Prof. Dr. rer. nat. habil. Martin O. Steinhauser
National Defense Academy, Yokosuka, Japan

Frankfurt University, Germany
October 2024

Computational Materials Science with Atomistic and Coarse-Grained Methods

Handout 1

A Survival Guide to the Language C

1 Introduction

This handout provides you with some basic features and command syntax of the language C which you have to learn and to know to be able to read, write and understand Molecular Dynamics (MD) programs written in this language. Learning a programming language is much like learning a human language. Once you have learnt the basic vocabulary and some grammar, you need to go out and apply it. The same goes for programming. You can only master it by practice. So, if you are not already experienced in programming, I recommend that you type the provided code examples (snippets) yourself and try to compile and run them despite their simplicity. If you don't get some code running, come and ask me during/after lecture. We focus here only on the absolutely essential elements of C that you will most likely need to write your own MD programs and to be able to understand code (or parts of it) written by others. While the language C is infamous for its capability of allowing to write extremely concise (and unreadable code), it just takes a little discipline to do much better.

Hence, it is important that you adopt from the beginning a good style of programming by adhering to some general (mostly unwritten) rules of software design, which do not only apply to C-programming but are valid for any software of more than trivial complexity. These general good programming practices or, more sophisticated: software design rules are general recommendations to ensure that your written code can be understood, maintained and extended by others (and by yourself), and I provide some of them here. You will (it is hoped) later understand why they make sense, when you might start writing yourself considerable complex code.

2 Software Design Rules

Before writing a single line of code, have the algorithm clarified and understood. It is crucial to have a logical structure of e.g. the organization and the flow of data before on starts writing anything. Put shortly: Think first, then start writing.

- Implement a working code with emphasis on design for extension and maintenance. Focus on the design of your code in the beginning and don't think too much about efficiency before you have verified your program. A rule of thumb is the 20-80 rule: 80% of the CPU time is spent in 20% of the code, and you will experience that typically only a small part of your code is responsible for most of the CPU expenditure. Therefore, spend most of your time in devising a good algorithm to solve your problem.

- The planning of the program should be *from top to bottom*, trying to keep the flow of commands as linear as possible. Avoid jumping back and forth in any program. First, arrange the major tasks to be done. Then try to break the major tasks into subtasks which can be represented using functions.

- Write your code in a modular way using different *.c*-files which collectively include the definitions of functions which logically belong together, e.g. *InputOutput.c*, *ReadInputFiles.c*, *Integrate.c*, etc. Then use appropriate *.h* header files to include the declarations of your functions into those *.c*-files, where these functions are used. By this way of modular organization of functions you will always make sure that the definition of your function is just at one particular place which you can easily find by using module names that are clear about the type of functionality implemented in the functions of the respective module. We will cover this topic later in the lecture and in the exercises. Modular programming improves maintainability of code.

- Never, NEVER, NEVER use FORTRAN-style *GOTO*-statements! Although all varieties of spaghetti are a culinaric temptation (albeit not really considered sophisticated food), spaghetti-like FORTRAN style with *GOTO* statements is to be avoided.

- Always use variables with a speaking name, i.e. use names that explain the specific meaning[1] – the same goes for function names. For example, don't use *v1=1.0* when you can use *speedOfLight=1.0*.

- Try to find some cases where an analytical solution exists or where simple test cases can be applied. If possible, devise different algorithms for the same problem. If you get the same answers, you may have coded things correctly (or made the same error twice).

- Avoid pointer-to-pointer-to...-pointer constructs such as ****particleArray[i][j][k]*, particularly inside loops.

- Avoid global variables whenever possible! Global variables make code harder to understand and to maintain, because, when a variable is global, it can be accessed and changed by any function anywhere in the program.

- Try to use the standard ANSI or C99 version of C. Avoid local dialects if you wish to port your code to other machines[2].

- Always add comment lines describing what your code does. This will help you to understand what you did, e.g. many months later.

