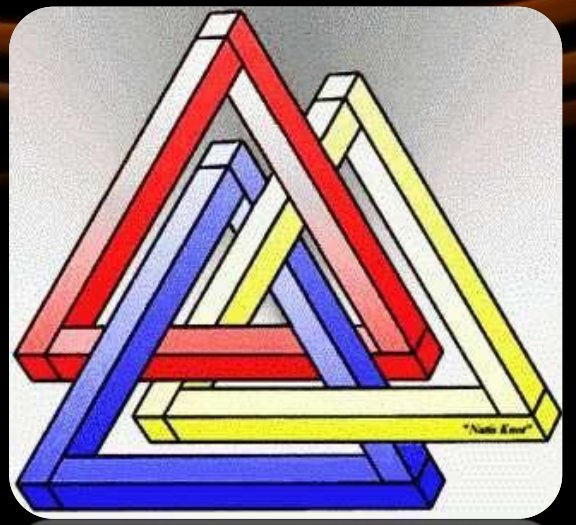
n=5 →

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

n=6 →

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

SOFTWARE UNIVERSITY
FOUNDATION



Printing Triangle

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Returning Values From Functions

Function Return Types

SOFTWARE UNIVERSITY FOUNDATION

- Type **void** - does not return a value (only executes code)

```
void add_one(int n)
{
    n += 1;
    printf("%d", n);
}
```

- Other types - return values, based on the return type of the function

```
int plus_one(int n)
{
    return n + 1;
}
```

Functions with Parameters and Return Value


```
double calc_triangle_area(double width, double height)
{
    return width * height / 2;
}

int main()
{
    double width, height;
    scanf("%lf %lf", &width, &height);
    double area = calc_triangle_area(width, height);

    printf("Area is %.2lf\n", area);

    return 0;
}
```

Power Function

```
double power(double number, int power)
{
    double result = 1;
    for (int i = 0; i < power; i++)
    {
        result *= number;
    }
    return result;
}
int main()
{
    double powerTwo = power(5, 2);
    printf("5^2=%.2lf\n", powerTwo);
    printf("7.45^2=%.2lf\n", power(7.45, 3));
    return 0;
}
```

