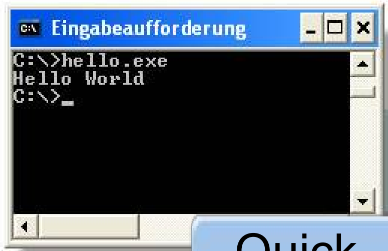
Einführung in C - Introduction to C

5. Functions and program structure

Prof. Dr. Eckhard Kruse

DHBW Mannheim

How to become a great C programmer?



```
C:\>hello.exe
Hello World
C:\>_
```

Quick overview

data types and operators

functions and program structure

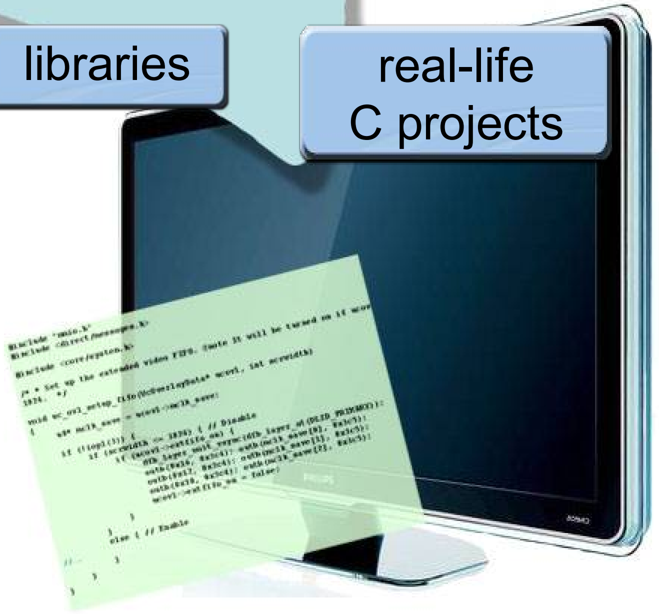
pointers and memory management

control flow

Start
Berechne Gewicht und Wert der Kombination
Initialisiere Beste Lösung = Leere Füllung
Ausgabe Beste Lösung
Merke aktuelle als beste Kombination
Ende

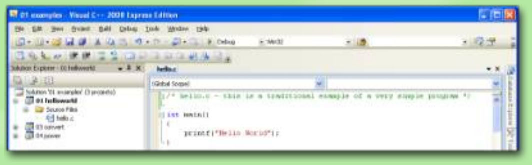
libraries

real-life C projects



Examples

Exercises



Functions

Definition

A **function** is a section of program code that performs a particular task. By defining functions, code can be structured by its purpose and made reusable, i.e. it can be called from different contexts with different input. Functions can accept **parameters** and **return** a result.

```
type function_name(type parameter1_name, type parameter2_name, ...)
{
  statements
  …
  return value;
}
```

```
int mean_value(int a, int b)
{
  int c;

  c = (a + b)/2;
  return c;
}
```

old C style

```
int mean_value(a, b)
  int a, int b;
{
  ...
```

Many useful functions are already provided in the **standard libraries**: `printf()`, `scanf()`, `strlen()`, `fopen()`, `rand()`, `getchar()` ...

Function parameters

no or one
out parameter

no, one, or many
in parameter(s)

```
float distance(float x1, float y1, float x2, float y2)
{
  float di;
  x1-=x2;
  y1-=y2;

  di=sqrt(x1*x1+y1*y1);
  return di;
}

main()
{
     float x1=2.0, y1=3.0, x2=7.0, y2=5.0;
     printf("%f", distance(x1, y1, x2, y2));
     // x1==2.0, y1==3.0 ....
}
```

return:
Exit the function immediately and return the provided value.

in-parameters are not modified by the function

Function prototype

A **function prototype** declares the function name, its parameters, and its return type to the rest of the program prior to the function's actual definition.

```
#include <stdio.h>

void main()
{
    printf("%d\n",add(3));
}

int add(int i, int j)
{
    return i+j;
}
```

This code can be compiled with just a warning, then the program gives undefined results.

```
#include <stdio.h>

int add(int, int); // Prototype for add

void main()
{
    printf("%d\n",add(3));
}

int add(int i, int j)
{
    return i+j;
}
```

This is better, because the error can be detected at compile-time.

stdio.h

Beispiel

```
/*
 * Function prototypes
 */
…
FILE * fopen(const char *, const char *);
FILE * wfopen(const wchar_t *,const wchar_t *);
int fprintf(FILE *, const char *, ...);
int fputc(int, FILE *);
int _fputchar(int);
int fputs(const char *, FILE *);
size_t fread(void *, size_t, size_t, FILE *);
FILE * freopen(const char *, const char *, FILE *);
int fscanf(FILE *, const char *, ...);
int fsetpos(FILE *, const fpos_t *);
size_t fwrite(const void *, size_t, size_t, FILE *);
int getc(FILE *);
int getchar(void);
char * gets(char *);
int getw(FILE *);
int printf(const char *, ...);
int putc(int, FILE *);
int putchar(int);
int puts(const char *);
int remove(const char *);
int rename(const char *,const char *);
void rewind(FILE *);
int scanf(const char *, ...);
```

Header files provide function prototypes (and other definitions) of libraries which are linked to the program.

