

University of Adrar

Faculty of Material Sciences, Mathematics and Computer Science

Department of Mathematics and Computer Science

Algorithms and Data Structure 2

C
O
U
R
S
E

H
A
N
D
O
U
T

Full Algorithms and Data Structure 1 Course

Level : 1st YEAR LICENCE (LMD) in Mathematics and
Computer Science

Semester: 2nd Semester (S2)

Dr. KADDI Mohammed

Associate professor

University of Adrar

Foreword

This handout, a crucial resource, is specifically designed for first-year LMD students in the Mathematics and Computer Science field. It serves as a comprehensive course manual for the subject "Algorithmics and data structure 2", aiming to introduce the fundamental notions of functions and procedures, files, and linked lists. It's important to note that students should have a solid foundation in computer science and mathematics.

This handout is structured into three chapters as follows:

First chapter: Subroutines: Functions and Procedures

This chapter will define functions and procedures and then explain local and global variables. I will present the transmission of parameters and then address the concept of recursion.

Second chapter: The files

The fundamentals of files, their kinds, and the different operations will all be covered in this chapter.

Third chapter: Linked lists

This chapter will introduce pointers, dynamic memory management linked lists, operations on linked lists, doubly linked lists, and special linked lists.

A list of **bibliographical references** is given at the end of this manuscript.

Table of contents

	Page
Foreword	01
Table of contents	02
Chapter 1: Subprograms: functions and procedures	03
1. Introduction	03
2. Definitions	04
3. Local and global variables	08
4. Passing parameters	10
5. Recursivity	13
6. Conclusion	16
Chapter 2: The files	17
1. Introduction	17
2. Why files are needed?	17
3. Types of Files	17
4. File Handling in C	18
5. Functions for file handling	19
6. File operations	22
7. Conclusion	31
Chapter 3: Linked Lists	32
1. Introduction	32
2. Pointers	33
3. Pointer Operations	38
4. Dynamic Memory Management	38
5. Linked Lists	48
6. Operations on Linked Lists	51
7. Double linked list	58
8. Special Linked Lists	61
9. Conclusion	68
References	69

Chapter 1: subprograms : functions and procedures

1 Introduction

A program is a set of sequential instructions for solving a specific problem. In order to find the solution method (algorithm), the problem must be divided into different sub-problems whose solution is less complicated. Partial problems can be solved using sub-programs.

2 Definitions

2.1 Sub-programs:

Is a set of independent instructions that have a name and are called for execution. The caller is either the main program or another subprogram. When the program, during its execution, reaches the instruction that calls the procedure, the execution context becomes the contents of the subprogram, and once it has finished executing the subprogram, it returns to executing the instruction immediately following the invocation.

The subprograms are also known as procedures, functions, methods and routines.

2.1.1 Procedure:

Procedure is a sub-program which returns no values in its name, but can return results via arguments. The procedure name can be used as a complete instruction, for example:

algorithm	C
SomeProc	SomeProc ();
OtherProc (x)	OtherProc (x) ;

2.1.2 Function:

Function is a sub-program that necessarily returns a result in its name, as its name is considered to be a variable that carries a certain value. Consequently, the function call can be used as a variable in assignment operations and other expressions. For example:

Algorithm	C
Y← SomeFunction (X) *5	Y= SomeFunction (X) *5 ;

Note: Any procedure that returns a single result as an argument can be converted into a function.

2.1.3 Advantages of using subroutines :

- Readability: the use of subroutines organizes and simplifies the program, making it easier to understand the program code.
- Programming speed: don't repeat the same sequence of instructions several times within the program.
- Reduce program size
- Facilitates the maintenance process
- Reuse: it can be stored in libraries for reuse in other programs.

2.2 Declarations

Procedure: the declaration of a procedure takes the following form:

algorithm	C
<pre> procedure proc_name (parameter list) local variables begin instructions end. </pre>	<pre> void proc_name (parameter list) { local variables ; instructions ; } </pre>

Function: the declaration is similar to the declaration of a procedure, except that the type of the result value returned must be specified. It takes the following form:

Algorithm	C
<pre> function func_name (parameter list) : type local variables begin instructions end. </pre>	<pre> type func_name (parameter list) { local variables ; instructions ; } </pre>

- `proc_name`, `func_name`: valid identifiers.
- Parameter list (optional): a set of variables through which data is transmitted and results are retrieved, separated by a comma ",", and which are enclosed in parenthesis () and are of the form `paramName: type`, such as `(a:integer, b:real)` and are called "formal parameters".
 - In C, the list of arguments takes the form of `type paramName (int a, float b)`. Parentheses () are required even if they contain no arguments.
- Local declarations (optional) : A list of local variables of the form: `var varLoc : type`
- Instructions: a set of instructions of any type, which will be executed when the subprogram is called. Where all variables declared in the parameter list or in the local declaration, which are called local variables, and variables declared in the main program, called global variables, can be used.
- Result_Type : When the program is a function, the type of value that the function will return to the program that called it must be specified, and a value must be assigned to the function name. This is generally the function's last instruction, and is of the form `func_name ← expression` where the function name acts as a special variable that contains the return value by the function.
 - In the C language, you can dispense with the result type if the subprogram is a procedure, but some versions use the word **void**, which means that the function returns nothing, and the word **return** is used to assign a value to the function name.
- **return**: the **return** instruction exits the sub-program and returns it to the program that called it at the instruction immediately following the invocation. It can return a value to the program that called the sub- program if it was a function.

Format:

```
return [ <expression> ] ;
```

Example:

```
return 5*x ;      If a function  
return ;         if it's a procedure (i.e. a void function)
```

Important notes:

- To find the arguments, we ask what we're giving the subprogram as input and what it's returning as output.
- The list of parameters in the definition part of the sub-program must be identical in number and type to that used in the sub-program invocation.
- The first line of a function or procedure declaration, i.e. function type, function name, type, order and number of arguments, except their names, is called the header or prototype.
- Arguments are not grouped together if they are of the same type, as in (x, y:integer), but we put (x:integer, y:integer) (int x, int y)
- Any return type other than void indicates that the program is a function and not a procedure.
- void main() or simply main() is a procedure, while int main() is a function, so you need to use return.
- scanf() and printf() are two functions declared in the stdio library

2.3 Where to declare subprograms :

In the algorithm, it is located after the declaration of variables and before the begin of the main program. In a C program, it is declared before the main() function.

