Scientific Computing I

Concise Introduction to the C Programming Language

Martin Köhler

Computational Methods in Systems and Control Theory (CSC) Max Planck Institute for Dynamics of Complex Technical Systems

Winter Term 2024/2025

This Lecture:

The C Basics

History

History

- **1972:** C was developed by Dennis Ritchie at Bell Labs as an evolution of the B programming language to reimplement MULTICS as UNIX.
- **1978:** The first edition of "The C Programming Language" book by Brian Kernighan and Dennis Ritchie was published, often referred to as "K&R C."
- **1983:** The American National Standards Institute (ANSI) formed a committee to establish a standard specification for C, leading to the development of ANSI C.
- **1989:** The ANSI C standard, known as C89, was officially published, providing a standardized version of the language.
- **1990:** The International Organization for Standardization (ISO) adopted the ANSI C standard, resulting in the ISO C standard, also known as C90.
- **1999:** A major update to the C standard, known as C99, introduced new features such as inline functions, variable-length arrays, and new data types.
- **2011:** The C11 standard was released, adding features like multi-threading support, improved Unicode support, and type-generic macros.
- **2018:** The C18 standard was published, mainly focusing on bug fixes and clarifications without introducing new features.

The Programming Environment

Overview

The Programming Environment

Overview

Four steps are necessary to transform the human readable source code to an executable program:

1. **The Preprocessor** searches the source code for special directives beginning with **#**. The output of this phase stays human readable but the code is filled with additional statements and data from other files.
2. **The Compiler** is the main tool. It checks whether the source code is syntactically correct and translates it into assembler code. The assembler output is still human readable and it expresses the same instruction as the C source on a much lower abstraction level.
3. **The Assembler** turns the assembler output into machine code. This can theoretically be executed by the CPU, but missing external libraries prevent this.
4. **The Linker** finally combines the object files and the libraries to an executable program.

These four steps are usually performed by a single compiler call in GCC.

Compiler Invocation

The Programming Environment

Compiler Invocation

The C compiler is invoked in the shell:

```
gcc <opts> -o outputfilename input1.c ... <libs>
```

This compiles all given input files to one executable. If the output filename is omitted the compiler uses **a.out**.

If a program consists of many source files or they need different compiler options it is more convenient to create the single **object files** first:

```
gcc -c input1.c
gcc -c input2.c
...
```

Afterwards the **object files** are linked with libraries to the final executable:

```
gcc -o output input1.o input2.o ... <opts> <libs>
```

The Programming Environment

Compiler Invocation: External Libraries

- External libraries are added using the **`-l`** option.
- A library named **`libNAME`** is linked using **`-lNAME`**.
- Libraries must be specified in the order they depend on each other.
- Cyclic dependencies are resolved by adding the libraries more than once, to the linker invocation.

The Programming Environment

Compiler Invocation: External Libraries

- External libraries are added using the **-l** option.
- A library named **libNAME** is linked using **-lNAME**.
- Libraries must be specified in the order they depend on each other.
- Cyclic dependencies are resolved by adding the libraries more than once, to the linker invocation.

Example

A program depends on **libone**, **libtwo** and **libthree**, where **libtwo** depends on **libone**. The resulting compiler call is:

```
gcc -o output input.c -ltwo -lone -lthree.
```

The Programming Environment

Compiler Invocation: External Libraries

- External libraries are added using the **-l** option.
- A library named **libNAME** is linked using **-lNAME**.
- Libraries must be specified in the order they depend on each other.
- Cyclic dependencies are resolved by adding the libraries more than once, to the linker invocation.

Example

A program depends on **libone**, **libtwo** and **libthree**, where **libtwo** depends on **libone**. The resulting compiler call is:

```
gcc -o output input.c -ltwo -lone -lthree.
```

Two types of libraries exist.

- static libraries (.a) may result in large binaries
- shared object libraries (.so) included only upon execution of the program

The Programming Environment

Compiler Invocation: External Libraries

- **.so** libraries are loaded dynamically
- these libraries need to be in default locations
- the search path can be extended by setting the **LD_LIBRARY_PATH** environment variable

Example

A program uses a library in a non standard location. It is compiled and linked using

```
gcc -o output input.c -L/path/to/the/library -lthelib
```

and executed with adding the path to **LD_LIBRARY_PATH**:

```
export LD_LIBRARY_PATH=/path/to/the/library:$LD_LIBRARY_PATH
./output
```

Development Tools

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.

ddd The Data Display Debugger is a graphical user interface for **`gdb`**.

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.

ddd The Data Display Debugger is a graphical user interface for **`gdb`**.

valgrind Is a suite of debugging tools which analyze the memory access, check for memory leaks, create call graphs,. . .

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

- **gdb** The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.
- **ddd** The Data Display Debugger is a graphical user interface for **`gdb`**.
- **valgrind** Is a suite of debugging tools which analyze the memory access, check for memory leaks, create call graphs,. . .
- **nm** Lists all symbols (functions or variables) in an object file or a library.

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.

ddd The Data Display Debugger is a graphical user interface for **`gdb`**.

valgrind Is a suite of debugging tools which analyze the memory access, check for memory leaks, create call graphs,. . .

nm Lists all symbols (functions or variables) in an object file or a library.

ldd Lists all external libraries required by a program. It also checks if they are found in the current search paths and shows which ones will be used upon execution of the program.

The Programming Environment

Development Tools

Many tools exists to support the programmer during development and debugging. The basic ones are:

gdb The GNU Debugger is a command line tool that helps executing a program step by step, enables to look into variable values at runtime, or view the machine code. It allows a deep analysis of what is going on in the program.

ddd The Data Display Debugger is a graphical user interface for **`gdb`**.

valgrind Is a suite of debugging tools which analyze the memory access, check for memory leaks, create call graphs,. . .

nm Lists all symbols (functions or variables) in an object file or a library.

ldd Lists all external libraries required by a program. It also checks if they are found in the current search paths and shows which ones will be used upon execution of the program.

make An automatic build utility.

Other Compilers

The Programming Environment

Other Compilers

Beside the GNU Compiler Collection, there a set of other compilers for the C programming language:

- **Clang** Part of the LLVM project. Known for its fast compilation and useful error messages.
- **MSVC (Microsoft Visual C++)** Provided by Microsoft for Windows development.
- **Intel OneAPI C++ Compiler** Optimized for Intel processors. Known for high performance.
- **Nvidia HPC SDK (formerly PGI)** Optimized for Nvida ARM CPUs. Advanced GPU offloading features.
- **TinyCC (TCC)** Lightweight and fast.
- **PCC (Portable C Compiler)** One of the oldest C compilers. Focuses on simplicity and portability.
- **IBM XLC** Compiler with focus on Power CPUs for Linux and AIX.

