# INF1004 procedural programming in C

DHBW Stuttgart
Christian Holz
christian.holz@lehre.dhbw-stuttgart.de

# Lecture 01

PART I

- Recap of the previous contents
- Functions
- Pointers
- Array

PART II

- Call by Value
- Call by Reference
- Structures
- Unions

# Lecture 01

**PART I**

- **Recap of the previous contents**
- **Functions**
- **Pointers**
- **Array**

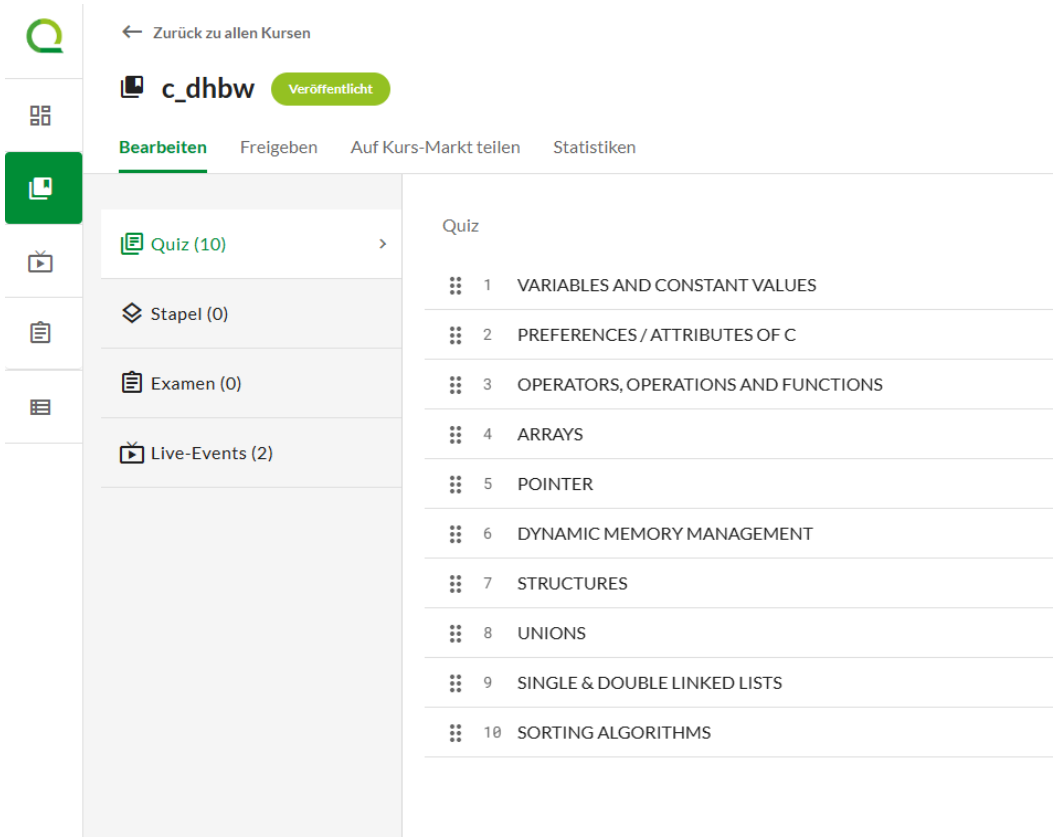
PART II

- Call by Value
- Call by Reference
- Structures
- Unions

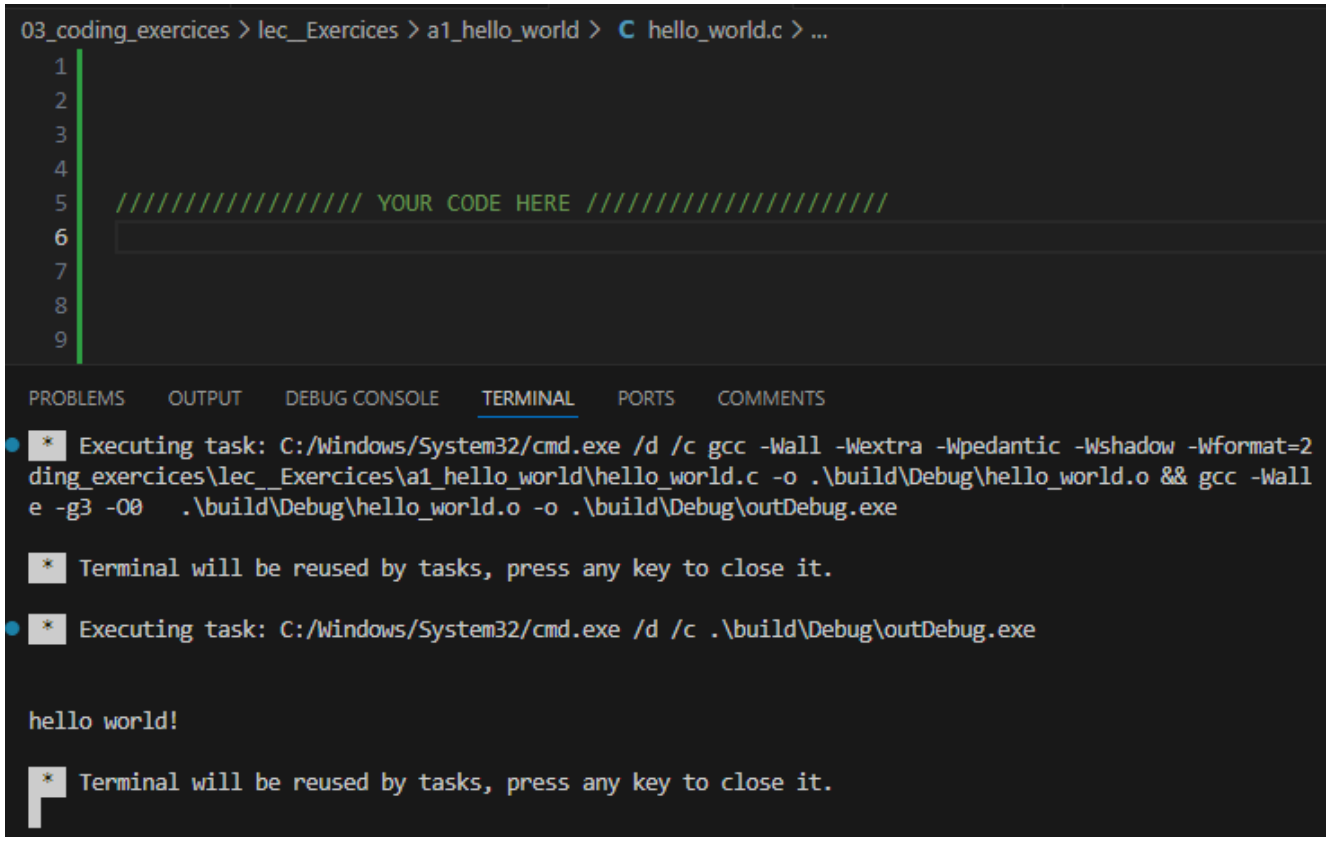
# Recap of the previous contents

Live event

# Let’s play

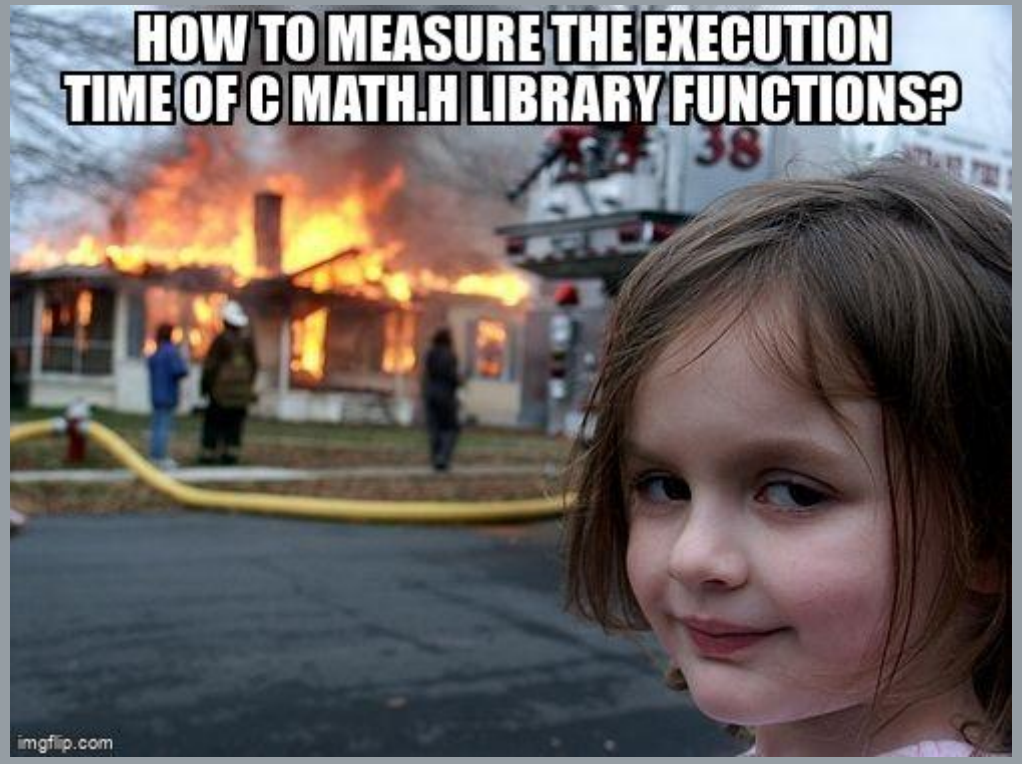


# Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# Functions and Libaries

**What is a Function in Programming?**



# Functions and Libaries
## Structure of a function

**Every function in C has the following components**

- Return value: The type of value that the function returns. If the function does not return a value, void is used.
- Function name: The name under which the function is called.
- Parameter list: A list of values that are passed to the function.
- Function body: The block of code that is executed when the function is called.

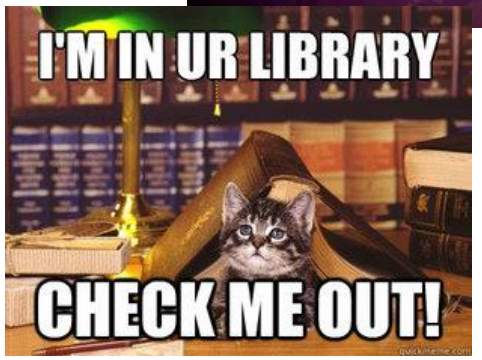
```c
size_t count(int32_t *array, size_t length, int32_t value)
{
    if (array == NULL)
    {
        return 0;
    }

    size_t counter = 0;

    for (size_t i = 0; i < length; i++)
    {
        if (array[i] == value)
        {
            counter++;
        }
    }

    return counter;
}
```

# Functions and Libaries
## What is a library?

- A library in C is
  - a collection of functions and other definitions
  - that are defined in separate files
  - and can be reused by different programmes
- C has standard libraries (e.g. stdio.h for input/output)

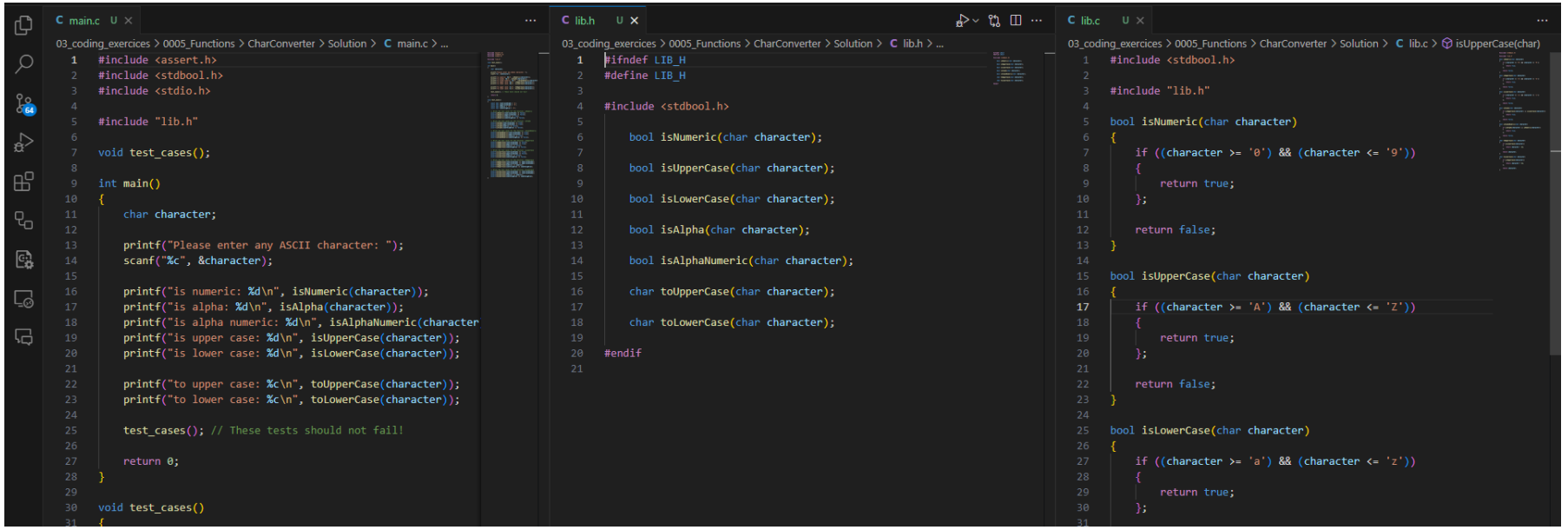
**We can also create your own libraries**

- header files (.h): Contain the function declarations (prototypes) and constants provided by the library.
- implementation files (.c): Contain the function definitions, i.e. the actual code of the functions.
- Include the header file: To use functions from a library, the associated header file must be included using #include.

