

Engineering Informatics

MSE WS 2020

Manuel Wendl

March 11, 2021

Contents

I	Coding Basic C	2
1	Basic Data Types in C	3
1.1	Examples for Datatypes	3
1.2	Type-Casting	3
2	Operators	4
3	Specifiers and Input Output	4
3.1	Specifiers	4
3.2	Input Output / IO	4
3.2.1	Printf	4
3.2.2	Gets Fgets and Scanf	5
4	Control Statements	6
4.1	If-Else	6
4.2	Switch	6
4.3	While loop	7
4.4	For loop	8
4.5	Do-While loop	9
4.6	Break and Continue	9
5	Functions	9
6	Scope	10
6.1	Internal Variables	10
6.2	Global Variables	10
6.3	Static Variables	11

7	Arrays	12
7.1	Number Arrays	12
7.2	Char Arrays - Strings	12
7.3	Multidimensional Arrays	13
8	Structures	14
9	Unions	14
10	Enumerations	15
11	Pointers	16
11.1	Pointers to Structures	18
11.2	Pointers and Arrays	18
11.3	Function Pointers	19
12	Typedef	19
13	Dynamic Memory Allocation	20
14	The Preprocessor	22
15	Data Structures	23
15.1	Stack	23
15.2	Queue	24
15.3	Graph	26
15.4	Binary Tree	27
15.4.1	Implementation	27
15.4.2	Sorted Binary Tree	28
15.4.3	Treetraversals	28
15.4.4	Checking for Balance in Tree	33
16	Sorting Algorithms	34
16.1	Search Algorithms on sorted Arrays	34
16.2	Bubble Sort	34
16.3	Merge Sort	35
16.4	Quicksort	36
17	Bitwise Operators	37
II	Theory of Programming Languages	39
18	Scanner and Parser	40
18.1	Scanner	40
18.2	Parser	40
19	Syntax Trees	41

20 Control Flow Diagrams	42
21 Reading C declarations	43

Part I

Coding Basic C

1 Basic Data Types in C

Definition 1.1 (Basic Data Types in C).

There are only four main datatypes in C:

Name	Keyword C	Size	Range	Additional
character	char	1 byte	$[-128; 127]$	Always stored in form of Ascii Code
integer	int	4 bytes	$[-2, 147, 483, 648; 2, 147, 483, 647]$	1. short int : 2 bytes range: $[-32, 768; 32, 767]$ 2. long int : 8 bytes range: $[-2^{63}, 2^{63}]$
float	float	4 bytes	$[\pm 1.2 \cdot 10^{-38}; \pm 3.4 \cdot 10^{38}]$	
double	double	8 bytes	$[\pm 2.3 \cdot 10^{-308}; \pm 1.7 \cdot 10^{308}]$	

For char and int there are also the two possible to declare them '**signed**' and '**unsigned**', which means that in unsigned state, there only exist positive variables for the computer.

1.1 Examples for Datatypes

In the following Code some variables in different datatypes have been initialized.

```
1 //Characters
2 char a = 'a'; //-> a
3 char b = 97; //-> a
4 unsigned char c = 97; //-> a
5 signed char d = 97; //-> a
6 //Integers
7 int e = 42;
8 short int f = 42; //-> 42 only 2 byte
9 long int g = 42; //-> 42 in 8 byte
10 //Floats
11 float h = 2.3456; //-> 2.3456
12 //Doubles
13 double i = 2.3456; //-> 2.3456
```

1.2 Type-Casting

Type-Casting is used to change the datatype of a variable or value. Here the fraction of two integers is used as an example.

```
1 //Problem:
2 float a = 3/4; //-> 0.000
3 //Solutions:
4 float b = 3.0/4; //-> 0.750
5 float c = 3/4.0; //-> 0.750
```

```

6 float d = (float)3/4; //-> 0.0750
7 float e = 3/(float)4; //-> 0.0750

```

2 Operators

Definition 2.1 (Operators).

There are different types of operators used in C.

Assignment operator	=
Arithmetic operators	+, -, *, /, % (modulo)
Relational operators	<, <=, ==, >=, >
Logical operators	&&, , !

3 Specifiers and Input Output

3.1 Specifiers

Definition 3.1 (Specifier).

Specifiers are used for example in print or scan functions, in order to define the output or input type. Here is a list of specifiers and their purpose.

Specifier	Purpose
d	signed decimal integer
u	unsigned decimal integer
x	unsigned hexadecimal integer
f	decimal floating point
c	character
s	string

3.2 Input Output / IO

The standard library `<stdio.h>` provides several functions used to print or scan data. In the following chapters these functions are explained.

3.2.1 Printf

This function can print text and also different data from variables. For printing variables values 'printf' needs a format specifier listed in the table of the previous chapter. In the following code there are listed several print statements and their outputs. Important is, that before we can run the print statements it is necessary to import the standard library. The import is always done right at the beginning of the code. It will also be shown in the code example.

```

1 #include <stdio.h>
2
3 int main(){

```

```

4   int a = -6;
5   char b = 120;
6   float c = 3;
7   double d = 5.36;
8
9   printf("Value of a: %d \n",a); //-> Value of a: -6
10  printf("Value of a: %u \n",a); //-> Value of a: 4294967290
11  printf("Value of a: %x \n",a); //-> Value of a: ffffffff
12  printf("Value of b: %c \n",b); //-> Value of b: x
13  printf("Value of b: %d \n",b); //-> Value of b: 120
14  printf("Value of c: %f \n",c); //-> Value of c: 3.000000
15  printf("Value of c: %d \n",c); //-> Value of c: 73896
16  printf("Value of d: %f \n",d); //-> Value of d: 5.360000
17  printf("Value of a: %d and b: %c \n",a,b); //-> Value of a: -6 and b: x
18 }

```

3.2.2 Gets Fgets and Scanf

There exist three functions to read the input typed into the console by the user. In case the char array in which we want to store the input has a smaller length than the input, the functions behave differently.

gets()	fgets()	scanf()
It doesn't control whether the input is too long for the array, such that there will be returned an error. It reads the input until newline or end of file.	This function does the same, except that we additionally give a length of the input that should be stored.	This function also reads the input and only stores a given number of characters in the chararray, however if the input is longer it remains in the memory and will get scanned in the next scan first. Therefore there has to be implemented a flush function.

The following code shows the different implementations of the functions. Char Arrays will be discussed in 7.2 on p. 12.

```

1  #include <stdio.h>
2
3  int main(){
4      int size = 10;
5      // gets()
6      char buffer1[size];
7      gets(buffer1);
8      printf("Scanned input: %s\n",buffer1);
9      // fgets()
10     fflush(stdin);
11     char buffer2[size];
12     fgets(buffer2,size,stdin);
13     printf("Scanned input: %s\n",buffer2);
14     // scanf()
15     fflush(stdin);

```

```

16     char buffer3[size];
17     scanf("%9s",buffer3);
18     printf("Scanned input: %s\n",buffer3);
19 }

```

4 Control Statements

4.1 If-Else

Definition 4.1 (If-Else Statements).