C Statements, Types and Operators

The Basic Structure of a C Program

C Statements, Types and Operators

The Basic Structure of a C Program

The basic structure of a C program looks like

```
#include <stdio.h>
#include <stdlib.h>
// more includes
...
// type definitions (see Section 3.4)
...
// function definitions (see Section 3.5)
...
int main ( int argc, char **argv) {
  // Here comes the code.
  return 0;
}
```

C Statements, Types and Operators

The Basic Structure of a C Program

The basic structure of a C program looks like

```
#include <stdio.h>
#include <stdlib.h>
// more includes
...
// type definitions (see Section 3.4)
...
// function definitions (see Section 3.5)
...
int main ( int argc, char **argv) {
  // Here comes the code.
  return 0;
}
```

stdio.h and **stdlib.h** are two header files from the standard C library (described later). They provide basic input and output, access to files, and other basic actions. They are necessary for essentially every program.

C Statements, Types and Operators

The Basic Structure of a C Program

The basic structure of a C program looks like

```
#include <stdio.h>
#include <stdlib.h>
// more includes
...
// type definitions (see Section 3.4)
...
// function definitions (see Section 3.5)
...
int main ( int argc, char **argv) {
  // Here comes the code.
  return 0;
}
```

main() is the function that is called when a program starts. All statements are executed in the order in which they appear. The **return 0;** statements exits the **main()** function and returns the status code **0** to the operating system.

Comments

Possible comment structures are:

```
// A single line comment

/* Another single line comment */

 /* This
    is
    a multi-line comment */

#ifdef GRAPHICS
   Some code fragment
#endif /*GRAPHICS*/
```

Statements and Blocks

C Statements, Types and Operators

Statements and Blocks

A **statement** in C can be one of the four kinds:

variable declaration

```
data-type varname;
```

function call

```
dosomething();
```

assignment

```
x = 3;
```

or control structure (decribed later)

C Statements, Types and Operators

Statements and Blocks

Statements are grouped to code blocks using { and }:

```
{ // begin of the code block
  Statement1;
  Statement2;
  ...
} // End of the code block
```

Basic Data Types and Variable Declaration

Rules for Variables

- Variables need to be declared before they may be used.
- Declarations consist of a data type followed by a variable name.
- Valid variable names begin with alphabetic charactes and contain no special characters except _.
- The name may not be used for another variable or function in the context.
- Variables need to be declared at the begin of a block or a function following the C89 standard.
- The C99 standard allows this everywhere.

C Statements, Types and Operators

Basic Data Types and Variable Declaration

Basic Data Types

`int`	Stores one signed integer value. Normally this is 4 byte large, that means it can store one 32-bit number.
`long`	Stores one large signed integer value. This must have at least the size of an **`int`** variable but it can be larger. On a 64-bit architecture this is normally 8 byte.
`unsigned int`	Stores an integer without a sign, that means only positive but larger numbers.
`unsigned long`	Stores a **`long`** without a sign, that means only positive but larger numbers.

Basic Data Types ctd.

char	Stores one character from the ASCII table. Internally it is a one-byte integer value and holds values from -127 to 128.
size_t	An unsigned integer value which is large enough to store the size of the largest theoretically possible memory object. Its size depends on the hardware of the platform used.
float	A single precision floating point number, 4 Bytes.
double	A double precision floating point number, 8 Bytes.
void	Non specified type for function with no return value or generic pointers.

C Statements, Types and Operators

Basic Data Types and Variable Declaration

There was no boolean data-type in C until the C99 standard. Boolean values are, therefore, expressed as integers where zero means **false** and all other values are evaluated as **true**. The definitions of variables of basic data types can also contain initial assignments.

```
int x = 1, y;
```

The above definition declares two integers **x** and **y** and initializes **x** with the value 1. The character type **char** is assigned using single quotes:

```
char c = 'A';
```

The single quotes implicitly convert the given character in to the corresponding ASCII value. We introduce strings in the complex data types section.

Operators

Binary Arithmetic Operators

- These are $+$, $-$, $*$, and $/$.
- For integers additionally % (modulo).
- Integer arithmetic is used when both arguments are integers.
- Basic arithemtic evaluation rules apply.
- () influence evaluation order.

Example

```
int x,y,z,r;  // Declares x,y,z, and r to be integers
x = 4;        // Sets x to 4
y = 3;        // Sets y to 3
z = x / y;    // Integer Division of x and y
r = x % r;    // Modulo, the remainder of the division
```

Short Form of Binary Operations

If the left side of an assignment is the same as the first operand of a binary operation this can be abbreviated as in:

```
x += y;  // same as x = x + y;
```

This is possible with all binary operators.

Unary Operators

- The **++** and **--** operators increment or decrement a variable by one.
- Used as pre- or postfix to a variable.
- The prefix increments the variable before its value is used.
- The postfix does it the other way around.

Example

```
int x = 1, y;
x++;              // x = 2;
y = ++x;          // y = 3; x = 3;
y = x++;          // y = 3; x = 4;
```

Bitwise Operators

`x & y`	Perform a bit-wise **and** operation.
`x | y`	Perform a bit-wise **or** operation.
`x ^ y`	Perform a bit-wise **xor** operation.
`~x`	Perform a bit-wise **not** operation.
`x << y`	Bit-Shift on **x**. Move **y** bits to the left.
`x >> y`	Bit-Shift on **x**. Move **y** bits to the right.

Typecasts

A **typecast** is used to convert one data-type into another one. It is performed by putting the new data-type in parentheses in front of a variable.

```
int y; double x;
x = (double) y;  //  converts y from int to double
```

Control Structures

Conditionals

Control Structures

Conditionals

The **`if`**-statement realizes an alternative. The simplest one is:

```
if ( condition ) {
  Statements; // evaluated if the condition is true
}
```

The **`if`** statement can be extended to an **`if-else`** construct. This full alternative is:

```
if ( condition ) {
  Statements; // evaluated if the condition is true
} else {
  Statements; // evaluated if the condition is false
}
```

Control Structures

Conditionals

If more than two cases are necessary this extends to:

```
if ( condition1 ) {
  Statements; // evaluated if the condition1 is true
} else if ( condition2) {
  Statements; // evaluated if the condition2 is true
} else {
  Statements; // evaluated if the condition1 and condition2 are false
}
```

This concept works for more than two conditions analogously.

Control Structures

Conditionals

A conditional assignment

```
if ( condition ) {
  a = value1;
} else {
  a = value2;
}
```

can be reduced with the help of the **?**-operator to:

```
a = (condition)? value1:value2;
```

This is the only ternary operator in C.

Control Structures

Conditionals

The discrete decision statement in C is **switch**. The syntax is

```
switch (variable) {
  case const_1:
    Statements; // evaluated if variable == const_1
    break;
  case const_2:
    Statements; // evaluated if variable == const_2
    break;
  default:
    Statements; // evaluated if none of the other cases matched
}
```

- If there is no **break**-statement the program runs trough all other following cases.
- **switch** only works on discrete data. Interval conditions like **x>4 && x<4.5** require an **if-else** construction.

