

C Data Types

Data Types, Variables, Declaring
and Initializing Variables



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



Table of Contents

1. Data Types

- Integer, Floating-Point
- Character, String

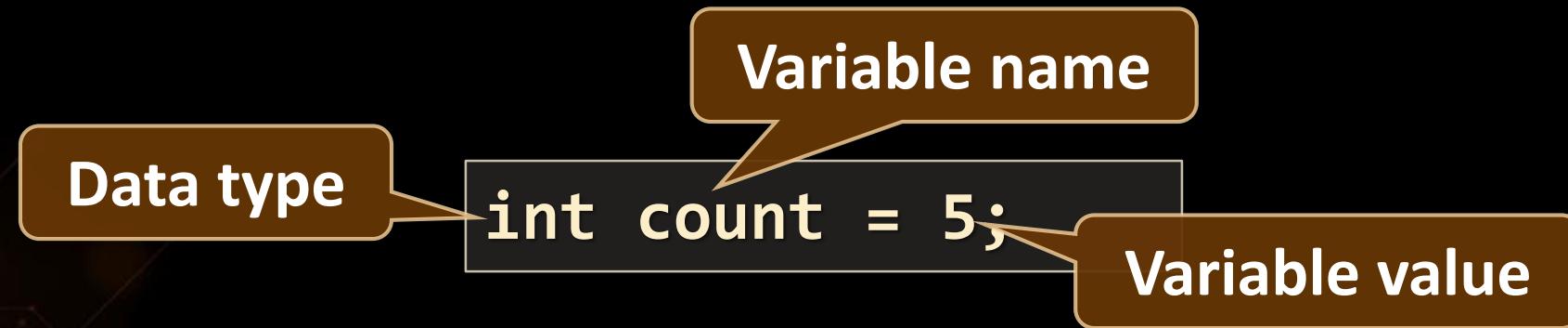
2. Declaring and Using Variables

- Identifiers
- Variables
- Literals



How Computing Works?

- Computers are machines that process data
 - Data is stored in the computer memory in variables
 - Variables have name, data type and value
- Example of variable definition and assignment in C



- When processed, data is stored back into variables

What Is a Data Type?

- A data type:
 - Is a domain of values of similar characteristics
 - Defines the type of information stored in the computer memory (in a variable)
- Examples:
 - Positive integers: **1, 2, 3, ...**
 - Alphabetical characters: **a, b, c, ...**
 - Days of week: **Monday, Tuesday, ...**



Data Type Characteristics

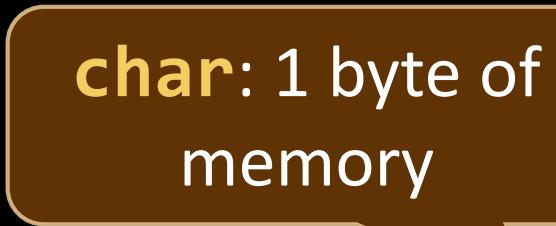
- A data type has:
 - Name (C keyword)
 - Size (how much memory is used)
- Example:
 - Characters in C
 - Name: **char**
 - Size: **8 bits** (1 byte)



A grid of binary digits (0s and 1s) representing a sequence of 8 bits. A small grey arrow points from the text "char: sequence of 8 bits in memory" to the grid.

```
0110000101100110011000010  
110001101100101011101101  
100010010100101101010111  
010100101101100100000110  
100001101001011010101100  
101011011100111010110010  
010010000010010101101000  
101101000011001011010101  
001000000100000011010010  
111001011001010110110011  
101101011100101011001111  
1100101110110001010110100  
0001110011110010101000001100  
110010101110001010100000110  
01010110000011001011010110  
00010000011100101100101001  
10110101111000101011101010  
00001110010010001010100000  
01100100011010101010100010  
00010100001010101010101001  
01010111000011010101101001  
11011100000010101010110010  
01010110000010101010110010  
00101010101010101010101010  
00101101011100110101110111  
01011101011100110101110111  
11001101011100110101110111  
1001100101001000001000000110  
1001100101001000001000000110  
1001100101001000001000000110  
1001100101001000001000000110
```

char: sequence of 8 bits in memory



A brown speech bubble pointing to a green computer RAM chip. The chip has several black rectangular components and yellow capacitors. The text "char: 1 byte of memory" is inside the bubble.

char: 1 byte of memory



A green glowing cube displays binary code in its faces: 10010, 100100, and 10.

int *sbyte*
uint *byte*
long **short**
ulong *ushort*

Integer Types

char, short, int, unsigned int, long

What are Integer Types?