- The naming conventions for variables and functions in this course (and in general) are as follows:

  - Function names always start with a lower-case letter which is followed by lower- and upper-case letters or underscores "_" to enhance readability. For example, use *print_all_local_variables();* or *printAllLocalVariables();* The latter version is preferred because it can be typed much faster.

  - Variable names always start with a lower-case letter which is followed by lower- and upper-case letters to enhance readability, e.g. *speed_of_light* or *speedOfLight*.

[1]There is this bad habit in e.g. FORTRAN code to use variables like *x1*, *x2*, etc., or for functions which go like *routine1*, *routine2*, etc. In FORTRAN, variable names used to be restricted to 5 or 6 characters. This is why old FORTRAN programs from the 70s or 80s (which are often still used today!) are painful to read because of the use of such short and cryptic variable and function names.

[2]Use the option *-ansi* for C99 or *-std=c++98* to ensure this.

– Preprocessor variables (#defines) are always written in capital letters, e.g.

```
#define MAX_ARRAY_LENGTH 30
```

Here, underscores are merely used to increase readability.

3 Basic Data Types in C

Basic data types in C are the following:

Data Type	**Description**
short	used for (2 byte) integer value (is the same as *signed short*).
int	used for (4 byte) integer value (is the same as *signed int*).
unsigned short	used for (4 byte) integer value (is the same as *signed int*).
unsigned int	positive (4 byte) number.
float	used for (4 byte) floating point value with single precision.
double	used for (8 byte) floating point value with double precision.
char	used to display printable characters from the ASCII table, nowadays almost only Unicode.

The sample program listing below shows the declaration and initialization of basic C data types and some simple operations done with them.

```c
/* Sample code using basic data types */

#include <stdio.h>

int main(void) {

  /* Initiailizations */
  int ival                        = -15;
  unsigned intUnsigned            = 456;
  short myShort                   = 34;
  double myDouble                 = 45.12345678901234567890l2345;
  float  myFloat                  = 45.12345678901234567890l2345;
  char thisIsAcharField     [4]  = "Hello";
  char thisIsACorrectString[6]  = "Hello";

  /* Decision using logical OR operator */
  if( (ival) || (intUnsigned)) {
      printf("\n At least ONE number is not equal zero: %i %d\n",ival, intUnsigned);
  }

  ival = 0;

  if ( (intUnsigned) && (ival) )
       printf("Is this printed?\n");

  printf(" myShort = %i\n", myShort);
  printf("\t Wrong string = \"%s\"   \n\t Correct string = \
          \"%s\"\n\t",thisIsAcharField, thisIsACorrectString);
  printf(" Double precision (16 valid digits!) Double: %.30f\n",myDouble);
  printf("\t Single precision (8  valid digits!) Float : %.30f\n\n",myFloat);
  return 0;
}
```

4 Operators for Logical Expressions (Decisions)

The following table provides the mathematical operators in C that can be used in control structures (see below) for decisions based on logical expressions:

Operator	Meaning	Example	Return value
<	smaller than	a < b	1 if a is smaller than b, 0 else
<=	smaller or equal	a <= b	1 if a is smaller or equal to b, 0 else
>	larger than	a > b	1 if a is larger than b, 0 else
>=	larger or equal	a >= b	1 if a is larger or equal to b, 0 else
==	smaller than	a == b	1 if a equals b, 0 else
!=	smaller than	a != b	1 if a is not equal b, 0 else

5 Streams – Input/Output Functionality

When developing source code for MD you need to be able to read and write data not only from and to the screen (called *standard output* or *standard error output*), but also from and to files on the file system. The input/output of data in C is realized by *data streams*. When opening a file, a new stream is created and removed again when closing the file. The single streams are administered by the operating system. In C there are *text streams*, *binary streams* and *standard streams*.

