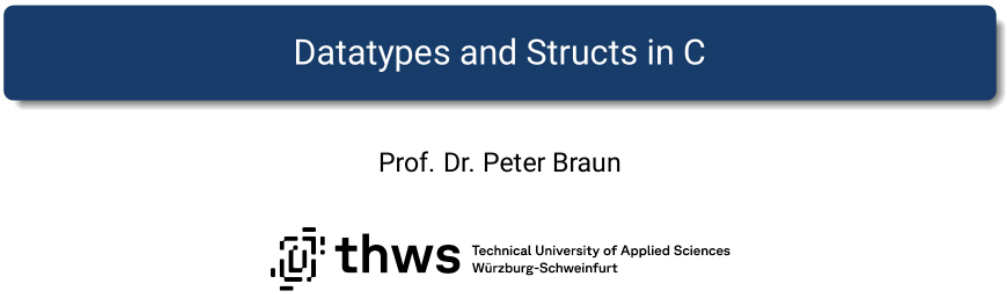
Datatypes and Structs in C

Prof. Dr. Peter Braun

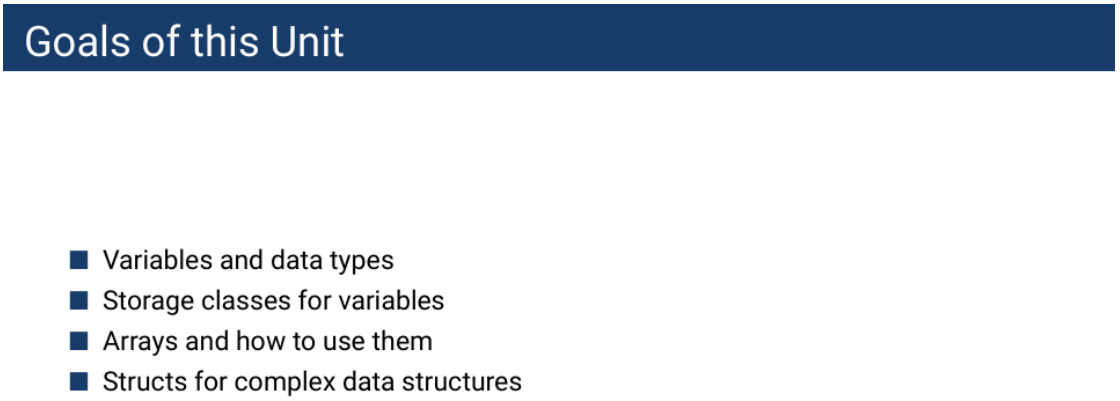
7. November 2024

1 Titlepage



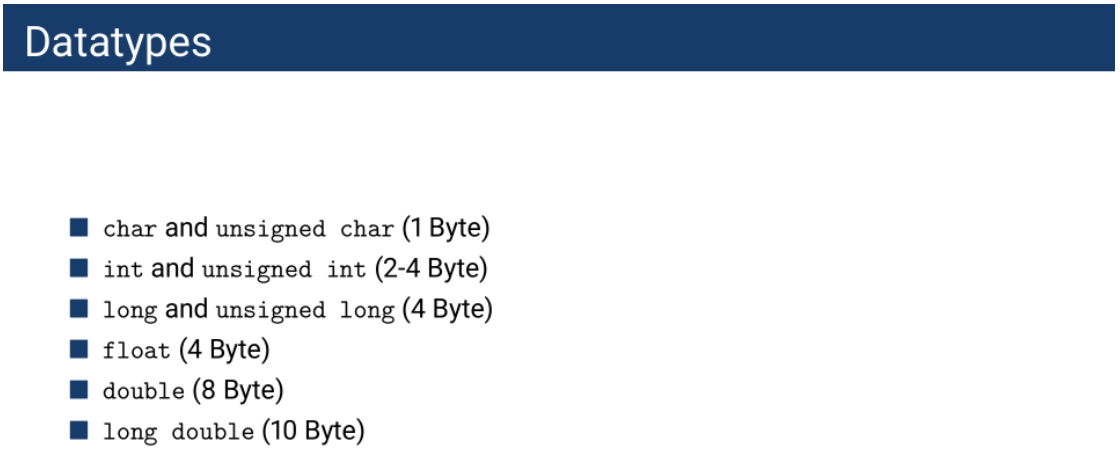
No Text – maybe some music :-)

2 Goals of this Unit



Welcome to this second unit of our introduction to programming with C. Today, we will be focussing on variables and data types. To begin with, we will summarise the data types that exist in the C programming language. We will then look at the various memory classes that a software developer can use to control data storage in memory. For example, knowledge of the different memory classes is necessary if you want extremely high performance. We then go on to look at arrays. We will show you how to create arrays and access the individual elements, and in which use cases working with arrays is important. Finally, we will show you how to define more complex data structures in C. A data structure is a collection of variables with different data types that should be accessible under one name.