Function result will evaluate first

Temperature Conversion – Example

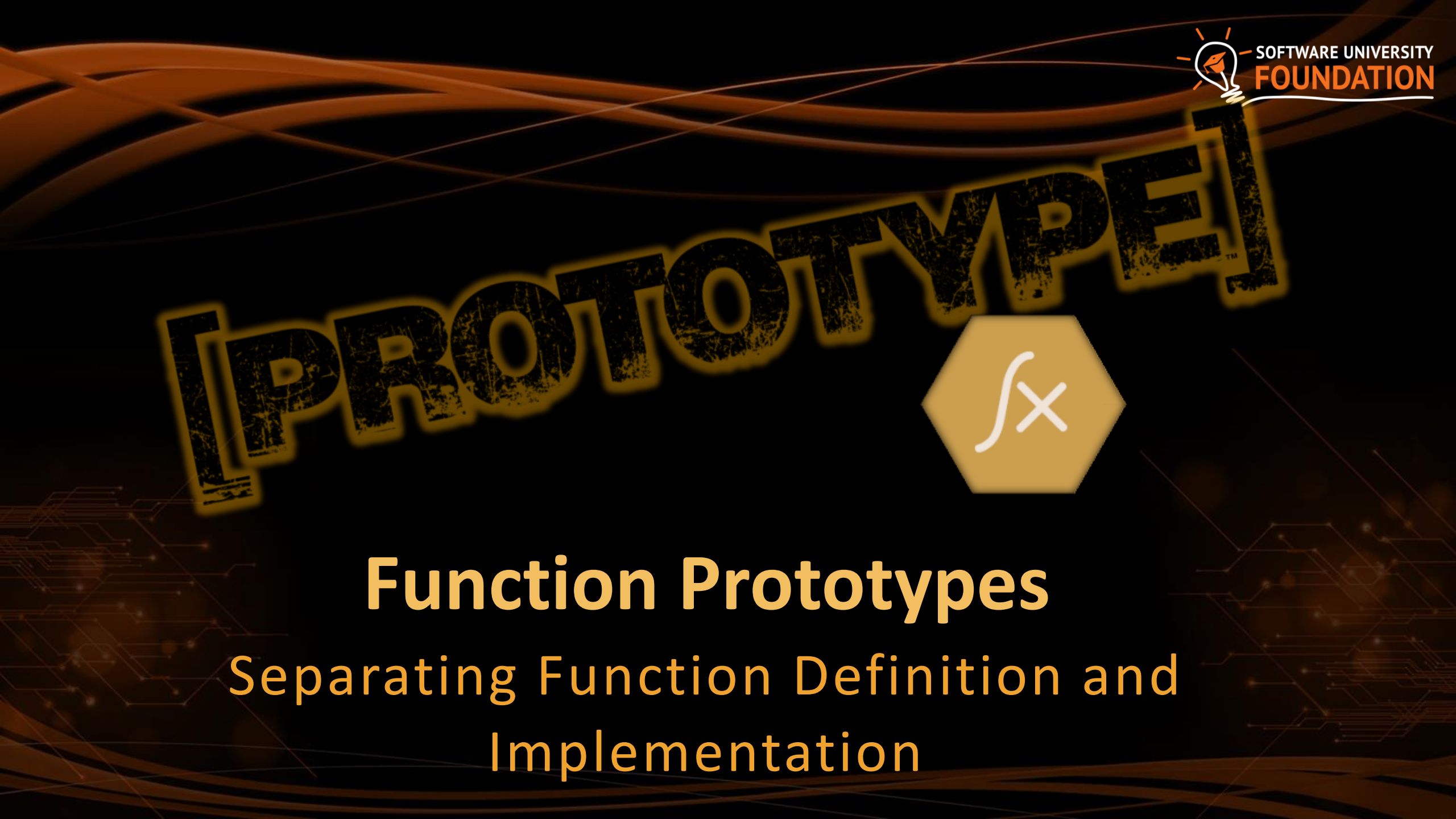
- Convert temperature from Fahrenheit to Celsius:

```
double fahrenheit_to_celsius(double degrees)
{
    return ((degrees - 32) * 5) / 9;
}

int main()
{
    double temperature;
    printf("Temperature in Fahrenheit: \n");
    scanf("%lf", &temperature);
    temperature = fahrenheit_to_celsius(temperature);
    printf("Temperature in Celsius: %lf\n", temperature);
}
```

SOFTWARE UNIVERSITY FOUNDATION

Returning Values From Functions

Live Demo



Function Prototypes

Separating Function Definition and Implementation

Function Prototypes in C

- A function prototype represents only the signature of a function
  - No body and implementation

```
int array_index_of(int* array, int arrLen, int num);
int main()
{
    int array[] = { 3, 4, 5 };
    int index = array_index_of(array, 3, 4);

    return 0;
}
int array_index_of(int* array, int arrLen, int num)
{
   ...
}
```

Function prototype

Function implementation

Why Use Function Prototypes?