Control Structures

Conditionals

Conditions

- Expressions that are evaluated to zero (false) or non-zero (true)
- Comparison operators exist for all numerical data-types such as **`int`** or **`double`**

`<`	smaller than
`<=`	smaller than or equal to
`==`	equal to
`!=`	not equal to
`>=`	greater than or equal to
`>`	greater than

Conditions

- Expressions that are evaluated to zero (false) or non-zero (true)
- Comparison operators exist for all numerical data-types such as **`int`** or **`double`**

Boolean operators combine different conditions:

`&&`	boolean and
`||`	boolean or
`!`	boolean negation, prefix operator

Loops

Control Structures

Loops

C provides three different loop constructions:

- **`for`**,
- **`while`**, and
- **`do-while`**.

A loop repeats a group of statements until certain conditions are met.

While Loop

- checks condition on entry,
- repeats execution of statements until condition is no longer met.

The syntax is

```
while (condition) {
  statements; // executed as long as the condition is true
}
```

The condition works exactly as in the **if**-statements.

Do-While Loop

- Executes the statements at least once,
- checks the condition upon exiting the loop block.

The syntax is:

```
do {
  Statements; // executed as long as the condition is true
} while (condition);
```

The semicolon at the end of the statement is untypical but mandatory.

For Loop

- Most general loop concept
- mostly used for enumerated loops
- can emulate both other loops

The syntax is:

```
for (initialization; condition; action){
  Statements; // inside the loop
}
```

- initialization executed before first loop block entry
- condition **for** continues as long as this is true, i.e., non-zero
- action executed at the end of every iteration.

Control Structures

Loops

A **for**-loop is equivalent to a **while**-loop of the form:

```
initialization;
while (condition){
  Statements; // inside the loop
  action;
}
```

Each of the three parts inside the **for**-definition can be made up of multiple expressions separated by commas. They are evaluated from left to right and represent the value of the last expression.

Control Structures

Loops

Example

Output all numbers squared for the sequence from 1 to 10:

```
int i;
for (i = 1; i <= 10; i++){
  printf("␣%d␣*␣%d␣=␣%d\n", i, i, i * i);
}
```

Control Structures

Loops

break-statement

- An emergency exit inside a loop.
- Exits the loop immediately ignoring the condition.
- The program continues with the first statement after the loop.

```
while ( condition ) {
  Statements;
  if ( special condition ) {
    break; // Exits the loop regardless of the while-condition
  }
}
// Control jumps here on the break
```

Control Structures

Loops

continue-statement

- causes the control to jump to the end of the code block defining the loop
- Skips the remaining statements
- Inside a **for**-loop it still evaluates the **action** statements

```
while ( condition ) {
  Statements;
  if ( special condition ) {
    continue;
  }
  Statements;
  // Control jumps here on the continue;
}
```

Control Structures

Loops

Remark 1

Control structures can be nested inside each other as often as desired.

Remark 2

If a control structure only executes one statement, the surrounding brackets `{ }` defining the code block can be omitted.

This Lecture:

Advanced C Topics

Complex Data Types and Arrays

Structures

Complex Data Types and Arrays

Structures

Data-structures are collections of different variables with in a common context. They are defined using the **struct**-statement:

```
struct NameOfTheStructure {
  data-type1 variable1;
  data-type2 variable2;
  ...
};
```

Variables of this type are defined by:

```
struct NameOfTheStructure variable;
```

The .-operator provides access to the components of a structure:

```
variable.member = ...;
x = variable.member;
```

Example

We define a structure representing a point in $\mathbb{R}^3$ and let $P = (0, 1, -1) \in \mathbb{R}^3$ of this type:

```
struct point3d {
  double x, y, z;
};

struct point3d P;
P.x = 0.0;
P.y = 1.0;
P.z = -1.0;
```

Example

We define a structure representing a point in $\mathbb{R}^3$ and let $P = (0, 1, -1) \in \mathbb{R}^3$ of this type:

```
struct point3d {
  double x, y, z;
};

struct point3d P;
P.x = 0.0;
P.y = 1.0;
P.z = -1.0;
```

Assignment is performed using: **`struct1 = struct2;`**

Complex Data Types and Arrays

Structures

Example

We define a structure representing a point in $\mathbb{R}^3$ and let $P = (0, 1, -1) \in \mathbb{R}^3$ of this type:

```
struct point3d {
  double x, y, z;
};

struct point3d P;
P.x = 0.0;
P.y = 1.0;
P.z = -1.0;
```

Comparison via **==** does **not** work.
It has to be performed member by member.

Arrays

Complex Data Types and Arrays

Arrays

Arrays provide a multi-dimensional storage for data of the same data-type.
A one-dimensional (static) array is declared using:

```
data-type name[NumberOfElements];
```

The bracket **[]**-operator provides the access to the elements:

```
x[0] = y;       // Assignment of the first element
h    = x[i-1]; // Access to the i-th element
```

The array-elements are indexed from 0 up to $NumberOfElements - 1$.

Example

We declare a vector $a \in \mathbb{R}^4$:

```
double a[4];
```

It consists of four values **a[0]**, **a[1]**, **a[2]**, and **a[3]**.

Multidimensional Arrays

- defined and accessed via repeated use of **[]**
- sorting of elements in memory uses rightmost index

```
double a[4][5][2];
```

defines a 3d array with $4 \times 5 \times 2$ entries.

The element **a[2][2][1]** and **a[2][2][2]** are stored next to each other in the memory, followed by **a[2][3][1]** and **a[2][3][2]**.

remark

A matrix (2d-array) is stored rowwise. This contrasts Fortran where it is done columnwise.

Every data-type can be made up to an array. Arrays of structures are possible and arrays can be used as members of structures.

Example

We declare an array of 10 Points in $\mathbb{R}^3$:

```
struct point3d {
  double x,y,z;
};
struct point3d points[10];
points[0].x = 10.0; // Set x value of the first point.
points[9].z = -1.0; // Set z value of the last point.
```

Strings

String

- equivalent to array of chars
- length at least number of chars +1
- 0-byte (ASCII **NIL**) terminates string
- assignment uses double quotes

```
char string[10] = "Hello!";
```

will be stored as

Index:	0	1	2	3	4	5	6	7	8	9
Value:	'H'	'e'	'l'	'l'	'o'	'!'	0	*	*	*

in memory.

Pointers

Pointers are the most powerful concept of C and at the same time the most difficult for beginners using the language.
Pointers are

- variables containing memory addresses instead of values,
- references to other memory locations where the actual data is located.

Declaration:

```
data_type *a_pointer_to_data_type;
```

- A pointer should always be assigned to a valid memory location, or **NULL**.
- Accessing an illegal memory region may kill the program.

Pointer Operators

- & **address-of operator** returns the address, i.e., the memory location of a variable.
- * **dereferencing operator** the counterpart of the above. Allows to access the value inside the memory cell pointed to.

```
int var_x, var_y; // declares two int variables
int *ptr_x;       // declares a pointer to an int
var_x = 2;        // Sets the value of var_x
ptr_x = &var_x;   // Assigns the location of var_x to the pointer
var_y = var_x;    // Assings the value of var_x to var_y
var_y = *ptr_x;   // equivalent to the previous
```

Dynamic Array Interpretation

A pointer is simply an array of undetermined size, i.e, a dynamic array.

```
int field[10];
int *ptr;
ptr = &field[0];
int x = ptr[3];
ptr[4] = 4711;
```

- Unused pointers should be set to **NULL** which represents 0 in the pointer context.
- The **void *** pointer is the generic pointer which can be type cast to any other pointer.
- **void *** pointers do not allow for the dynamic array style access.