3 Datatypes



C is a statically typed programming language. This means that variables must be declared before their first use by specifying a data type, and the variable type does not change over the variable's lifetime. C provides the following data types; you can see the list next to me. The size in bytes is given in brackets, although this can vary slightly, especially with older computers or operating systems. The data type integer for whole numbers only takes up 2 bytes on some computers and four bytes on others. Three data types for floating point numbers correspond to the precision according to IEEE Standard 754. As you can see, no separate data type for character strings exists. Other programming languages have something like the string data type, for example. There is also no separate data type for boolean values. In C, an integer value is typically used for boolean values. The one stands for true, and the 0 for false.

4 Storage Classes

Storage Classes

- Storage classes define how to handle a variable
- The default storage class is `auto`
- `static` variables keep values between invocations
- `register` variables should be mapped to registers
- `volatile` variables can change due to interference

```
1  auto int number;
2  static int number;
3  register int number;
4  volatile int number;
```

The so-called memory classes can be defined using additional keywords when declaring variables. With this information on the memory classes, we can support the compiler in optimizing our program, among other things. For example, a compiler has to decide which values should be stored in the main memory and which values should be stored in a register directly in the processor. As you probably already know, accessing registers is much faster than accessing the main memory. Compilers have algorithms for deciding in which memory class a variable is stored. However, software developers can help the compiler by using these keywords. Auto is used as the default setting if no explicit memory class is specified. It is, therefore, entirely up to the compiler to make this decision about the memory class. The keyword static has two meanings in C. It can be used for both global and local variables. If this keyword is used for local variables within a function, this variable retains its value even after the end of a function call. We will look at this later in an exercise. If the static keyword is used for global variables, we use it to limit the visibility. We will look at this on the next slide. The keyword register tells the compiler that we would find it useful if this variable is permanently stored in a register directly in the processor. We select this memory class, for example, if we know this variable will be accessed frequently. The last keyword, volatile, is a hint from the software developers to the compiler that this variable should be excluded from optimizations if possible. It could be that other parts of the program change this variable without this being obvious in the source code.

5 Static vs. Global Variables

Static vs Global Variables

- Global variables are defined outside of any function
- Global variables are similar to class variables in Java
- Static global variables are only visible in this file

```
1   void foo();
2
3   int NUMBER = 10;
4   static int LIMITED = 20;
5
6   int main() {
7     printf("%d",NUMBER); // result: 10
8     NUMBER = 11; // NUMBER modified globally
9
10    foo();
11  }
12  void foo() {
13    printf("%d",NUMBER); // result: 11
14  }
```

Let's now look at the difference between global and global static variables. A global variable is always declared or defined outside of functions. This variable can be accessed in every function. This creates side effects between functions that make the source code of our program difficult to read and prone to errors. Unfortunately, the use of global variables is not atypical for C. You can see an example of the definition of a global variable in the source code next to me in line three. Within the main function, this variable number is output in line seven and overwritten with a new value in line eight. The second function is then called. The screen output in line 13 then shows the value eleven. This variable number would also be accessible in other program files. In the previous units, we have not yet dealt with the concept that a program in C can consist of several program files. Therefore, only briefly here. A C program is made up of many functions, as well as definitions of data structures. These can be distributed over several program files, which are then combined by the compiler to form an executable program. As we have defined the variable global here, other parts of the program could also access this variable. If you want to prevent this, i.e., restrict access to the global variable to the functions of this file, place the keyword static in front of the declaration or definition.