- Integer types:
 - Represent whole numbers
 - May be signed or unsigned
 - Have range of values, depending on the size of memory used
- The default value of integer types is:
 - **0** – for integer types, except
 - **0L** – for the **long** type

int
long **short**

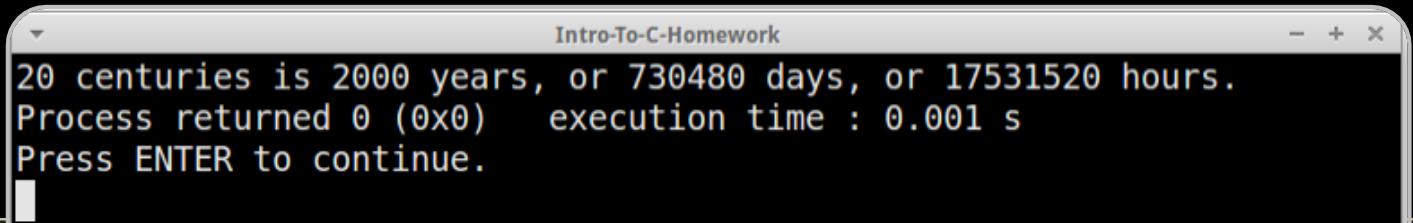


Integer Types

Type	Storage Size	Value Range
char	8-bit	-128 to 127
unsigned char	8-bit	0 to 255
short	16-bit	-32,768 to 32,767
unsigned short	16-bit	0 to 65,535
int	16- or 32-bit	$-2^{15}+1$ to $2^{15}-1$ or $-2^{31}+1$ to $2^{31}-1$
unsigned int	16- or 32-bit	0 to $(2*2^{15})-1$ or 0 to $(2*2^{31})-1$
long	32- or 64-bit	$-2^{31}+1$ to $2^{31}-1$ or $-2^{63}+1$ to $2^{63}-1$
unsigned long	32- or 64-bit	0 to $(2*2^{31})-1$ or $(2*2^{63})-1$
long long	64-bit	$-2^{63}+1$ to $2^{63}-1$
unsigned long long	64-bit	0 to $(2*2^{63})-1$

Measuring Time – Example

- Depending on the unit of measure we may use different data types:



```
Intro-To-C-Homework
20 centuries is 2000 years, or 730480 days, or 17531520 hours.
Process returned 0 (0x0)  execution time : 0.001 s
Press ENTER to continue.
```

```
char centuries = 20;           // A small number (up to 255)
unsigned short years = 2000;    // A small number (up to 32767)
unsigned int days = 730480;     // A large number (up to 4.3 billion)
unsigned long hours = 17531520; // A very big number (up to 18.4*10^18)
printf("%c centuries is %hu years, or %du days, or %lu hours.",
       centuries, years, days, hours);
```

- Note: Type range of **int** and **long** varies according to platform!

Integer Types

Live Demo

int *ushort*
unsigned int *long* **short**
long **unsigned long** *char*
unsigned long



Floating-Point

double



float

What are Floating-Point Types?

- Floating-point types:
 - Represent real numbers
 - May be signed or unsigned
 - Have range of values and different precision depending on the used memory
 - Can behave abnormally in the calculations

double

float



0.3425205
0.0000244
0.1044620
0.434780
0.1044780
0.2031416
0.56104473
0.46104408
0.25166416
0.02031416
0.0434780
0.1044620
0.0000244
0.3425205

Floating-Point Types

- Floating-point types are:
 - **float** ($\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$)
 - 32-bit, precision of 6 digits
 - **double** ($\pm 2.3 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$)
 - 64-bit, precision of 15 digits
 - **long double** ($\pm 3.4 \times 10^{-4932}$ to $\pm 1.1 \times 10^{4932}$)
 - 80-bit, precision of 19 digits

double

float



PI Precision – Example

- Difference in precision when using **float** and **double**:

```
float floatPI = 3.141592653589793238f;  
double doublePI = 3.141592653589793238d;
```