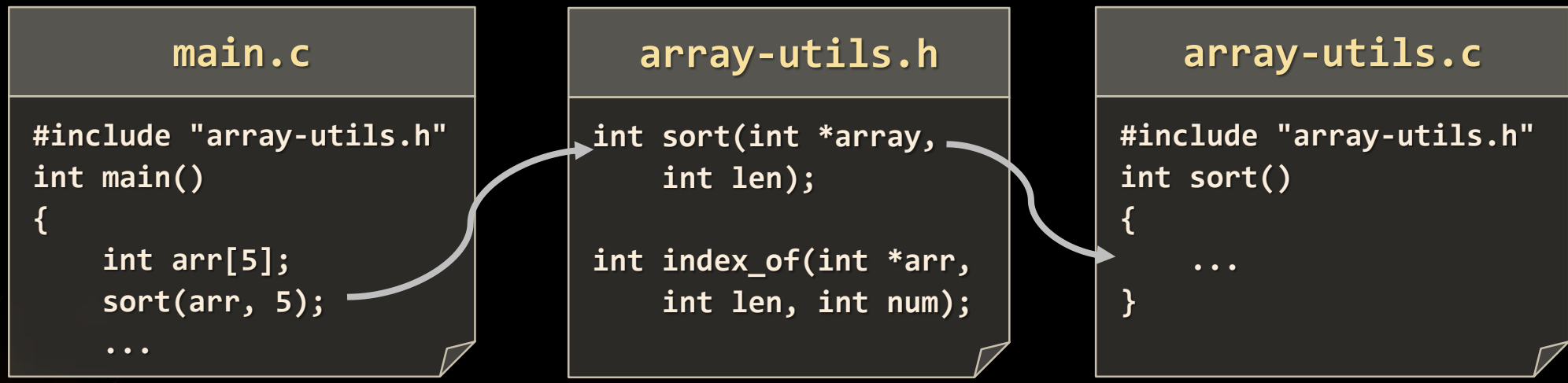
- Function prototypes explicitly declare the function signature
  - That way the compiler knows the exact arguments
  - Example without prototypes:

```
int main()
{
    int result = sum(4);
    printf("%d\n", sum);
    return 0;
}
int sum(int a, int b)
{
    return a + b;
}
```

The compiler does not correctly check the function input parameters

Header Files

- Function prototypes are usually put in header (`.h`) files
  - Calls functions
  - Defines function prototypes
  - Implements function body



- Separates function definition and implementation
- Makes code more modular and easier to maintain

SOFTWARE UNIVERSITY FOUNDATION

Function Prototypes

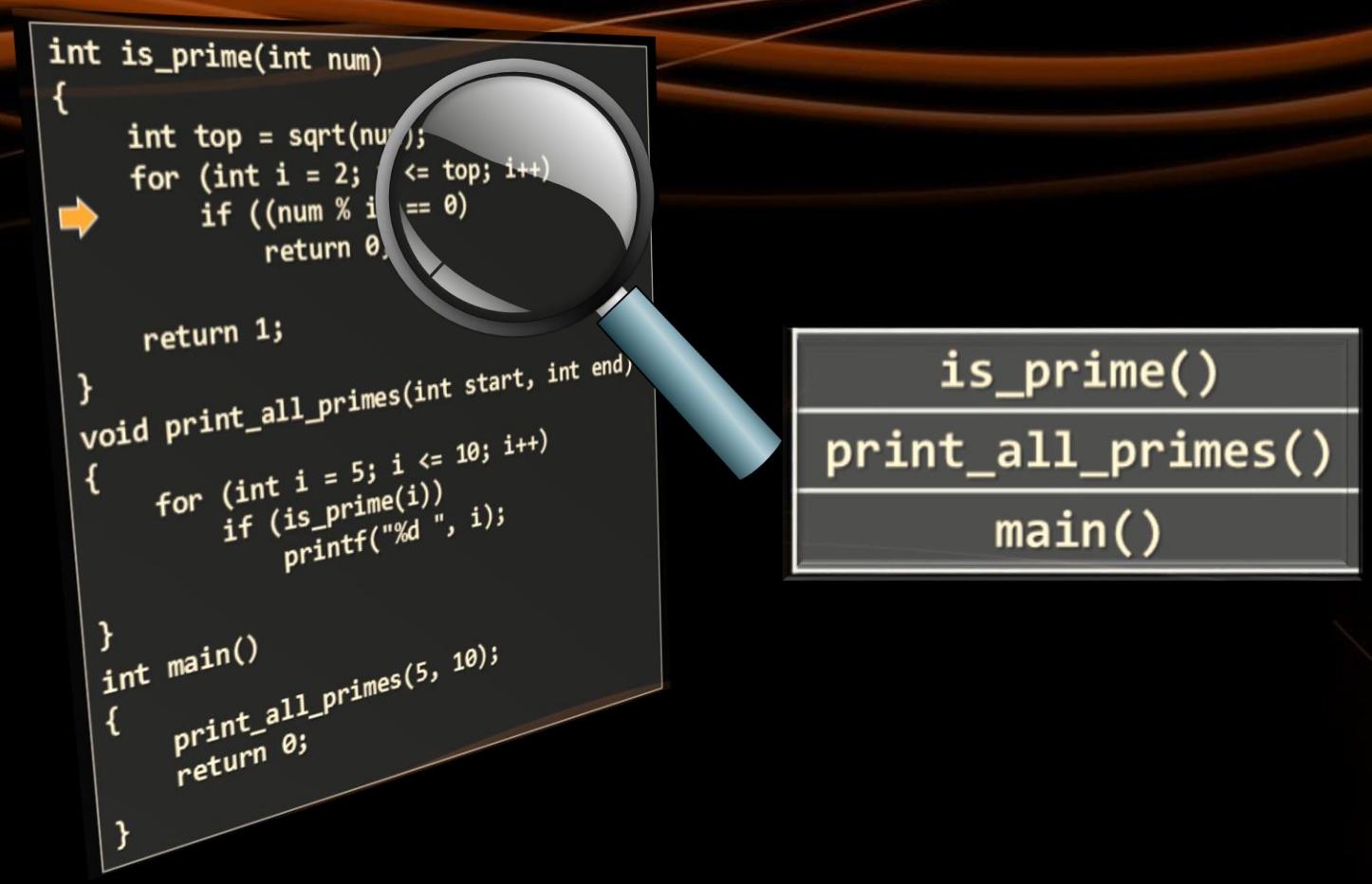
Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Exercises in Class