The if and else statements run some statement if some expression (usually consisting out of boolean logical operations) is satisfied.

The following code shows the pseudo code and an example.

```

1  #include <stdio.h>
2
3  int main(){
4      // Pseudo Code
5      if(expression1){
6          statement1;
7      }else if(expression2){
8          statement2;
9      }else{
10         stetement3;
11     }
12     // Example
13     int a = 3;
14     int b = 5;
15
16     if(a==b){
17         printf("a equals b");
18     }else if(a<b){
19         printf("a smaller than b");
20     }else{
21         printf("a greater than b");
22     }
23 }

```

Even though this is a very intuitive way to implement such conditions, the run time of the program increases with the number of conditions, as all conditions have to be checked after each other. Therefor there was implemented another control statement, decribed in the following section.

4.2 Switch

Definition 4.2 (Switch Statements).

The switch statement is made out of an expression (usually consisting out of logic operators) and different cases. To each case there is a given a result of the expression followed by the statements. The else case is called default however is different as it is executed independent of the other cases, such that if we only want to have a special case, we have to determine the switch with **break**.

The following code shows the pseudo code and an example.

```

1  # include <stdio.h>
2
3  int main(){
4  // Pseudo Code
5      switch (expression) {
6          case case_expression1: statement1; //default will also be executed
7          case case_expression2: statement2;
8          break; //default will not be executed because of break
9          default: statement3;
10     }
11 // Example
12     int a = 5;
13     int b = 6;
14
15     switch (a==b) {
16         case 1: printf("a equals b");
17             break;
18         default:
19             switch (a<b) {
20                 case 1: printf("a smaller than b");
21                 case 0: printf("a greater than b");
22             }
23     }
24 }
```

4.3 While loop

Definition 4.3 (While loop).

The while loop repeats a set of statements as long as a given expression is true. This statement again consists out of logic operators.

The following code shows the pseudo code and an example. The example here is the mathematical operation of the n^{th} partial sum of $\frac{1}{i}$. Defined as

$$\sum_{i=1}^n \frac{1}{i}$$

```

1 #include <stdio.h>
2
3 int main(){
4 // Pseudo Code
5 while (expression){
6     statements;
7 }
8 // Example
9 float sum = 0;
10 int i = 1;
11 int n = 10;
12 while (i<=n){
13     sum = sum + 1/(float)i;
14     i++;
15 }
16 printf("n-th partial sum = %f",sum);
17 }

```

4.4 For loop

Definition 4.4 (For loop).

The statement of the for loop also gets executed as long as the condition is true. Additionally we can initialize directly in the for loop head a loopcounter and condition, such that is an optimal solution for loops with known numbers of iterations.

The following code shows the pseudo code and an example.
It is again the same example as in 4.3 on p. 7.

```

1 #include <stdio.h>
2
3 int main(){
4 // Pseudo Code
5 for (assignment;condition;step){
6     statements;
7 }
8 // Example
9 float sum = 0;
10 int n = 10;
11 for (int i=1;i<=n;i++){
12     sum = sum + 1/(float)i;
13 }
14 printf("n-th partial sum = %f",sum);
15 }

```

4.5 Do-While loop

Definition 4.5 (Do-While loop).

The Do-While loop also runs certain statements until the termination condition jumps to false. However the difference to the other loops is, that the condition is proved after the execution of the statement.

The following code shows the pseudo code and the common use of the special loop in user input.

```
1 #include <stdio.h>
2
3 int main(){
4     // Pseudo Code
5     do {
6         statements;
7     }while (expression);
8     // Example
9     char grade;
10    do {
11        printf("Please enter your grade (number between 1 and 6):\n");
12        grade = getchar();
13        getchar();
14    }while(grade < '1' || grade > '6');
15 }
```

4.6 Break and Continue

1. Break:

As already used in 4.2 on p. 6, does the **break** statement terminate a loop or in the specific case the switch. The break statement terminates the **innermost** loop.

2. Continue:

The **continue** statement skips the statements after 'continue' in the particular loop.

5 Functions

Definition 5.1 (Functions).

The aim of using functions is to reuse code, that we otherwise would have to implement doubled, to simplify problems and to enable structuring.

A function consists of:

1. return-type
2. name of the function

3. list of provided parameters

4. function body

Different return types are:

void: does not return anything. int: returns an integer ... there can be chosen any datatype and pointer as return type.

A function is defined and called in the main function as follows.

```
1 // Defining Function
2 return_type function_name(parameters){
3     statements;
4 }
5
6 int main(){
7     // Calling Function
8     return_type x = function_name(parameters);
9 }
```

6 Scope

The scope of a variable describes for which functions a variable is visible. Basically we distinguish between two main types.

6.1 Internal Variables

Definition 6.1 (Internal Variables).

Internal variables are defined within a certain function and are only visible for the function itself.

In the following abstract of code there are initialized internal variables.

```
1 double function(int a,char b,double c){
2     double d;
3     int e;
4     ...
5 }
```

6.2 Global Variables

Definition 6.2 (Global Variables).

Global variables are defined outside of any function and are visible for any function of the file and any included file. This leads to the problem, that the naming of the variable can get confusing, as these variables names are in every function the same. This often leads to errors.

In the following abstract of code there are initialized global variables.

```
1 int e;  
2 double d;  
3  
4 int function(){  
5     ... //e and d visible  
6 }  
7  
8 int main(){  
9     ... //e and d visible  
10 }
```

6.3 Static Variables

It is important to distinguish between calling a global or internal variables as static. This has got different meanings.

Definition 6.3 (Static Variables).

1. **Static global variables:**
Restriction of the code to the current source file.
2. **Static internal variables:**
Value of variable retained between calls.
(Values of variables don't get set back to initializing value, but remain in changed value.)

The following abstract of code shows how to set a variable static.

```
1 #include <stdio.h>  
2 static int a = 2; //only accessible in this c file  
3  
4 int sum(){  
5     static int sum = 0; //value of sum stays 55  
6     for(int j=1;j<=10;j++){  
7         sum = sum + j;  
8     }  
9     return sum;  
10 }  
11  
12 int main(){  
13     printf("Sum first run: %d\n",sum());  
14     printf("Sum second run: %d\n",sum());  
15 }
```

The output:

Sum first run: 55
Sum second run: 110

7 Arrays

7.1 Number Arrays

Definition 7.1 (Arrays).

The purpose of arrays is that it is possible to store a number of variables of same type without initializing each for itself.

An array gets initialized with:

1. data type (look for nearer description in 1 on p. 3).
2. array name
3. size of array: This is the number of values that can be stored in the array.