# Functions and Libaries
## Advantages of Functions and Libaries

- Reusability: Once written, functions can be used in different parts of the program
- Clarity: Functions help to divide the code into smaller, clear sections
- Maintainability: Changes to a function only need to be made in one place instead of everywhere in the code
- Modularity: Programmers can be made more modular by giving each function a clearly defined task



# Functions and Libaries
## Compilationsteps

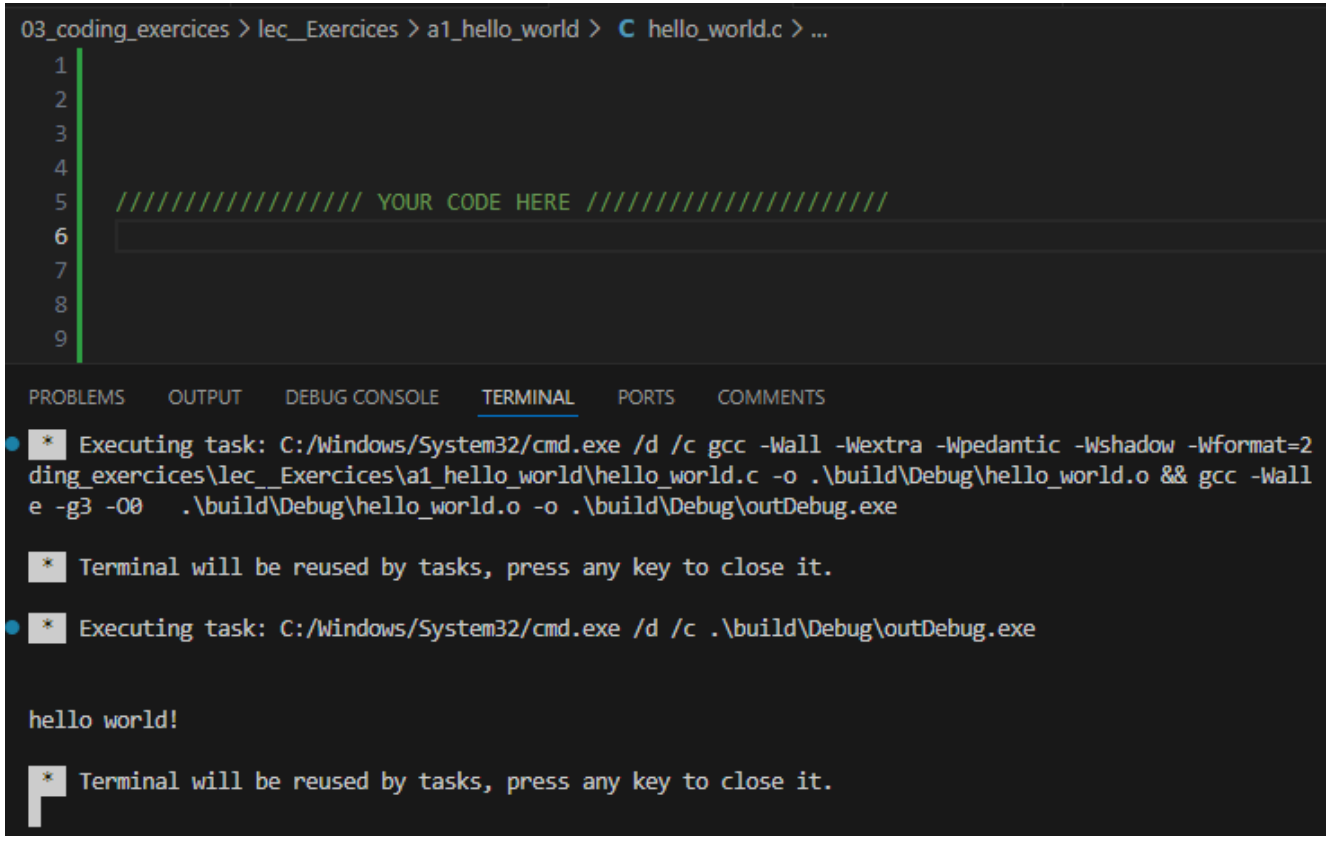
To compile your main.c, lib.h, and lib.c files into one executable file, proceed as follows:

1. Compile the lib.c into an object file (lib.o): `gcc -c lib.c -o lib.o`

2. Compile main.c including lib.h and left with lib.o: `gcc main.c lib.o -o my_program`


1+2: `gcc main.c lib.c -o my_program`

# Let's code

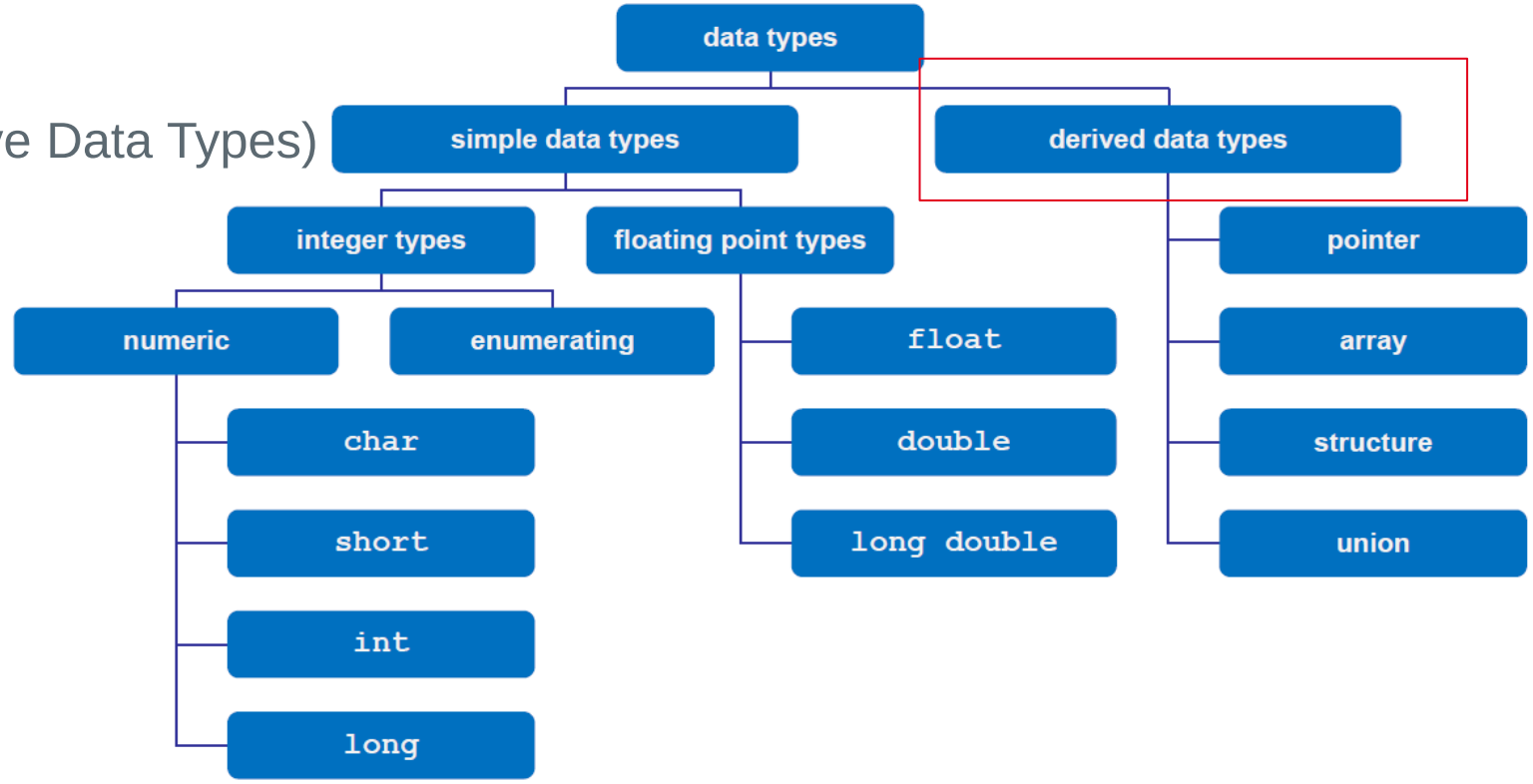

03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# Classes of Data Types

**What Data Type Classes do you know?**

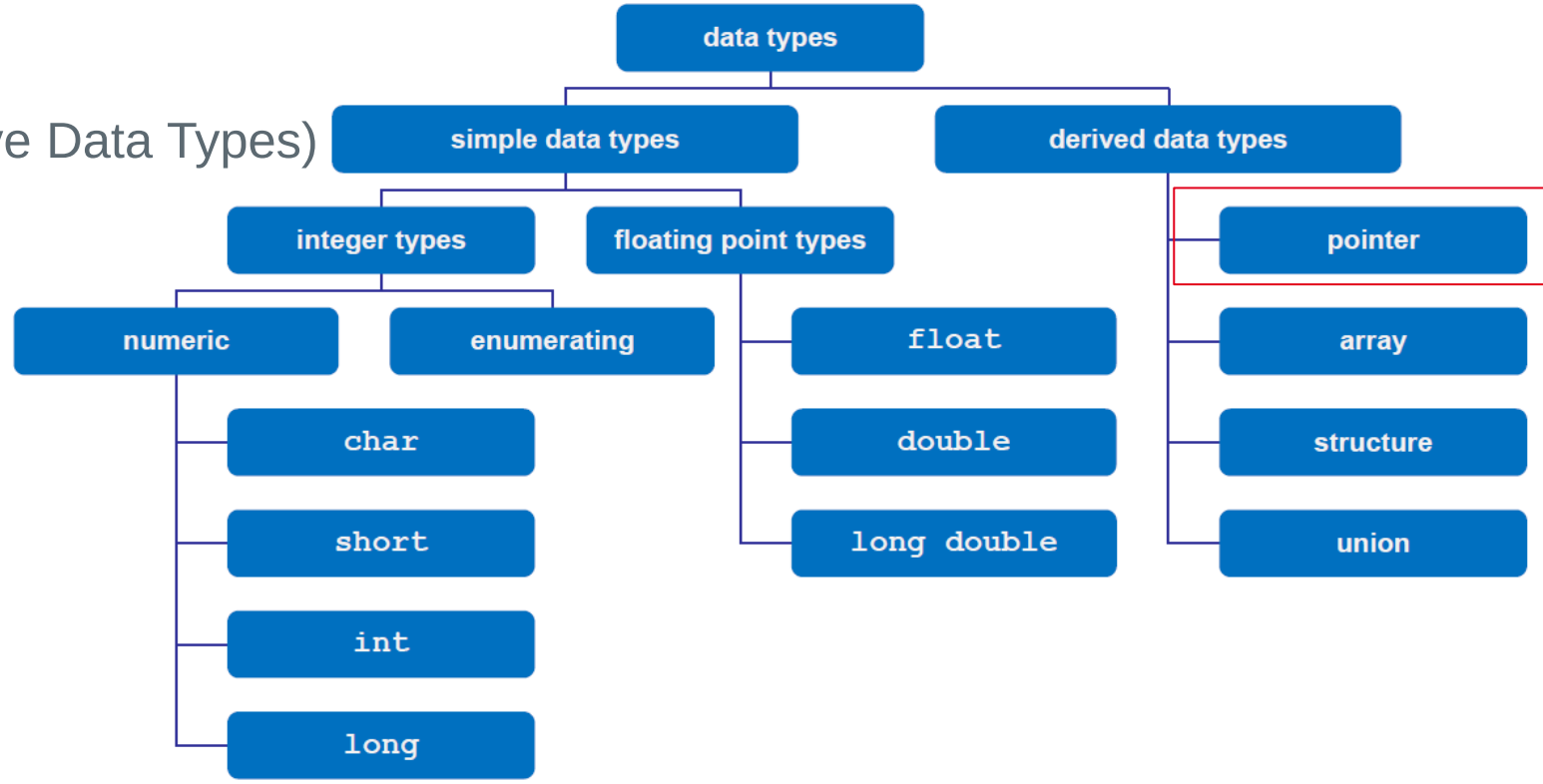
# Classes of Data Types
## Overview

(Primitive Data Types)