Note: The order of subroutines is important, as each function must be defined before it can be used. In other words, if function f1() calls function f2(), then function f2() must be defined before function f1().

2.4 The invocation

To call and execute a procedure, we use its name as a separate instruction and assign values and/or variables to the arguments in brackets, called effective parameters. Parentheses can be omitted in the absence of any arguments, but in C, they are mandatory.

The same goes for calling a function, where its name is considered a variable that carries a certain value, so the function call can be used as a variable in assignment operations and other expressions.

The parameters must correspond in number, type and order with the formal parameters.

2.5 Examples

Examples of procedures

- If numbers below a certain limit are displayed on the screen, it takes the upper limit and returns nothing.
`procedure displayNbs(n : integer)`
- Display array values on screen takes an array and returns nothing
`procedure displayTab(t :real array, n :integer)`
- Solve a quadratic equation that takes three coefficients and returns two solutions
`procedure eq2(a : integer, b : integer, c : integer, var x1 : integer, var x2 : integer)`

Examples of functions

- Square a number Takes a number and returns its square
`function square(x :real) : real`
- The area of a rectangle takes two numbers and returns the area
`function area(long :real, wide :real) : real`
- Solving a first-order equation takes two coefficients and returns a solution
`function eq1(a :real, b :real) : real`
- The sum of an array takes an array and returns the sum
`function sum(t :array of real numbers, size :integer) : real number`
- whether the number is prime or not
`function isPrime(x : integer) : Boolean`

Example Algorithm

<code>algorithm Test;</code>	Program name
<code>var z : real;</code>	Global variable
<code>procedure displayNbs(n:entire)</code>	The name of the procedure that takes an integer variable n as argument
<code>var i:integer;</code>	local variable
Begin	The begin of the procedure
<code>for i←1 to n do</code> <code>Write(i);</code> <code>Endfor;</code>	Procedural instructions
End procedure.	end of procedure
<code>Function sumNbrs</code> <code>(x:integer,y:integer):integer</code>	The name of the function that takes two integer variables and returns an integer result. x and y are not grouped even if they are of the same type.

Begin	The begin of the
sumNbrs ←x+y;	The function name acts as a variable and takes the result of the sum
End function.	end of function
Begin	Begin of main program
displayNbs(5);	Call the displayNbrs procedure, where 5 is assigned to n, and the procedure displays the numbers from 1 to 5.
z←sommeNbrs (5, 3);	Calling sumNbrs, the program assigns the value 5 to x and the value 3 to y, then calculates the sum and assigns it to z
Write("the sum is ", z)	It displays the sum is 8
End.	End of main program

Examples C

#include <stdio.h>	utilizing the stdio library
float z ;	Global variable
void displayNbs (int n)	The name of the procedure that takes an integer variable n as an argument
{	The begin of the procedure
int i ;	local variable
for (i=1; i<=n; i++) printf("%d\t",i);	Procedural instructions
}	end of procedure
int sumNbrs (int x, int y)	The name of the function that takes two integer variables and returns an integer result. x and y are not grouped even if they are of the same type.
{	The begin of the
return x+y ;	The function name acts as a variable and takes the result of the sum
}	end of function
int main(){	Begin of main function
displayNbs (5);	Call the displayNbrs procedure, where 5 is assigned to n, and the procedure displays the numbers from 1 to 5.
Z=sumNbrs (5, 3);	Calling sumNbrs, the program assigns the value 5 to x and the value 3 to y, then calculates the sum and assigns it to z
printf("sum is %d", z);	It displays the sum is 8
return 0 ;}	End of main function

3 Local and global variables

A **global variable** is a variable declared outside the body of any sub-program, and therefore usable anywhere in the program. Since a variable is global, it is not necessary to pass it as a parameter to use it in subprograms. As for its lifetime, i.e. its existence in memory, it is created when the program is loaded into memory, and is only deleted at the end of program execution.

A **local variable** is a variable that can only be used in the subprogram or block where it is defined. The variable is created when the function is called and deleted when execution is complete.

- We recommend using local variables and parameters rather than global variables to avoid errors and ensure function independence.

Example Algorithm:

algorithm glob_loc;	
Var glob, b : integer;	global variables
Procedure tst	
Var b, loc : integer;	local variables
Begin	
glob←11;	Global variables are accessible within the
b←22;	Local variable b hides global variable b
loc←33;	
Write("in tst: glob=", glob, "b=", b, "loc=", loc);	
End.	
Begin	
glob←1;	
b←2;	Variable b is a global variable
Write("before tst : glob=", glob, "b=", b);	Local variables such as loc are not accessible
tst	Procedure call
Write("after tst : glob=", glob, "b=", b);	
End.	

Example in C

#include <stdio.h>	
int glob, b ;	global variables
tst(){	
int b, loc ;	local variables
glob=11;	Global variables are accessible within the
b=22;	Local variable b hides global variable b
loc=33;	
printf("in tst: glob=%d b=%d loc=%d", glob, b, loc);	
}	
int main(){	
glob=1;	
b=2;	Variable b is a global variable
printf("before tst : glob=%d b=%d", glob, b);	
//Local variables such as loc are not accessible	
tst();	Procedure call
printf("after tst : glob=%d b=%d", glob, b);	
return 0 ;}	

Screen :

```

before tst :  glob=1    b=2
in tst:      glob=11   b=22 loc=33
after tst :  glob=11   b=2

```

Explanation:

before calling tst	During tst call	after calling tst						
glob b <div style="display: flex; gap: 20px;"> <div style="border: 1px solid green; padding: 2px 5px;">1</div> <div style="border: 1px solid green; padding: 2px 5px;">2</div> </div>	glob b <div style="display: flex; gap: 20px;"> <div style="border: 1px solid green; padding: 2px 5px;">11</div> <div style="border: 1px solid green; padding: 2px 5px;">2</div> </div> <div style="margin-left: 100px;"> <table border="1" style="border-collapse: collapse;"> <tr> <td style="padding: 2px 5px;">tst</td> <td style="padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">loc</td> </tr> <tr> <td></td> <td style="padding: 2px 5px;">22</td> <td style="padding: 2px 5px;">33</td> </tr> </table> </div>	tst	b	loc		22	33	glob b <div style="display: flex; gap: 20px;"> <div style="border: 1px solid green; padding: 2px 5px;">11</div> <div style="border: 1px solid green; padding: 2px 5px;">2</div> </div>
tst	b	loc						
	22	33						

Before the call, there are only two variables glob and b, but when the tst procedure is called, the processor reserves two more variables, loc and b. The procedure can access global variables, but the local variable b hides the global variable b, and when the procedure is terminated, the processor deletes all local variables.