Dynamic Array Interpretation

A pointer is simply an array of undetermined size, i.e, a dynamic array.

```
int field[10];
int *ptr;
ptr = &field[0];
int x = ptr[3];
ptr[4] = 4711;
```

Note that in expressions as **ptr[3]** above the brackets represent a dereferencing operation for the element chosen by the enclosed index and thus no additional * is needed

Complex Data Types and Arrays

Pointers

Dereferencing the pointer to a **struct** is done using the *-operator and the access to a components is uses the .-operator:

```
struct point3d p;
struct point3d *sptr;
sptr = &p;
(*sptr).x = 0.0;
```

This type of notation **(* sptr).x** has an equivalent representation as in:

```
sptr->x = 0;
```

- pointers can be nested (**int **ptr;**)
- this corresponds to multidimensional arrays
- multiply dereferencing accesses the different levels
- dynamic usage requires **malloc()** and **free()** (both in **stdlib.h**) to claim or free additional memory

Memory Management

`sizeof(type)`

- memory allocation needs to be done relative to sizes of data types
- **`sizeof(type)`**-operator returns the size of a data-type in bytes
- it can be applied to basic data types as well as structures

Example

Print the size of the **`double`** and the **`struct point3d`** type:

```
printf("sizeof(double) = %lu\n", sizeof(double));
printf("sizeof(struct point3d) = %lu\n", sizeof(struct point3d));
```

Complex Data Types and Arrays

Memory Management

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

- may allocate contiguous memory blocks of arbitrary size[1]
- returns **void*** pointer to a **size** bytes large memory segment
- needs to be transformed to the desired data-type using a type cast

```
double *x;
x = (double *) malloc(sizeof(double));
```

If a memory location is no longer used it should be made available again. The **free**-function deallocates the memory referred to by a pointer:

```
void free(void *ptr);
```

[1]Only restricted by the availability of memory.

Example

Allocate an array with 100 **double** entries, sum them up, and free the array:

```
double *array;  // declare the pointers

// Allocate 100*sizeof(double) bytes memory
array = (double *) malloc(sizeof(double)*100);

// sum them up
double sum = 0.0;
for (i = 0; i < 100; i++) {
  sum += array[i];
}

free(array); // free the memory
```

If an allocated memory location is too small or too large it can be resized using the **realloc**-function:

```
(void *) realloc(void *oldptr, size_t newsize);
```

- inputs: current location and desired size of the segment
- output: (possibly) new location of the resized segment
- data in the part that is kept remains untouched
- if **oldptr** is **NULL**, then behavior is as in **malloc()**

valgrind is an excellent tool to detect errors with wrong access to pointers or wrong usage of the memory management function.

Functions

Trivia

Functions

Trivia

- The **`main`**-function is the starting function of every program.
- It is called automatically when a program is executed.
- Statements like **`printf`** and **`scanf`** are functions, too.
- Some important standard functions are introduced in Section 2.6.

Functions are called using their name followed by a list of arguments in parentheses. If the return-value is needed it is used like a variable in an expression or a function in a mathematical context.

Example

Check if **`scanf`** has read two integers correctly:

```
int i1, i2, r;
r = scanf("%d %d", &i1, &i2);
if ( r != 2 ) {
  printf("scanf did not read 2 integers successfully.\n");
}
```

Definition of Own Functions

Functions

Definition of Own Functions

A function consists of two parts:

- header defines the input/output arguments and the return type
- body code block implementing the functions behavior

```
return-type function-name(argument-list) {
  // Local declarations
  Statements;
  Statements;
  return return-value;
}
```

- **`return-type`** can be any simple data-type, structure, or pointer
- **`void`** is used for functions without return value
- naming conventions for variables also apply to functions
- argument list is a comma-separated list of the format **`data-type variable`**

- function header without the body is called **signature of a function**
- compiler checks if the calling sequence for a function is compatible with its signature

Example

Define a function named **sqr** operating on a double precision number and returning the square of the argument:

```
double sqr(double x) {
  double a;
  a = x * x;
  return a;
}
```

The signature of this function is **double sqr(double x);**

Argument Behavior

- By default arguments are copied to the function
- function works on a copy of the data not modifying the original
- behavior is called **Call by Value**
- to change a given argument at its original location the arguments needs to be a pointer to the variable
- behaviour is called **Call by Reference** because only a reference to a variable is passed
- A function can return more than one value or complex data types using this technique

Functions

Definition of Own Functions

We define a function which takes two integer values as arguments and swaps their values.

Example

```
void swap(int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
// in main()
int x = 4;
int y = 5;
swap(x, y);
```

Wrong!

Only exchanges the local copies inside the function.

Functions

Definition of Own Functions

We define a function which takes two integer values as arguments and swaps their values.

Example

```
void swap(int *a, int *b) {
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
// in main()
int x = 4;
int y = 5;
swap(&x, &y);
```

Correct!

Call by reference usage changes data in the original location.

Example

The **main**-function of a C program is a special case of a function that takes two arguments:

- **int argc** contains the number of command line arguments passed to the program
- **char **argv** is an array of strings
- Each string contains one command line argument
- **argv[0]** contains the name of the program

```
int main(int argc, char **argv)
```

This Lecture:

Advanced C Topics II

Introduction to the Standard Library

The ISO C Standard

Introduction to the Standard Library

The ISO C Standard

- defines a standard library to provide basic functions on every platform and allows portable programming
- consists of about 20 different header files
- around 200 function for input/output, basic math, string manipulation, and memory management

POSIX C Library

- important extension to the standard C library
- provides more operating system dependent operations
- contains functions for networking, inter process communication, threading, and many more

Starting with the C11 standard, threading has also become part of the standard C library.

`stdio.h` and **`stdlib.h`**

Introduction to the Standard Library

`stdio.h` and **`stdlib.h`**

These two headers files provide the basic functionality of the C library. They provide input/output operations, control statements, and memory management.

The file-io operations will be demonstrated by examples in a separate section.

The input/output functions introduced later in this section contain **format strings** determining what is to be read or printed. These format strings contain **format specifiers** for the representation of the variables contents. They will be introduced first.

Introduction to the Standard Library

stdio.h and **stdlib.h**

d	integers of the type **int**
ld	integers of the type **long**
u	integers of the type **unsigned int**
g	float pointing numbers of the type **float** or **double**
e	float pointing number in [-]d.ddde+dd notation
c	a single character of type char
s	strings (see Section 2.4 in lecture notes)
%	the % sign.

The full format specification has the form

```
% [flags][width][.precision][l]type
```

The **[l]type** part is what is shown above. The **[flags]** influence the alignment and printing of signs. All bracketed specifiers are optional.

Introduction to the Standard Library

Example

```
double pi = 3.14159265;
printf("pi␣=␣%8.6g\n",pi);
```

prints:

```
pi = 3.141593
```

Note that the decimal dot is consuming one of the 8 digits.