6 Exercise 1: Static Variables

Exercise 1: Static Variables Within Functions

- Write a function with a static and global int variable
- Initialise this variable with some number
- Increment this variable and print the value
- Call this function several times

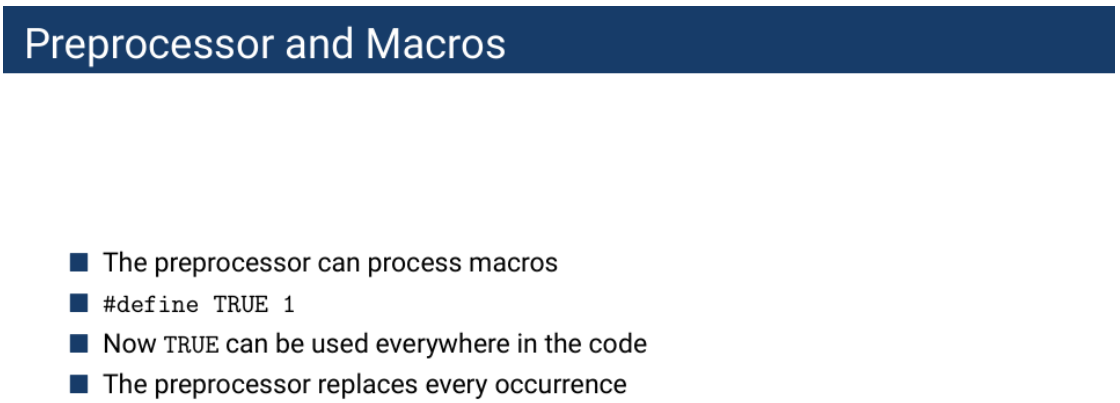
In today's first exercise, you must deal with static variables within functions. Please write a C program with a function where a local variable with the data type integer and the storage class static is declared. Initialize this variable with a value. Now, change the variable's value within the function; for example, you could increase the value by one. Display the value of the variable on the screen. Now, call this function several times in the main program. Please try to understand what is happening here. Please press the button when you have finished or want to see the solution.

7 Screencast

- todo

This exercise was about the static variables in C and local functions. So we write a first C program here. First we need to include standard I.O. because we want to print something on the screen. Then we have to write a function foo that accepts a parameter. We declare and define a static local variable with type integer. And the name is counter and we initialize this variable with zero. Then we add to this counter variable the given parameter and we print the value of counter to the screen. Next we need a main function in our program. And in this main function we first call foo with parameter 10, next with parameter 20. And let's see what happens when we execute this program. First we have to compile the program and then we execute it. And the values are 10 and 30. So let's have a look at the source code again. The initial value of counter is zero. We first call function foo with value 10. So we increase counter by 10 and the first output is clear. And then in the second run of this program we pass as parameter the value 20. And now we can see that counter keeps its value even if the scope of this variable counter is bound to the function foo. But the value is stored nevertheless. So when we call foo with parameter 20 this value is added to the current value of 10 and so the output equals 30.

8 Preprocessor and Macros



Program files for the C programming language can contain so-called preprocessor directives. You have already learned an example of this at the beginning of your study of the C language. You use the include directive to insert other source code files into your file before the compiler starts the translation. Here, you will familiarize yourself with the "define" directive. In the second line next to me, we define a macro with the name TRUE. We then write the number 1, which now has the following meaning. At all places in the source code where this word "true" is used, the preprocessor replaces this character string with the number 1. This is a mindless "search and replace" of character strings in the source code. When programming in C, it is good to avoid using numbers in the source code as much as possible and use such macros instead. You might now think you can define constants with these macros. And this impression is correct. Constants should be defined in C with such macros.

9 Better Use Constants Than Macros

Constants and Not Really Constant in C

- For unchangeable variables use the keyword `const`

```
1  const int number = 0;
2  void foo(const int value){
3    number = 11; //Error
4    value = 11; //Error
5  }
```

- But constants cannot be used when a constant expression is expected

But now, there is the keyword CONST, with which we can also define constant values. You can see an example in line one in the source text next to me. What is the difference? You use this keyword to define a variable you protect from being overwritten. Memory is still required for this variable, and when the variable is accessed, the memory must also be accessed in a complex manner. The compiler may be able to optimize this in some places. But, this is now a major disadvantage or misunderstanding of this keyword. Variables defined as constants with this keyword can still not be used in places where the C language requires a constant expression. This is the case, for example, when declaring arrays. We will see this later. The size of an array must be defined using a constant and must not be a variable. So please note that despite this keyword, no constant has been defined. In such cases, macros are better, as explained on the last slide. Let's take another look at this live on the computer.