Locals	
floatPI	3,14159274
doublePI	3,1415926535897931

- The default value of floating-point types:
 - Is **0.0F** for the **float** type
 - Is **0.0D** for the **double** type
- NOTE: The "**f**" suffix in the first statement!
 - Real numbers are by default interpreted as **double**!
 - One should **explicitly** convert them to **float**

Abnormalities in the Floating-Point Calculations

- Sometimes abnormalities can be observed when using floating-point numbers
 - Comparing floating-point numbers can not be performed directly with the `==` operator

```
double a = 1.0f;  
double b = 0.33f;  
double correctSum = 1.33f;  
double actualSum = a + b;
```

Locals	
a	1
b	0,33000001311302185
correctSum	1,3300000429153442
actualSum	1,3300000131130219



Floating-Point and Decimal Floating-Point Types

Live Demo

Boolean Type

true

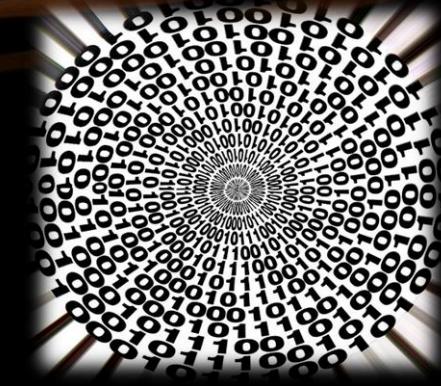


false



The Boolean Type

- The Boolean data type:
 - Has two possible values: **true** and **false**
 - Is useful in logical expressions
- In C there is no native boolean data type (no **bool** keyword)
 - **int** variables are used instead (**1** for true, **0** for false)



```
int isHappy = 1;  
int hasMoney = 0;
```

Boolean in C

- As an alternative **bool** can be defined as a macro

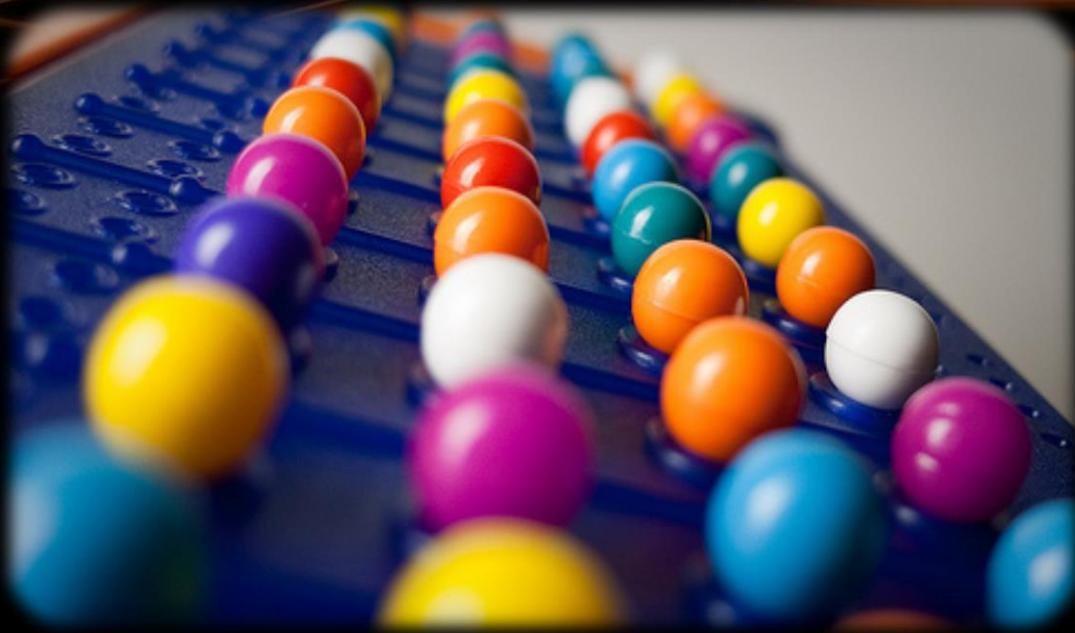
```
typedef int bool;  
#define true 1  
#define false 0  
  
int main()  
{  
    bool isHappy = true;  
    bool hasMoney = false;  
    return 0;  
}
```