The placeholders and modifiers are described in **`man 3 printf`**

Introduction to the Standard Library

stdio.h and **stdlib.h**

```
int printf(const char *formatstring, arguments, ...);
int fprintf(FILE *f, const char *formatstring, arguments, ...);
int sprintf(char *buf, const char *formatstring, arguments, ...);
```

The **printf**-function writes a text to the standard output. The **fprintf**-function is the equivalent for files, whereas **sprintf** stores the result in the output string **buf**. The return-value is the number of characters written.

stdio.h defines **stdout** and **stderr** file descriptors to use **fprintf** for printing output and error messages separately.

Introduction to the Standard Library

stdio.h and **stdlib.h**

```
int scanf(const char *formatstring, arguments, ...);
int fscanf(FILE *f, const char *formatstring, arguments, ...);
int sscanf(const char *string, const char *formatstring, arguments, ...);
```

- **scanf**-function reads a formatted input from the standard input. This is the keyboard in most cases. The arguments are pointers to the variables where the values read from the input are stored.
- **fscanf**-function is the equivalent to read data from a file.
- **sscanf** reads from another string.
- **fscanf** stops reading when either the end of a line, or the end of the file is reached.
- **sscanf** terminates upon reaching the 0-byte.

Introduction to the Standard Library

stdio.h and **stdlib.h**

```
FILE *fopen(char *filename, char *mode);
```

The **fopen**-function opens the file specified by the **filename** and returns a pointer to the file stream. **mode** is a string determining the access to the file:

Mode	Meaning	Remarks
r	open for reading	Only possible if the file exists otherwise NULL is returned.
w	create a file for writing	If the file already exists the content is destroyed.
a	append data to a file	If the file already exists, the new data is appended to the end. If it does not exist, the behavior is like "w".

fopen returns NULL in case of an error.

Introduction to the Standard Library

stdio.h and **stdlib.h**

```
int fclose(FILE *stream);
```

The **fclose**-function closes a given file stream. Any buffered data is written to the file. The **stream** is no longer associated with the file.

```
int feof(FILE *stream);
```

The **feof**-function returns true if the given file stream reached the end of the file otherwise false is returned.

```
void perror(const char *s);
```

The **perror**-function displays the most recent error from the C library. The string **s** is used as a prefix to the error message.

Introduction to the Standard Library

stdio.h and stdlib.h

```
void *malloc(size_t size);
void *realloc(void *ptr, size_t new_size);
void free(void *ptr);
```

The memory management functions explained earlier.

```
void abort();
void exit(int exit_code);
```

- **abort** terminates a program immediately without any clean up
- **exit** terminates a program immediately with clean up

```
int atoi(char *s):
double atof(char *s);
```

The **atoi**-function converts a string to an integer if possible. The **atof**-function does the same with a floating point number.

`math.h` and **`complex.h`**

math.h and **complex.h**

- Provide common mathematical functions and constants
- A program that uses at least one of them needs to be linked with **-lm**
- All of the following functions take **double** arguments and produce **double** return values

Introduction to the Standard Library

`math.h` and `complex.h`

`fabs(x)`	absolute value of `x`
`exp(x)`	returns e^x
`exp2(x)`	returns 2^x
`log(x)`	returns $\ln x$
`log10(x)`	returns $\log_{10} x$
`log2(x)`	returns $\log_2 x$
`sqrt(x)`	returns $\sqrt{x}$
`hypot(x,y)`	returns $\sqrt{x^2 + y^2}$
`pow(x,y)`	returns x^y
`sin(x)`	returns $\sin x$
`cos(x)`	returns $\cos x$
`tan(x)`	returns $\tan x$
`asin(x)`	returns $\sin^{-1} x$
`acos(x)`	returns $\cos^{-1} x$
`atan(x)`	returns $\tan^{-1} x$

Introduction to the Standard Library

math.h and **complex.h**

- The C99 standard introduces the data types **float complex** and **double complex** for handling complex numbers.
- The header file **complex.h** defines these data types along with the imaginary unit as **I** and the following functions for double precision complex arguments and return values.

Introduction to the Standard Library

`math.h` and `complex.h`

`creal(x)`	real part of **`x`**
`cimag(x)`	imaginary part of **`x`**
`carg(x)`	computes the phase angle of a complex number
`cabs(x)`	computes the magnitude of a complex number
`conj(x)`	returns $\bar{x}$
`cexp(x)`	returns e^x
`clog(x)`	returns $\ln x$
`csqrt(x)`	returns $\sqrt{x}$
`cpow(x,y)`	returns x^y
`csin(x)`	returns $\sin x$
`ccos(x)`	returns $\cos x$
`ctan(x)`	returns $\tan x$
`casin(x)`	returns $\sin^{-1} x$
`cacos(x)`	returns $\cos^{-1} x$
`catan(x)`	returns $\tan^{-1} x$

Introduction to the Standard Library

`math.h` and `complex.h`

- The list of mathematical functions presented here is not complete.
- More can be found in the man pages or the C standard.
- For nearly all double precision functions there exists a corresponding single precision function with an **`f`** as suffix.
- For example the single precision square root is computed by **`sqrtf(x)`**.

Some predefined constants are:

M_PI	$\pi = 3.14159265358979323846$
M_PI_2	$\frac{\pi}{2} = 1.57079632679489661923$
M_E	$e = 2.7182818284590452354$
M_SQRT2	$\sqrt{2} = 1.41421356237309504880$

string.h

Introduction to the Standard Library

string.h

The **string.h**-header file contains various functions to manipulate and work with strings. The important ones are:

```
size_t strlen(char *s);
```

The **strlen**-function returns the length of the string not including the terminating 0 character.

```
char *strcpy(char *dest, char *src);
```

- Copies a string from **src** to **dest** and returns the **dest** pointer.
- **dest** needs to be a preallocated string with at least **strlen(src)+1** elements.
- The destination string is not 0-terminated if the source string does not contain the 0-byte within the length of the destination string.
- The behavior in case the destination is to short is unspecified and may depend on the actual implementation of the compiler.

Introduction to the Standard Library

string.h

```
char *strcat(char *dest, char *src);
```

The **strcat**-function appends the string from **src** to **dest** and returns the **dest** pointer again. **dest** needs to be a preallocated string with at least **strlen(src)+strlen(dest)+1** elements.

```
int *strcmp(char *lhs, char *rhs);
```

The **strcmp**-function compares two strings lexicographically. It returns a negative value if **lhs < rhs**, a positive value if **lhs > rhs** and 0 if they are equal.

Additional Memory Manipulation Functions in **`string.h`**

Beside the string operations **string.h** defines a variety of memory actions like:

```
void *memcpy(void *dest, void *src, size_t n);
```

The **memcpy**-function copies **n** bytes from **src** to **dest** and returns the **dest** pointer again. **dest** needs to be preallocated with **n** bytes. **src** and **dest** must not overlap each other. **memmove** does the same but allows overlapping. It is slower than **memcpy**.

```
void *memset( void* dest, int ch, size_t count );
```

The **memset**-function converts the value **ch** to an **unsigned char** and copies it into each of the first **count** characters of the location referred by **dest**.