10 Screencast

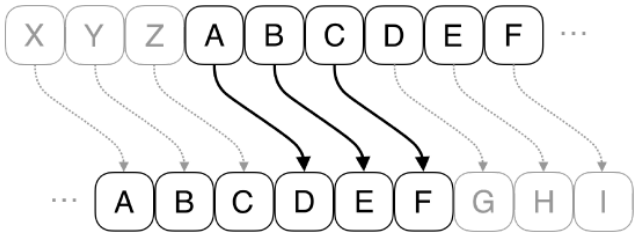
- todo

I would like to demonstrate this relation between constants and macros. We start a new C file and we first include standard I O as usual and then we define a constant variable and we make it a global one here. The data type is integer, the name is size and we initialize this variable with a value 20. Then we declare an array here, although you don't know what arrays are in C, but you know the principle of arrays and we use this size constant here to declare the size of the array. Finally, we need some main function here just to make the program do something. So hello world as usual we can now start the compiler and here you see the problem. The compiler complains about this data array and the size value is in some sense variable now. It's declared as a constant, but this more means that the value cannot be changed, but the compiler cannot see its actual value of 20. When we type in 20 here, then it is clear that this value is actually allowed. Finally, I want to show you how this can be done using macros. We define a macro named size in uppercase letters and then we can use this macro inside the variable declaration. The difference here is that although it is named constant, these constant variables are not allowed in places where the compiler actually needs a really constant value. For this you must type in the literal value 20 or a macro.

11 Caesar Code

Caesar Code

- Caesar Code can `encrypt` strings
- Each character is moved three places to the right
- Users can choose the input they want to encrypt
- The result is printed to console afterwards



Now, we come to the topic of arrays or fields. A field is a variable that can store several pieces of data, whereby the data must all have the same data type. The elements of a field are accessed via an index value. To motivate us to deal with fields, we use a very typical task, the so-called Caesar Code. The name supposedly refers to Julius Caesar, who developed this method to encrypt messages. Please do not take the term encryption too seriously here. By today's standards, it does not describe a real encryption process. The basic idea is that a fixed number of positions in the alphabet shifts one letter of the input. You can see the basic principle in the diagram next to me. If the letter A is in the input, it becomes a D in the output. Similarly, a B becomes an E, and a C becomes an F, and so on. Now that we have explained the basics of working with fields, you should write a corresponding function in the C programming language in the next exercise. But before that, we need to introduce more about arrays.

12 Arrays (1)

Introduction to Arrays

- Define an array: `char input[255];`
- Character arrays require `\0` at the end
- There is no `length` value to read the size of an array
- A loop counting until the null-terminal `\0` is required

The definition of an array or field in the C programming language is similar to Java. The size specification is in square brackets. However, in C, this is after the variable's name and not after the data type, as in Java. Character strings are defined in C as a data field of the type character. There is no separate String data type. Character strings in C are limited by a zero byte. This is also a difference to Java. There, we store the length specification explicitly in the object.

13 Arrays (2)

Accessing Array Elements

- There is no `IndexOutOfBoundsException` in C
- If an array is defined with: `int field[255];`
- Accessing `field[300]` is not an error
- However, it leads to undefined program behavior

The elements of a field are accessed as in Java with square brackets and an index position. You can access the first element with index zero. Unlike in Java, however, access to the elements of a field is not checked. For example, if you have defined a field with 255 elements, as in the example next to me, the compiler still allows access with an index value of 300. You can easily imagine that the behavior of such a program is no longer controllable. You are accessing some position in the memory where you can no longer know the data. They might overwrite other important data. Incidentally, variables of the array type cannot be assigned. Instead of an assignment, you must copy arrays yourself.