Geometry - Exercise

- Create a `geometry.h` header file and define the following function prototypes:
  - Function takes `x`, `y` and `r` of 2 circles and determines if they intersect
  - Function takes sides `a`, `b` and `c` of a triangle and returns if the triangle is valid
- Create a `geometry.c` source file, include `geometry.h` and implement the functions

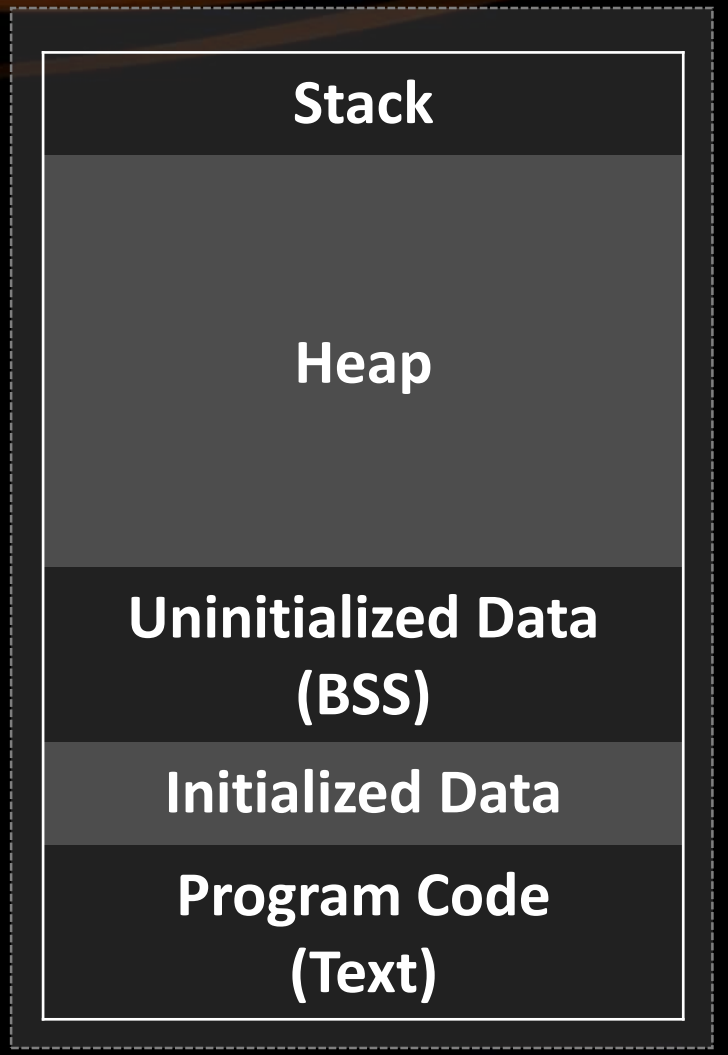


Program Execution Stack

Process Memory

- All Linux processes are granted a memory space to execute on, comprised of:

1. Stack – holds called functions and their local variables
2. Heap – holds dynamically allocated objects
3. Unitialized Data – uninitialized global variables