File I/O

Examples

fopen

opens a specified file in the desired mode. To avoid undefined behavior we have to check if **NULL** was returned.

Example

We create file "test.txt" for writing:

```
FILE *fp;
fp = fopen("test.txt","w");
if ( fp == NULL ) {
  perror("can␣not␣open␣test.txt␣for␣writing.");
  return -1;
}
```

If we want to read data from a file we have to use "r" instead.

File I/O

Examples

The **fprintf** and **fscanf** functions in the following are only useful for human readable files. For individual access to binaries we refer to **fread**, **fwrite** and other functions from **stdio.h**.

Example

The access modes "w" and "a" open files for writing. **fprintf** is used like **printf** on this file:

```
int x = 10;
double y = 145.1;
fprintf(fp, "x␣=␣%d␣,␣y␣=␣%lg\n", x, y);
```

File I/O

Examples

The access mode `"r"` allows **`fscanf`** to read data from it. If the **`feof()`**-function evaluates to true, no more data can be read from the file.

Example

We consider a human-readable file with the following layout:

```
x1 y1
x2 y2
...
```

Example

The code-snippet to read all values and print them to the screen will be:

```
FILE *fp;
double x, y;
fp = fopen("test.txt","r");
if ( fp == NULL ) {
  perror("can not open test.txt for reading.");
  return -1;
}
while (!feof(fp)){
  fscanf(fp, "%g %g",&x,&y);
  printf("x=%g \t y=%g\n",x,y);
}
```

After reading or writing to a file it needs to be closed by `fclose(fp)`.

The Preprocessor and Header Files

`#include`

The Preprocessor and Header Files

`#include`

- used to include other files into the current source code
- mostly used for header files of libraries containing function-headers, data-structures or constants
- entire content of the included file is temporarily copied to the position of the **`include`**-statement

Two Versions of Includes

- ```
  #include <header.h>
  ```
  searches default include path first and then all directories specified with **`-I`** at the **`gcc`** command line for **`header.h`**
- ```
  #include "header.h"
  ```
  checks the local project directory and afterwards the default and the **`-I`** paths. Can also be used to include other **`.c`**-files.

#define

The Preprocessor and Header Files

#define

1.) Constants:

Example

The preprocessor statements:

```
#define PI 3.14519
#define SQRT2 sqrt(2)
```

will replace any occurrence of **PI** with **3.14159** and of **SQRT2** with **sqrt(2)** in the current source file.

2.) Preprocessor Macros

Example

The following macro will give the absolute value of the parameter:

```
#define ABS(X) (((X)>0)?(X):(-(X)))
```

This replaces **y = ABS(z+1);** with:

```
y = (((z+1)>0)?(z+1):(-(z+1)));
```

If **X** is not enclosed with parentheses this is evaluated to:

```
y = ((z+1>0)?z+1:-z+1));
```

This is not the desired behavior because the minus in the second part is only applied to **z** and not to the whole expression as it was intended.

The Preprocessor and Header Files

`#define`

3.) boolean variables for the **`#ifdef`**-statement.

- evaluates to true when the define exists
- preprocessor variables can be set using the **`-D`** command line option of the compiler, i.e., **`gcc -DDEBUG ...`**, makes the Macro-variable **`DEBUG`** set, i.e., evaluate to true, in the preprocessor.

The Preprocessor and Header Files

`#define`

3.) boolean variables for the **`#ifdef`**-statement.

- evaluates to true when the define exists
- preprocessor variables can be set using the **`-D`** command line option of the compiler, i.e., **`gcc -DDEBUG ...`**, makes the Macro-variable **`DEBUG`** set, i.e., evaluate to true, in the preprocessor.

Remark

The preprocessor acts stupid on all replacements of **`define`**. It does not check whether or not the resulting code is valid C code. The programmer has to make sure that the **`define`** statements are extended to correct C code.

#if

The Preprocessor and Header Files

#if

if-directive

- allows conditional compiling of the source code based on a conditional
- the conditional must consist on boolean operations or integer math
- **else** and **elif** (else-if) available as well
- works like the **if-else** construct, but is evaluated by the preprocessor at compile time

```
#if CONDITION1
// Code compilied if CONDITION1 is true
#elif CONDITION2
// Code compiled if CONDITION2 is true
#else
// Code compiled otherwise
#endif
```

#ifdef

The Preprocessor and Header Files

#ifdef

ifdef-directive

- allows conditional compiling of the source code based on the definition of a preprocessor variable
- if the variable does not exists or evaluates to 0, false is assume
- short-hand for **if defined(...)**

```
#ifdef PREPROCESSOR_DEFINE
// Code compilied if PREPROCESSOR_DEFINE exitsts
#else
// Code compiled otherwise
#endif
```

This technique is used to handle different environment situations in a single source file

The Preprocessor and Header Files

#ifdef

Example

In order to debug a program easily somebody defined an **INFO**-macro which prints the given parameter to the screen. In the final version of the program this is not necessary. However, removing all outputs in the code may be unwanted to be able to insert them again for later debugging purposes:

```
#ifdef DEBUG
#define INFO(X) printf(X)
#else
#define INFO(X)
#endif
```

If **DEBUG** is defined the **INFO**-macro is expanded to a **printf**-statement, otherwise it is replaced with nothing.

The **#ifndef** statements is the opposite of **#ifdef**. It simply negates the condition of the **#ifdef** statement.

Header-Files

Header-Files

- tell the compiler which functions, data-structures, and constants exist in other source files
- compiler can only check the function headers and the calling sequence in the current file
- similar to a normal source file but consist only of definitions
- come without any implementation

Cyclic inclusions should be avoided using the preprocessor commands **`#define`** and **`#ifndef`**

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
// Your header code
#endif
```

In the literature: Header Fence, Include Guard

The Preprocessor and Header Files

Header-Files

Example

exfct.c implements the function **something**:

```
#include <math.h>  // for sqrt
#include "exfct.h" // Ensure that the function header
                   // fits to the one from exfct.h
double something(double x, double y, double z){
  return sqrt(x*x+y*y+z+z*z);
}
```

Example

exfct.h contains:

- the function header (its signature)
- a preprocessor trick preventing double inclusion in one file:

```
#ifndef EXFCT_H
#define EXFCT_H
double something(double x, double y, double z);
#endif
```

The main program can now include the header and knows how the function **something** is called correctly.

The Preprocessor and Header Files

Header-Files

C and C++ compilers understand the same code, but symbol names are not compatible.