14 scanf and Arrays

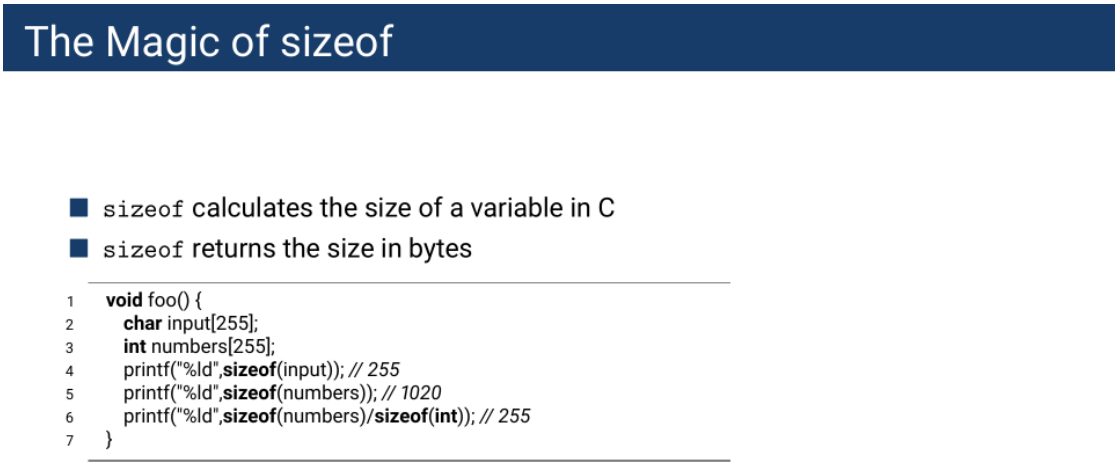
scanf and Arrays

- `scanf` reads whole words but not lines
- It stops after encountering a space character
- Therefore, use `fgets` to read whole lines from `stdin`

```
1  int foo() {
2    char input[255];
3    fgets(input, 255, stdin); // no &, stdin is from stdio.h
4    printf("%s",input);
5    return 0;
6  }
```

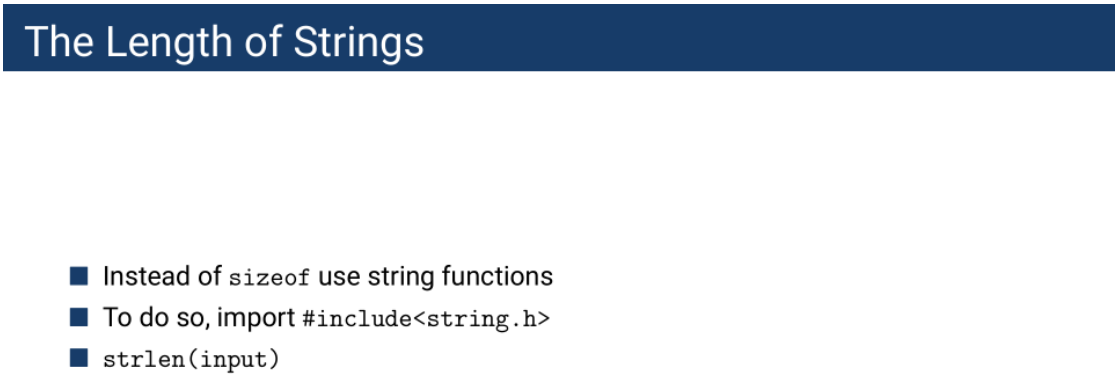
Data fields of the character type are suitable for receiving input from the user. You can see an example of this next to me in line two. Let's create a field with 255 characters. In the next line, we wait for an input from the user and save the input in the variable we defined previously. In line four, we display the input on the screen again. The F GET S function reads in characters from the user until the user presses the Return key or the maximum number specified as a parameter has been reached. Incidentally, this differs from the Scan F function, which you may remember from the previous unit. Scan F only reads in one word.

15 The Magic of sizeof



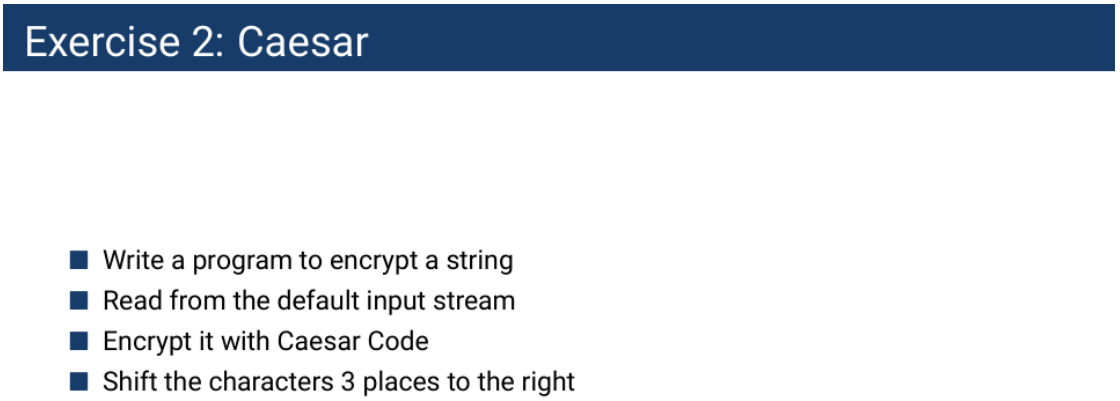
Once a data field has been declared, it is no longer so easy to determine the total number of elements. In C, there is the "sizeof" function, which you can use to determine the number of bytes a variable occupies in memory. Let's take a look at the example next to me. In line two, we create a data field of type character with 255 elements. In line three, we create a data field of the same size but with an integer data type. In line four, we determine the number of bytes, i.e., the size of the input field in the memory. As one character occupies exactly one byte, the memory size corresponds to the number of elements. In line five, however, the result is strange. The data field has 255 elements, but the "sizeof" function returns a result of 1020. This result becomes clear when you realize that 4 bytes are required for an Integer value. 255 elements multiplied by 4 bytes per element equals exactly 1020. So, to determine the number of elements in a field, you must calculate this as shown in line six. First, determine the size of the occupied memory and divide it by the size of the data type.