4. Initialized Data – initialized global variables
5. Program Code (binary)



The Stack

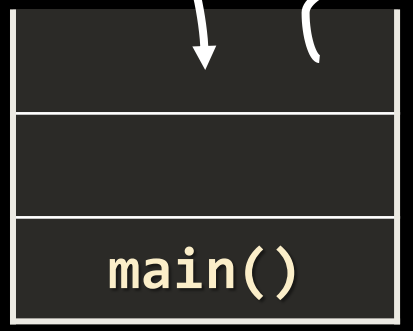
- The stack is a small fixed-size chunk of memory (e.g. 1MB)
  - Keeps the currently called functions in a stack data structure
  - Changes as the program enters / exits a function

```
int main()
{
    printf("Hello");

    return 0;
}
```

When **printf** is called it is pushed onto the stack

printf()

When **printf** returns it gets popped from the stack



Stack

The Stack – Example

- When the program executes line 5 the stack is as follows:

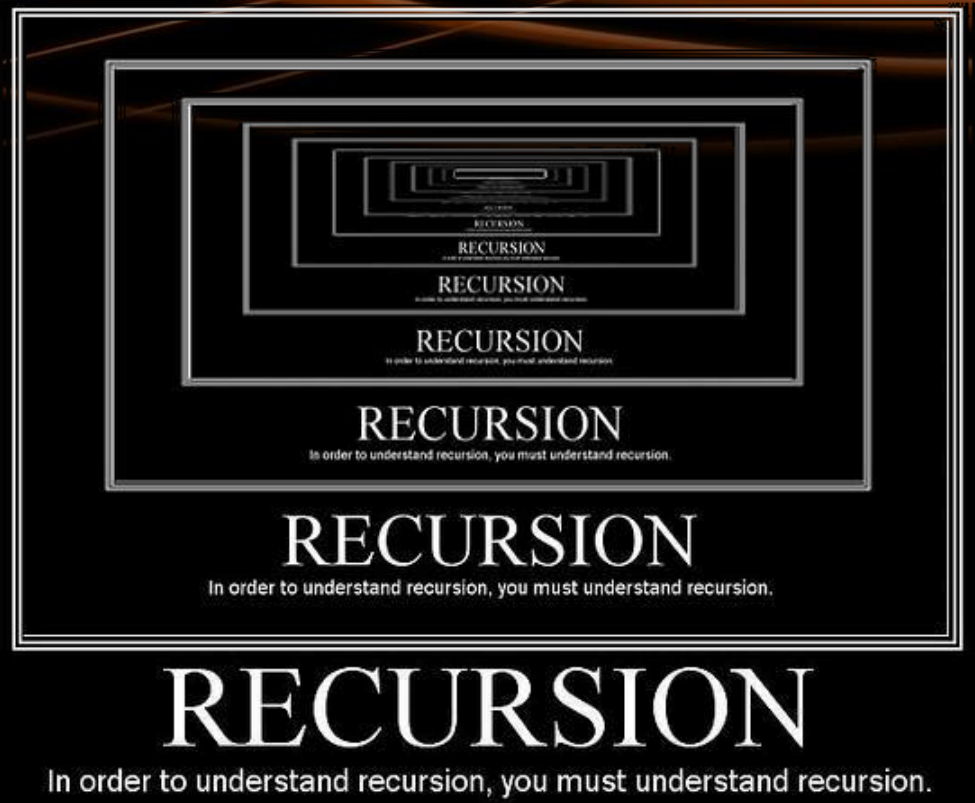
is_prime(5)	line 5
print_all_primes(5, 10)	line 13
main()	line 18