A specific element of an array is addressed by the index. Such that there can be changed specific elements of the array. **Important:** C starts counting the indices from zero!

In the following abstract of code there is shown how to initialize an array, address an element and print the array.

```
1 #include <stdio.h>
2 //Pseudo Code
3 data_type array_name[size];
4 // Example
5 int main(){
6     int example[3] = {1, 2, 3}; // Initializing
7     example[0]=example[2]; // Setting the first element
8                             // to value of third.
9     for(int i=0;i<3;i++){
10         printf("example[%d] = %d\n",i,example[i]);
11     }
12 }
```

The output:

```
example[0] = 3
example[1] = 2
example[2] = 3
```

Also important for operations with arrays is, that there isn't existing a return type of arrays, however because an array is a pointer, which is described in the modifications of the array in another array also affects the array in the main function.

7.2 Char Arrays - Strings

Definition 7.2 (Strings).

Strings are nothing different than character arrays. However there has to be determined where the string ends, such that the last element of a string always has to be the terminating element 'NULL'. This leads to the array size of numbers of characters plus one for the 'NULL'. Whenever there are made changes on character arrays the string has to remain **null terminated**.

```

1 | #include <stdio.h>
2 |
3 | void tocapital(char array[]){ // array with variable length.
4 |     int i = 0;
5 |     while(array[i]!=0){
6 |         if(array[i]<123 && array[i]>96) { // Range of small letters
7 |             array[i] -= 32; // Difference capital small in Ascii
8 |         }
9 |         i++;
10 |    }
11 | }
12 |
13 | int main(){
14 |     char string[] = "Good Luck In The Exam!";
15 |     printf("Before tocapital(): %s\n",string);
16 |     tocapital(string);
17 |     printf("After tocapital(): %s\n",string);
18 | }

```

The output:

Before tocapital(): Good Luck In The Exam!

After tocapital(): GOOD LUCK IN THE EXAM!

7.3 Multidimensional Arrays

Multidimensional arrays can be created as follows.

```

1 | // Pseudo Code
2 | datatype a[x][y]; // Array size x*y with x rows and y columns
3 | // Example
4 | int a[5][20];
5 | a[0][0] = 1; // Addresses upper left element

```

Giving an array to a function as argument is done as follows.

```

1 | int a[5][20];
2 |
3 | void func(int a[5][20]){
4 |     ...
5 | }
6 | void func(int a[][20]){
7 |     ...
8 | }
9 | void func(int (*a)[20]){
10 |    ...
11 | }

```

8 Structures

Definition 8.1 (Structs).

For storing variables of multiple data types we use structs.

1. For declaring structs there is used the keyword 'struct'.
2. Name of the declared structure.
3. Variables that are contained by the struct.

In the following abstract of code there is shown how to initialize and define structs. There are two different ways of doing this.

```
1 #include <stdio.h>
2
3 struct example1{
4     int a;
5     double b;
6 }; // Initializing in main
7
8 struct example2 {
9     int c;
10    double d;
11 }; // Directly Initializing all values = zero
12
13 int main() {
14     struct example1 x = {10, 13.9};
15     printf("struct x: %d and %f\n",x.a,x.b);
16     printf("struct e before: %d and %f\n",e.c,e.d);
17     e.c = 10;
18     e.d = 13.9;
19     printf("struct e after: %d and %f\n",e.c,e.d);
20 }
```

The output:

struct x: 10 and 13.900000

struct e before: 0 and 0.000000

struct e after: 10 and 13.900000

A variable of a certain struct is addressed as in code line 17 and 18.

9 Unions

Definition 9.1 (Unions).

A Unions purpose is to store a variable in different kinds of datatypes. However only provides space for the largest member. Unions are defined with the keyword 'union' and can be treated analogous to structs.

In the following Code there is shown how to initialize and access a union.

```
1 // Initializing
2 union union_name{
3     declarations;
4 } unions ;
5
6 // Access in main
7 int main(){
8     unions.declarations = value;
9 }
```

An additional example will be given in the next section 10 on p. 15.

10 Enumerations

Definition 10.1 (Enumerations).

Enumerations enable us to assign a name to a constant. For declaration we use the keyword 'enum'. It is possible to define own values for a name, however if we don't enums always start with zero and every additional name gets assigned previous constant+1. It is also possible to store more names assigned to the same constant.

In the following abstract of code the initializing process and access is shown.

```
1 // Initializing
2 enum enum_name{
3     name = constant;
4     name = consnatant;
5     ...
6 } enums ;
7
8 // Accessing in main
9 int main(){
10     datatype_constant x = name;
11 }
```

The following abstract of code deals with the problem of having a variable of different types, we want to print out. Therefore the classic printf() function lacks, such that there is a possible workaround consisting out of a struct, union and enum. For detailed infos on structs and enums have a look at 8 on p. 14 and 9 on p. 14.

```
1 #include <stdio.h>
2
3 struct variable{
4     enum type{
5         INT, //-> 1
6         CHAR, //-> 2
7         FLOAT //-> 3
8     }t;
```

```

9     union value{
10         int a;
11         char b;
12         float c;
13     }v;
14 };
15
16 void different_type_printer(struct variable x){
17     switch(x.t){
18         case INT:
19             printf("int = %d\n",x.v.a);
20             break;
21         case CHAR:
22             printf("char = %c\n",x.v.b);
23             break;
24         case FLOAT:
25             printf("float = %f\n",x.v.c);
26             break;
27     }
28 }
29
30 int main(){
31     struct variable var1;
32     var1.v.a = 123;
33     var1.t = INT;
34     different_type_printer(var1);
35
36     var1.v.b = 'a';
37     var1.t = CHAR;
38     different_type_printer(var1);
39
40     var1.v.c = 123.456;
41     var1.t = FLOAT;
42     different_type_printer(var1);
43 }

```

The output:

int = 123

char = a

float = 123.45600

11 Pointers

Definition 11.1 (Pointers).

A pointer is a variable that contains the address of another variable.

In the following abstract of code it is shown how to declare a pointer of a specific datatype or an arbitrary datatype. This is done first in pseudo code and an example.

```

1 // Pseudo Code
2   data_type *pointer;
3 // Example
4   // Declare
5   int *int_pointer;
6   void *pointer;
7   // Initialize
8   int x = 31;
9   char y = 'a';
10  int_pointer = &x;
11  pointer = &y;
12  // Read Pointers
13  x = *int_pointer;
14  y = *pointer;

```

'&' is the address operator, that gives the address of a certain variable. '*' accesses the value the pointer points to.

The advantage of using pointers is, that not the value of a variable, but the address can be given to functions. This is the reason why it is possible to change the variables also in other functions without being global.