5.1 Text Streams

A text stream reads and writes single characters of a text. Usually the text is separated in single lines. The internal representation of text is independent of the operating system, on which the program is executed. Text streams can use all printable ASCII-characters and several control codes, e.g. new line ‘\n‘ or tab ‘\t‘. On Windows systems, the end of line is denoted with the control sign \r\n, but UNIX/Linux systems use only \n for that. Thus, here, the compiler does an automatic conversion. The end of a text is usually denoted by ^Z (ASCII-Code 26), which can also be sent to the screen by Strg+Z or Strg+D under UNIX/Linux.

5.2 Binary Streams

With binary streams the content is processed byte by byte and not character by character. Thus, data which have been processed in a binary stream are available for reading in exactly the same way. No automatic conversions are done by the compiler.

5.3 Standard Streams

Three streams, the three standard streams, are always available in any C-program. Standard streams are pointers to a FILE object. These are the standard streams:

- **stdin** The standard input, which is usually connected with the keyboard.
- **stdout** The standard output is connected with the screen. The output is buffered linewise.
- **stderr** The standard error output is connected with the screen, just like stdout, but the output is not buffered.

When a file is opened in C with the command *fopen()*, a memory object of type *FILE* is generated and initialized. A successfully opened file returns a pointer to a *FILE* memory object which is connected with the stream. Whether a binary or text stream is used can be determined with an additional option in the function *fopen()*. The FILE object is a structure which is declared in the header file *<stdio.h>*. Using the returned *FILE* pointer one can read or change data of the stream using the standard function *printf()*. The prototype of the function *fopen()* is the following:

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

fopen() opens a file with name filename. If a file could not be opened, this function returns the *NULL* pointer. The second argument mode determines the access mode, which has to be one of the strings of the following table:

Mode	**Meaning**
"r"	Opens a file for reading.
"w"	Opens a file for writing.
"a"	Like "w" but appends data instead of overwriting existing data in the file.
"r+"	Open file for reading and writing.
"w+"	Like "r+", but if the file does not exist, a new file is created.
"a+"	Open file for reading and writing. If it doesn't exist, a new file is created.

5.3.1 Sample Code for File Handling (Input/Output Functionality)

The following sample code is a simple working example that shows how to use the *fopen()* function to open a text file and *fclose()* to close a text file. The code uses the *puts()* (put string) and *gets()* (get string) functions which are declared in *<string.h>*. I recommend to you copying this code by hand, and trying to get it compiled and running! It also teaches you several concepts of good programming practice, for example modular organisation of code, avoiding of magic numbers and file name conventions.

The program asks you whether you want to read or write a file named "test.tx" First chose option (2) and write as input several lines of text. When you input a single period as only input of a line this is recognized at the last line. Then the program writes the input lines into a file"test.txt" which you can now read using an ordinary text editor or by restarting the program and choosing option (1).

```c
/* Sample code for file handling */
#include <stdio.h>
#include <string.h>

#define DATNAME "test.txt"
#define MAX_LINE_LENGTH 160

void write(void)
{
  FILE *fhd;
  char s[MAX_LINE_LENGTH];

  fhd=fopen(DATNAME,"w");
  if(!fhd)
    {
      printf("File could not be generated!\n\n");
    }
  else
    {
      printf("Please write several lines and input not more than %i characters per line.\n",MAX_LINE_LENGTH);
      printf("End your input by typing only a \".\" in a line and pressing <return>\n\n");
      do
        {
          printf(">");
          gets(s);
          if(strcmp(s,"."))
            {
              fputs(s,fhd);
              fputs("\n",fhd);
            }
        } while(strcmp(s,"."));

      fclose(fhd);
      printf("\nEnd of input!\n");
    }
}

void read(void)
{
  FILE *fhd;
  char s[MAX_LINE_LENGTH];
  int x=1;

  fhd=fopen(DATNAME,"r");
  if(!fhd)
    {
      printf("File could not be opened!\n\n");
    }
  else
    {
      printf("The file has the following content\n");

      fgets(s,MAX_LINE_LENGTH,fhd);
      do
        {
          printf("%i:%s",x++,s);
          fgets(s,MAX_LINE_LENGTH,fhd);
        } while(!feof(fhd));

      fclose(fhd);
      printf("\nEnd of file!\n");
    }
}

void main(void)
{
  int input;

  printf("Do you want to (1) read or (2) write a file?\n");
  scanf("%i",&input);

  if(input==1)
    {
      read();
    }
  else
    {
      if(input==2)
        {
          write();
        }
      else
        {
          printf("\nWrong input!\n\n");
        }
    }
}
```