  - **main** called **print_all_primes** on line 18
  - **print_all_primes** called **is_prime** on line 13
  - **is_prime** currently executing line 5

```
1   int is_prime(int num)
2   {
3       int top = sqrt(num);
4       for (int i = 2; i <= top; i++)
5           if ((num % i) == 0)
6               return 0;
7
8       return 1;
9   }
10  void print_all_primes(int start, int end)
11  {
12      for (int i = 5; i <= 10; i++)
13          if (is_prime(i))
14              printf("%d ", i);
15  }
16  int main()
17  {
18      print_all_primes(5, 10);
19      return 0;
20  }
```

Debugging a Program

Call Stack Live Demo



Recursion

Recursive Functions In C

What is Recursion?

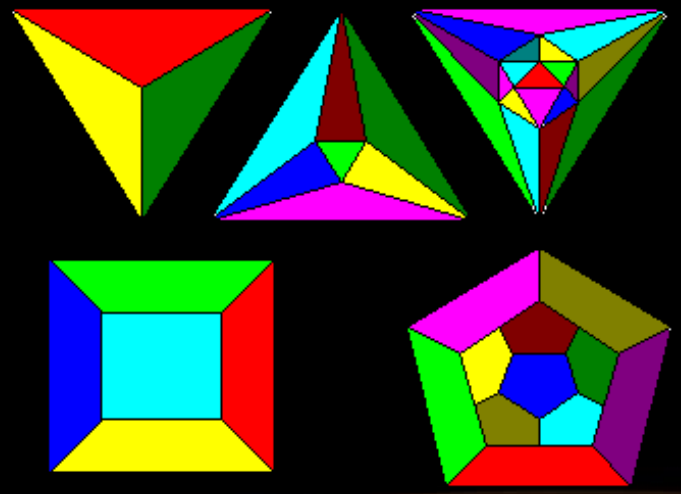
- Recursion is when a method calls itself
  - Powerful technique for combinatorial and branched search algorithms design
- Recursion should have:
  - Direct or indirect recursive call
    - The method calls itself directly
    - Or through other methods
  - Exit criteria (bottom)
    - Prevents infinite recursion

Recursive Factorial – Example

- Recursive definition of **n!** (n factorial):

```
n! = n * (n-1)! for n > 0
0! = 1
```

  - 5! = 5 * 4! = 5 * 4 * 3 * 2 * 1 * 1 = 120
  - 4! = 4 * 3! = 4 * 3 * 2 * 1 * 1 = 24
  - 3! = 3 * 2! = 3 * 2 * 1 * 1 = 6
  - 2! = 2 * 1! = 2 * 1 * 1 = 2
  - 1! = 1 * 0! = 1 * 1 = 1
  - 0! = 1