# Classes of Data Types
## Overview

(Primitive Data Types)

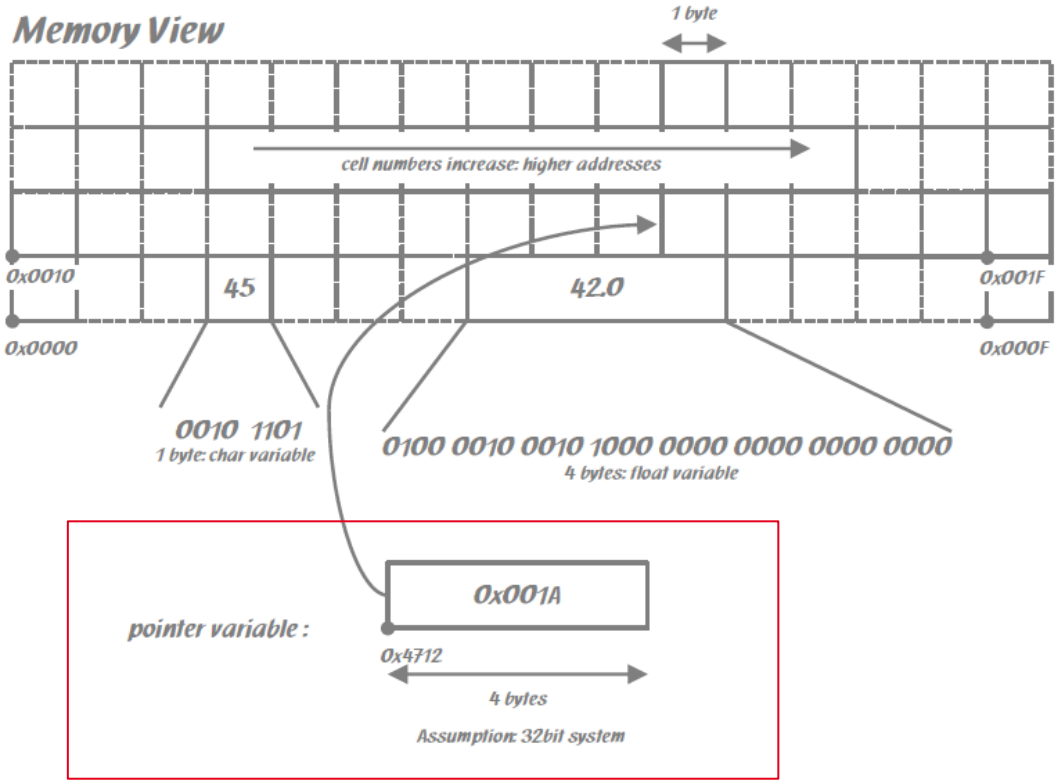



# Classes of Data Types
## Derived Data Types – Pointers

A pointer is a variable that stores the memory address of another variable

This means:

- instead of saving the value of a variable directly
- a pointer contains the address of the memory location where this value is located




The Computer's memory is divided into dedicated **cells of storage**. Each storage cell has an **number**, which we call an **address**. The computer's storage can be accessed by addresses **byte by byte** (Remember: 1 byte = 8 bits).

Simplified: Each byte of the computer's memory has and address. You cannot access single bits.

# Classes of Data Types
## Derived Data Types – Pointers

A **pointer is a variable** has the four known characteristics:

- data type
- address
- name
- value

It's value is an address.

The pointer that references another variable must **comply with the variable's type**.

A pointer to int (`int`) variable must be of the pointer type int (`int *`).

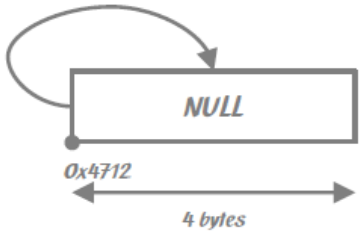
```
#include <stdio.h>

int main (void)
{
   int a;                  // int variable
   int * ptr;              // pointer variable
   float f;                // float variable
   float * f_prt;          // float pointer

   ...

   return (0);
}
```

# Classes of Data Types
## Derived Data Types – Pointers

The value that a pointer hosts is valid as long as is a **defined address** in the system.

**NULL** is a specific value. Assigned to a pointer variable the pointer is set to “not valid”.

If the pointer is not set to specific address, best practise is, always to **initialize a pointer** with the value NULL.




Information: NULL is a pre-defined symbolic constant.

```
#include <stdio.h>

int main (void)
{
   int * ptr = NULL;                    // int variable

   ...

   return (0);
}
```

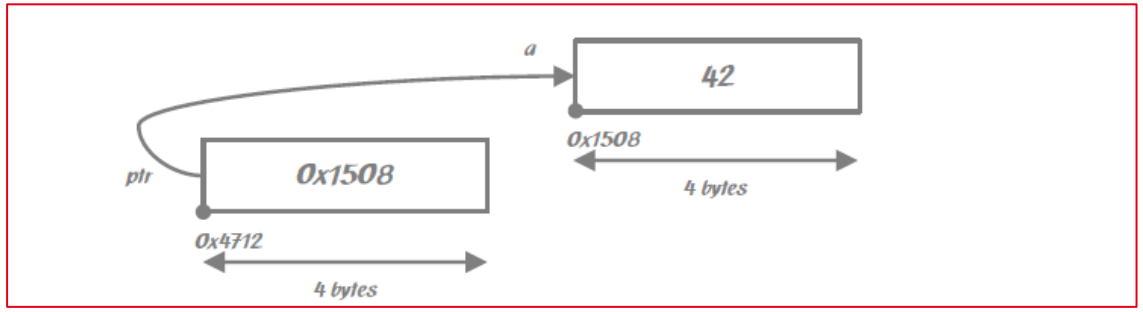
A defined pointer variable **only** allocates memory space for an address. There is no space reserved to store any other variable value.

# Classes of Data Types
## Derived Data Types – Pointers Operators

For accessing pointer there are two important operators available in C:

- `&` referencing operator
- `*` de-referencing operator

The **referencing operator** returns the address of a variable. The **de-referencing operator** returns the value of the referenced variable (content).



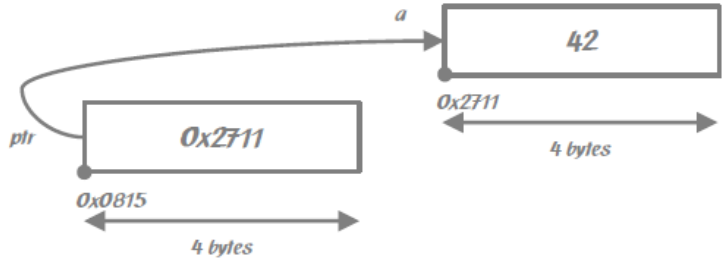

```
#include <stdio.h>

int main (void)
{
   int a = 42;
   int * ptr = NULL;
   ptr = & a;

   printf ("value: %d\n", a);
   printf ("value: %d\n", *ptr);

   ...

   return (0);
}
```

# Classes of Data Types
## Derived Data Types – Pointers Operators

Manipulating data with pointers:

```
int a = 42;
int * ptr;
ptr = &a;
*ptr = *ptr + 1;
```

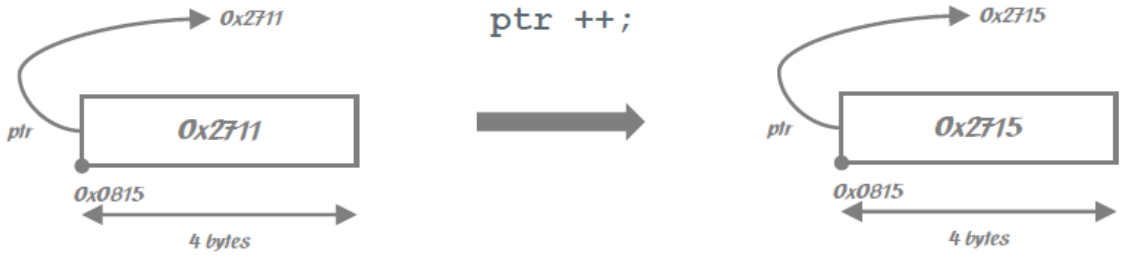
The statement `*ptr = *ptr + 1;` increases the value of a by `1`.




**Increasing** the pointer variable will change the address store with the pointer **by one step**.

The address is no necessarily increased by `1`.
The increase depends on the pointer's type:

- type char: increase by 1
- type int: increase by 4
  (assumption 32bit system)



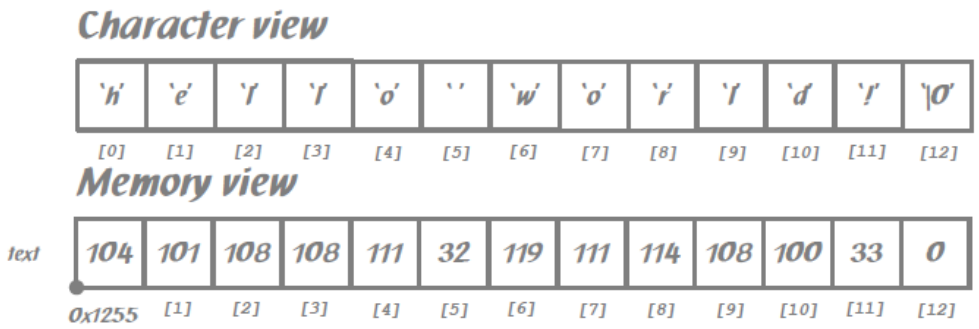
# Classes of Data Types
## Derived Data Types – Pointers Array

Summary:

An array

- combines variables with the **same data type**
- Contains **multiple** variable

The `[]` operator is congruent with the de-referencing operator.




Pre-Requisite:

```
char text [13] ="hello world!";
char * ptr;
```

If the `[]` operator **de-references** the variable, the variable name itself (without the `[]` operator) is the reference:

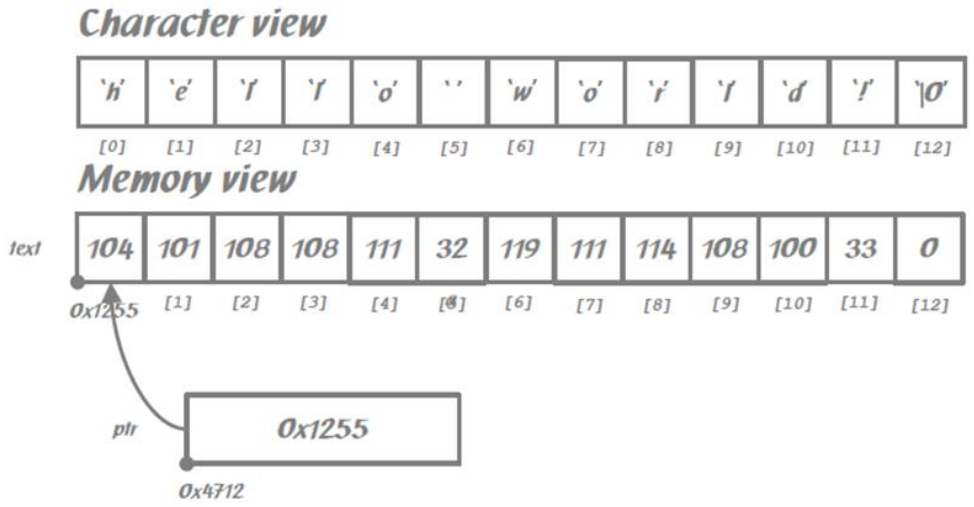
```
ptr = text;
```

The name of an array returns an address. The **array's name** also called a **vector**.

# Classes of Data Types
## Derived Data Types – Pointers Array

A pointer can be used to **run through an array**.

Base for this is the fact, that all elements of an array are **next to each other** and increasing an pointer will point to the next elements.