4 Passing parameters

Arguments are the variables through which information can be exchanged between programs, i.e. the input of data from the calling program to the subprogram and/or the output of results from the subprogram to the calling program.



There are two ways of passing parameters or arguments

Passage by value:

In this mode, the value of the original variable is copied into the (formal) parameter, and this copy is used (a local variable), leaving the original variable unchanged. In this mode, a constant value or expression can be passed, and need not be a variable.

This mode is only used to enter information into the sub-program and is not used to receive results.

Passage by reference, address or variable:

Not only is the value passed, but the place of the original variable (address) is passed to the formal variable, so they become a single variable, and any modification of the parameter in the sub called program results in the modification of the original variable that was passed as a parameter.

In this mode, it's not possible to pass a constant value or an expression, but it must be a variable, so it's called pass by variable.

This mode is used to enter information for the sub-program, especially large variables such as arrays and matrices, to avoid copying. It is also used to receive results.

In algorithm the word “**var**” is used before declaring the name of the argument to indicate that the pass is a pass by variable or pass by reference.

To pass arguments with address in C, we use the pointers we'll see in the third chapter of this course, where the name of the formal parameter is preceded by * when declared and when used, but when the function is called, this variable is preceded by “&”.

Declaration `int f(int *x)`

Usage `*x=5;`

Call `f(&a);`

In C++, pointer management is masked by using the “&” symbol in the declaration only, and this is called a reference.

Declaration `int f(int &x)`

Usage `x=5;`

Call `f(a);`

Note: We don't use the word **var** (* in C) to enter data and display results.

Example Algorithm:

by value	passage by reference, address or variable
<pre> algorithm Passage_value var a, c: real Procedure square (x: real, y: real) Begin y← x*x end begin c←0 a←3 write("before square c=", c) square(a ,c) // we can use square(3,c) write("after square c=", c) end </pre>	<pre> algorithm Variable_passage var a, c: real Procedure square (x: real, var y: real) Begin y← x*x end begin c←0 a←3 write("before square c=", c) square(a,c) write("after square c=", c) end </pre>
the screen	
before square c=0 after square c=0	before square c=0 after square c=9

Example C:

passage by value	passage by reference, address or variable
<pre> #include <stdio.h> void square(float x, float y){ y= x*x; } int main(){ float a, c; c=0; a=3; printf ("before square c=%f ", c); square(a ,c); // we can use square(a,5) printf ("after square c=%f ", c); return 0 ;} </pre>	<pre> #include <stdio.h> void square(float x, float *y) { *y=x*x; } int main(){ float a, c; c←0; a←3; printf ("before square c=%f ", c); square(a,&c); // square(a,5) cannot be used printf ("after square c=%f", c); return 0 ;} </pre>
the screen	
before square c=0 after square c=0	before square c=0 after square c=9

Switching from a procedure to a function:

Any procedure that returns a single result can be converted into a function, where we change the word Procedure into function and transform the argument that the procedure returns into a local variable and define the type of the function as the type of this argument and before terminating the function, we assign the value of the variable to the name of the function.

For example, the sub-program that calculates the absolute value of a real number:

In the form of a procedure	In the form of a function
<pre> Procedure abs (x: real, var y: real); Begin if x<0 then y← -x; else y← x; end if; end. </pre>	<pre> function abs (x: real): real var y: real; Begin if x<0 then y← -x; else y← x; end if abs←y; end. </pre>
call	
abs(-5, z);	z←abs (-5);

In C

<pre> void abs (float x, float *y){ if (x<0) *y= -x; else *y= x; } </pre>	<pre> float abs (float x){ float y; if (x<0) y= -x; else y= x; return y; } </pre>
<p>The variable y can be omitted You can omit else, which comes after return.</p>	<pre> float abs (float x){ if (x<0) return -x; return x; } </pre>
call	
abs(-5, &z);	z=abs (-5);

5 Recursivity

The recursion is a simple and elegant way of solving certain problems of a recurring nature.

A recursive program is any program that recalls itself. Whereas a defined program is used to define itself. In concrete terms, a recursive program is one that does part of the work and then recalls itself to complete the rest.

Note: Any **for** or **while** loop can be transformed into a recursive program.

Stop condition

Since the recursive program calls itself, it is necessary to provide a condition for stopping the recursion, which is the case when the program doesn't call itself or it will never stop.

It is preferable to test the stop condition first, then, if the condition is not met, to call the program back as the call leads to the stop condition.

Example:

<pre> Procedure display (i :integer); begin write(i) display (i +1); end. </pre>	<pre> void display (int i) { printf("%d",i); display (i +1); } </pre>
---	---

For example, we invoke `display(1)`, so it displays 1, then it invokes `display` for $i=i+1=2$, so it displays 2, then to infinity, so the algorithm must have a stop condition, by Example:

<pre> Procedure display (i :integer) begin if (i<10) then write(i) display (i +1) endif end. </pre>	<pre> void display (int i) { if (i<10) { printf("%d",i); display (i +1); } } </pre>
--	---

The general form of the recursive program:

<pre> procedure Recursive (parameters); begin if (stop condition) then <stop point instructions>; else <instructions>; Recursive call (parameters changed); <Instructions>; Endif; End. </pre>	<pre> void recursive(parameters) { if (stop condition) <stop point instructions>; else { <instructions>; Recursive call (parameters changed) <Instructions>; } } </pre>
---	--

Example:

1. Factorial

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n - 1) & \text{if } n > 0 \end{cases}$$