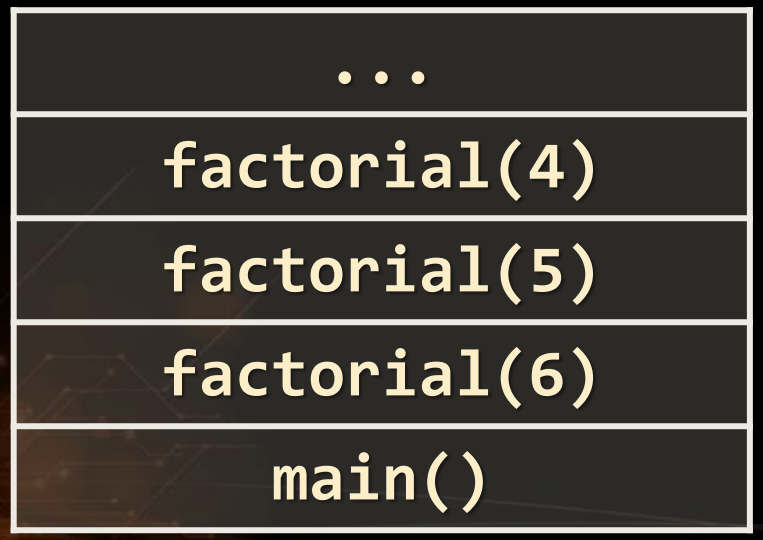
Recursive Factorial – Example

- Calculating factorial:
  - 0! = 1
  - n! = n* (n-1)!, n>0

```
long factorial(int num)
{
    if (num == 0)
        return 1;
    else
        return num * factorial(num - 1);
}
```

The bottom of the recursion

Recursive call: the method calls itself

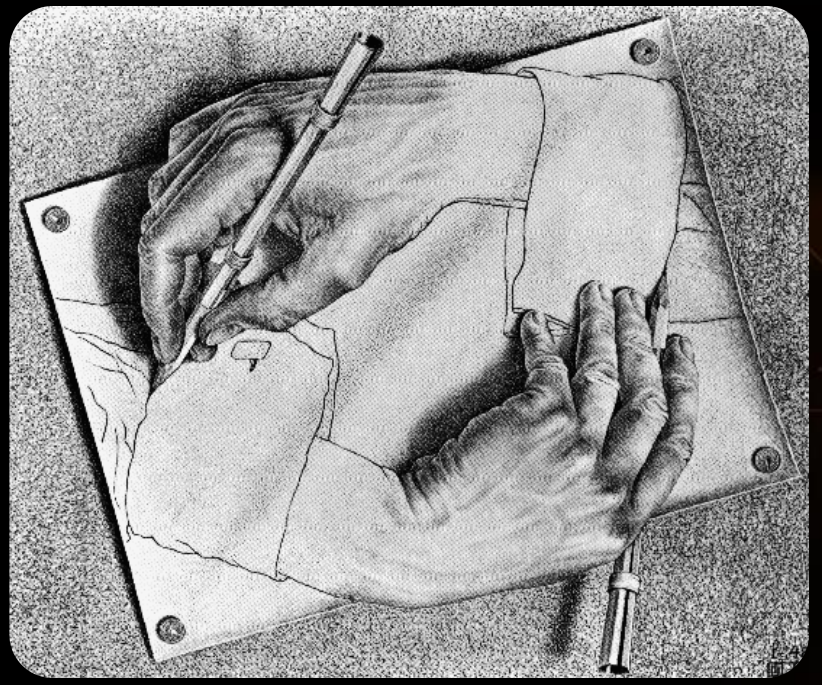


- Note: Recursion without a bottom can cause a stack overflow (i.e. stack runs out of memory)!

SOFTWARE UNIVERSITY FOUNDATION

Recursive Factorial

Live Demo



SOFTWARE UNIVERSITY FOUNDATION

Inline Functions

Injecting Function Code

Inline Functions in C

SOFTWARE UNIVERSITY FOUNDATION

- When a function is called a stack frame is allocated on the stack
  - That takes extra CPU time
  - Inline functions reduce this overhead by having their code injected into the calling function body
    - Suitable when small functions are called many times
    - Declared with the **`inline`** keyword

```
inline int multiply(int a, int b)
{ ... }
```

Inline Function – Example

```
inline int multiply(int a, int b)
{
    return a * b;
}

int main()
{
   int a, b, char sign;
   scanf("%d %c %d", &a, &b, &c);
   switch (sign)
   {
      case '*':
          int result = multiply(a, b);
          break;
      ...
```

```
int main()
{
   int a, b, char sign;
   scanf("%d %c %d", &a, &b, &c);
   switch (sign)
   {
      case '*':
          int result = a * b;
          break;
      ...
```