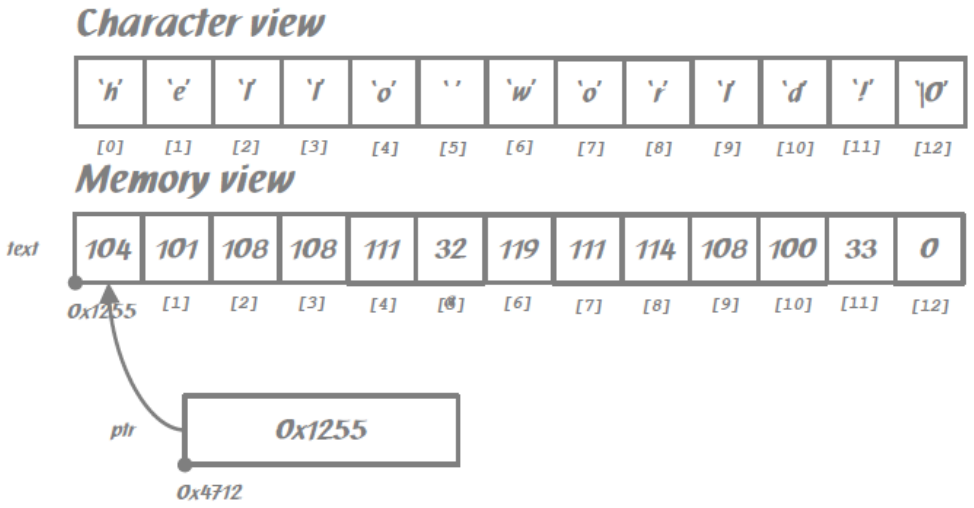



```c
#include <stdio.h>

int main (void)
{
   int i;

   char text[13] = "hello world!";
   char * ptr;

   ptr = text;
   printf ("Text: ");

   for (i=0;i<12;i++) {
      printf ("%c", *ptr++);
   }

   printf ("\n");


   return (0);
}
```

# Classes of Data Types
## Derived Data Types – Pointers Array

A pointer can be used to **run through an array**.

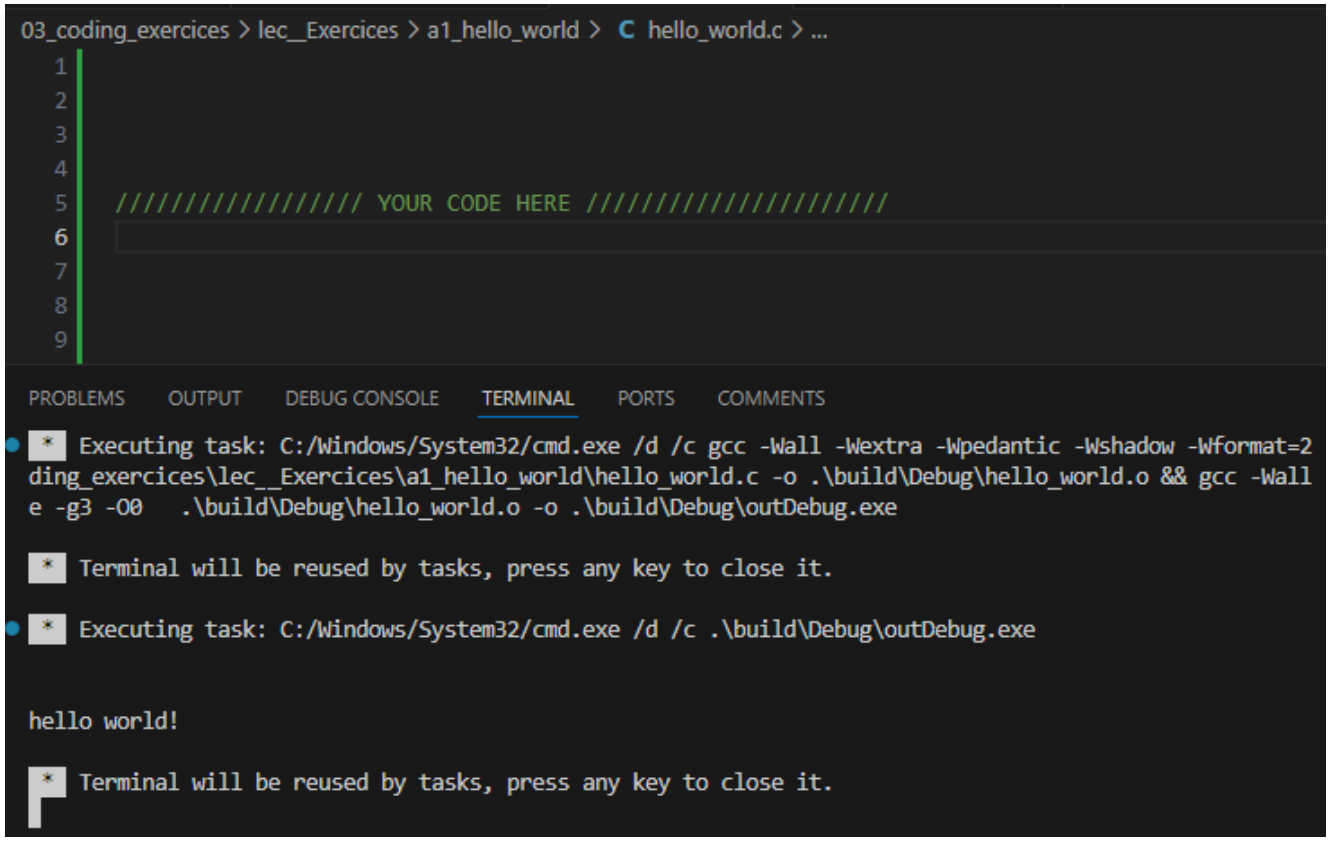
The example is the same but the solution is different. The previously used **`for`**-loop has been changed into a **`while`**-loop.




```c
#include <stdio.h>

int main (void)
{
   int i;

   char text[13] = "hello world!";
   char * ptr;

   ptr = text;
   printf ("Text: ");

   while (*ptr) {
      printf ("%c", *ptr++);
   }

   printf ("\n");


   return (0);
}
```

loop works?
why / why not

# Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2
ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall
e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# ~~Classes of Data Types~~ Intro to `assert()` in C
For Developers

The assert macro in C is a debugging tool provided by the standard library, which helps developers

- catch logic errors early in the development process

```c
#include <assert.h>

void someFunction(int value) {
    assert(value != 0);  // Program terminates if value equals 0
    // Rest of the function
}
```

Key Features of assert:

- If the condition provided to assert is true (non-zero), nothing happens, and the program continues
- If the condition is false (zero), the program prints an error message and terminates
- assert is included in the `<assert.h>` header file.

# ~~Classes of Data Types~~ Intro to `assert()` in C
For Developers

Use Cases for assert

- Testing Pre-conditions:
  You can use assert to verify function arguments.
  Example: Ensure a pointer is not NULL before dereferencing it.
- Invariants:
  Invariants are conditions that must always hold true during the execution of a program.
  Example: In a sorting algorithm, you might assert that the array remains within certain bounds.
- Debugging:
  During the development phase, assert helps to catch logic errors early.

```c
#include <assert.h>

void someFunction(int value) {
    assert(value != 0);  // Program terminates if value equals 0
    // Rest of the function
}
```

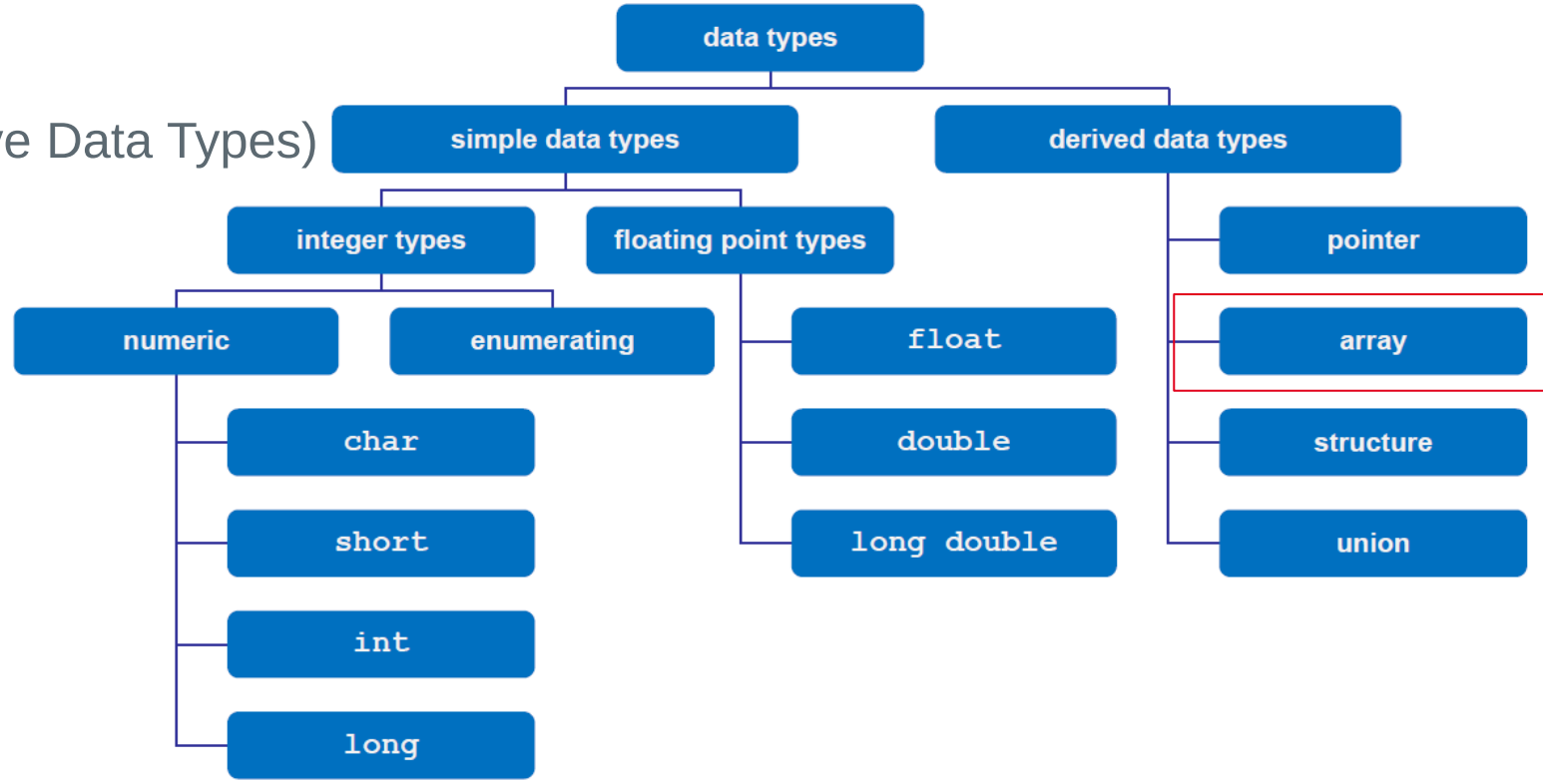
In production builds, assertions can be disabled by defining `NDEBUG` before including `<assert.h>`.

This turns assert into a no-op (no operation).

```c
#define NDEBUG
#include <assert.h>
```

# Classes of Data Types
## Overview

(Primitive Data Types)

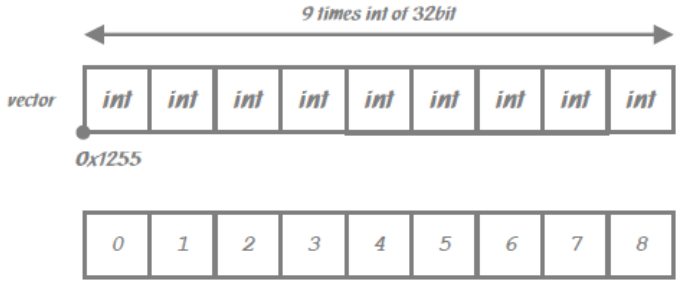



Accessing elements in an array in C/C++ be like

# Classes of Data Types
## Derived Data Types – Arrays

- An **array** belongs to the **derived data types**
- able to host **multiple elements** (variables) of the **same type**

(The derived data type "structure" can hos multiple elements of different data types.)




- The **array** itself -once defined-is a **single variable**
- has the same characteristics as a variable (name, address, data type, value).
- The single values of an array have an **index number**
- **Index number** use to access the single values.

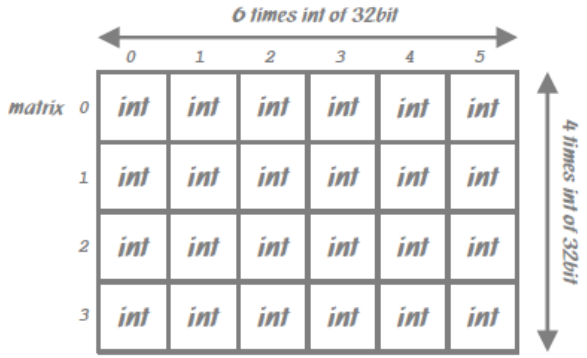
Definition of the array above:

```
data_type array_name [dimension];
int vector [9];
```

**Important:**
The index numbering starts with 0 (zero).

# Classes of Data Types
## Derived Data Types – Arrays

Arrays can have multiple dimension.




Definition of the array above:

```
int matrix [6][4];
```

Remember: If you do not initialize a variable it has a random value.