Game of life

Exercise

`game_of_life.c`

Do this as teamwork (e.g. with teams of 2 or 3). The different functions should be developed by different team members and then integrated into the final program.

Code snippet 501

Call by value

Definition

```
int calc(int a, int b, float c)
{
   b=b+1;      // this has no impact on 'value' below
   …
}
…
    ret=calc(5+7, value, other_calc());
```

Call by value means that parameters are passed by evaluating the expressions in the function call and copying the results into the local variables of the function. Even if the local variables are modified in the function, the values in the function call are not modified. (e.g. above: **`value`** is copied and can not be modified by **`calc`**).

In C, arrays/strings cannot be passed by value. (it would require that the whole data structure is copied, which could be very inefficient.)

How to pass strings/arrays as parameters?
How to get back more than one return value?

Call by reference

Definition

```
void swap(int *a, int *b)
{
   int helper;
   helper=*a;   *a=*b;   *b=helper;
}
…
    int var1=3, var2=5;
    swap(&var1, &var2);  // Exchange values of var1 and var2
```

See e.g.:

```
scanf("%d", &value);
```

Call by reference means passing references of the in-parameters to the function.
No local copies are created, the function directly modifies the referenced variables.

Call by reference is realized by passing pointers:

- The **address operator &** provides the reference to (= address of) the variable.
- The **dereference operator *** 'dereferences a pointer' i.e. it provides the value stored at the indicated address.

(more on this later → Pointers and memory management)

Refactoring and reuse

Exercise

`refactoring_reuse`

Refactoring means improving/restucturing a program to make it more readable, maintainable etc. - but without changing its functionality.

Have a look at previous programs from the lecture and extract portions of them into re-usable functions.

- 205-bitwise_operators.c:
  Write a function "void print_binary(int value)" which prints integer values in binary format.
- 204-ascii_art.c:
  Write a function "void ascii_art(int x, int y, int parameter)" which prints an ascii-art of size x * y, which look depends on 'parameter'.
- 401-switch_case.c

Strings and arrays as parameters

```
int addArray (int a[], int n)
{
   int i;
   int sum = 0;
   for(i=0; i<n; i++)
      sum = sum + a[i];
   return sum;
}

int main() {
   int y[10] = {1, 5, 13, 42, 0, 60, 71, 82, 93, 10};
   printf("Sum is %d\n", addArray(y, 10));
}
```

Array size may be omitted. At runtime, C does not know the array size, it needs to get it with n.

(Alternative: int a[10], 10 „hardcoded“ in loop)

Strings and arrays are passed by reference.
No address operator (&) is required in the call, as the name without brackets already acts as pointer. In the function, the brackets [ ] can be considered as special form of dereferencing operator.

String functions

Exercise

`string_functions`

Write a set of string functions (based on the previous programs working with strings):

- `int string_length( char txt[] )`
  Returns the length of the string txt.
- `int devowelize( char txt[] )`
  Removes the vowels within txt and returns the number of removed vowels.
- `void decharacterize( char txt[], char remove[], int cnt[])`
  Removes from txt the characters listed in remove (terminated by a 0-character) and returns the number of removals for each character in cnt[ ].
- Write a main-function with test code for each of the above functions.
- Discuss the risks / error potential regarding the maximum length of the strings.

Code snippet 503

Matrix functions

Exercise

`matrix_functions`

Write two functions (based on matrices.c):

- `void add( float a[3][3], float b[3][3], float sum[3][3])`
  Adds matrices a and b and returns the result in sum.
- `void multiply( float a[3][3], float b[3][3], float product[3][3])`
  Multiplies matrices a and b and returns the result in product.

Code snippet 504

main

Definition

The **main-function** is the entry point to the C program. Every C program must have exactly one main function.

- Optionally, main can receive parameters which are taken from the command line /invoking process.
- main() returns an integer value, which may be used by the invoking process, command shell etc.