The function can be written as a recursive relationship:

$$b_0 = 1$$

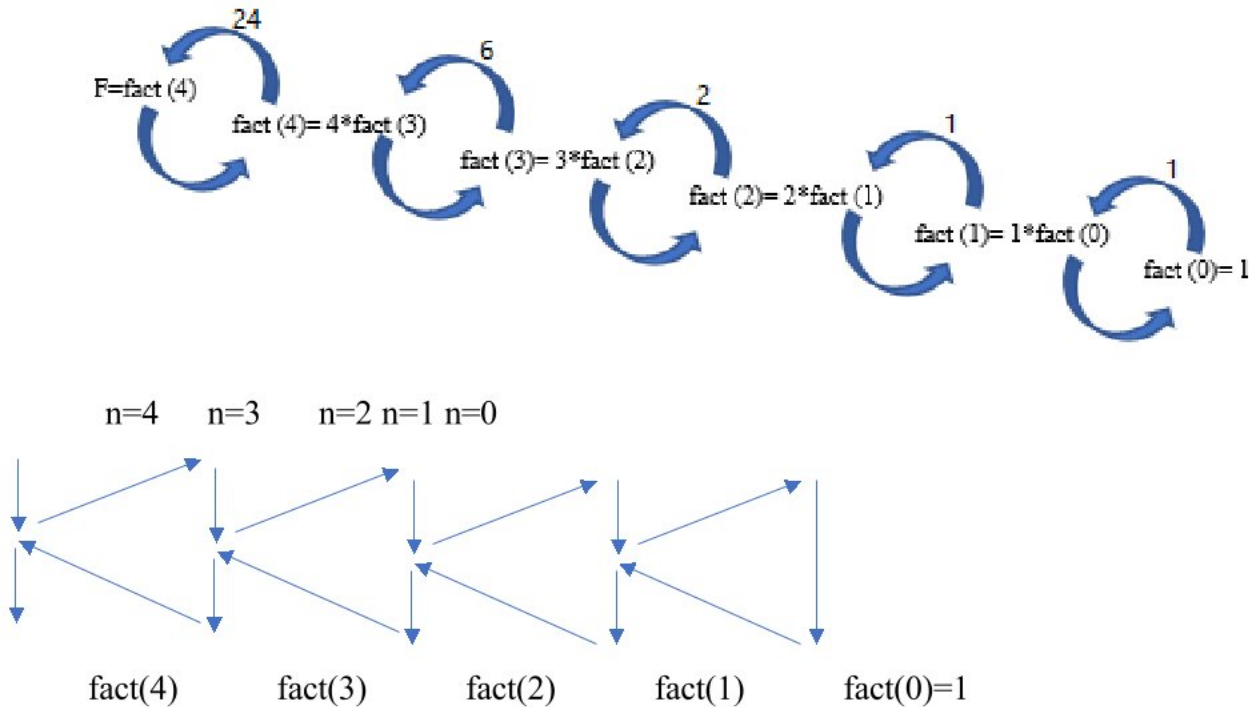
$$b_n = nb_{n-1}$$

iterative	recursive
<pre> Function fact (n : Integer) : Integer; var i, f: Integer; begin f←1; for i←2 to n do f← f * I; endfor; fact←f; end. </pre>	<pre> Function fact (n : Integer) : Integer begin if (n = 0) then fact←1; else fact←n*fact (n-1); endif end. </pre>
<pre> int fact (int n){ int f=1; for (i=2 ;i<= n ; i++) f← f * I; return f; } </pre>	<pre> int fact (int n){ if (n == 0) return 1; return n*fact(n-1); } </pre>

How does it work?

We call the function fact for $n=4$ to calculate $4!$

We call $F=\text{fact}(4)$ which in turn calls $\text{fact}(3)$ which calls $\text{fact}(2)$ until it calls $\text{fact}(0)$ which terminates and returns 1 allowing $\text{fact}(1)$ to be calculated which allows $\text{fact}(2)$ to be calculated until $\text{fact}(4)$ is calculated $\text{fact}(4)$. See below.



The execution stack :

A memory location designated to hold parameters and local variables, and where the result is stored for each running sub-program.

Usually, programming in recursive mode is easier and more readable, but it consumes a lot of memory, for example to calculate $4!$ We reserve a place in the stack for the result, another for the parameter $n=4$, then another place for the result of $3!$ And the parameter $n = 3$ and so on until $0!$ is calculated. The parameter $n=0$ is deleted, then the parameters and results are deleted in the reverse order in which they were created.

Mutual recursive: a recursive program can call itself directly or indirectly, because it calls another program, which in turn calls the first program.

Example:

To calculate π , we use the following relationship $\pi/4=1-1/3+1/5-1/7+1/9\dots$ We create two recursive functions, the first adding $1/n$, calling the second for $n=n-2$, then subtracting $1/n$ which in turn calls the first to add and so on until n becomes zero.

<pre>function f1(n: integer); begin if n<=0 then f1←0; else f1←1/n+f2(n-2); endif end. function f2(n: integer); begin if n<=0 then f2←0; else f2←-1/n+f1(n-2); endif end.</pre>	<pre>#include <stdio.h> float f2(int n); float f1(int n) { if (n <= 0) return 0; return 1. / n + f2(n - 2); } float f2(int n) { if (n <= 0) return 0; return -1. / n + f1(n - 2); } void main() { printf("%f\n", 4*f1(2*100+1) * 4); }</pre>
--	--

The f1 function calculates $\pi/4$, and to calculate π , we multiply the result by 4.

Important note: Since function f1 calls function f2, which is not yet defined in C, the header of function f2 must be added without its body (the first line) before defining function f1, knowing that its definition comes after.

6 Conclusion

This chapter introduced functions and procedures, local and global variables, parameter passing, and the concept of recursion. The next chapter will cover files, their types, and file manipulation.

Chapter 2: The files

1. Introduction

A file is a container in computer storage devices used for storing data.

2. Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

3. Types of Files

When dealing with files, there are two types of files you should know about:

- Text files
- Binary files

3.1 Text files

Text files are the normal **.txt** files. You can easily create text files using any simple texteditors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text.

You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

3.2 Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

4. File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the localfile system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

5. Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions is given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

5.1 Opening File: fopen()

We must open a file before it can be read, write, or update. The fopen() function is used to open a file. The syntax of the fopen() is given below.

1. **FILE *fopen(const char * filename, const char * mode);**

The fopen() function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "**c://some_folder/some_file.ext**".

- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the `fopen()` function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The `fopen` function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

1. `#include<stdio.h>`
2. `void main()3. {`
4. `FILE *fp ;`
5. `char ch ;`

```

6. fp = fopen("file_handle.c","r");
7. while ( 1 ) {
8. {
9. ch = fgetc ( fp );
10. if ( ch == EOF )
11. break ;
12. printf("%c",ch);
13. }
14. fclose (fp );
15. }

```

Output

The content of the file will be printed.

```

#include; void main( )
{
FILE *fp; // file pointer
char ch;
fp = fopen("file_handle.c","r");
while ( 1 )
{
ch = fgetc ( fp ); //Each character of the file is read and
stored in the character file.
if ( ch == EOF ) break; printf("%c",ch);
}
fclose (fp );
}

```

5.2 Closing File: fclose()

The fclose() function is used to close a file. The file must be closed after performing all the operations on it. The syntax of fclose() function is given below:

```

1. int fclose( FILE *fp );

```

6. File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

6.1 Opening a file - for creation and edit

Opening a file is performed using the `fopen()` function defined in the `stdio.h` header file.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("filename", "mode");
```

For example,

```
fopen("E:\\cprogram\\newprogram.txt", "w");
```

```
fopen("E:\\cprogram\\oldprogram.bin", "rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode `'w'`.

The writing mode allows you to create and edit (overwrite) the contents of the file.

- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`.

The second function opens the existing file for reading in binary mode `'rb'`.

The reading mode only allows you to read the file, you cannot write into the file.

Mode	Meaning of Mode	During Inexistence of file
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. Data is added to the end of the file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. Data is added to the end of the file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

Opening Modes in Standard I/O

6.2 Closing a File

Closing a file is performed using the `fclose()` function.

```
fclose(fp);
```

Here, `fp` is a file pointer associated with the file to be closed.

6.3 Reading and writing to a text file

For reading and writing to a text file, we use the functions `fprintf()` and `fscanf()`.

They are just the file versions of `printf()` and `scanf()`. The only difference is that `fprintf()` and `fscanf()` expects a pointer to the structure `FILE`.

Example 1: Write to a text file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num;
    FILE *fp;
    // use appropriate location if you are using MacOS orLinux
    fp = fopen("C:\\program.txt", "w");
    if(fp == NULL)
    {
        printf("Error!");exit(1);
    }
    printf("Enter num: ");scanf("%d",&num);
    fprintf(fp,"%d",num);fclose(fp);
    return 0;
}
```

This program takes a number from the user and stores in the file `program.txt`.

After you compile and run this program, you can see a text file `program.txt` created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file

```
#include <stdio.h> #include <stdlib.h>
int main()
{
    int num;
    FILE *fptr;
    if ((fptr = fopen("C:\\program.txt", "r")) == NULL) {
        printf("Error! opening file");
        // Program exits if the file pointer returns NULL.
        exit(1);
    }
    fscanf(fptr, "%d", &num);
    printf("Value of n=%d", num);
    fclose(fptr);
    return 0;
}
```

This program reads the integer present in the `program.txt` file and prints it onto the screen.

If you successfully created the file from Example 1, running this program will get you the integer you entered.

Other functions like `fgetchar()`, `fputc()` etc. can be used in a similar way.

6.4 Reading and writing to a binary file

Functions `fread()` and `fwrite()` are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Example 3: Write to a binary file using fwrite()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
    int n1, n2, n3;
};
int main()
{
    int n;
    struct threeNum num; FILE *fptr;
    if ((fptr = fopen("C:\\program.bin", "wb")) == NULL) {
        printf("Error! opening file");
        // Program exits if the file pointer returns NULL.
        exit(1);
    }
}
```

```
}
for(n = 1; n < 5; ++n)
{
num.n1 = n; num.n2 = 5*n; num.n3 = 5*n + 1;
fwrite(&num, sizeof(struct threeNum), 1, fptr);
}
fclose(fptr);
return 0;
}
```

In this program, we create a new file `program.bin` in the C drive.

We declare a structure `threeNum` with three numbers - `n1`, `n2` and `n3`, and define it in the main function as `num`.

Now, inside the for loop, we store the value into the file using `fwrite()`.

The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.

Since we're only inserting one instance of `num`, the third parameter is 1. And, the last parameter `*fptr` points to the file we're storing the data.

Finally, we close the file.

Reading from a binary file

Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```


Example 4: Read from a binary file using fread()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
    int n1, n2, n3;
};
int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;
    if ((fptr = fopen("C:\\program.bin", "rb")) == NULL) {
        printf("Error! opening file");
        // Program exits if the file pointer returns NULL.
        exit(1);
    }
    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
    }
    fclose(fptr);
    return 0;
}
```

In this program, you read the same file `program.bin` and loop through the records one by one. In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

You'll get the same records you inserted in Example 3.

6.5 Getting data using `fseek()`

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using `fseek()`.

As the name suggests, `fseek()` seeks the cursor to the given record in the file.

Syntax of `fseek()`

```
fseek(FILE * stream, long int offset, int whence);
```

The first parameter `stream` is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

Whence	Meaning
<code>SEEK_SET</code>	Starts the offset from the beginning of the file.
<code>SEEK_END</code>	Starts the offset from the end of the file.
<code>SEEK_CUR</code>	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
    int n1, n2, n3;
};
int main()
{
    int n;
    struct threeNum num;
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");
        // Program exits if the file pointer returns NULL.
        exit(1);
    }
    // Moves the cursor to the end of the file
    fseek(fptr, -sizeof(struct threeNum), SEEK_END);
    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2,num.n3);
        fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
    }
    fclose(fptr);
    return 0;
}
```

This program will start reading the records from the file `program.bin` in the reverse order (last to first) and prints it.

7. Conclusion

The basics of files, file types, and different operations on files have been introduced in this chapter. The next chapter will be devoted to linked lists.

Chapter 3: Linked Lists

1. Introduction:

In the first semester, we learned that a program comprises a set of data and a set of instructions, with the data stored in memory as variables. A **variable** is a memory location characterized by an address, name, type, and value.

- **Address:** Each variable stored in memory is identified by an address, a natural number indicating its location. Typically expressed in hexadecimal (e.g., 0x5A63).
- **Name:** An identifier used by programmers to reference the stored value; the variable's name is manipulated instead of the address (e.g., "weight").
- **Type:** In computing, everything is represented in 0s and 1s. The type dictates how to interpret these binary values and specifies the size to be reserved in memory, including the number of bits and allowable operations (e.g., "int" for a 32-bit integer).
- **Value:** The content of the bits composing the variable's value, often changing during program execution (e.g., "15").

When the program encounters a variable declaration statement (e.g., `int age;`), it instructs the operating system (Windows) to allocate a memory space of size `x`, depending on the type. After reservation, the system returns the memory address usable as a variable.