There are different way to initialize an array:

- at definition time
- at runtime

Initialization at definition time:

```
int matrix [6][4]={0};
int matrix [6][4]={0,0,0,0,0};
```

# Classes of Data Types
## Derived Data Types – Arrays

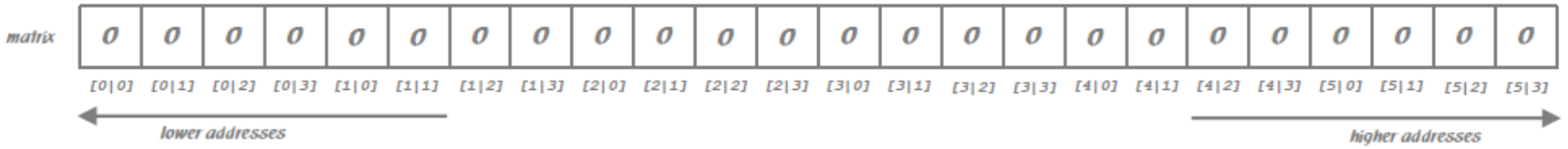
Initialization at runtime / during the program:

```
for (i=0;i<6;i++)
{
   for (j=0;j<4;j++)
   {
      matrix [i][j]=0;
   }
}
```

*Logical view*

| matrix | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |

*Memory view*



# Classes of Data Types
## Derived Data Types – Arrays

### Formatted Output of Array Elements

Usage of **`printf()`** with arrays depends on the data type of the array elements.

```
int matrix[6][4]={0};
matrix[1][1]=10;
printf ("value is : %d", matrix[1][1]);    // output of an int array element


float matrix[6][4]={0.0};
matrix[1][1]=2.5;
printf ("value is : %5.2f", matrix[1][1]); // output of a float array element
```

# Classes of Data Types
## Derived Data Types – Arrays

**Caution:**

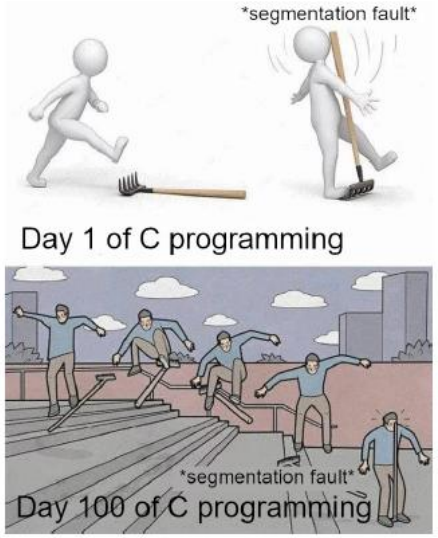
- When using loops to run through an array
- **no mechanism to detect the end of an array**

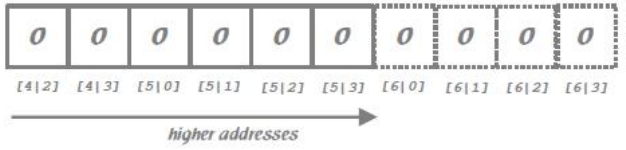
(Especially when writing values the loop can exceed the border of the array and will write into unallocated memory space.)

- C Programmers need to take care about the correct index numbers

**Example Code:**

```
for (i=0;i<7;i++)
{
   for (j=0;j<4;j++)
   {
      matrix [i][j]=0;
   }
}
```



# Lecture 01

PART I

- Recap of the previous contents
- Functions
- Pointers
- Array

**PART II**

- **Call by Value**
- **Call by Reference**
- **Structures**
- **Unions**

# Call by Value vs. Call by Reference

**What is meant by this?**

# Call by Value vs. Call by Reference
## Passing arguments to functions

- When programming in C (and other languages), there are two main methods for passing arguments to functions:

1. Call by value
   - Definition: With call by value, a copy of the argument is passed to the function.
   - Functionality:
     - **Changes to the copy within the function have no effect on the original.**
     - The original remains unchanged as the function only works with the copy.

2. Call by Reference
   - Definition: With Call by Reference, the address of the argument is passed to the function.
   - Functionality:
     - **The function works directly with the original value.**
     - Changes within the function have a direct effect on the original.

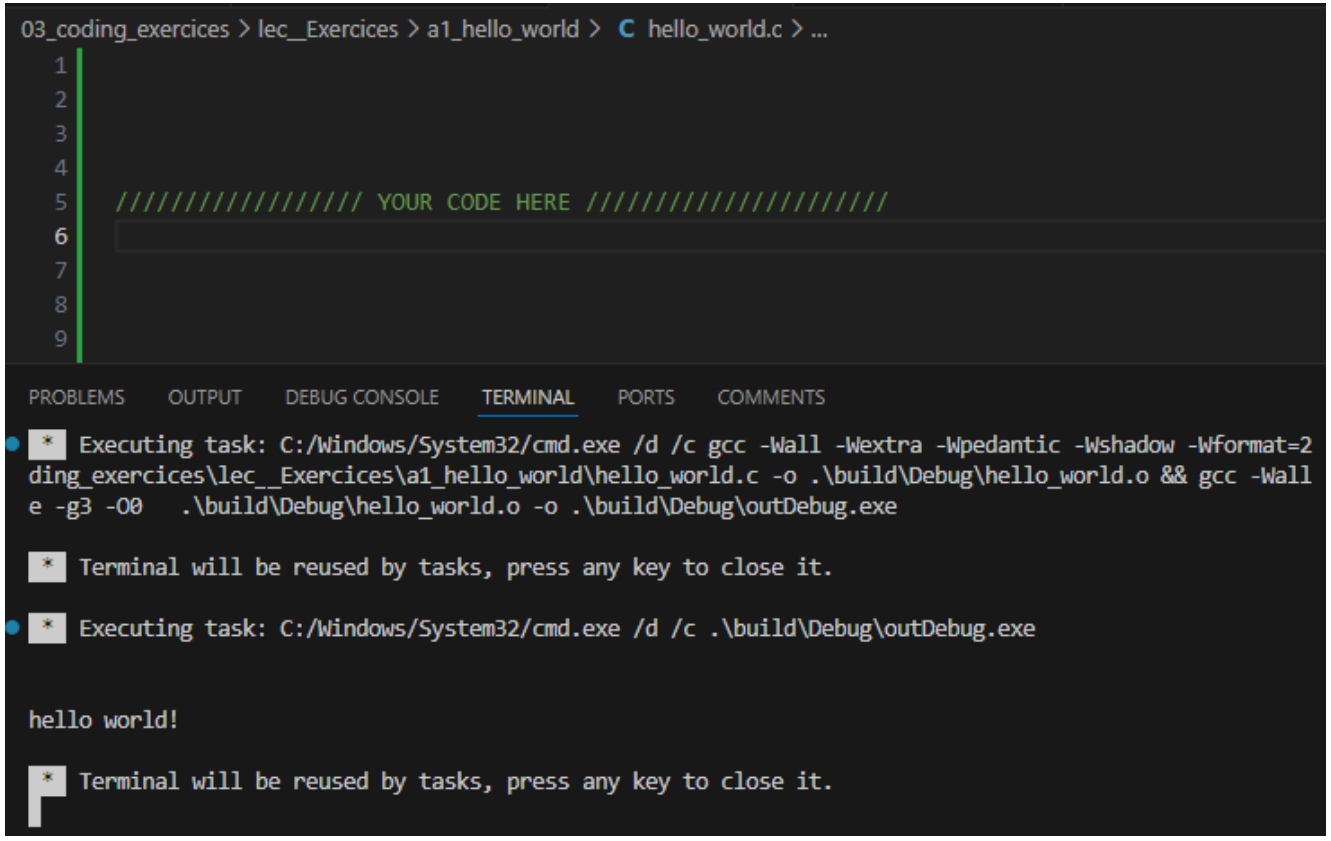
# Call by Value vs. Call by Reference
Passing arguments to functions

| Feature | Call by Value | Call by Reference |
|---|---|---|
| **Data transmission** | Copy of the value | Address of the value |
| **Influence** | Does not affect the original | Influences the original directly |
| **Memory requirement** | Higher for large data structures | More efficient with large data structures |
| **Side effects** | None | Possible |

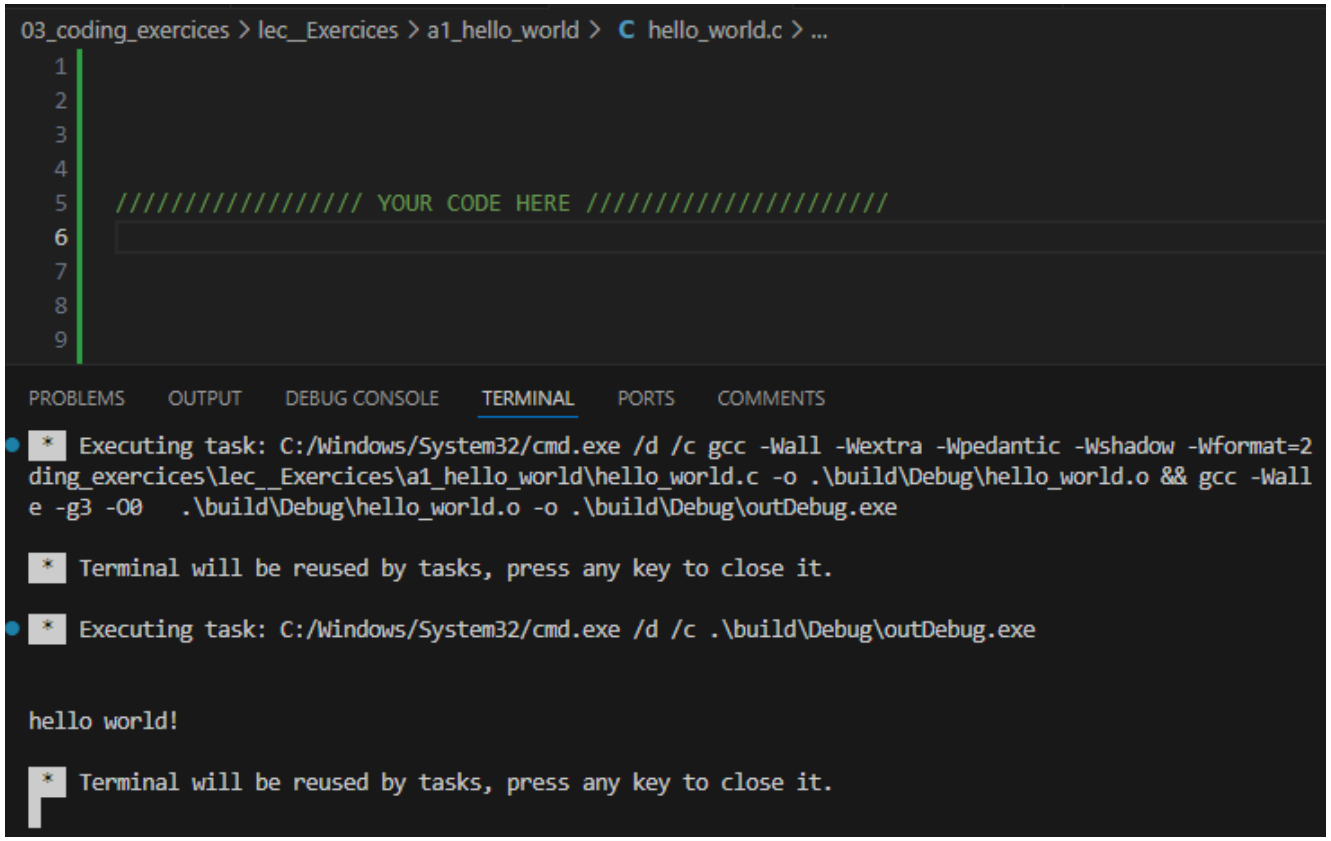
- Call by value and call by reference are fundamental concepts in programming
- influence the code efficiency and security / robust and efficient programmes

When do I use which method?

# Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2
ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall
e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# Let's code

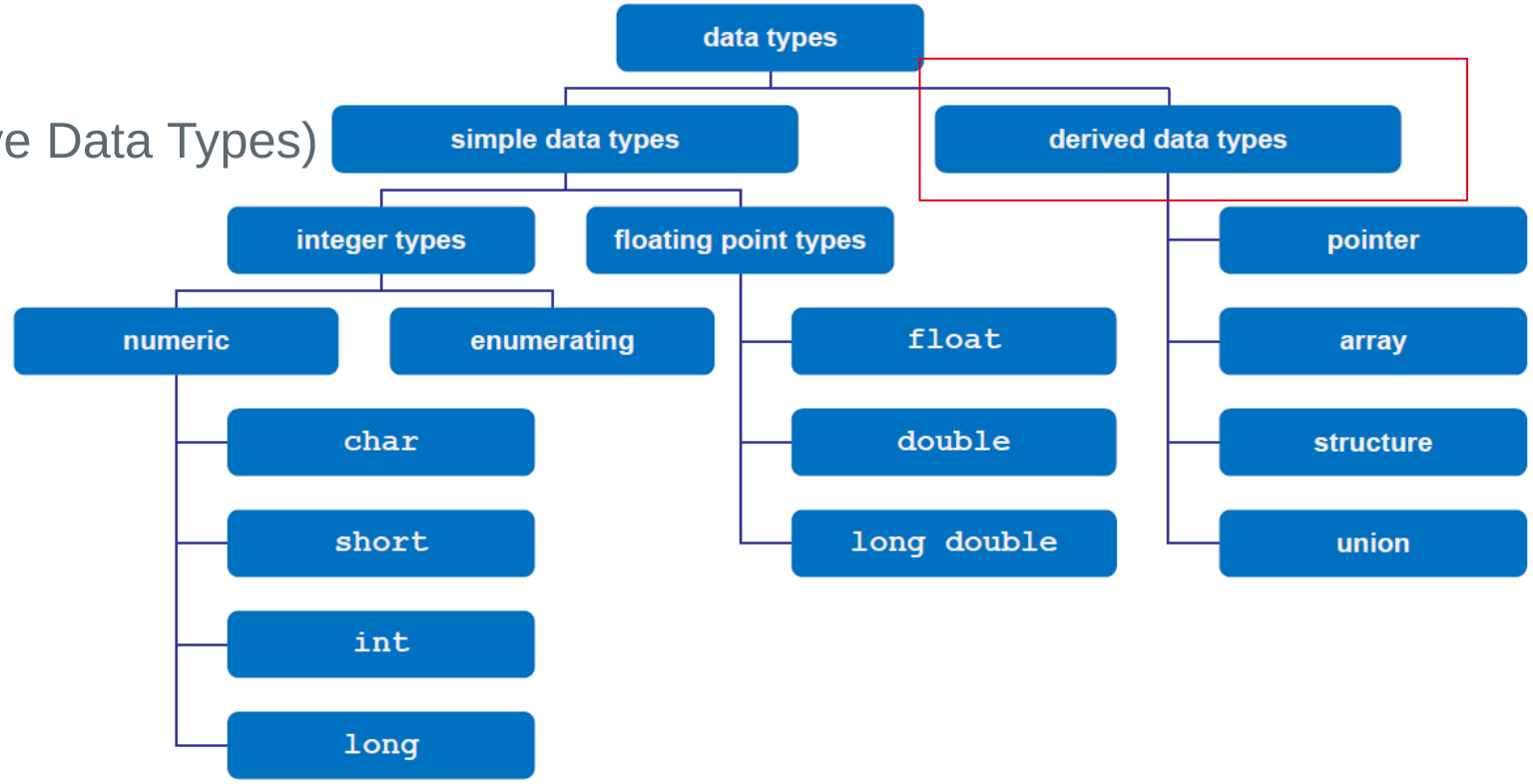

03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# Classes of Data Types

**What Data Type Classes do you know?**

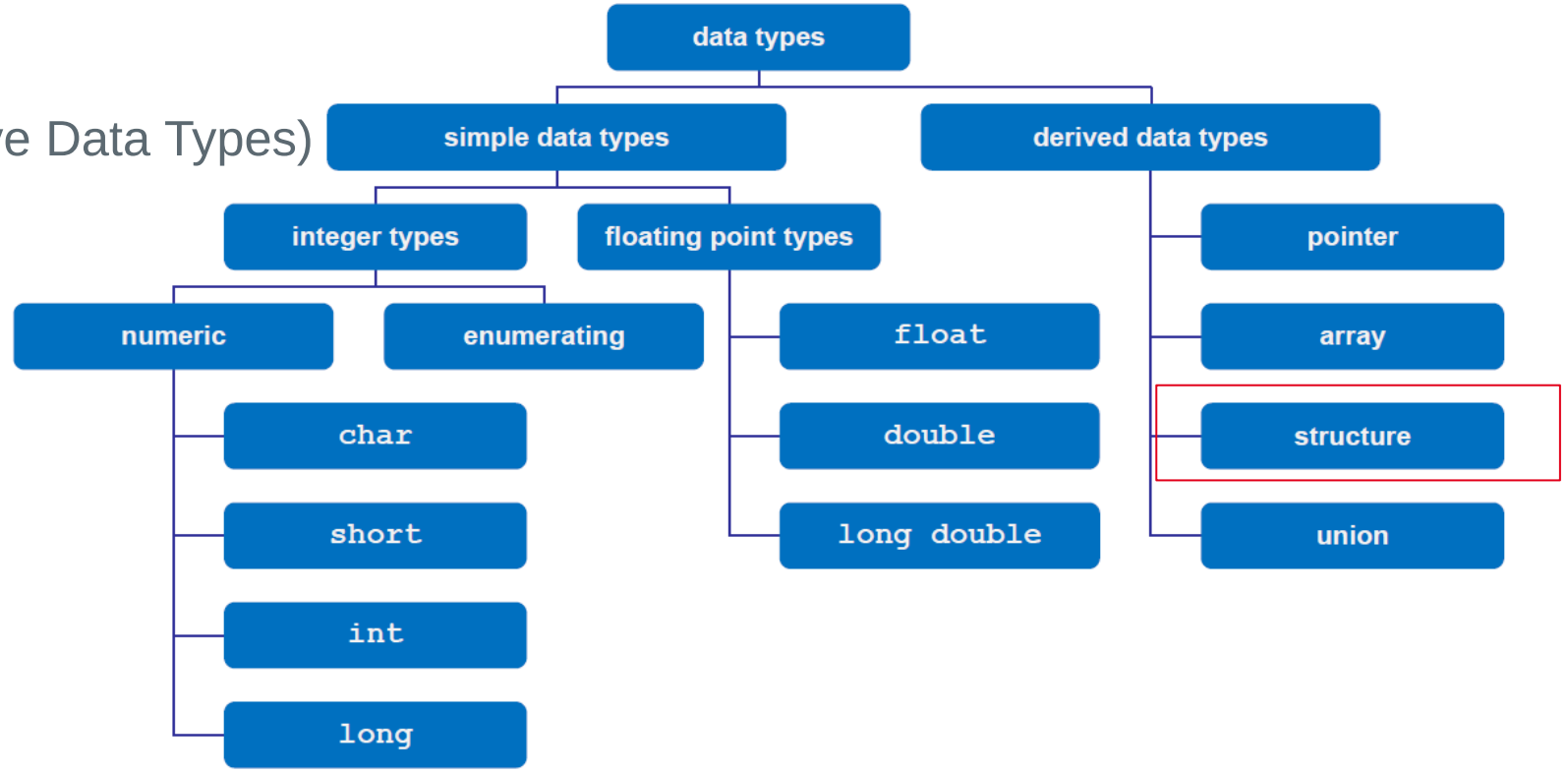
# Classes of Data Types
## Overview

(Primitive Data Types)



# Classes of Data Types
## Overview

(Primitive Data Types)



# Classes of Data Types
## Derived Data Types – Structure

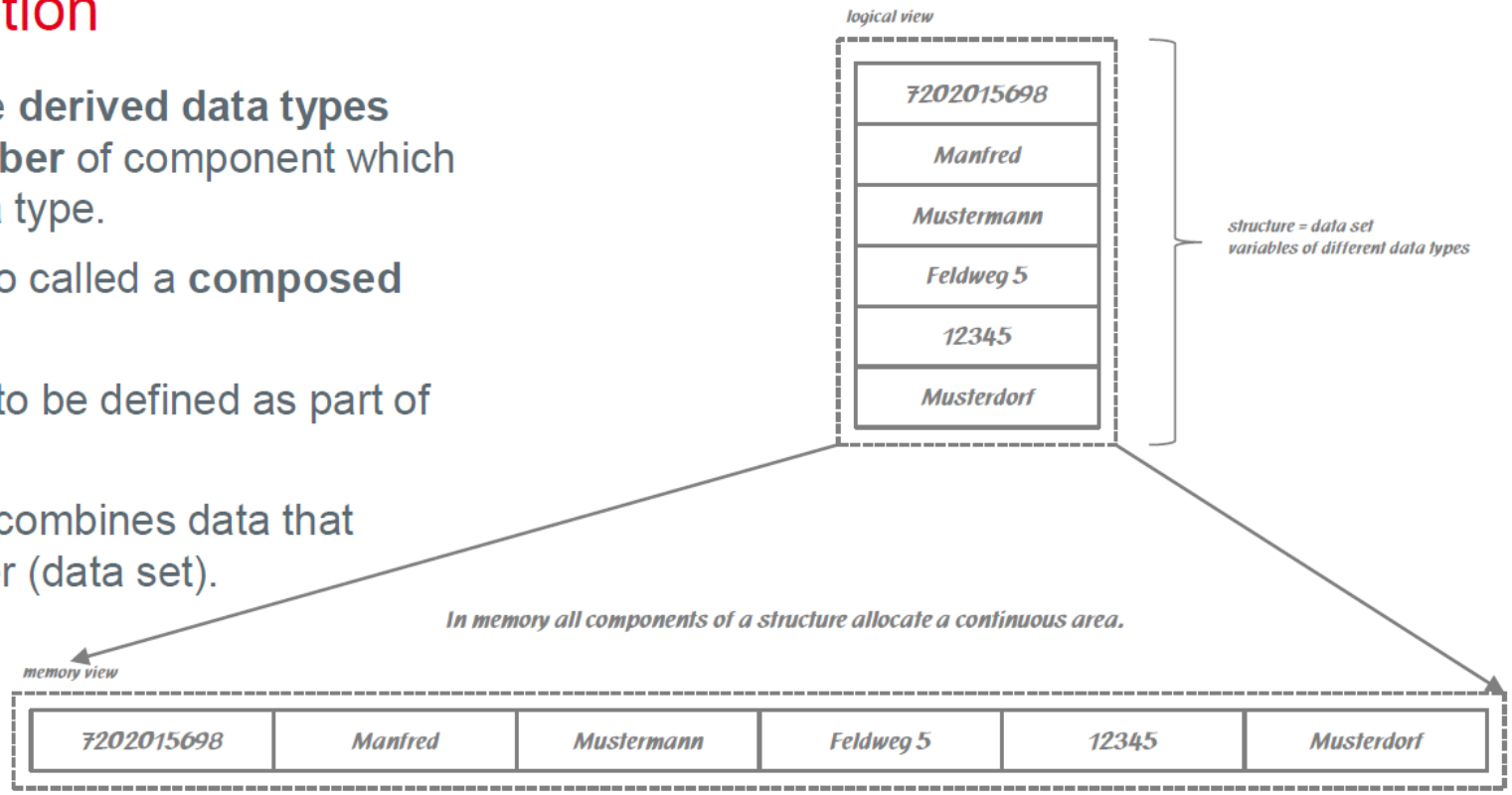
### Structures: Definition

**Structures** belong to the **derived data types** and host a **defined number** of component which can have a different data type.

The structure itself is also called a **composed data type**.

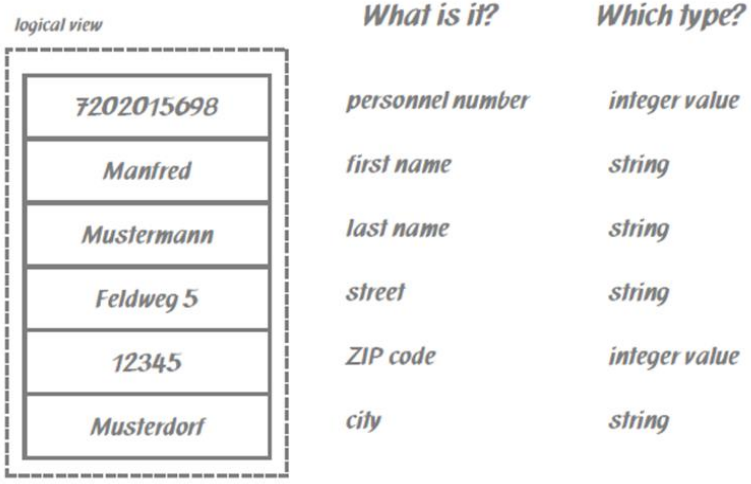
Each component needs to be defined as part of the structure.

In particular, a structure combines data that logically belongs together (data set).



# Classes of Data Types
## Derived Data Types – Structure

### Structures: Definition




The data type structure is defined with the keyword `struct` followed by the name and a block with all sub components.

General format:

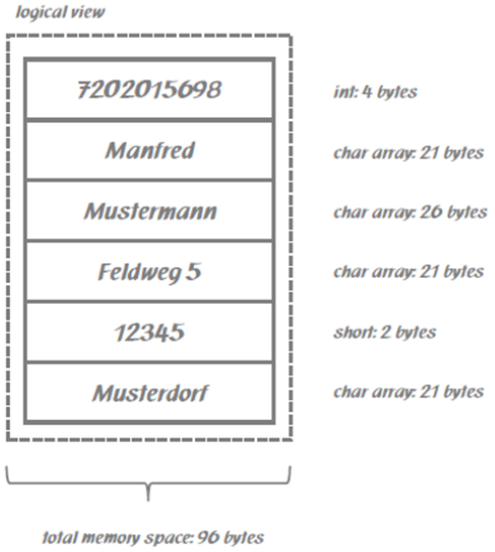
```
struct name {
   data_type_1 component_1;
   data_type_2 component_2;
   data_type_3 component_3;
   ...
   data_type_n component_n;
};
```

This is the data type **declaration**.