- Or simply **#include <stdbool.h>**

Note: Compiler must support C99 standard

true

false



Boolean Type

Live Demo

The Character Data Type

- The **character** data type:
 - Represents symbolic information
 - Is declared by the **char** keyword
 - Can be assigned either a letter or an integer:

```
char letterA = 'A';
char letterB = 66;

printf("%c", letterA);
printf("%c", letterB);
```



A a B b C č Đ d Ě
N Θ Y Δ Μ Ж Й ڦ ڻ
س ع ڏ あ に サ
タ ڪ ڳ ۾ ۾ ٻ ٻ ٻ ٻ

char



Character Type

Live Demo



String Type

Donec eris felix, multos numerabis amicos
Μῆνιν δέιδε θεὰ Πηληϊάδεω Ἀχιλῆος
Παύδαν γετασπιαδή φίμην δεοραν σραεft, αnd . . .
phonetician /,fəʊnə'tɪʃən/ dog /dɒg/ bird /bɜ:d/
Й рече бгъ: да бѓдетъ свѣтъ. Й быств свѣтъ.
אֶבְרָאַשְׁיָה בְּרָא אַלְקִים וְאֵת הַאֲדֹנָיוֹת אֲבָגָדְעָזְקִים נְכָפָאַת צְחֹזָה
अथ कलेन महता स मत्स्यः सुमहानभूत्।

char*

char*

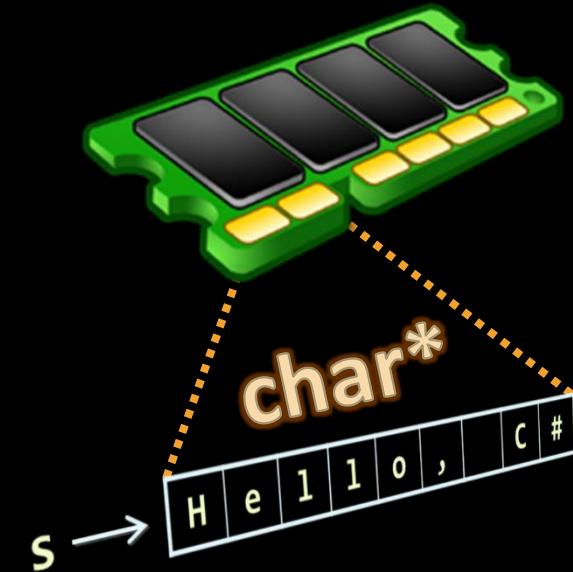
The String Data Type

- 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 terminator ('**\0**')

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

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

Null terminator character



String Type

Live Demo

char*





Introducing Variables



What Is a Variable?

- Variables keep data in the computer memory
- A **variable** is a:
 - Placeholder of information that can be changed at run-time
- Variables allow you to:
 - Store information
 - Retrieve the stored information
 - Change the stored information



Variable Characteristics

- A variable has:
 - Name
 - Type (of stored data)
 - Value
- Example:

```
int counter = 5;
```

- Name: **counter**
- Type: **int**
- Value: **5**



Declaring and Using Variables

counter



$$f(x) = e^x$$

$$f(x) = \sqrt[3]{x} * \sin(x)$$

$$(x) = 1 + x + x^2 + x^3 + x^4$$

$$f(x) = \arctan(\tan(x))$$

$$f(x) = \cos(\pi - x)$$

Declaring Variables

- When declaring a variable we:

height

- Specify its type
- Specify its name (called identifier)
- May give it an initial value



int

- The syntax is the following:

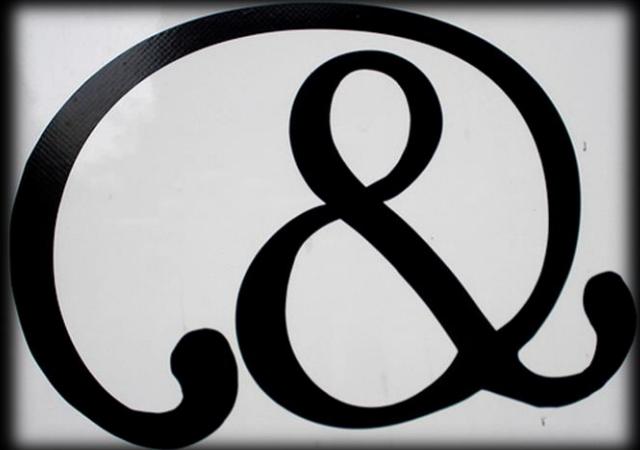
```
<data_type> <identifier> [= <initialization>];
```