6 Casts

C allows explicit converting of data types using the *cast operator*. Why is that needed? Look at the following code snippet:

```
01 int value1 = 10, value2 = 3;
02 float result = value1 / value2;
03
04 printf(‘‘%f\n‘, result);     /* The printed result will be 3.000000 */
```

In line (02), an integer division is performed. Because two integers are used for the division, there are no decimal places displayed (filled with 0‘s instead of all 3s). The reason for this is, that the integer result of the division (which is 3) is implicitly assigned to a float in line (02), so when result is printed, the value 3 is printed as a decimal number with all zeros right of the comma. Here, an explicit type cast is needed which has the following syntax:
(*type*) expression

Here, first *expression* is evaluated and then converted into the datatype *type*. With respect to the above example, the needed casts would be the following:

```
01 int value1 = 10, value2 = 3;
02 float result = (float) value1 / (float) value2;
03
04 printf(‘‘%f\n‘‘, result);     /* The printed result will be 3.333333 */
```

Now, *value1* and *value2* are converted to floats in line (02). However, this cast is ONLY valid at this command line! *value1* and *value2* are still of type *int*! Also note the following common error in the usage of casts:

```
01  int val1 = 10, val2 = 3;
02
03 /* in the next line, first calculate the division, THEN do the cast */
04  float erg = (float) (val1 / val2);
05
06  /* But now the result is wrong again ! */
07 printf(‘‘%f \n‘‘,erg);          /* This prints again 3.000000 */
```

Because the calculation before the type cast was put into parentheses, it is evaluated first. Then, the result (=3) is explicitly converted into a float, which, however, is too late now; so the result is again 3.0 and not 3.333333.

7 Control Structures

Control structures are available in every high-level language because they are needed to control the flow of program in various directions depending on the result of certain decisions. Here, we introduce *four* main control structures of C, namely:

- if-else-statement
- for loops
- while loops
- do-while loops

7.1 Decisions: if-else

We first look at the if-statement as a control structure. It has the following syntax:

```
if ( expression ) {
     command1;
}
command2;
```

An example of the simple *if*-statement is demonstrated in the following piece of C-code.

```c
/* Example if-statement */
#include <stdio.h>

int main(void) {
  int ival;
  printf("Please provide an integer: ");
  scanf("%d", &ival);

  if( ival ) {
     /* This command block is executed, when \
      * the condition is true
      */
     printf("The number is not equal to zero\n");
  }
  printf("Now, I am outside if-command block\n");
  return 0;
}
```

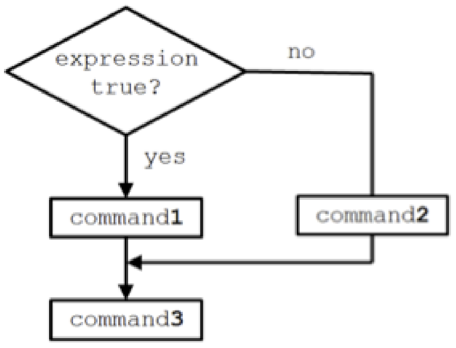

Figure 1: The simple *if*-statement in C as flow chart.

Figure 1 shows the corresponding flow chart.