To retrieve a variable's **value**, you simply use its **name**. However, to obtain its **address** (location in memory), the algorithm utilizes the "@" symbol before the variable name, and in C, the "&" symbol precedes the variable name.

Example:

```
write("value of age=", age, " its address=", @age);  
printf("value of age = %d its address = %p", age, &age);
```

Here, `age` is the variable value, and `&age` is its memory address. The `%p` format treats `&age` as a hexadecimal memory address, which can also be displayed in decimal using `%d`. It's important to note that the address may change each time the program is run.

2. Pointers

A **pointer** is a variable whose value points to an address in the computer's memory. This address can be associated with either a variable or a program. Pointers are employed for various purposes, including passing parameters by address, dynamically reserving memory, defining recursive types (such as lists, stacks, and queues), and other applications.

Example:

Memory can be conceptualized as an array numbered from 0 to the memory capacity minus one. In the following illustration, two variables have been allocated. The first is an integer named "age," situated at address 0x0276, holding the value 19. Here, the "0x"p denotes that the number is expressed in the hexadecimal system (16) – specifically, 0x0276 corresponds to 630 in the decimal system.

The second variable, denoted as "p," holds the value 0x0276. This value signifies the location of the variable "age." In other words, we can state that "p" points to "age."

Variable Name	Memory address	Content
	0x0000	
	0x0001	
	0x0002	0x0276
	0x0003	

age	0x0276	19
	0x0277	
	0x0278	

The Creation

To create a pointer variable in the algorithm, we prefix the variable type with the symbol \wedge . This results in the following format: `var p1, p2 : \wedge type`

To create a pointer variable in C, we add * before the variable name `Type *P1, *P2;`

Here \wedge or * indicates that the variable is of the pointer type, i.e. a memory address, while type is the type of the contents of that location.

Example : We declare six variables x and y of integer type, p1 and p2 of type pointer to integer, z of type real, and pz of type pointer to real.

<code>int x, *p1, y, *p2;</code>	<code>Var x, y: integer p1, p2: \wedgeinteger</code>
<code>float z, *pz;</code>	<code>z : real pz : \wedgereal</code>

When declaring a variable, it initially holds an undefined value. It is advisable to set it to NULL in uppercase, signifying that the pointer does not point anywhere (defined within stdio.h, representing the number 0).

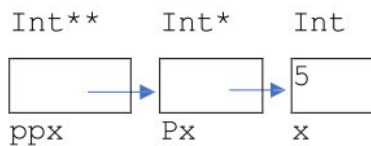
```
p1= NULL;
```

The variable p1 can take the **address** of variable x or the **value** of variable p2, but it cannot take the address of variable z, the address of p2, or the value of pz.

Valid Transactions	Invalid transactions	The Explanation
p1=&x;	p1=x;	p1 is a pointer and x is an integer
p2=p1;	p1=&z;	p1 is an integer pointer and &z is a real address
pz=&z;	pz=p1;	pz is a pointer to a real and p1 is a pointer to an integer
	p2=&p1 ;	P2 is a pointer to an integer, but &p1 is the address of a pointer to an integer.
	p1=&(0x0276) ;	Must be a variable, not a number.

It is crucial to distinguish between the address stored in the pointer and the address of the pointer itself. The pointer, being a variable, possesses an address similar to other variables. Consequently, its address can be assigned to another pointer. However, in such a scenario, the second pointer type must be the address of a pointer of the first type.

For example: x is of integer type (int), and px contains the address of x, so its type is (int*) and ppx contains the address of px, so its type is (int**) as shown in the following diagram:



It is declared as follows:

```
int x, *px, **ppx; x=5;
px=&x; ppx=&px;
```

typedef can be used to create new types and the above statement becomes something like this:

```
typedef int* pint; typedef int** ppint; pint px;
ppint ppx;
```

Usage:

It's rare that we treat memory addresses as direct numbers, but we treat them as addresses for existing variables. To get the address of a variable, we use the @ operation in the algorithm or & in the C programming language **before** the variable name, and to retrieve the value of the variable (Dereference) from its address stored in a pointer, we use the symbol ^ **after** the variable name in the algorithm and * **before the** name of the variable in the C programming language.

$$p \leftarrow @x \Rightarrow p^{\wedge} \Leftrightarrow x \quad p = \&x \Rightarrow *p \Leftrightarrow x$$

Example:

C	The Algorithm	memory	The Explanation
<pre>int x, *p1, y, *p2;</pre>	<pre>Var x, y: integer p1, p2 : ^ integer</pre>		
<pre>x=3; y=4;</pre>	<pre>x←3 y←4</pre>		
<pre>p1=&x; p2=&y;</pre>	<pre>p1←@x p2←@y</pre>		Here p1 contains the address of x and p2 contains the address of y
<pre>*p1=5;</pre>	<pre>p1^←5</pre>		We assign the number 5 to the variable whose address is at p1, and at this point it is the variable x, as if the variable x had a second name, which is *p1 can be replaced by the x=5 statement;
<pre>p1=p2;</pre>	<pre>p1 p2←</pre>		We assign the value of p2, which represents the address of y, to p1, so that y, *p1, and *p2 become the same variable at that time.
<pre>*p1=6;</pre>	<pre>p1^←6</pre>		We assign the digit 6 to the variable whose address is in p1 and at this point it is the variable y can be replaced by the y=6 statement; or *p2=6;

Notes:

- To comprehend pointers better, it is always advisable to visually represent variables, with the pointer depicted as an arrow pointing to the variable carrying its address. Additionally, we symbolize a pointer with a value of NULL, indicating that it does not point to any

location.

- A pointer is always of a simple type, whereas the variable whose address it holds can be of a complex type, such as an array or structure.
- Attempting to retrieve the value of an uninitialized pointer or a NULL value will cause the program to terminate.
 - A value (variable address) must be assigned to the pointer before attempting to retrieve the value it points to.
 - Before retrieving the value that the pointer points to, it is crucial to ensure that the pointer is not null.
- Understanding the passing of parameters by address in subroutines becomes possible with these concepts.