- Example:

```
int height = 200;
```

Identifiers

- Identifiers may consist of:
 - Letters (Unicode)
 - Digits [0-9]
 - Underscore "_"
- Examples: **count, firstName, Page**
- Identifiers
 - Can begin only with a letter or an underscore
 - Cannot be a C keyword (like **int** and **void**)



Identifiers (2)

- Identifiers
 - Should have a descriptive name
 - E.g. **firstName**, not **dasfas** or **p17**
 - It is recommended to use only Latin letters
 - Should be neither too long nor too short
- Note:
 - In C small letters are considered different than the capital letters (**case sensitivity**)



Identifiers – Examples

- Examples of syntactically correct identifiers:

```
int New = 2; // here N is capital
int _2Pac;   // this identifier begins with underscore _

char greeting[] = "Hello"; // more appropriate name

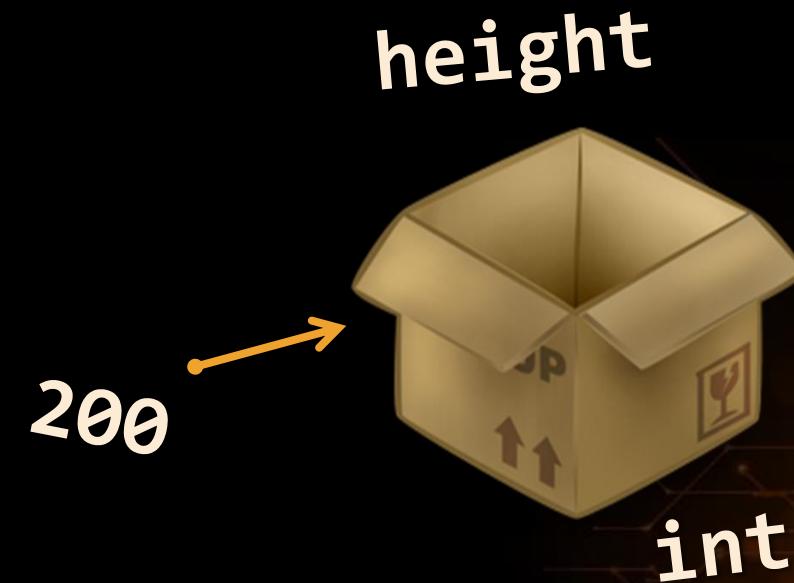
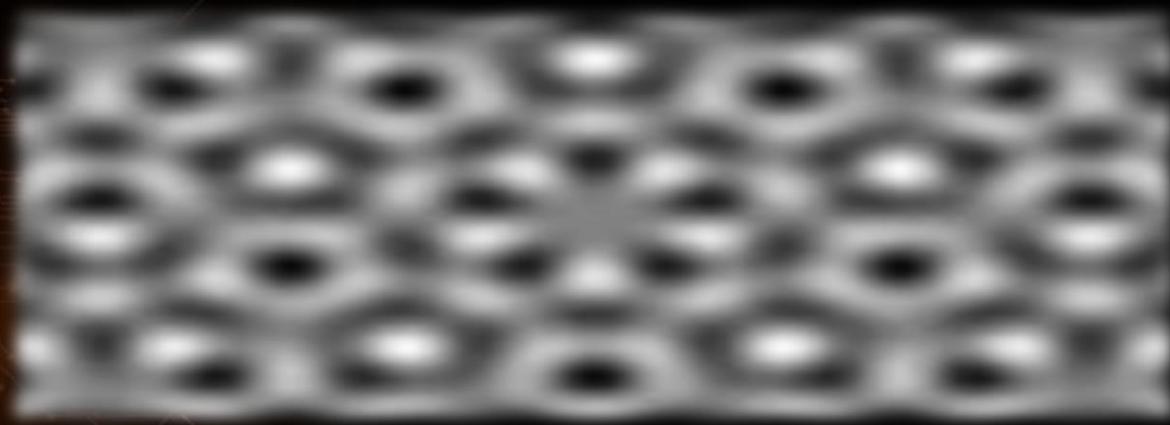
int n = 100; // undescriptive
int numberOfClients = 100; // good, descriptive name

int numberOfPrivateClientsOfTheFirm = 100; // overdescriptive
```