7.1.1 Sample code for the if-statement

The source code above shows a sample C-source code of the simple *if*-statement along with a flow chart in Fig. 1. Note, that the expression *if (ival)* is the same as *if (ival !=0)*. In the *if*-statement, the logical condition[3] for *expression* within the brackets () is evaluated first. Depending on whether *expression* is true (!=0) the command *command1* in the command block between the curly brackets {} is executed. Note, that *command1* can also consist of many individual commands, even many function calls – it doesn‘t have to be only one single command. Then, the program is continued with *command3*. In case *expression* is not true (==0), the command block between the brackets {} is not executed, but rather the program directly continues execution with *command2*, see Fig. 1.

7.1.2 Else-branch

Normally, after one or several *if*-statements in a program there will be an optional branch of execution.

This branch will be executed if no *expression* is fulfilled. Such a branch is realized with an alternative *else*-branch with the following syntax:

```
if ( expression ) {
      command1;
}
else {
  command2;
}
command3;
```

Figure 2 displays a flow chart of the *if-else* control structure. Note, that an *else*-branch can only be used with a precedent *if*-statement.

```c
/* Example if-else-statement */
#include <stdio.h>

int main(void) {
  int ival;
  printf("Please provide an integer: ");
  scanf("%d", &ival);

  if( ival != 0) {
     /* This command block is executed, when
      * the condition is true
      */
     printf("The number is not equal to zero\n");
  }
  else {
      printf("The number is equal to zero\n");
  }
  printf("Now, I am outside if-command block\n");
  return 0;
}
```

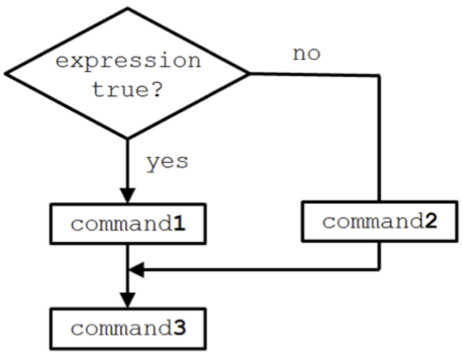

Figure 2: The *if-else*-statement in C as flow chart.

[3]The brackets () denote a logical expression here. Logical in C always means **integer**! Thus, the expression in the if-statement can have any numerical integer value; however, ONLY 0 is interpreted as false and any other value as true.

In case an *if*-statement has only one command within the command block {}, the parentheses can be left out and the single command can be written directly in the next line:

```
if ( ival != 0)
    printf(‘‘The number is not equal to zero\n‘‘);
else
   printf(‘‘The number is equal to zero\n‘‘);
printf(‘Now, the program continues...\n‘‘);
```

7.2 Loops - Repeating Program Parts: *for*-statement

When you want to repeat a group of commands, in C there are several loops, so-called iteration-commands available. We discuss first for loops which have the following syntax:

```
for ( expression1; expression2; expression3) {
        command(s);
}
```

Before entering the *for* loop, *expression1* is executed once. Normally, *expression1* is simply the initialization of the loop variable, but it could be any valid C-command! *expression2* is the logical condition which regulates the loop, i.e. it determines the exit-condition of the loop: As long as *expression2* is true, the loop is executed. Once *expression2* returns 0 (not true) the loop ends and the program is continued with the commands after the loop. As a rule, *expression3* is used for re-initializing of the loop variable (but, again, in principle, *expression3* can be any valid C- command!), see Fig. 3.