16 The Length of Strings



But even this only helps if you want to determine the length of a character string. A character string may not occupy all the elements of the data field. So, we need another function called STRLEN, which stands for String Length. This function counts the number of characters from the first to the terminating null symbol. But beware, this function may never terminate if no null character terminates the string due to an error.

17 Exercise 3: Caesar



You should now have all the necessary prerequisites to write a function that encrypts a string using the Caesar method. If you expect an input from the user, save the input in a data field. You can restrict yourself to entering lowercase letters and ignore other characters. Then call up your function. This function can move the characters by three places directly in the array. Make sure you move the characters correctly at the end of the alphabet. At the end, your program should display the result on the screen. When you have finished the task and want to see the solution, please click the button.

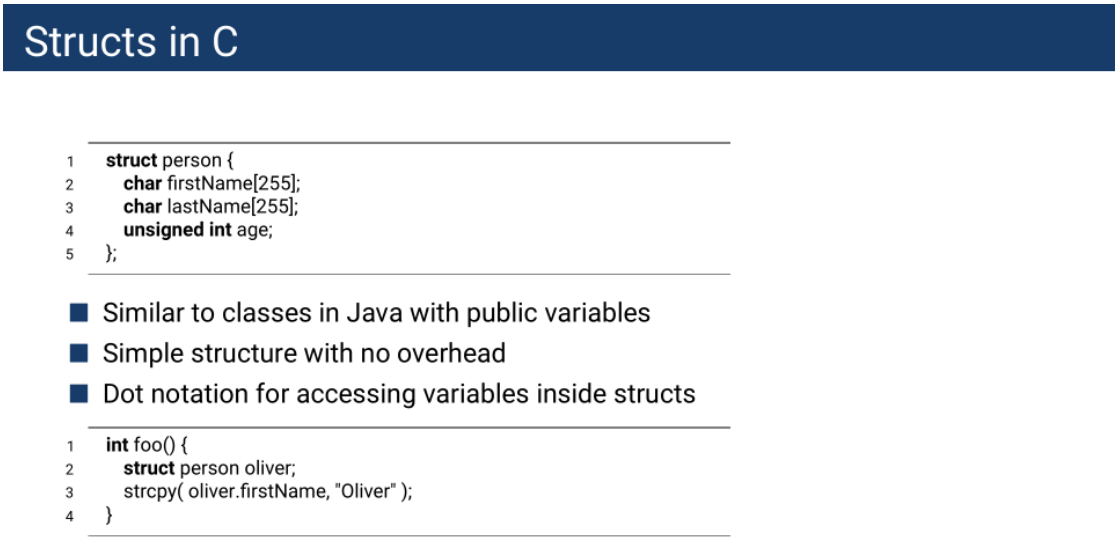
18 Screencast

- todo

So let's have a look at the solution. We open a new file here and we need two headers. One for the printf statement in standard IOU and we need some string manipulation things here. We implement the algorithm just in the main function. First we have to reserve some memory for the input array. So we expect at maximum 255 characters. Then we print some requests to the user and we read the input from standard in. Next we loop through all elements of the array. And here we use this function string length. In variable C we store the character that is stored in the array input at position I. And now we implement this main idea of this Caesar code algorithm. So we first limit ourselves just to characters and we don't take care about any digits or special symbols. So we first ask for the uppercase characters. So we have an if statement here. And we check if the character C is in this interval in this range of uppercase letters. We could have used also the ASCII values instead of the symbols capital A and capital Z. But C allows us to use the symbols here as well. And then now we have to implement this shift operation. So first we subtract from C the value of A, which is 65, according to the ASCII table, to map this in the range of 0 to 25. Then we shift the value by three positions. Now we have to take care of any overflows. So we use the model operator. And we have 26 symbols in our alphabet. And then we move the character back to the range of uppercase characters, so plus 65. And if this is not true, we check for lowercase letters. The lowercase A starts at 97. And we move the character by three positions, modulo 26, and back to the range of lowercase letters. And all the other characters we don't care. And now we write back this modified character into the character array. And we do this for all the characters in the array. Finally, we print the result. And now we check what happens with this program. So we compile it. And we just use a very simple word here, hello. And now you could use the ASCII table, for example, on the Wikipedia page to double check that the capital

H plus three positions is then the capital K and so on. We could also check some more simple cases. So ABC. And when we shift this by three positions, it's DEF. And finally, we should also check what happens when we have this overflow situation. So we have characters at the end of the character range. So let's start with X or W, X, Y, Z. And this becomes Z, A, B, C.