- Examples of syntactically incorrect identifiers:

```
int char; // char is a keyword
int 2Pac; // cannot begin with a digit
```

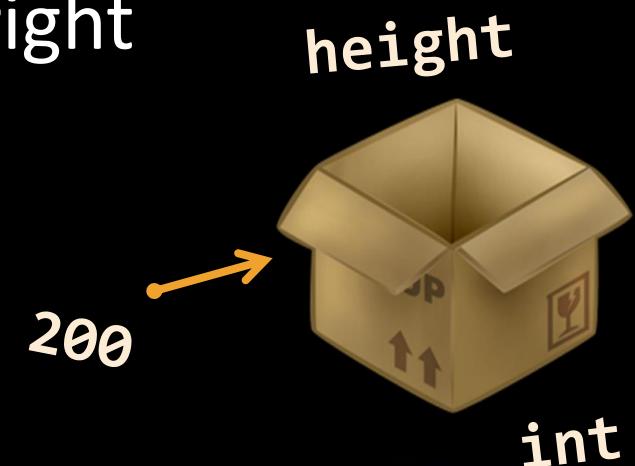
Assigning Values To Variables



Assigning Values

- Assigning values to variables
 - Use the `=` operator
- The `=` operator
 - Holds a variable identifier on the left
 - Value of the corresponding data type on the right
 - Or expression of compatible type
 - Could be used in a cascade calling
 - Where assigning is done from right to left

`=`



Assigning Values – Examples

```
int firstValue = 5;  
int secondValue;  
int thirdValue;
```

```
// Using an already declared variable:  
secondValue = firstValue;
```

```
// The following cascade calling assigns  
// 3 to firstValue and then firstValue  
// to thirdValue, so both variables have  
// the value 3 as a result:
```

```
thirdValue = firstValue = 3; // Avoid cascading assignments!
```



Initializing Variables

- Initializing
 - Means "to assign an initial value"
 - Must be done before the variable is used!
- Two ways of initializing:
 - By using a literal expression
 - By referring to an already initialized variable



Initialization – Examples

- Example of variable initializations:

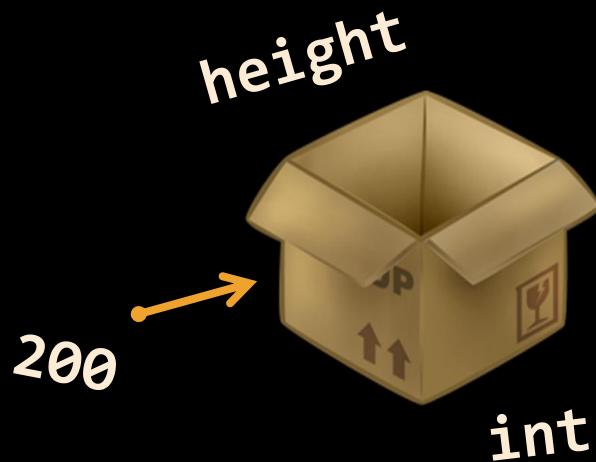
```
// The following would assign the default
// value of the int type to num:
int num; // num = 0

// This is how we use a literal expression:
float heightInMeters = 1.74f;

// Here we use an already initialized variable:
char greeting[] = "Hello World!";
char *message = greeting;
```