The following code shows the application of a function that can swap values. Therefore it is useful and necessary to use pointers.

```

1 #include <stdio.h>
2
3 void swap(int *a,int *b){
4     int temp = *a;
5     *a = *b;
6     *b = temp;
7 }
8
9 int main(){
10     int a = 10;
11     int b = 20;
12     printf("value of a: %d and b: %d\n",a,b);
13
14     swap(&a,&b);
15     printf("value of a: %d and b: %d\n",a,b);
16 }

```

The output:

value of a: 10 and b: 20
value of a: 20 and b: 10

11.1 Pointers to Structures

We can also declare a pointer to a structure or a pointer to an element of struct. This is analogous to normal pointers.

The following abstract of code shows how to declare and initialize a pointer to a struct.

```
1 struct example{
2     int a;
3     int b;
4 }e;
5 // Pointer to a struct
6 struct example *p1 = &e;
7 // Pointer to an element of a struct
8 int *p2 = &e.a;
9
10 // Access a member of a struct
11 ea = (*p1).a;
12 ea = p1->a;
```

11.2 Pointers and Arrays

Pointers and Arrays are considered by many programmers as the same, however there exist several differences.

Array	Pointer
Not a variable	Is a variable
Holds data	Holds an address
Data is accessed directly	Data is accessed indirectly
Address immutable (unchangeable)	Address variable
Space implicitly allocated and dellocated	Space for data must be reserved
Size of data is known	Size of data usually unknown

On compiler level array are treated as pointers in those three cases.

1. Arrays used within an expression are treated as a pointer to the first element of the array.
2. '[x]' displays the offset from a pointer.
3. Arrays used in declaration of a function parameter are treated as a pointer of the first element of an array.

In the following abstract of code there are listed examples for the three cases above.

```
1 // Example for 1)
2 int a[5] = {1,2,3,4,5};
3 int *p = 0;
4 p = a;
5 // Example for 2)
6 char a[5] = {1,2,3,4,5};
7 x = a[2];
8 x = *(a+2); // Equal to statement above
```

```

9 // Example for 3)
10 void equal(int str[]){
11     ...
12 }
13 void equal(int *str){
14     ...
15 }

```

11.3 Function Pointers

Definition 11.2 (Function Pointers).

There can also be created pointers to functions, such that the *f_pointer* contains the address of a certain function.

In the following abstract of code the implementation of a function pointer will be shown.

```

1 #include <stdio.h>
2
3 // Pseudo Code
4 return_type *variable_name(parameters){
5     ...
6 }
7 // Example
8 int *f_pointer(int *a, int *b){
9     int x = *a + *b;
10    int *ptr = &x;
11    return ptr;
12 }
13 int main(){
14     int a = 1;
15     int b = 2;
16     printf("%d", *f_pointer(&a, &b));
17 }

```

12 Typedef

Definition 12.1 (Typedef).

As C declarations may become complex, there is the possibility to simplify code with typedef. It enables us to give custom names to data types. Variables which are initialized based on the name have the same properties as with declaration by type.

In the following code there is shown how to use typedef.

```

1 #include <stdio.h>
2 #include <stdlib.h>

```

```

3 // Pseudo Code
4 typedef type newname;
5 int func{
6     ...
7     newname x = value;
8     ...
9 };
10 // Example
11 typedef unsigned int uint;
12
13 typedef struct{ // typedef of a struct
14     uint Matr_num;
15     uint Semester;
16 } student;
17
18 int main(){
19     student s;
20     s.Matr_num = 14561;
21     s.Semester = 1;
22 };

```

13 Dynamic Memory Allocation

In programming there occurs often the problem, that the amount of needed memory prior runtime is not known. Therefore we use dynamic memory allocation.

Definition 13.1 (Dynamic Memory Allocation).

Dynamic memory allocation allows to allocate additional memory during runtime.

For a better understanding, there can be said that usual variables are stored from top to bottom such that they have limited memory (stack). Dynamical Memory is stored from bottom to top in the so called 'heap'.

C offers in the standard library functions to allocate and free dynamic memory.

1. Allocating memory:

We additionally have to give the allocating function the size of our type we want to store (in case of `calloc`: and the number of elements). The size of our type we want to store can be difficult to determine, as every compiler or program can handle memory sizes differently. Therefore we use the function `sizeof()`.

(a) `malloc()`:

`Malloc` doesn't initialize the content of the memory, such that it will throw a segmentation fault if it is accessed before initializing.

(b) `calloc()`:

Initializes all content of the memory to zero. If we access memory from `calloc` it will return 0, if it was not overwritten.

2. Free memory:

Because of the manually allocated memory, this memory will be kept forever, as the compiler doesn't know when the memory gets redundant. Because of the so called leaking memory, if a function only allocates memory but doesn't reset the used memory, there has to be used the free method, which frees/de-allocates the memory. This function is called by 'free()'.

In the following code there will be shown how to dynamically allocate memory and free it again. Additionally there is shown an example application of dynamic memory allocation, as it is dependent on the user input whether it is necessary to allocate memory.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 // Pseudo Code
4 //void* malloc(size_t size);
5 //void* calloc(size_t num, size_t size);
6 //free(memory_name);
7 // Example
8 typedef struct{
9     char matr_nr[10];
10    int semester;
11 }student;
12
13 int main(){
14     printf("Enter one if you want to insert a student\n");
15     int x = getchar();
16     if(x=='1'){
17         student *new = calloc(sizeof(student),1);
18         printf("Please enter the matr_nr.\n");
19         char *matr_num = malloc(10*sizeof(int));
20         fflush(stdin);
21         scanf("%9s",matr_num);
22         matr_num[9]=0; // have to initialize determining element manually as malloc
                        // doesn't initialize to zero.
23         for(int i=0;i<10;i++) {
24             new->matr_nr[i] = matr_num[i];
25         }
26         free(matr_num); // Memory of help array can be freed
27         printf("Please enter semester.\n");
28         fflush(stdin);
29         char s = getchar();
30         new->semester = s;
31         printf("Inserted student: matr_nr: %s semester: %c\n",new->matr_nr,new->semester
32                );
33         free(new->matr_nr); // freed after print if you want to keep then don't free!
34     }else{
35         printf("You didn't want to insert a student\n");
36     }
```

14 The Preprocessor

Definition 14.1 (The Processor).

C provides a **preprocessor**, which is invoked during compilation. It is possible to give commands to the preprocessor in form of **directives**, which always start with '#'. The three most important uses are:

1. Inclusion of files
2. Macro Substitutes
3. Conditional Inclusions

Following there will be given examples for the listed applications.

1. Inclusion of files:

Enables use of external declarations. Mostly inserted at the beginning of the file.

There are two different notations for file inclusion:

- (a) `< ... >`:
This searches for the filename in by the IDE/Compiler predesignated directories. Is normally used to include standard libraries.
- (b) `" ... "`:
This searches for the file in the same directory and if not found follows the search path of '`<...>`'.

A code example:

```
1 #include <filename>
2 #include "filename"
```

2. Macro Substitution:

It allows to define macros (names, that replace a piece of code), that will be replaced at compile time.

In the following abstract of code an example application is given.