```
#ifdef MY_HEADER_H
#define MY_HEADER_H

#ifdef __cplusplus
extern "C" {
#endif

//Your C header codes goes here

#ifdef __cplusplus
} // extern "C"
#endif
#endif
```

The **`extern "C"`** statement is only evaluated by the C++ compiler and tells him to treat the following code with the naming rules of the C compiler.

Makefiles

Make

Makefiles

Make

- automates build procedures
- controlled by a textfile usually called **`Makefile`**
- **`Makefile`** contains the build instructions and interdependencies
- deals with dependencies
- only recompiles files that really changed

Different Vendor Versions

- GNU Make
- BSD Make
- Microsoft **`nmake`**

Makefile

- works as a simple dependency tree
- compiles the files that are outdated in the order they depend on each other
- consists of so called targets, which may depend on each other

A target is defined by a rule:

```
targetname: dependencies
    command1
    command2
    ...
```

The indentation in front of the commands must be a `<tab>` and not spaces!

Makefile

- works as a simple dependency tree
- compiles the files that are outdated in the order they depend on each other
- consists of so called targets, which may depend on each other

A target is defined by a rule:

```
targetname: dependencies
    command1
    command2
    ...
```

The **targetname** should be equal to or closely related to the output file generated by the commands.

Makefile

- works as a simple dependency tree
- compiles the files that are outdated in the order they depend on each other
- consists of so called targets, which may depend on each other

A target is defined by a rule:

```
targetname: dependencies
    command1
    command2
    ...
```

dependencies is a space separated list of other targets that need to be compiled prior to the target or names of files which need to exist.

Example

Consider a small software project consisting of **main.c**, **file1.c** and **file1.h**. A makefile to create the final program **prog** looks like:

```
prog: main.c file1.c file1.h
    gcc -c main.c
    gcc -c file1.c
    gcc -o prog main.o file1.o
```

If the makefile is named **Makefile** or **makefile**, use:

```
make targetname
```

If the makefile has another name, use:

```
make -f makefilename targetname
```

If no **targetname** is specified, the first one in the makefile is used.

Variables

- **make** supports definition of variables
- often they contain lists of files
- or they are used to inherit compiler settings from include files

A variable is set by

```
VARNAME=VALUE
```

and accessed with **$(VARNAME)**. To change the extension of all files listed in a variable the substitute command is used. The syntax is

```
NEWVAR = ${OLDVAR:.old=.new}
```

This replaces the extension of every file ending with **.old** in **OLDVAR** to **.new** and stores the list to **NEWVAR**. This is normally used to create a list of object files form the list of source files.

Suffix Rules

- avoid separate rules for all input files
- create targets for all files matching the suffix
- apply to all files that have not been processed by a separate rule before

```
.SUFFIXES: .in .out
.in.out:
    command1
    command2
    ...
```

- create a target for every file ending on **.in**
- transform it into the same filename with the extension **.out**
- used to compile source code from **file.c** to an object file **file.o**

Makefiles

Make

- Two placeholders exist referring to the input and the output filenames. The input file is referred to using **`$<`** and the output file using **`$@`**.
- Finally, we define a clean up target. The target **`clean`** removes all object files or intermediate outputs. Because this target does not produce an output file and does not depend on a file called **`clean`**, it needs to be declared as **`.PHONY`** target.
- Other techniques extend the make file such as automatic dependency creation using the GCC compiler, pattern rules as a generalization of the suffix rules, include statements, if directives, and many more.
- Other tools like CMake[2] or the GNU Autotools[3] provide high level scripting languages to create complex makefiles automatically.

---

[2] `https://www.cmake.org`
[3] `https://en.wikipedia.org/wiki/GNU_build_system`

Example

```
SRC=main.c file1.c
OUTPUT=prog
CC=gcc
CFLAGS= -O2
OBJECTS=${SRC:.c=.o}
.PHONY: clean
.SUFFIXES: .c .o

$(OUTPUT): $(OBJECTS)
	$(CC) -o $(OUTPUT) $(CFLAGS) $(OBJECTS)

.c.o:
	$(CC) -c -o $@ $(CFLAGS) $<

clean:
	rm -f $(OBJECTS)
```

This Lecture:

Advanced C Topics III

Writing own Libraries

General Reminder

Writing own Libraries

General Reminder

Libraries

- collections of precompiled functions, datastructures and predifined constants together with the header files providing their signatures
- do not provide a **`main`** function
- standard C library is the most prominent and important example for a library
- Two different types of libraries exists
  - **static libraries** are easy to create but need more space on the mass storage and cause problems with cyclic dependencies between libraries
  - **dynamic libraries** are a bit more complicated to create but take less space on the mass storage and can in some cases be exchanged without recompiling the program

Static Libraries

Static Libraries

- collection of object files combined in a specially structured archive
- classical UNIX **ar**-file containing all **.o**-files of the library and a search index.
- source code needs to be compiled to object code using the -c option
- all object files are combined to a **.a**-file together:

```
ar crs libNAME.a *.o
```

  - **c** option creates an archive
  - **r** option replaces existing files inside the archive
  - **s** option adds an object index to speed up linking procedures

Writing own Libraries

Static Libraries

A static library is linked to a program by adding the **.a**-file to the compiler call:

```
gcc -o program main.c libname.a
```

Example

We consider the minimal external function example again. The following steps create a static library and link it against a program.

```
gcc -c -fPIC exfct.c
...
ar crs libexfct.a *.o
gcc -o prgm main.c libexfct.a
```

Remark

Static libraries used in conjunction with dynamic ones or on a 64-bit architecture must be compiled with the **-fPIC** flag.

Dynamic/Shared Libraries

Dynamic/Shared Libraries

- are almost the same as standard programs
- contain no **`main()`** function
- when linked to a program only a crossreference is added
- dynamic loader loads the symbols from the library to the address space of the program upon execution
- external functions from the library are then called from the program memory and the library code is executed

- The dynamic linker/loader typically searches **`/lib`**, **`/usr/lib/`**, and **`/usr/local/lib/`** for shared libraries.
- The **`LD_LIBRARY_PATH`** environment variable is used tospecify additional search paths

Writing own Libraries

Dynamic/Shared Libraries

- Dynamic libraries can be replaced without relinking program as long as they use a compatible binary interface.
- If at least one function head, i.e. signature, changed or a data structure in a header file has changed, the program needs to be recompiled and relinked.

Writing own Libraries

Dynamic/Shared Libraries

Creation

- created using compiler and linker
- source code needs to be compiled with **`-fPIC`**
- the **`-shared`** option advises the compiler and the linker options to create a shared library
- output file name must follow the **`libNAME.so`** naming convention

Example

We reconsider the minimal external function:

```
gcc -shared -fPIC -o libexfct.so exfct.c
gcc -o prgm -L. -lexfct main.c
```

The **`libexfct.so`** can be modified without relinking it to the output program as long as the function signature does not change.

Interfacing Fortran

Fortran

Interfacing Fortran

Fortran – Why?

- A high-level programming language used **primarily for numerical and scientific computing**.
- Developed: In the 1950s by IBM, led by John Backus. First release **1957**.
- Key Features:
  - Strong support for numerical computation and scientific computing.
  - Efficient execution of mathematical operations.
  - Extensive libraries for scientific calculations (BLAS, LAPACK, SCALAPACK, ARPACK, FFTPACK, ELPA, . . . , FEM Package)
- Compilers produce more efficient code compared to C by default.
- Internal support for Vectors and Matrices.

Interfacing Fortran

Fortran

- recent Fortran provides an interface to C[4]
- however, this is not supported by all compilers and only works with recent standards
- mathematical software typically relies on Fortran 77 (an old standard) or Fortran 90
- Fortran files usually end on **`.f`**, **`.f90`**, or **`.f95`**
- the compiler for Fortran in the GCC is **`gfortran`** and supports most of the switches that we know from **`gcc`**

[4] `https://de.wikibooks.org/wiki/Fortran:_Fortran_und_C`

An Example

Interfacing Fortran

An Example

The DAXPY[5] operation from the Basic Linear Algebra Subroutine library (BLAS)[6] is used as an example to explain how a Fortran subroutine is called from C. The DAXPY operation computes

$$y = y + \alpha x$$

for two vectors $x, y \in \mathbb{R}^n$ and a scalar $\alpha \in \mathbb{R}$. The Fortran function header is

```
SUBROUTINE DAXPY(N,DA,DX,INCX,DY,INCY)
      DOUBLE PRECISION DA
      INTEGER INCX,INCY,N
      DOUBLE PRECISION DX(*),DY(*)
```

[5] https://www.netlib.org/blas/daxpy.f

[6] https://www.netlib.org/blas (see also Chapter 6 in Lecture Notes)

Interfacing Fortran

An Example

- Fortran passes values using **Call by Reference**
- all arguments will be pointers no matter if they are scalar values or vectors
- data-types of the arguments translate following:

Fortran type	C type
`INTEGER`	`int`
`REAL`	`float`
`REAL*8`	`double`
`DOUBLE PRECISION`	`double`
`COMPLEX`	`float complex`
`COMPLEX*16`	`double complex`
`DOUBLE COMPLEX`	`double complex`

Function Name Translation

For the GNU Compiler Collection (under Linux, *BSD, MacOS) the rules are:

- The function name is translated to lower case.
- A trailing underscore _ is added to the function name.
- If the function name contains an underscore, a second underscore is added.

- Fortran subroutines compare to C functions with a **`void`** return-type
- for Fortran functions instead the return-type needs to be translated according to the previous list
- return variables are not pointers

Interfacing Fortran

An Example

Applying these rules to the DAXPY subroutine gives:

```
void daxpy_(int *N, double *DA, double *DX, int *INCX,
                                double *DY, int *INCY);
```

This function header is necessary in every C source code which uses the Fortran routine. It can also be moved to a header file.

The following code computes

$$y = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \qquad y = y + 2 \cdot \begin{pmatrix} 4 \\ 3 \end{pmatrix}$$

using the DAXPY subroutine:

Interfacing Fortran

An Example

```
#include <stdio.h>
#include <stdlib.h>
void daxpy_(int *N, double *DA, double *DX, int *INCX, double *DY, int *
    INCY);

int main ( int argc, char *argv){
  double x[2] = { 4 ,3 };
  double y[2] = { 1 ,2 };
  double alpha = 2.0;
  int n = 2, incx = 1, incy = 1;

  daxpy_(&n, &alpha, x, &incx, y, &incy);

  printf("y = [ %g, %g ]\n",y[0],y[1]);

  return EXIT_SUCCESS;
}
```

Interfacing Fortran

An Example

The program is compiled calling:

```
gfortran -c daxpy.f
gcc -c main.c
gcc -o prgm main.o daxpy.o -lm -lgfortran
```

The math (**-lm**) and the Fortran runtime library (**-lgfortran**) need to be added to the program.

Interfacing other Fortran subroutines works analogously.

The lazy way

Interfacing Fortran

The lazy way

If you are using a newer verion of the GNU compiler collection, **gfortran** assists the translation of the function name and its arguments:

```
gfortran -fsyntax-only -fc-prototypes-external daxpy.f > daxpy_header.h
```

The header file **dapy_header.h** contains the C compatible header:

```
#include <stddef.h>
#ifdef __cplusplus
extern "C" {
#else
#endif
void daxpy_ (int *n, double *da, double *dx, int *incx, double *dy,
    int *incy);

#ifdef __cplusplus
}
#endif
```

The include guard is not generated.

Interfacing Fortran

The lazy way

If you are using a newer verion of the GNU compiler collection, **gfortran** assists the translation of the function name and its arguments:

```
gfortran -fsyntax-only -fc-prototypes-external daxpy.f > daxpy_header.h
```

The header file **dapy_header.h** contains the C compatible header:

```
#include <stddef.h>
#ifdef __cplusplus
extern "C" {
#else
#endif
void daxpy_ (int *n, double *da, double *dx, int *incx, double *dy,
    int *incy);

#ifdef __cplusplus
}
#endif
```

The include guard is not generated.

Automatic Generation of Documentations Using DOXYGEN

DOXYGEN in Short

DOXYGEN

- is a documentation generator tool
- allows to write the documentation directly inside the source code
- extracts the documentation from specially structured comments
- generates HTML files, a LaTeX document, an RTF document, or man pages
- supports, e.g.,
  - C
  - C++
  - Java
  - Fortran
  - Python

Automatic Generation of Documentations Using DOXYGEN

DOXYGEN in Short

- uses modified comments to control the documentation generation
- in C, multiline comments starting with `/**` are evaluated
- comments in front of objects like structures, functions, ... refer to those objects
- documentation is improved by special keyword statements inside those comment blocks:

@brief	Set the brief documentation of the object.
@param	Document a parameter of a function.
@return	Document the return value of a function.
@author	Set the author of a function.
@version	Set the version of an object.
@see	Create a cross reference to an other function, struct,...

Automatic Generation of Documentations Using DOXYGEN

DOXYGEN in Short

- keywords can start with a \ instead of the `@` character
- lines not beginning with a **`doxygen`**-command are considered normal documentation text
- standard C comments are not recognized by **`doxygen`**
- HTML tags or LaTeX-style formulas can be used in the documentation
- a LaTeX formula is enclosed by **`\f$`** or **`\f[`** and **`\f]`**
- for HTML output the LaTeX-formulas are rendered and included as images
- if the output is a LaTeX document the basic HTML tags are converted to the corresponding LaTeX-commands

Automatic Generation of Documentations Using DOXYGEN

DOXYGEN in Short

```
/**
  \brief Squares a given double value.
  \param x Input value.
  \return the square of the input value x.

  The sqr function returns the square \f$ x^2 \f$ of a
  given number x. <i>The intermediate result is stored
  in an internal variable.</i>
*/
double sqr(double x) {
  /* This is not for doxygen. */
  double a;
  a = x * x;
  return a;
}
```

Automatic Generation of Documentations Using DOXYGEN

DOXYGEN in Short

Beside the special comments inside the source code **`doxygen`** is controlled by a so called **`Doxyfile`**. This specifies the source directory, the output format, and other in- and output related options. A template of this file is generated using:

```
doxygen -g config_filename
```

The generated file is well documented and easily customizable using a normal text editor. The documentation of a software project is created by simply calling

```
doxygen config_filename
```

If **`doxygen`** is invoked without any configuration file, it searches for a file named **`Doxyfile`** in the current directory.