# Classes of Data Types
## Derived Data Types – Structure

## Structures: Definition


```c
#include <stdio.h>

// type declaration

struct persontype {
   int pn;
   char firstname [20+1];
   char lastname [25+1];
   char street [20+1];
   short zip;
   char city[20+1];
};

int main (void)
{

   struct persontype employee;

   return (0);
}
```




type declaration

variable definition

# Classes of Data Types
## Derived Data Types – Structure

### Structures: Access the Structure

To work with structures the following operations are available:

- assignment
- selection of a sub-component
- determine the allocated space
- Determine the address of the variable

The **assign operator** "=" is used for simple data types. To assign text to char-arrays (as component of a structure) use **`strcpy`**.

To select a sub-component of a structure the **dot-operator** " . " separated the name of the structure from the sub.-component.

```
struct persontype {
   int pn;
   char firstname [20+1];
   char lastname [25+1];
   char street [20+1];
   short zip;
   char city[20+1];
};

int main (void)
{
   struct persontype employee;

   strcpy (employee.firstname "Manfred");

   employee.zip = 12345;

   return (0);
}
```

# Classes of Data Types
## Derived Data Types – Structure

### Structures: Access the Structure

The `sizeof()` operator returns the number of allocated bytes and can be used for a single component or the whole structure.

```
sizeof (employee);
sizeof (struct persontype);
```

A pointer is a derived data type and inherited from its base type. A pointer to a structure gets the characteristics from the structure by using the reference operator "`&`".

```
struct persontype * ptr;
ptr = & employee;
```

```
int main (void)
{
   struct persontype employee;
   struct persontype * ptr;
   ptr = & employee;
   strcpy (ptr->firstname "Manfred");
   ptr->zip = 12345;
   return (0);
}
```

Access a structure by using pointers (not its name) works with the **arrow-operator**. The **dot-operator** is only used with the original name of the variable.

# Classes of Data Types
## Derived Data Types – Structure

### Structures: Type Definition

The **declaration** shows how the data type is designed. The name of the **data type** is (referring to the example):

`struct persontype`

C allows to give a **type definition** a different name. The keyword is

`typedef`

It is not mandatory to use typedef with structures.

```
#include <stdio.h>

// type declaration

struct persontype {
   int pn;
   ...
   char city[20+1];
};

#typedef struct persontype PERSON;


int main (void)
{
   struct persontype employee;

   PERSON employee_2;

   return (0);
}
```

*alternatively:*

```
typedef struct {
   int pn;
   ...
   char city[20+1];
} PERSON;
```

# Classes of Data Types
## Derived Data Types – Structure

### Structures: Special Structures

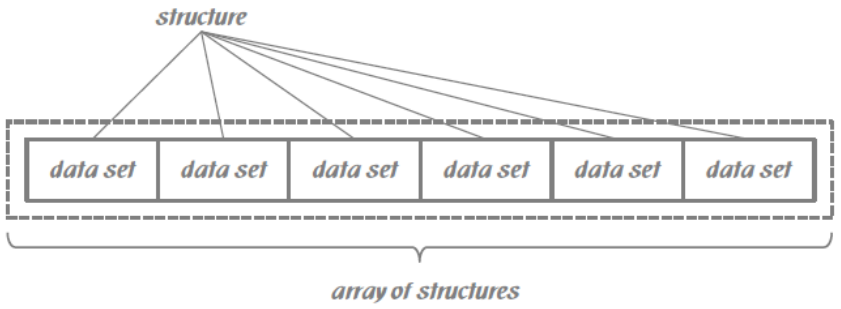
The system has some pre-defined structures. For example the time information is stored in a pre-defined structure:

```
struct tm {
   int tm_sec;    // sec
   int tm_min;    // min
   int tm_hour;   // hours
   int tm_mday;   // day
   int tm_mon;    // month
   int tm_year;   // year since 1900
   int tm_wday;   // days since Sunday
   int tm_yday;   // days since Jan 1st
   int tm_isdst;  // daylight saving
};
```

# Classes of Data Types
## Derived Data Types – Structure

### Structures: Array of Structures

A single structure can host one single data set. With an array of structure the base data type for a file card system is available.

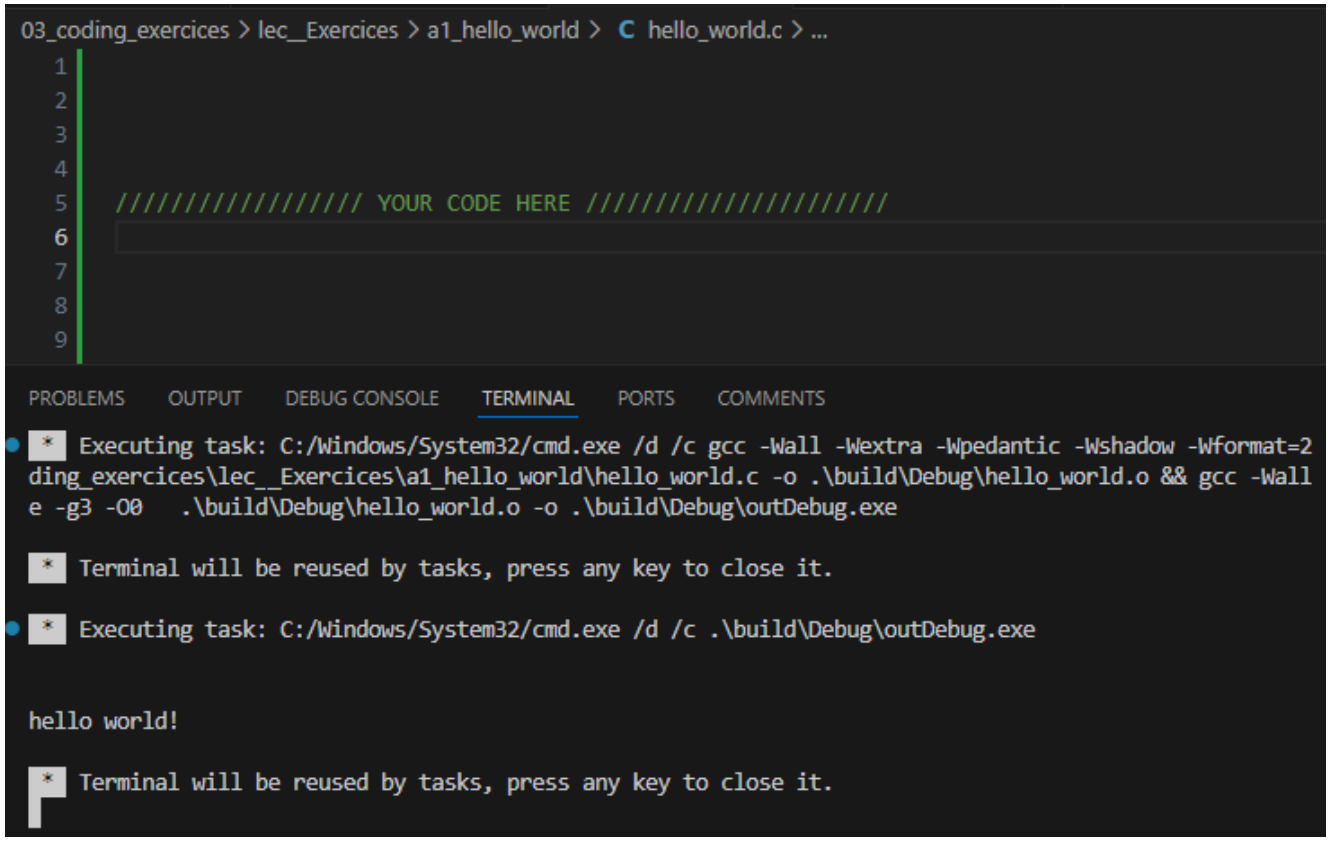



A structure and an array of structures is not initialized when defining the variable.

```
int main (void)
{
   struct persontype people[10];
   ...
   strcpy (people[3].firstname "Hugo");
   people[3].zip = 99999;
   ...
   return (0);
}
```

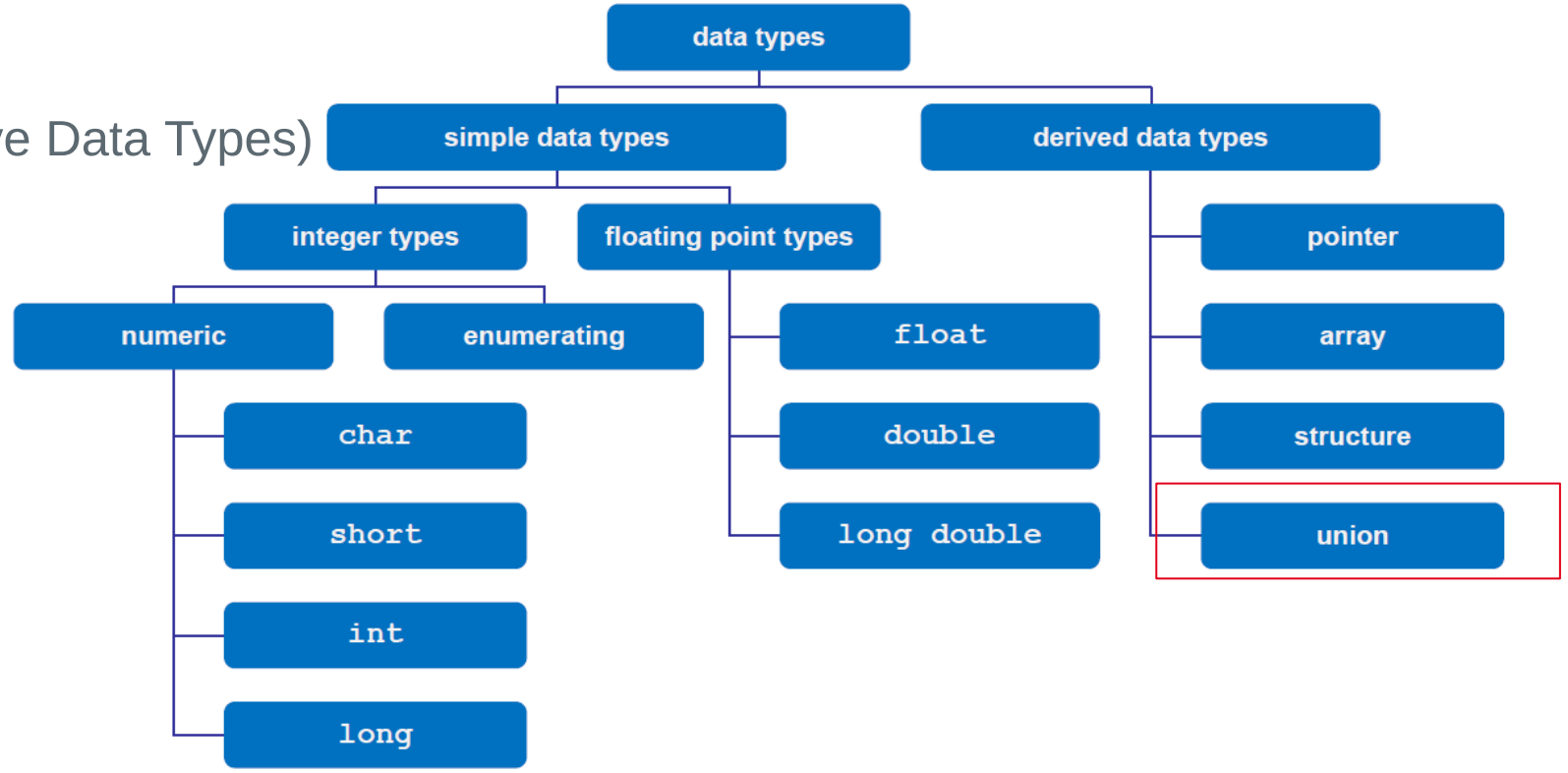
| | | | | | | |
|---|---|---|---|---|---|---|
| 7202015698 | 7202015698 | 7202015698 | 7202015698 | 7202015698 | | 7202015698 |
| Manfred | Manfred | Manfred | Hugo | Manfred | | Manfred |
| Mustermann | Mustermann | Mustermann | Mustermann | Mustermann | | Mustermann |
| Feldweg 5 | Feldweg 5 | Feldweg 5 | Feldweg 5 | Feldweg 5 | | Feldweg 5 |
| 12345 | 12345 | 12345 | 99999 | 12345 | | 12345 |
| Musterdorf | Musterdorf | Musterdorf | Musterdorf | Musterdorf | | Musterdorf |
| [0] | [1] | [2] | [3] | [4] | ... | [n] |

# Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

# Classes of Data Types
## Overview

(Primitive Data Types)

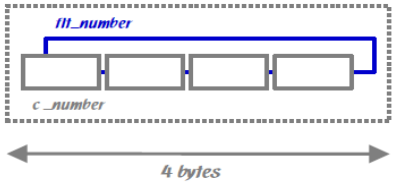


# Classes of Data Types
## Union

The derived data type **`union`** combines different data type, similar to s **`struct`**.