```
1 #include <stdio.h>
2 #define multiply(x,y) (x)*(y)
3 #define PI 3.14159265
4 #define circlearea(x) (PI*x*x)
5
6 int main(){
7     int a = 12;
8     int b = 10;
9     printf("%d multiplied with %d = %d\n",a,b,multiply(a,b));
10    float rad = 3;
11    printf("For circle with radius %f this is the area: %f\n",rad,circlearea(rad));
12 }
```


3. Conditional Inclusion:

It controls the processing with several statements, which can be used in the preprocessor.

```
1 // Most popular Conditional operators
2 #if condition
3 #define macro
4 #else
5 #elif condition
6 #endif
7 #ifdef macro
8 #ifndef macro
```

15 Data Structures

15.1 Stack

Definition 15.1 (Stack).

A stack is a data structure with the **Last in First out (LIFO)** principle. There exist the following operations on a classical stack structure.

1. **Push:** Add element on top of stack
2. **Pop:** Remove element on top of stack
3. **Top:** View element on top of stack

The following Code shows how to implement a stack and the basic operations listet above.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 struct Stack{
5     struct Element *top;
6 };
7
8 struct Element{
9     int data;
10    struct Element *prev;
11 };
12
13 struct Stack *init_stack(){
14     struct Stack *newstack = calloc(sizeof(newstack),1);
15     newstack->top = NULL;
16     return newstack;
17 }
18
19 void push(struct Stack *stack, int data){
20     struct Element *element = calloc(sizeof(element),1);
21     element->data = data;
```

```

22     element->prev=stack->top;
23     stack->top=element;
24 }
25
26 void pop(struct Stack *stack){
27     if(stack->top!=NULL){
28         struct Element *temp = stack->top->prev;
29         printf("Popping Element with data: %d \n", stack->top->data);
30         free(stack->top);
31         stack->top = temp;
32     }
33 }
34
35 void top(struct Stack *stack){
36     if(stack->top!=NULL){
37         printf("Top Element has data: %d \n", stack->top->data);
38     }else{
39         printf("Stack empty. \n");
40     }
41 }
42
43 int main(){
44     struct Stack *stack = init_stack();
45     push(stack,5);
46     push(stack,10);
47     top(stack);
48     pop(stack);
49     top(stack);
50     pop(stack);
51     top(stack);
52 }

```

The output:

```

Top Element has data: 10
Popping Element with data: 10
Top Element has data: 5
Popping Element with data: 5
Stack empty.

```

15.2 Queue

Definition 15.2 (Stack).

A queue is a data structure with the **First in First out (FIFO)** principle. A queue structure consists the following operations.

1. **Enqueue:** Add element to the end of the queue

2. **Dequeue:** Remove element from the front of the queue
3. **Printqueue:** View all elements of queue

The following Code shows how to implement a queue and the basic operations listed above.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Queue{
5      struct Element *head;
6      struct Element *tail;
7  };
8
9  struct Element{
10     int data;
11     struct Element *next;
12 };
13
14 struct Queue *init_queue(){
15     struct Queue *queue = calloc(sizeof(queue),1);
16     queue->head=NULL;
17     queue->tail=NULL;
18     return queue;
19 }
20
21 void enqueue(struct Queue *queue, int data){
22     struct Element *element = calloc(sizeof(element),1);
23     element->data=data;
24
25     if(queue->head==NULL){
26         queue->head=element;
27         queue->tail=element;
28     }else{
29         queue->tail->next = element;
30         queue->tail = element;
31     }
32 }
33
34 void dequeue(struct Queue *queue){
35     if(queue->head!=NULL){
36         struct Element *temp = queue->head;
37         free(queue->head);
38         queue->head = queue->head->next;
39         printf("Dequeuing Element with data: %d \n", temp->data);
40     }else{
41         printf("Queue empty. \n");
42     }
43 }
44
45 void printqueue(struct Queue *queue){
```

```

46     struct Element *current = queue->head;
47     if(current==0){
48         printf("Queue empty. \n");
49     }else {
50         while (current != NULL) {
51             printf("Element with data: %d \t", current->data);
52             current = current->next;
53         }
54         printf("\n");
55     }
56 }
57
58 int main(){
59     struct Queue *queue = init_queue();
60     enqueue(queue,5);
61     enqueue(queue, 10);
62     enqueue(queue,15);
63     printqueue(queue);
64     dequeue(queue);
65     printqueue(queue);
66     dequeue(queue);
67     dequeue(queue);
68     printqueue(queue);
69 }

```

The output:

Element with data: 5 Element with data: 10 Element with data: 15
 Dequeueing Element with data: 5
 Element with data: 10 Element with data: 15
 Dequeueing Element with data: 10
 Dequeueing Element with data: 15
 Queue empty.

15.3 Graph

Definition 15.3 (Graph).

It is a data structure consisting of vertices and edges, that connect those vertices.

1. **Directed Graph:** With directed edges (acyclic if no circles)
2. **Tree:** Exactly one path between every pair of vertices.

A **tree** consists of a:

1. **root:** vertex without incoming edges
2. **leaf:** vertex without outgoing edges
3. **parent:** vertex of incoming edge

4. **children:** vertices at outgoing edges
5. **ancestors:** all vertices from root to the vertex
6. **descendants:** all vertices that are traversed until reaching all leafs of vertex.
7. **siblings:** all siblings have same parent node
8. **neighbours:** all vertices with the same vertex depth.

Additional terminology:

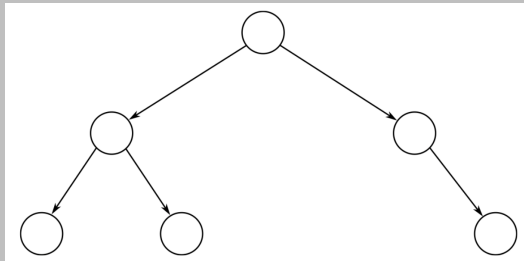
1. **tree height:** length of longest part
2. **vertex depth:** length of path from root to vertex
3. **subtree:** tree formed by vertex and all its descendants
4. **balanced tree:** height of all children's subtrees differ no more than one for all vertices.
5. **binary tree:** Every vertex has maximal two outgoing edges.

15.4 Binary Tree

15.4.1 Implementation

Definition 15.4 (Binary Tree).

The binary tree is a special type of graph with maximal two outgoing edges from every vertex.



In the following Code abstract there is shown how a tree is implemented in C.