Example

C	memory	The Explanation
<pre>void exchange(int *x, int *y){ int t; t=*x; *x=*y; *y=t; } int a=5,b=3; exchange(&a, &b);</pre>	<p>The diagram illustrates the state of memory during the execution of the C code. It shows two pointer variables, <code>x</code> and <code>y</code>, and two integer variables, <code>a</code> and <code>b</code>. Variable <code>a</code> contains the value 5, and variable <code>b</code> contains the value 3. The pointer variable <code>x</code> holds the address of <code>a</code> (indicated by a blue arrow pointing to the box containing 5), and the pointer variable <code>y</code> holds the address of <code>b</code> (indicated by a blue arrow pointing to the box containing 3). The labels <code>*x</code> and <code>*y</code> are placed next to the pointer variables, indicating the values they point to.</p>	<p>Here <code>x</code> and <code>y</code> are two pointers and when calling the function we assign <code>x</code> the address of variable <code>a</code> i.e. <code>x=&a</code> and <code>y</code> the address of variable <code>b</code> i.e. <code>y=&b</code> and inside the function <code>exchange</code> to obtain the variable whose address <code>x</code> carries we use the operation <code>*</code> where <code>*x</code> at this moment represents the variable <code>a</code> and <code>*y</code> represents the variable <code>b</code></p>

3. Pointer Operations

Suppose that P and Q are pointers and i is an integer. The following table summarizes the operations that can be performed on pointers:

Algorithm operation	Operation C	Type of 2nd Operator	Type of result	Example	Observation
+	+	Int	Pointer	P + i	Returns a pointer to the i th element after P in an array
	++		Pointer	P++	Returns a pointer to the next immediately P element in an array
-	-	Int	Pointer	P - i	Returns a pointer to the i th element before P in an array
	--		Pointer	P--	Returns a pointer to the element immediately preceding P in an array
-	-	Pointer of the same type	Int	P - Q	Returns the number of items between P and Q where P and Q should point to the same array
=	==	Pointer	Boolean	P == Q	This is true if P and Q have the same address, i.e. they point to the same place
≠	!=	Pointer	Boolean	P != Q	This is true if P and Q are different
^	*		Value Type	*P	To retrieve the value whose address it contains

4. Dynamic Memory Management

The method we've known for reserving variables in memory so far is called static reservation. In static reservation, the variable is declared at the beginning of the program, and the compiler automatically reserves the necessary memory. The variable persists until the end of the program's execution (or until the end of a subroutine in the case of a local variable). However, there are situations where we need to allocate a dynamic amount of memory, such as an array with N elements, and N is only known at runtime. In such cases, we declare a pointer, and when N becomes available, we dynamically reserve the array.

Developers have a set of functions that enable dynamic memory management during runtime.

In algorithm:

There are three procedures for dynamic memory management:

1. **allocate():** Used to reserve an array, taking the pointer's name (array name) and the number of elements as parameters.

```
allocate(array_name, num_elements)
```

Example:

```
allocate(T, 10)
```

2. **realloc():** Changes the size of the array, either by increasing or decreasing. It takes the pointer's name (array name) and the new number of elements (new size) as parameters. It preserves the values of the previously reserved elements and removes excess or adds new elements to the array.

```
realloc(array_name, new_size)
```

Example:

```
realloc(T, 15)
```

3. **dealloc():** Deletes the reserved array created with `allocate()`. It takes the pointer's name (array name) as a parameter.

```
dealloc(array_name)
```

Example:

```
dealloc(T)
```

After creating an array "t" using `allocate()`, its elements can be accessed either by square brackets [] or by the retrieval operation \wedge , where the pointer "t" contains the address of the first element, i.e., $@t[0] = t$ and $t^\wedge = t[0]$. To get the address of the second element, "t[1]," add 1 to "t," i.e., $@t[1] = t + 1$, and $(t + 1)^\wedge = t[1]$. Therefore, the address of "t[i]" is $t + i$, i.e., $@t[i] = (t + i)$ and $(t + i)^\wedge = t[i]$.

Example:

algorithm	memory	The Explanation
var t : ^real n:integer	t n □ □	A pointer “t” and a variable “n” representing the number of its elements are declared
begin write("enter number of elements") read(n)	t n □ □ 3	Let “n” take 3
allocate (t ,n)	t □ → □ □ □	allocate() reserves an array of three elements and sets its address to t
t[0] ← 1 t[1] ← 2 t[2] ← 3 t^ ← 1 (t+1)^ ← 2 (t+2)^ ← 3	t □ → □ 1 □ 2 □ 3	We fill in the table where we can use the square brackets [] or use ^ where t[i] ⇔ (t+i)^
reallocate(t,n+2)	t □ → □ 1 □ 2 □ 3 □ □	Calling reallocate() resizes the array to 5
t[3] ← 4 t[4] ← 5 (t+3)^ 4 (t+4)^ ← 5 ←	t □ → □ 1 □ 2 □ 3 □ 4 □ 5	We fill in the two added elements
deallocate(t)	t n □ □ 3	We call deallocate() to remove the array

In C:

Memory management in C differs slightly from algorithms. Before delving further into it, we need to familiarize ourselves with "sizeof" and type casting.

4.1. The "sizeof" operation

A variable occupies more or less memory space depending on its type. For instance, a variable of type char takes up one byte, while a variable of type int requires either two or four bytes, depending on the C version. To determine the size required for a specific type, we use sizeof(), which takes the name of the variable or the name of the type as an argument and returns the number of bytes it needs in memory.

```
int sizeof type;
```

Example:

```
float t[20];
printf("char: %d bytes\n", sizeof(char));
printf("int : %d bytes\n", sizeof(int));
printf("double: %d bytes\n", sizeof(double));
printf("the size of t: %d bytes\n", sizeof(t));
printf("the size of t:%d bytes\n", 20*sizeof(float));
```

that displays on the screen

```
char: 1 byte
int: 4 bytes
Double: 8 bytes
T size: 80 bytes
T size: 80 bytes
```

The size of an array can be found by multiplying the size of a single element by the number of elements.