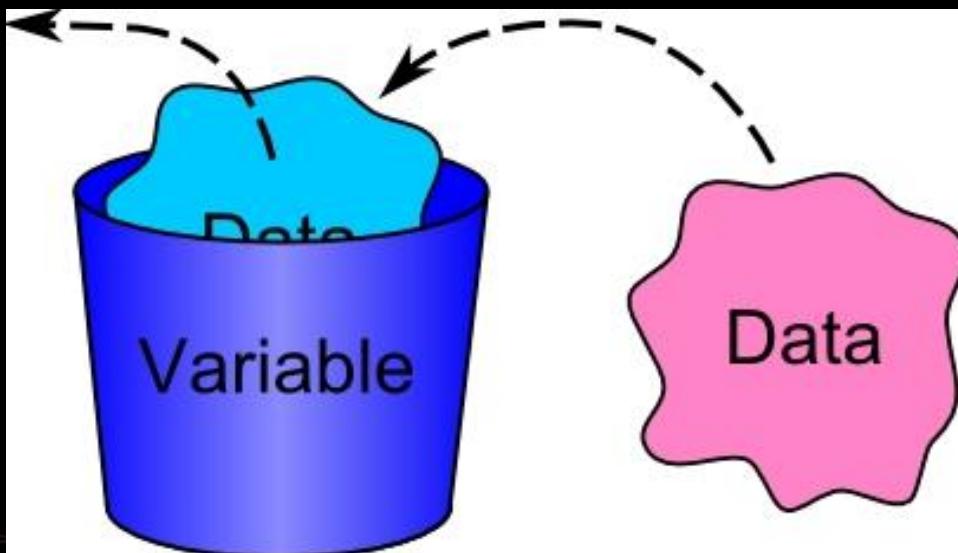
Assigning and Initializing Variables

Live Demo



Assigning and Initializing Variables

Exercise



Literals

true

null

-0.307f

@"Hi\r\n"

2034567324L

3.14159206

"\t\r\n\r\n\r\n\r\n"

0xFE

12.76m

6.02e+23

"\uE52B"

What are Literals?

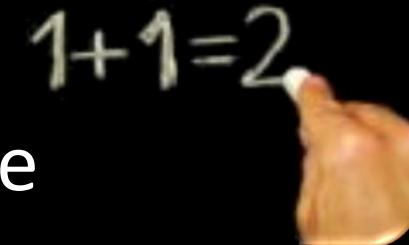
- Literals are:
 - Representations of values in the source code
- There are several types of literals
 - Integer
 - Real
 - Character
 - String
 - The **NULL** definition



Integer Literals

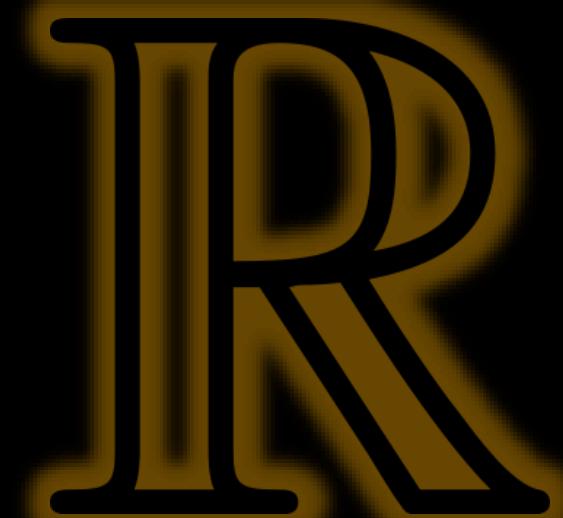
- Examples of integer literals:

- The '**0x**' and '**0X**' prefixes mean a **hexadecimal value**
 - E.g. **0xFE**, **0xA8F1**, **0xFFFFFFFF**
- The '**u**' and '**U**' suffixes mean an **unsigned long** or **unsigned int**
 - E.g. **12345678U**, **0U**
- The '**l**' and '**L**' suffixes mean a **long** or **unsigned long**
 - E.g. **9876543L**, **0L** , **1853287583258321ULL**



Real Literals

- The real literals:
 - Are used for values of type **float** and **double**
 - May consist of digits, a sign and ". "
 - May be in exponential notation: **6.02e+23**
- The "**f**" and "**F**" suffixes mean **float**
- The "**d**" and "**D**" suffixes mean **double**
- The default interpretation is **double**