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

int main(int argc, char *argv[])
{
    int n;
    for(n=0; n < argc; n++)
       printf("Argument %d is %s\n", (n+1), argv[n]);
    return EXIT_SUCCESS; // Standard return code in stdlib
                         // Alternative: EXIT_FAILURE
}
```

Passing args to main

Exercise

`main_args.c`

Code snippet 505

Variables: Local, global, static

Definition

Local variables are declared within the function header or body. They are only visible within the function. When the function is called, a new instance of the variable is created, when the function finishes, the variable disappears. This is done automatically, thus these variables are also called **auto / automatic.**

Global variables are declared outside the functions. They can be accessed from any part of the program (even from other files, using the `extern` keyword) and exist throughout the whole lifetime of the program.

Static variables (keyword `static`) can be declared inside or outside of functions. They exist throughout the whole lifetime of the program but are only visible in the scope, in which they are defined (i.e. the function or the file).

Important concepts:

- Lifetime: When is the variable created, when is it destroyed?
- Scope: From where can the variable be accessed?

Variables: Local, global, static

Beispiel

```
// file example.c
int global_var;         // visible everywhere
static int example;     // visible in file example.c
extern int other;       // Variable defined in a different file

int function(int a)     // a: local variable
{
    int b;              // b: local variable
    static int c=0;     // c: exists only once,
                        // keeps its value between function calls
                        // but is only locally visible
    c++;
    example++;
    global_var++;
    return ...;         // a and b are destroyed when returning
}

int main()
{
    int n=0;         // local var, only exists in main.
    n++; example++; global_var++;
}
```

Recursion

Definition

From a computer dictionary...

Recursion:

If you still don't get it, see: "Recursion".

Recursion is a technique, where a function calls itself:

- The recursive call should have different ("simpler") parameters to stepwise approach a solution.
- There must be simple base cases, which can be solved directly without recursive calls.

Recursion

Exercise

ackermann.c

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Code snippet 506

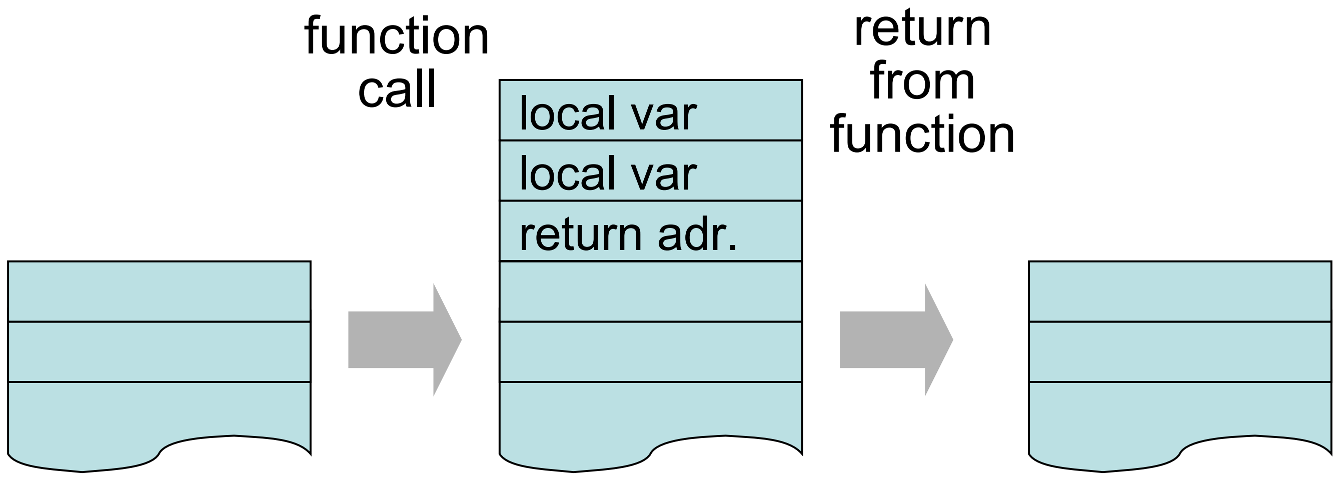
The stack

Definition

The **stack** is a dynamic data structure which stores information about the active functions of the program:

- Return address: From where has the function been called, i.e. which is the command to return to after the function is completed.
- Local variables (including function parameters)

When a function is completed its data is removed from the stack



Towers of Hanoi

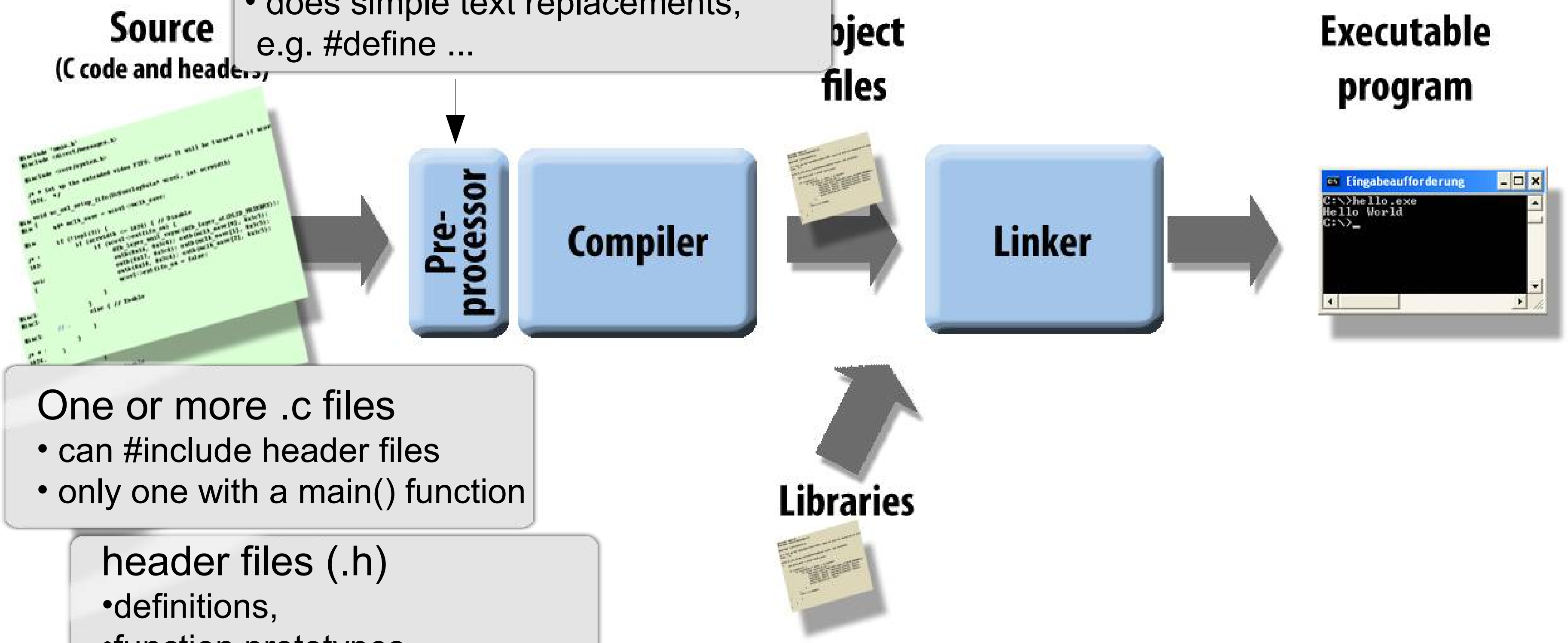
```
void hanoi(int n, int start, int goal, int helper)
{
    if(n>0) {
        hanoi(n-1,start,helper,goal);
        printf("move disk from %d to %d\n", start, goal);
        hanoi(n-1,helper,goal,start);
    }
}