The simplest and standard usage of for loops is demonstrated in the following program:

```
/* Sample for-loop in a C-Program */
#include <stdio.h>

  int main( void ){
  int counter;
  for (counter = 1; counter <= 5; counter ++) {
  printf(‘‘%d. Number of for loops\n‘‘,counter);
  }
  return 0;
}
```

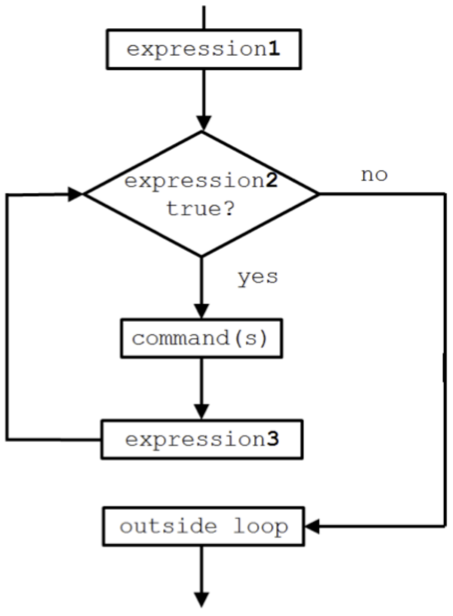

Figure 3: The for-statement in C.

8 Loops: *while*-statement with entry condition

The while loop has an entry condition expression that is checked first. As long as expression is true, the while loop is executed. The basic syntax is:

```
while ( expression ) {
command(s)
}
```

Before an execution of the loop is started, *expression* is checked. If expression is true (i.e. != 0) the commands within the while loop are executed. Once expression is not true, the program continues to execute with the *command(s)* following the while loop. Here is sample code snippet for the while loop:

```
/* Sample while-loop */
#include <stdio.h>

  int main( void ){
  int counter = 2000;

  while (counter > 1000) {
    printf(``counter = \%i``,counter--);
  }
  return 0;
}
```

9 Loops: *do-while*-statement with exit condition

The counterpart to the while loop is the do-while loop with exit condition. The basic syntax is:

```
do {
   command(s)
} while ( expression );
```

When executing a *do-while* loop, the commands are first executed. So, this loop type is guaranteed to execute the commands at least once. Then, expression is evaluated and if it is true the commands after the key word *do* are executed again. If *expression* is not true the program continues execution with the *command(s)* after the loop.

10 Preprocessor Directives

Before the compiler translates the source code into assembly, the pre-processor parses the source code and substitutes pre-processor directives which can be recognized by the symbol *#*. The preprocessor does the following:

- String literals are concatenated,
- Commentaries are removed and replaced by blanks,
- Header and source files are copied in the source code (*#include*),

- Symbolic constants are copied into the source code (*#define*),
- Conditional compilation is performed (*#ifdef*, *#elseif*, *#endif*),

The directive *#include* copies other include files into the program. As a rule, these are header files with the file ending *.h. There are two ways to use *#include*:

```
#include <header>
#include ''header''
```

The preprocessor removes the include lines and substitutes them by the corresponding source code. The compiler is then passed the modified source code for translation. The brackets <> are used, when the header file is in the standard directory *usr/include* in the file system, whereas double quotes " are used when the file is located in the current working directory.

The command *#idefine* allows to write texts (e.g. symbolic constants) which can be substituted by other texts before compilation, e.g.

```
#define PI 3.14
#define VALUE 10000
```

Besides defining symbolic constants one can use *#idefine* as parameterized macros. The following C source code shows an example for the use of parameterized macros. Parameterized macros are recognized by the parentheses which follow the macro name:

```c
/* Sample code for the use of macros */
#include <stdio.h>
#include <stdlib.h>

#define NOT_EQUAL(x, y) (x != y)
#define XCHANGE(x, y) {int j; j=x; x=y; y=j;}

int main(void){
    int integerValue1, integerValue2;

    printf("Value 1:");
    scanf("%d", &integerValue1);
    printf("Value 2:");
    scanf("%d", &integerValue2);

    if ( NOT_EQUAL(integerValue1, integerValue2)){
        XCHANGE(integerValue1,integerValue2);
    }
    printf("Value 1:  %d  |  Value2:  %d\n",integerValue1,integerValue2);
    return EXIT_SUCCESS;
}
```