```

1 struct Tree {
2     struct Node *root;
3 };
4
5 struct Node {
6     int data;
7     struct Node *left_child;
8     struct Node *right_child;
9 };
  
```

15.4.2 Sorted Binary Tree

These tree structures are often used to perform sorting. In the next abstract of code the implementation of a sorted binary tree is shown. (Ordered Output in Inorder Depth first search).

```
13 struct Node *init_node(int data) {
14     struct Node *new = calloc(sizeof(struct Node), 1);
15     new->data = data;
16     return new;
17 }
18
19 struct Tree *init_tree(int data) {
20     struct Tree *tree = calloc(sizeof(struct Tree), 1);
21     tree->root = init_node(data);
22     return tree;
23 }
24
25 void add_sorted_node(struct Node *current, int data) {
26     if (data < current->data) {
27         if (current->left_child == NULL) {
28             current->left_child = init_node(data);
29         } else {
30             add_sorted_node(current->left_child, data);
31         }
32     } else if (data > current->data) {
33         if (current->right_child == NULL) {
34             current->right_child = init_node(data);
35         } else {
36             add_sorted_node(current->right_child, data);
37         }
38     }
39 }
```

15.4.3 Treetraversals

1. Depth First Search:

Traversing through tree from root to leave before backtracking.

- (a) Inorder:
Left child subtree, current vertex, right child subtree.
- (b) Preorder:
Current vertex, left child subtree, right child subtree.
- (c) Postorder:
Left child subtree, right child subtree, current vertex.

In case of reversed order, the right child subtree is traversed before the left.

The following abstract of code contains the Depth first search implementation of inorder, preorder, postorder.

```

40 void DFS_inorder(struct Node *root) {
41     if (root == NULL) {
42         return;
43     }
44     DFS_inorder(root->left_child);
45     printf("%d \t", root->data);
46     DFS_inorder(root->right_child);
47 }
48
49 void DFS_preorder(struct Node *root) {
50     if (root == NULL) {
51         return;
52     }
53     printf("%d \t", root->data);
54     DFS_preorder(root->left_child);
55     DFS_preorder(root->right_child);
56 }
57
58 void DFS_postorder(struct Node *root) {
59     if (root == NULL) {
60         return;
61     }
62     DFS_postorder(root->left_child);
63     DFS_postorder(root->right_child);
64     printf("%d \t", root->data);
65 }

```

This first implementation was just recursive. In the following there will be used the data structure of the stack.

Again a sorted binary tree was implemented first. However in this example only the inorder Traversal is coded.

For the preorder an postorder, the statements in the DFSalg(...) function have to be switched.

```

1 // Implementation of DFS as Stack
2 struct Stack{
3     struct Element *top;
4 };
5
6 struct Element{
7     struct Node *node;
8     struct Element *prev;
9 };
10
11 void push(struct Stack *stack, struct Node *node){
12     struct Element *element = calloc(sizeof(element),1);
13     element->node = node;
14     element->prev=stack->top;
15     stack->top=element;
16 }
17

```

```

18 void pop(struct Stack *stack){
19     if(stack->top!=NULL){
20         struct Element *temp = stack->top->prev;
21         printf("Popping Element with data: %d \n", stack->top->node->data);
22         free(stack->top);
23         stack->top = temp;
24     }
25 }
26
27 void DFSalg(struct Node *current, struct Stack *stack){
28     if(current->left_child!=NULL) {
29         push(stack, current->left_child);
30         DFSalg(current->left_child,stack);
31     }
32     pop(stack);
33     if(current->right_child) {
34         push(stack, current->right_child);
35         DFSalg(current->right_child, stack);
36     }
37 }
38
39 void DFSframe(struct Tree *tree){
40     struct Stack *stack = calloc(sizeof(stack),1);
41     push(stack,tree->root);
42     DFSalg(tree->root, stack);
43 }
44
45 int main() {
46     struct Tree *tree = init_tree(5);
47     add_sorted_node(tree->root, 10);
48     add_sorted_node(tree->root, 0);
49     add_sorted_node(tree->root, 8);
50     add_sorted_node(tree->root, 12);
51     add_sorted_node(tree->root, 7);
52     add_sorted_node(tree->root, 9);
53     DFSframe(tree);
54 }

```

The output:

```

Popping Element with data: 0
Popping Element with data: 5
Popping Element with data: 7
Popping Element with data: 8
Popping Element with data: 9
Popping Element with data: 10
Popping Element with data: 12

```


Depth first search algorithms in sorted binary Trees

It is possible to use the order in the tree elements to find a specific value in the graph. Therefore we distinguish in three different cases. First it has the same data as the current vertex it returns true. Otherwise it is given to either the left or right subtree depending on whether the search value is less or greater than the vertex.

The abstract of code listed below contains two types of functions, the first one only checks if the value is contained in the tree. The second also inserts a new tree node if the value doesn't exist yet.

```
66 int Binary_search(struct Node *root, int search) {
67     if (root == NULL) {
68         return 0;
69     } else if (root->data == search)
70         return 1;
71     else if (search < root->data)
72         return Binary_search(root->left_child, search);
73     else
74         return Binary_search(root->right_child, search);
75 }
76
77 int insert_search(struct Node *root, int search) {
78     if (search < root->data) {
79         if (root->left_child)
80             return insert_search(root->left_child, search);
81         else {
82             struct Node *new_child = init_node(search);
83             root->left_child = new_child;
84             return 0;
85         }
86     }
87     else if (search > root->data) {
88         if (root->right_child)
89             return insert_search(root->right_child, search);
90         else {
91             struct Node *new_child = init_node(search);
92             root->right_child = new_child;
93             return 0;
94         }
95     }
96     else
97         return 1;
98 }
```

2. Breadth First Search:

Traversing the tree in layers, such that it explores all of the neighbours before moving in depth. In this implementation the queue structure was used. The implementation of the tree is the same as in 15.4.1. The queue implementation was adapted to this problem. The data is now a pointer to a tree node.

```

1  /* Implementation of BFS with Queue
2  * Pseudo Code:
3  * Enqueue root node
4  * Dequeue the head and add all children of dequeued element to the queue
5  * Repeat this until the queue is empty.
6  */
7
8  struct Queue {
9      struct Element *head;
10     struct Element *tail;
11 };
12
13 struct Element {
14     struct Node *node;
15     struct Element *next;
16 };
17
18 struct Queue *init_queue() {
19     struct Queue *queue = calloc(sizeof(queue), 1);
20     queue->head = NULL;
21     queue->tail = NULL;
22     return queue;
23 }
24
25 void enqueue(struct Queue *queue, struct Node *data) {
26     struct Element *element = calloc(sizeof(element), 1);
27     element->node = data;
28
29     if (queue->head == NULL) {
30         queue->head = element;
31         queue->tail = element;
32     } else {
33         queue->tail->next = element;
34         queue->tail = element;
35     }
36 }
37
38 void dequeue(struct Queue *queue) {
39     if (queue->head != NULL) {
40         struct Element *temp = queue->head;
41         free(queue->head);
42         queue->head = queue->head->next;
43         printf("%d \t", temp->node->data);
44     }
45 }
46
47 void BFS(struct Node *node, struct Queue *queue) {
48     dequeue(queue);
49     if (node->left_child != 0)
50         enqueue(queue, node->left_child);

```

```

51     if (node->right_child != 0)
52         enqueue(queue, node->right_child);
53     if (queue->head != 0)
54         BFS(queue->head->node, queue);
55 }
56
57 void fullBFS(struct Node *node) {
58     struct Queue *queue = init_queue();
59     struct Element *element = calloc(sizeof(element), 1);
60     element->node = node;
61     queue->head = element;
62     BFS(node, queue);
63 }

```

15.4.4 Checking for Balance in Tree

The following Code checks for balance in trees and returns the height of the tree if balanced and -1 if not balanced. This was implemented recursively.