main( ) {
    int n;
    printf("How many disks?");
    scanf("%d", &n);
    hanoi(n, 1, 2, 3);
}
```

hanoi.c

Code snippet 507

Building programs



```
C:\>hello.exe
Hello World
C:\>_
```

parses #... instructions:
- merges header files into source code
- does simple text replacements, e.g. #define ...

One or more .c files
- can #include header files
- only one with a main() function

header files (.h)
- definitions,
- function prototypes
  → for other .c source files
  → for libraries

Preprocessor

The **C preprocessor** handles #-directives in the source file by doing some simple parsing and text replacement before the compilation is done.

- **#include <....>** merges a file into the source file.
  → Typically used to merge definition and function prototypes from header files.
  ```
  #include <stdio.h>        // < >: file is in standard include dir
  #include "mainfunction.h"  // " ": file can also be in local dir
  ```
- **#define *identifier(pars ...) replacement***
  Defines a string to be replaced with a different string.
  Typically used to define constants and macros:
  ```
  #define COUNT 100
  #define MIN(a,b) ((a)>(b)?(b):(a))
  ```
  gefährlich...
- **#if, #ifdef, #ifndef, #else, #elif** and **#endif** are used to conditionally include/exclude segments of the source file:
  ```
  #ifdef WIN32 // WIN32 is defined by all Windows 32 compilers
      system("CLS");    // Clear screen on Windows console
  #else
      // Do something else...
  #endif
  ```

Preprocessor

Exercise

`preprocessor.c`

Use the preprocessor #define directive to modify your previous programs, e.g.

- define constants such as maximum values / array sizes, shortcuts for special ascii characters etc.
- Optionally, try to define macros by using parameters with #define.

Have a look at <stdio.h> and other standard header files and see how preprocessor directives are used there

Code snippet 508