As the structure uses dedicated memory space for each component, the union starts string the single elements **at the same memory area**.

The programmer must follow up with type is currently store in the union.




```
union number_type
{
   float flt_number;
   unsigned char c_number[4];
};

union number_type num;
```

*A practical example is the conversion of a float number from decimal into binary format.*

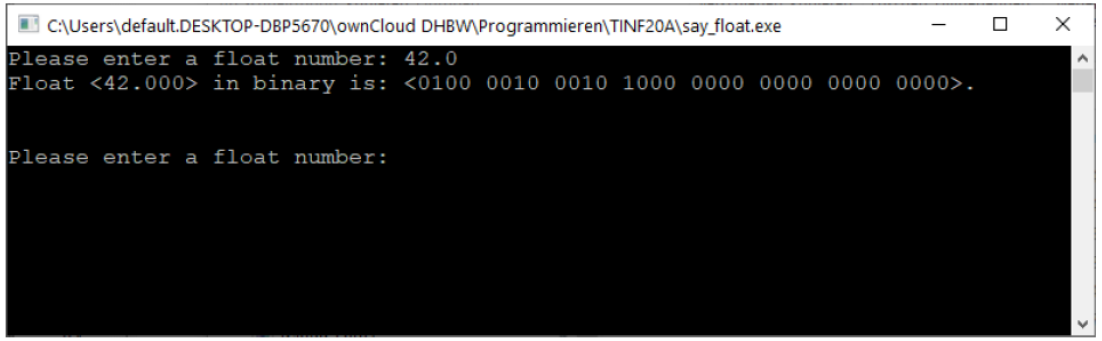
# Classes of Data Types
Union

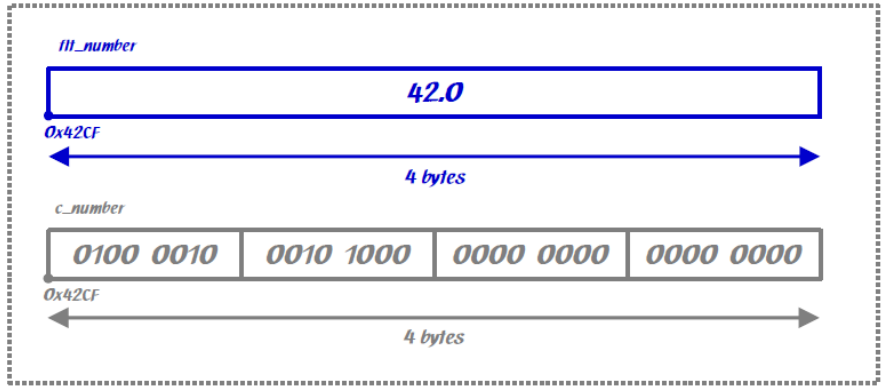
## Union: Exercise

The idea is to read a `float` number from keyboard, store it as `float` in a `union` an read the bits from the `union` as `char`.

The memory is **byte addressable** and a single bits cannot be read. The **operator &** (bitwise OR) can extract the information about a single bit using a mask.

Print each single bit on the screen. For better reading print a space every 4 bits.

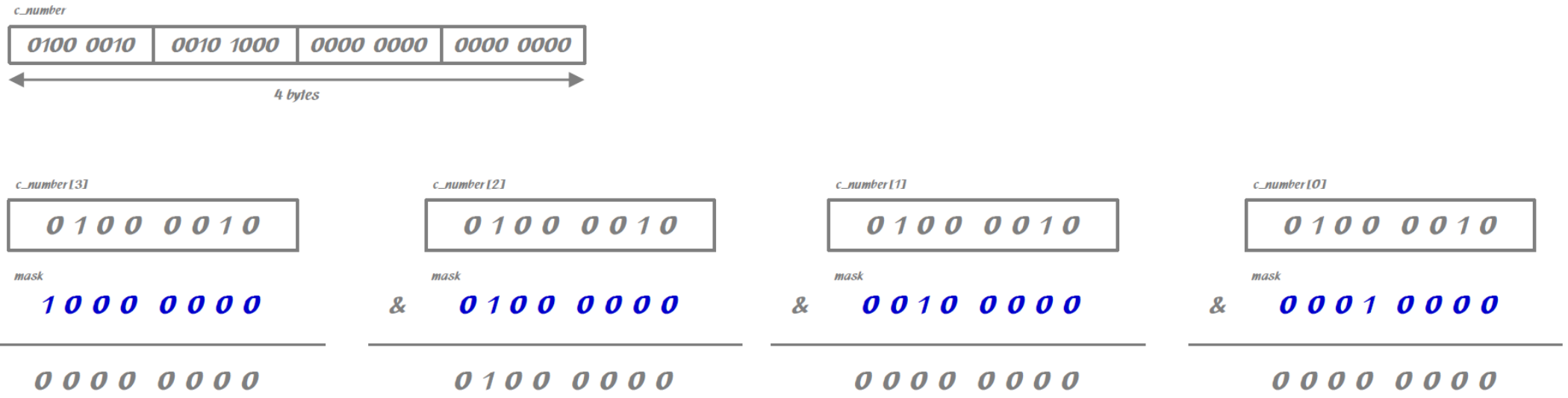


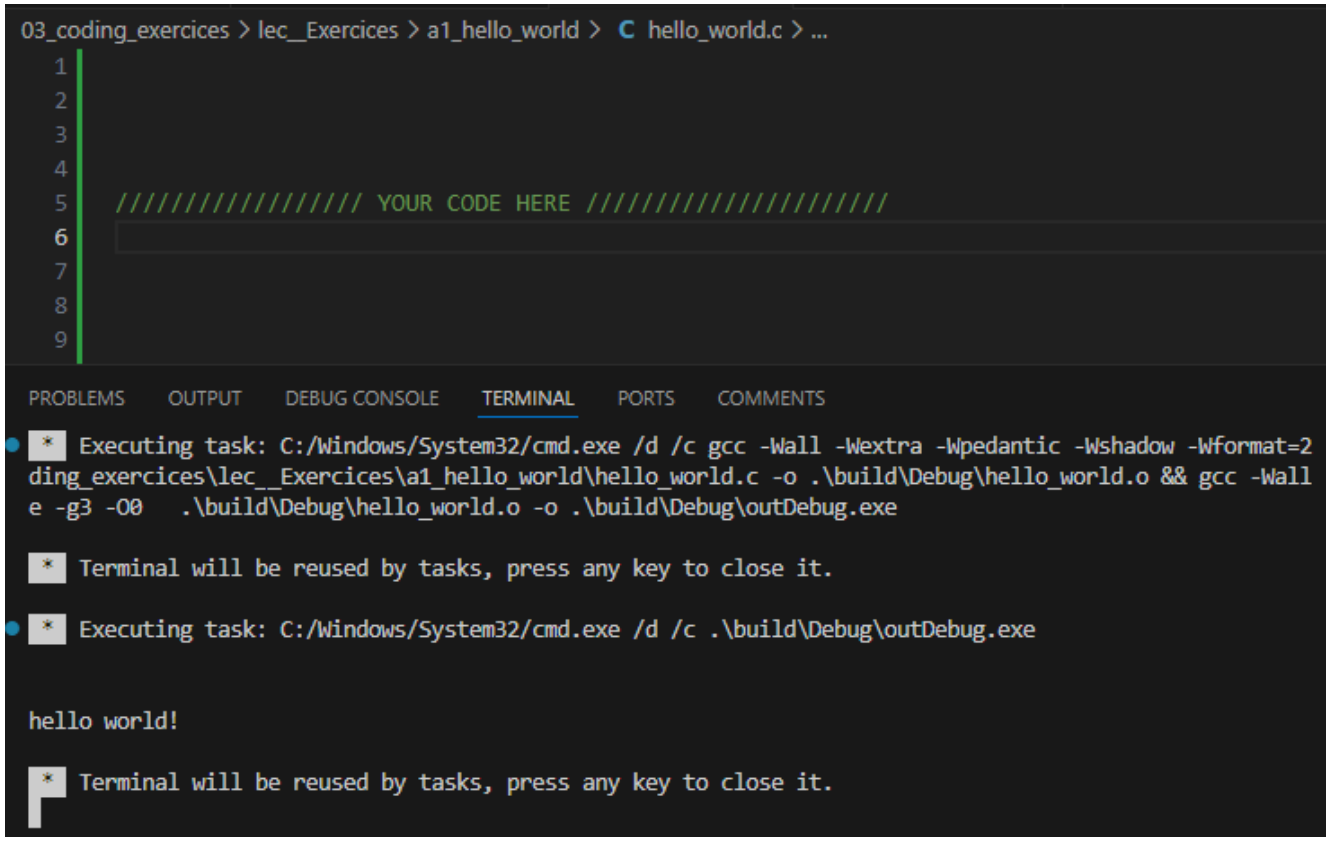
# Classes of Data Types
Union

## Union: Exercise

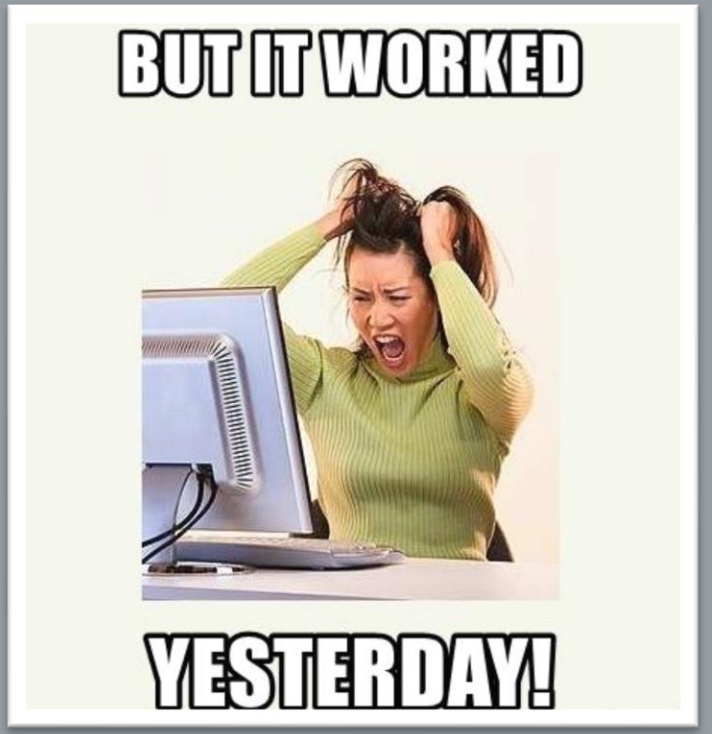
The **operator &** (bitwise and) can extract the information about a single bit using a mask.



# Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2
ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall
e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

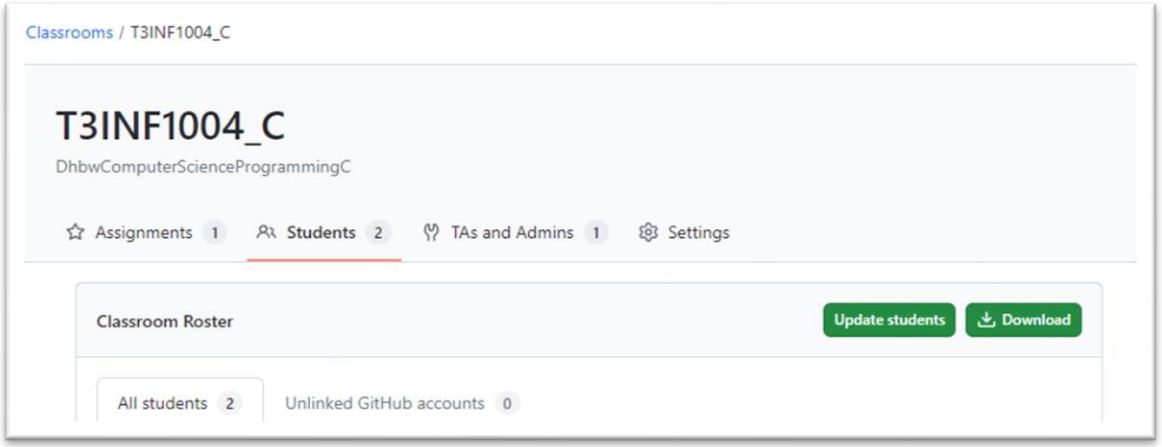
# BACK TO GIT AGAIN



Live demo

# Your Classroom for C coding Assignments

- Let's come together in the Classroom
- **01-Assignment** will be available for you
- Get the Repository
- **01-Assignment** Q&A




https://classroom.github.com/classrooms/182848101-t3inf1004_c/roster