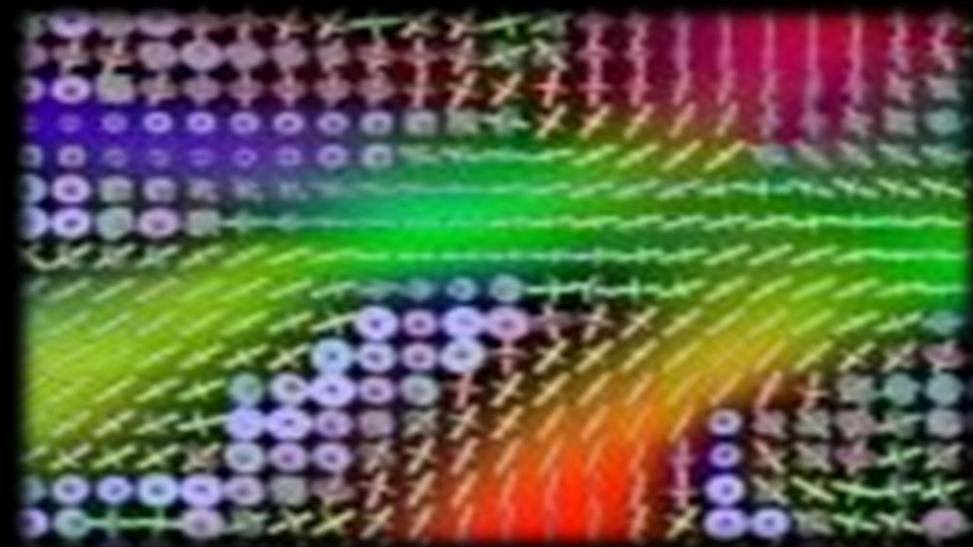
Character Literals



- The character literals:
 - Are used for values of the **char** type
 - Consist of two single quotes surrounding the character value:

```
'<value>'
```

- The value may be:
 - Symbol
 - The code of the symbol
 - Escaping sequence



Escaping Sequences

- Escaping sequences are:
 - Means of presenting a symbol that is usually interpreted otherwise (like ')
 - Means of presenting system symbols (like the new line symbol)
- Common escaping sequences are:
 - \' for single quote \" for double quote
 - \\ for backslash \n for new line

Character Literals – Example

- Examples of different character literals:

```
char symbol = 'a'; // An ordinary symbol  
  
symbol = '\''; // Assigning the single quote symbol  
  
symbol = '\\'; // Assigning the backslash symbol  
  
symbol = '\n'; // Assigning new line symbol  
  
symbol = '\t'; // Assigning TAB symbol  
  
symbol = "a"; // Incorrect: use single quotes
```

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:

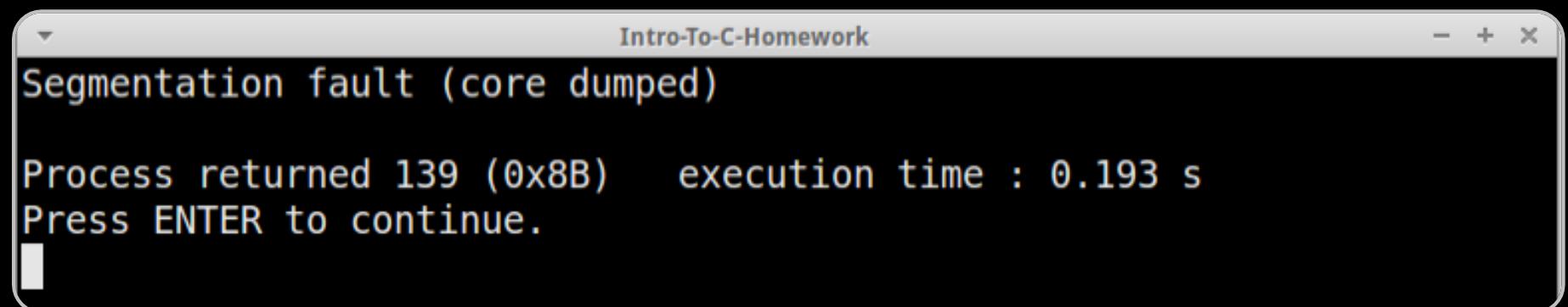
```
char firstName[6] = "Petar";
char lastName[] = "Georgiev";
```

String Literals (2)

2. Pointer declaration:

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

- **Warning:** string cannot be modified, will cause segmentation fault because it is located in read-only memory



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

String Literals

Live Demo



C Programming – Data Types



Questions?

SUPERHOSTING.BG



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