19 Structs in C



Let's now move on to the last topic in today's teaching unit. With the so-called structures, we summarise several data of different data types under one name. These structures are slightly similar to classes and public attributes within classes in object-oriented languages. One major difference is that structures do not contain any functions. Nowadays, however, this is imitated by pointers to functions, which are then part of the structure. We will talk about this later. However, C is not an object-orientated programming language, so there is no inheritance. In the example at the top, you can see the definition of a structure with the name Person. It contains three elements. The "first name" and the "last name" are data fields with 255 characters each. In addition, an unsigned number with the name age is also part of the structure. You can see how this data structure is used in line two of the example below. Here, we create a variable named Oliver. At this moment, memory is reserved for the entire structure, i.e., 514 bytes in this example. Two data fields with 255 bytes each and a number with four bytes. The data structure elements are then accessed using the so-called dot notation. You can see an example in line three. Here, the name Oliver is copied into the element "first name" of the structure Oliver. A dot separates the variable's name and the element's name. Let's take another look at this live on the computer.

20 Screencast

- todo

I would like to show you this live here on my computer. We start with a new C file. We first include standard I.O. as usual. Then we define a data structure. And we name this data structure person and it contains the first name of 255 bytes and the last name of 255 bytes and an integer value named H. In the main function we declare a variable named Oliver, which is a data structure of type person. And we copy this value of Oliver into this attribute here. And to check if it works we print the value of this first name element. This string copy function is defined in this string.h file so we have to include this. So let's compile this. And when we start you can see the output is correct. We add another new line character after the print of the name here. And now I would like to talk a little bit more about this size of data structures. So we output the size of this Oliver data structure. We use the sizeof operator. And we save this and start the compiler. And the output is 516. So on the slide I mentioned that we have to add the size of the attributes. So 255 plus 255 plus 4 bytes for the H makes 514. And the question is now what about this difference? 16 was the output, 514 is the sum of the size. And the reason for this difference is named padding. Padding means that the compiler puts the boundaries of the attributes to memory positions that are dividable by 4. So instead of 255 bytes, actually 256 bytes are used. So we have one byte more and this is true for the first name and the last name. And therefore we have 256 times 2 plus the 4 bytes for the H and this makes 516.

21 Structs

Assigning Structs in C

- Assigning new data to a struct copies the data

```
1  #include<string.h>
2  ...
3  int main() {
4     struct person oliver, steffen;
5     strcpy( oliver.firstName, "Oliver" );
6     steffen = oliver;
7     strcpy( oliver.firstName, "Oliver Hans");
8     printf("%s",steffen.firstName); // Output: Oliver
9  }
```

We had just mentioned that data fields cannot be assigned to each other. It may, therefore, come as a surprise that this is possible with structures. To demonstrate this, please look at the source code next to me. In line four, we have two variables named Oliver and Steffen as data structures of type Person. We set the first name in the variable Oliver to the text Oliver. Now, we assign Oliver's data structure to the variable Steffen and overwrite Oliver's first name in line seven. However, the first name of the data structure, Steffen, remains at the value of Oliver. The assignment has, therefore, worked, and all data is automatically copied from one data structure to the other.

22 Make Things Beautiful

Make Things Beautiful

- We can define custom datatypes in C using `typedef`

```
typedef unsigned long ulong;
ulong l2 = 1;
```

- This also applies to structs

```
typedef struct person {
  char firstName[255];
  char lastName[255];
  unsigned int age;
} Person;
Person oliver;
```

In the C programming language, creating alias names for data types is possible. The keyword typedef is used for this. You can see in the example above that we define the data type unsigned long with the alias name U-Long. From this instruction onwards, we can use this alias name whenever a data type is expected. This option of assigning alias names is particularly used when dealing with data structures. The definition of a data structure is often combined with the assignment of an alias name. We can see an example of this below. For the data structure with the name Person, here, it is now written in lowercase, and the alias name Person is now set to Written. This makes the source code easier to read because we can dispense with the struct keyword.

23 Exercise 4: Structs

Exercise 3: Structs and Functions

- Write a function that accepts a `struct` as parameter
- Modify the values in the struct
- Are these modifications visible for the caller?
- Is it possible for a function to return a struct?
- What do you learn from these two examples?