Inlined by the compiler

- Note: The compiler may decide not to inline

SOFTWARE UNIVERSITY FOUNDATION

```
int main()
{
    …
}
```

Inline Functions

Live Demo

SOFTWARE UNIVERSITY FOUNDATION

Function Error Handling

Function Error Handling

SOFTWARE UNIVERSITY FOUNDATION

- C has no built-in exception mechanism
  - Programmers have to manually check and handle errors
  - Error handling strategies:
    - Return `NULL` to signal error (e.g. `malloc()` when out of memory)
    - Take pointer and give it some status before the function returns (e.g. `0` -> success, `1` -> format error, `2` -> invalid range, etc.)
    - Set global `errno` variable some error code (e.g. `strtol()` on overflow sets `errno` to `ERANGE`)

Array Element Index – Example

- Returns the index of a element in array

```
int index_of(int array[], size_t size, int num)
{
    int i;
    for (i = 0; i < size; i++)
    {
        if (array[i] == num)
        {
            return i;
        }
    }

    return -1;
}
```

Returns -1 if no such element is found

Quadratic Equation Roots – Example

```
int calc_quadratic_equation(double roots[2], double a, double b, double c)
{
    double discriminant = (b * b) - (4 * a * c);
    if (discriminant < 0)
        return -1;
    else if (discriminant == 0)
    {
        double root = -b / 2 * a;
        roots[0] = root;

        return 1;
    }

    roots[0] = (-b + sqrt(discriminant)) / (2 * a);
    roots[1] = (-b - sqrt(discriminant)) / (2 * a);

    return 2;
}
```

Returns -1 if no roots exist

Array Sum – Example

```
double arr_sum(double arr[], size_t size, int *error)
{
    *error = 0;
    if (arr == NULL)
    {
        *error = 1;
        return 0;
    }

    int i, sum = 0;
    for (i = 0; i < size; i++)
    {
        sum += arr[i];
    }

    return sum;
}
```

Takes a pointer to a error variable

Sets error code **1** to error variable

SOFTWARE UNIVERSITY FOUNDATION

Calculating Quadratic Equation

Live Demo

SOFTWARE UNIVERSITY
FOUNDATION

Parsing Number with `strtol()`

Live Demo

Why Use Functions?

- More manageable programming
  - Splits large problems into small pieces
  - Better organization of the program
  - Improves code readability
  - Improves code understandability
- Avoiding repeating code
  - Improves code maintainability
- Code reusability
  - Using existing functions several times saves development time

Functions – Best Practices

- Each function should perform a single, well-defined task
- The function's name should describe that task in a clear and non-ambiguous way
  - Good examples: `calculate_price`, `readName`
  - Bad examples: `f`, `g1`, `Process`
  - In C there is no definite naming convention (camelCase, under_score, etc.) – just pick one and be consistent
- Avoid functions longer than one screen
  - Split them to several shorter functions

Summary

- Break large programs into simple functions that solve small sub-problems
- Functions consist of declaration and body
- Functions are invoked by their name and can take parameters
- Functions can return a value or nothing (void)
- The program stack keeps the currently called functions in a stack data structure

C Programming – Functions

SOFTWARE UNIVERSITY FOUNDATION

xsoftware

SmartIT

Komfo
SOCIAL MEDIA MARKETING AT SCALE

NETPEAK
интернет-маркетинг для бизнеса

Questions?

LUXOFT

SUPERHOSTING.BG

SoftwareGroup
doing it right

INDEAVR
Serving the high achievers

INFRAGISTICS
DESIGN / DEVELOP / EXPERIENCE

https://softuni.bg/trainings/1212/C-Programming-October-2015

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
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University @ YouTube
  - youtube.com/SoftwareUniversity
- Software University Forums – forum.softuni.bg