```

1  /* Check for Balancing of Tree */
2  int balanced_Tree(struct Node *root){
3      if(root == 0)
4          return 0;
5      int leftSubtree = balanced_Tree(root->left);
6      if(leftSubtree == -1)
7          return -1;
8      int rightSubtree = balanced_Tree(root->right);
9      if(rightSubtree == -1)
10         return -1;
11     if(abs(leftSubtree-rightSubtree)>1)
12         return -1;
13
14     if(leftSubtree>rightSubtree) {
15         int max = leftSubtree;
16         return max+1;
17     }
18     else {
19         int max = rightSubtree;
20         return max+1;
21     }
22 }

```

16 Sorting Algorithms

16.1 Search Algorithms on sorted Arrays

It is easier to implement search algorithms as the Binary search on sorted data. This searching algorithm is shown in the following code.

```
1 int Binary_Search(int search, int array[], int size){
2     if(size !=0){
3         int middle = size/2;
4         if(array[middle]==search)
5             return 1;
6         else if(array[middle]>search)
7             return Binary_Search(search,array,middle);
8         else if(array[middle]<search)
9             return Binary_Search(search,array+middle+1,size-middle-1);
10        else
11            return 0;
12    }else
13        return 0;
14 }
```

This is the reason why we need the sorting algorithms explained in the next sections.

16.2 Bubble Sort

The Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Due to the nature of the Bubble sort algorithm, the largest number moves to the end of the array immediately.

Therefore you can reduce the length of the unsorted array by one with each iteration.

Consequently the worst case runtime can be reduced stays however $O(n^2)$.

```
1 void bubble_sort(int array[], int size){
2     if(size == 0)
3         return;
4     for(int i=0;i<size-1;i++){
5         if(array[i]>array[i+1]) {
6             int temp = array[i];
7             array[i]=array[i+1];
8             array[i+1]=temp;
9         }
10    }
11    bubble_sort(array,size-1);
12 }
```

16.3 Merge Sort

The Merge Sort is a divide and conquer algorithm. The program can be split into two halves.

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted)
2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining.

The following code shows the implementation in two functions `merge_sort` and the helper function `merge` which merges.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void merge(int array1[], int array2[], int size1, int size2, int array[]) {
5     int a = 0, b = 0, c = 0;
6
7     while (a != size1 && b != size2) {
8         if (array1[a] < array2[b]) {
9             array[c++] = array1[a++];
10        } else {
11            array[c++] = array2[b++];
12        }
13    }
14    while (a != size1) {
15        array[c++] = array1[a++];
16    }
17    while (b != size2) {
18        array[c++] = array2[b++];
19    }
20 }
21
22 void merge_sort(int array[], int size) {
23     if (size > 1) {
24         int size1 = size/2;
25         int size2 = size - size1;
26         int *temp_array = calloc(sizeof(int), size);
27
28         for(int i=0; i<size; i++)
29             temp_array[i] = array[i];
30
31         merge_sort(temp_array, size1);
32         merge_sort(temp_array+size1, size2);
33         merge(temp_array, temp_array+size1, size1, size2, array);
34     }
35     else
36         return;
37 }
38
39 int main(){
```

```

40     int array[5]={9, 6, 4, 7, 1};
41     merge_sort(array,5);
42     for(int i=0;i<5; i++){
43         printf("%d\t",array[i]);
44     }
45 }

```

The output is:

1 4 6 7 9

16.4 Quicksort

The Quicksort is also a divide and conquer algorithm. The program consists of the following steps:

1. Pick pivot element.
2. Reorder the array, such that all elements greater are on the right side of the pivot and all smaller ones on the left side of the pivot.
3. Apply step 1. and 2. to those subarrays.

The implementation is shown in the following abstract of code.

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void swap(int *a, int *b){
5      int tmp = *a;
6      *a = *b;
7      *b = tmp;
8  }
9
10 void quicksort(int *array, int size){
11     if(size<2){
12         return;
13     }
14     int *pivot = array;
15     int *ptr = array+1;
16     int count = 0;
17
18     while(ptr!=array+size){
19         if(*ptr<*pivot){
20             swap(ptr++,pivot++);
21             count++;
22         }else
23             ptr++;
24     }
25     quicksort(array,count);
26     quicksort(array+count,size-count);
27 }
28

```

```

29 int main(){
30     int array[5]={9, 6, 4, 7, 1};
31     quicksort(array,5);
32     for(int i=0;i<5; i++){
33         printf("%d\t",array[i]);
34     }
35 }

```

The output is:
1 4 6 7 9

17 Bitwise Operators

Definition 17.1 (Bitwise Operators).

Bitwise operators operate on bit level. There are the following main operators:

Operator	Meaning
&	bitwise AND
	bitwise OR
^	bitwise XOR
«	leftshift
»	rightshift
~	ones complement

AND			OR			XOR		
A	B	A& B	A	B	A B	A	B	A ^ B
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Left and Right Shift operators perform a left or right shift by the given number of bits. Left shifting can be seen as multiplying with 2^n and right shifting as $\frac{1}{2^n}$. This is shown in the following code example.

```

1 #include <stdio.h>
2 int main() {
3     int x = 2;
4     int y = x << 1;
5     printf("%d -> %d \n", x, y);
6     y = x << 2;
7     printf("%d -> %d \n",x, y);
8     int z = y >> 1;
9     printf("%d -> %d \n",y, z);
10 }

```

The output is:

```

2 -> 4
2 -> 8
8 -> 4

```

Some easy applications with bitwise operators.