In the last exercise of today's unit, please experiment with the structures you have just learned in connection with functions. Please write a function that receives a data structure as a parameter. The caller has already written data to this data structure. The function should now change this data. Are these changes visible to the caller? Then, define a function that returns a data structure as a return value. What do you learn from these two examples? How are data structures passed to and returned by functions?

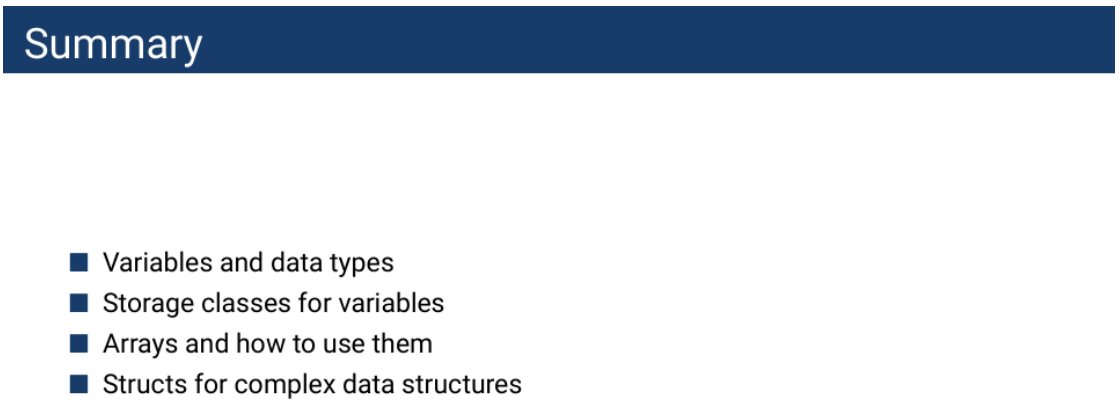
24 Screencast

- todo

So in this exercise we want to study handling of data structures and functions. We start by including standard I O as usual and string.h for some string functions. We define the same data structure as before. The name is person. We have a first name and last name. Both are 255 bytes long and we have this age. Then we define a function foo. We start with a void return type. The function accepts a parameter of type struct person. And we want to demonstrate how data structures are passed as parameters. We modify the data structure and we copy into this first name element of the structure a value Oliver. That's all. Next we implement the main function. We have a local variable here and we name this Oliver. We first store some name into this data structure. So the first name of this variable Oliver is now James. Then we call our function foo. Which should change now this value of the first name. Finally we check this by printing the first name to the screen. We compile this. When we start this you can see the result is James. Calling this function foo did not have any effect. What happened here? When we pass this data structure as parameter to foo the whole data structure is copied. We learned from the last demo that this data structure has 516 bytes. This is a huge copy procedure here. A lot of effort to copy this parameter. Then at the end this change to Oliver did not have any effect. Why is this? Because we don't have a return value here. And this data structure since it is copied all modifications are just visible inside of this function foo. The result or the solution for this is now to return the modified data structure. A simple way is now to change from void and we return the structure person. We have to modify the main function here a little bit. After we call foo we store the modified value into Oliver. When we compile this and start the program you can see now it works. It only works because of two very expensive copy functions. We have to first copy the parameter into this function. Then because it is returned it is also very expensive to copy this modified

data back to the main function. But actually it works at the end.

25 Summary



In today's unit, we dealt intensively with variables and data types. We talked about the different memory classes and when it is helpful to give the compiler support in optimizing the program. In addition to the primitive data types for characters and numbers, the C programming language also uses data fields. Data fields always summarise several variables of the same kind and allow access via an index. Data structures, on the other hand, summarise different data types. In the last exercise, you saw that the principle of call by value is always applied when calling a function or returning data in this programming language. This means that data is copied, which can be very time-consuming with large arrays or large data structures. In the next unit, we will look at the very important concept of pointers. This allows us to pass data to functions by reference, for example. But we'll look at that next time.

26 Literature

Literature

- Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language. Prentice Hall, 1988.
- Avelino J. Gonzalez: Computer Programming in C for Beginners. Springer, 2020.
- Helmut Schellong: Moderne C-Programmierung. 2. Auflage, Springer Vieweg, 2013.
- Jürgen Wolf: C von A bis Z. 4. aktualisierte und überarbeitete Auflage, Rheinwerk Computing, 2020. (Die 3. Auflage ist als Open Book verfügbar.)