```

1 #include <stdio.h>
2 int main(){
3     // Even or Odd numbers
4     int x = -19;
5     (x & 1) ? printf("Odd\n") : printf("Even\n");
6
7     // Numbers of power of two
8     int y = 8;
9     if((y & (y-1))==0)
10         printf("%d is a power of two\n", y);
11
12     // Swapping two numbers without temp variable
13     int a = 4, b = 1;
14     printf("%d and %d\n", a, b);
15     a^=b;
16     b^=a;
17     a^=b;
18     printf("%d and %d\n", a, b);
19 }

```

The output is:

```

Odd
8 is a power of two
4 and 1
1 and 4

```


Part II

Theory of Programming Languages

18 Scanner and Parser

18.1 Scanner

Definition 18.1 (Scanner).

The first step of the compiling process is done by the so called 'scanner'. Its tasks are:

1. Identifying:
 - (a) reserved words (keywords as: if, while, for, datatypes, ...)
 - (b) variable-, function-, structnames
 - (c) constants as: (numbers, strings ...)
2. Removing:
 - (a) white spaces (only for our visual understanding not important for compiler)
 - (b) comments

The scanner recognizes numbers and names from digits $. = 0, \dots, 9$ and letters $:= _, a, \dots, z, A, \dots, Z$.

A number is recognized by the scanner as:

$$number := -?digit\ digit\star (.digit\ digit\star)?$$

The '?' means that the element before is optional. The ' \star ' means 'repeat 0 or more times'.

A name is recognized by the scanner as:

$$name := letter(letter\ |\ digit)\star$$

The '|' is the symbol for alternation.

18.2 Parser

Definition 18.2 (Parser).

The recognized numbers and names get set together into an expression and checked for validity in the parser. The notation we will use here is called the **EBNF** - Extended Backus-Naur Form.

The parser additionally defines operators: $unop := -$ and $binop := -\ |\ +\ |\ *\ |\ /\ |\ \%$ to the given numbers and names from the scanner.

A valid expression is recognized by the scanner as:

$$expr := number\ |\ name\ |(expr)\ |\ unop\ expr\ |\ expr\ binop\ expr$$

This is a single line expression, so to the parser there are added additional elements, such that a whole program can be analyzed.

program	$:= \text{decl} \star \text{stmt} \star$
decl	$:= \text{type name } (, \text{name}) \star;$
type	$:= \text{int} \mid \text{float} \mid \text{bool}$
stmt	$:= ; \mid \{ \text{stmt} \star \} \mid \text{name} = \text{expr}; \mid \text{name} = \text{read}(); \mid \text{write}(\text{expr}); \mid$ $\text{if}(\text{cond}) \text{stmt} \mid \text{if}(\text{cond}) \text{stmt} \text{ else } \text{stmt} \mid \text{while}(\text{cond}) \text{stmt}$
cond	$:= \text{true} \mid \text{false} \mid (\text{cond}) \mid \text{expr comp expr} \mid \text{cunop cond} \mid \text{cond cbinop cond}$
cunop	$:= == \mid != \mid <= \mid < \mid >= \mid >$
cunop	$:= !$
cbinop	$:= \&\& \mid \parallel$

19 Syntax Trees

Syntax trees are a visual representation of a program. All the information can be taken from the parser. The following code will be visualized in a syntax tree.

```

1 int x;
2 x = read();
3 if(x>0){
4     write(1);
5 }else{
6     write(0);
7 }

```

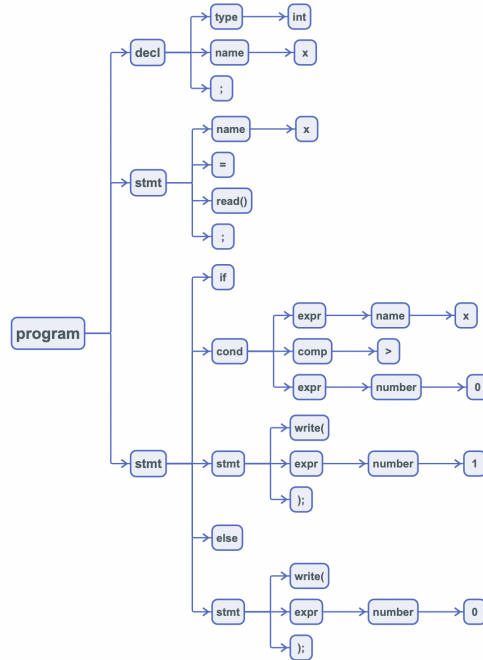




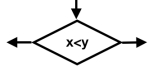



Figure 1: Syntax tree for given program

20 Control Flow Diagrams

A control flow diagram describes the path of the processor through the program during execution. There exist different nodes in a control flow diagram.

Type:	Start	Stop	Function	Expression	Branch	Join
Nodes:						

In the following example the following code will be illustrated as a control flow diagram

```

1 int x;
2 int y;
3 x = read();
4 y = read();
5
6 if(x<y){
7     while(x!=y){
8         x++;
9     }
10    write(x);
11 }else
12 {
13     write(x);
14 }

```

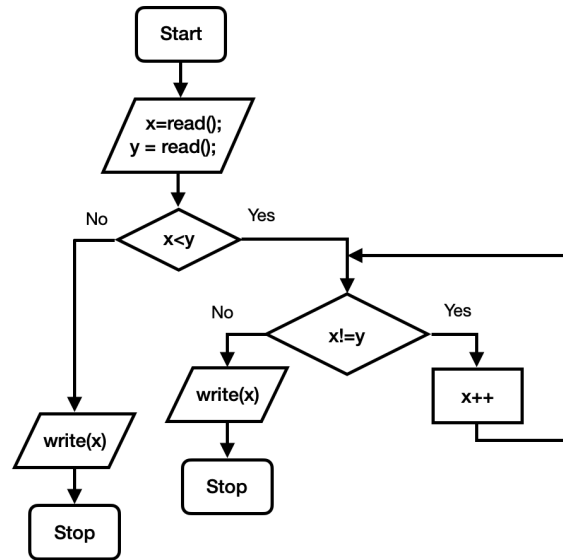


Figure 2: Control Flow Diagram for given program

21 Reading C declarations

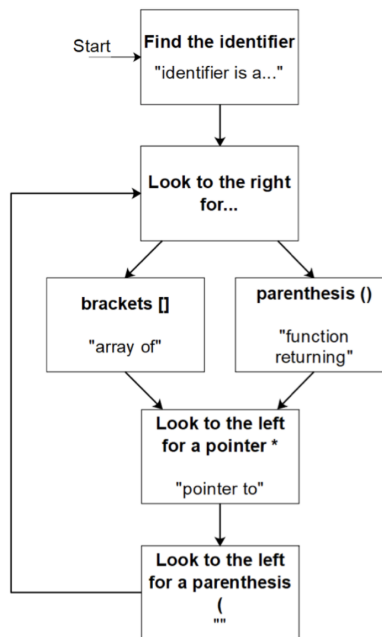


Figure 3: Read C declarations

```
char * (* (* * x[][8])())[];
```

x is an
array of an array
of 8 pointers
to pointers
to function
returning pointers
to array
of char pointers

Figure 4: Read C declarations example