4.2. Type Change: Casting

Sometimes, we need to convert a specific value from one type to another. To force the compiler to change the type of a specific value, we use the following formula:

(type) expression

Where the expression is converted to type

Example 1

<code>int A=8,B=3;</code>	
<code>float R=A/B;</code>	Since operators A and B are integers, the / operation performs integer division, resulting in R = 8/3.
<code>printf("no casting R=%f \n",R);</code>	no casting R=2.000000
<code>R=(float)A/B;</code>	We convert the value of A (not the variable A) to a real number, and then we perform the division process, so the operation becomes R = 8.0/3.
<code>printf("with casting R=%f \n",R);</code>	with casting R=2.6666666

Example 2

<code>int x,*p1;</code>	An integer and a pointer to an integer
<code>float y=2,*p2;</code>	A real number and a pointer to a real number
<code>x=(int)y;</code>	It converts the value of y to an integer and puts it in x, so x takes the value 2
<code>p2=&y;</code>	p2 takes the address of y
<code>p1=(int*)p2;</code>	Converting the address of a float to the address of an int, but the address of the variable remains in both variables, which is the address of y <div style="text-align: center;"> <p style="margin-left: 100px;"> x y 2 2.0 *p2/*p1 p1 p2 </p> </div>
<code>printf("x=%d \n",x);</code>	Displays x=2
<code>printf("*p2=%f \n",*p2);</code>	Displays *p2=2.000000 the same as y

<pre>printf("*p1=%d \n", *p1);</pre>	<p>Displays *p1=1073741824</p> <p>Because translating the bits of a real number into an integer does not give the same number</p>
--------------------------------------	---

4.3 Memory Management in C

Dynamic memory management in C is done using four functions defined in the stdlib library:

- ``malloc()`` (memory allocation, meaning to reserve memory): It instructs the operating system to reserve the required amount of memory.

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

It takes the required memory size (number of bytes) as a parameter and returns a pointer to the reserved memory. If the process fails due to insufficient available size, it returns NULL.

Example:

```
float *t;
```

```
t=(float *)malloc(10*sizeof(float));
```

	<code>t=</code>	<code>(float *)</code>	<code>malloc(</code>	<code>10*</code>	<code>sizeof(</code>	<code>float</code>	<code>)</code>	<code>);</code>
Table	Convert to	To reserve	Number of	The size of each	Type of each			
Name	Pointer Type	the table	items	element	element			

- ``free()``: This function is used to return memory previously reserved by the operating system's ``malloc()``, allowing it to be used by other programs.

```
void free( void * pointer );
```

It takes a previously reserved pointer as a parameter. It is recommended to set the pointer to NULL after calling ``free()`` to ensure that the pointer is no longer pointing to valid memory and to avoid potential errors.

Example:

```
free(t);
```

- ``realloc()``: This function is used to change the size of the reserved memory, either by increasing or decreasing it.

```
void * realloc(void * pointer, int new_size);
```

Where the function calls ``malloc()`` to reserve a new block of memory with the size of ``new_size``, then copies all the values from the "pointer" array to the new location (or deletes the extra elements if ``new_size`` is smaller than the old size). After that, it deletes the old

reserved array by calling `free()`. If the operation succeeds, it returns a pointer to the new location; otherwise, it returns `NULL`.

Example:

```
t=(float*)realloc(t, 20*sizeof(float));
```

- `calloc()`: Similar to `malloc()`, but this function puts zeros in the reserved memory.

```
void * calloc(int nb_element, int element_size);
```

It takes `nb_element`, representing the number of items in the array, and `element_size`, representing the size of each element. It returns a pointer to the allocated memory with zero-initialized values.

Example:

```
t=(float*)calloc(10, sizeof(float));
```

Observation:

- In the function lesson, we learned that `void` means the function returns nothing, while `void*` means the function returns a pointer of an undefined type.
- The `void*` type needs to be converted to the specific pointer type that will hold the address. This is done by placing the pointer type in parentheses before the `malloc`, `calloc`, and `realloc` function names. However, this conversion is not necessary in C++.
- To use these functions, you need to include the `stdlib` or `alloc` library by using the following statement:

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

The `sizeof` operation is not a function, so parentheses can be omitted. When reserving memory, we follow these steps:

1. Reserve memory with `malloc`.
2. Ensure that the allocation process has completed successfully by using `if (pointer != NULL)`.
3. When finished using the allocated memory, return it to the system using `free`.

Example

C	The Explanation
<code>#include <stdio.h></code> <code>#include <stdlib.h></code>	Inclusion of the STDLIB library
<code>int main(void) {</code> <code>char *str;</code>	Declaring a char pointer
<code>str = (char *) malloc(4*sizeof char);</code>	allocating an array for 4 characters:
<code>str[0]='A'; str[1]='S'; str[2]='D';</code> <code>str[3]='\0';</code>	Populating the array with the string "ASD" using [] and the symbol '\0' to indicate the end of the string:
<code>*str='A'; *(str+1)='S'; *(str+2)='D';</code> <code>*(str+3]='\0';</code>	Populating the array with the literal string "ASD" using the retrieval operation * where $*(str+i) \Leftrightarrow str[i]$
<code>printf("String is %s\n Address is %p\n",</code> <code>str, str);</code>	To display the string and its address, noting that & is not used because str is already an address:
<code>str = (char*) realloc(str, 5*sizeof char);</code>	Changing the capacity of the array from 4 to 5:
<code>str[3]='2'; str[4]='\0';</code> <code>*(str+3)='2'; *(str+4]='\0';</code>	Filling in the last two characters so that the string becomes "ASD2":
<code>printf("String is %s\n New address is</code> <code>%p\n", str, str);</code>	Displaying the string "ASD2" and its new address:
<code>free(str); return 0;</code> <code>}</code>	Returning reserved memory:

4.3. Pointers and matrices in C

Matrices in C are arrays in which each element is an array. We want to create an $M[3][4]$ matrix with three rows and four columns. Suppose we have three arrays: M0, M1, and M2.

```
float M0[4],M1[4],M2[4] ;
```


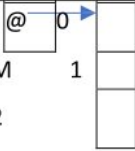
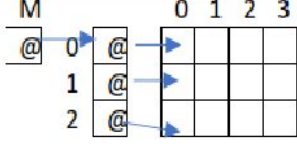
These arrays can be created using pointers

```
float *M0,*M1,*M2;
M0=(float *)malloc(4*sizeof(float));
M1=(float *)malloc(4*sizeof(float));
M2=(float *)malloc(4*sizeof(float));
```

Note that M0, M1 and M2 are all of the same type (float *), so they can be replaced by an array M of type (float *).

```
float * M[3];
for(int i=0; i<3; i++)
M[i]=(float *)malloc(4*sizeof(float));
```

Now, pointers can be used to create the array ‘M’

C	memory	The Explanation
<pre>float **M;</pre>		An M pointer is declared to be of type float **
<pre>M=(float**) malloc(3*sizeof(float*));</pre>		
		An array M is created, which contains 3 elements representing the number of rows. The type of each element is float*.
<pre>for(int i=0; i<3; i++) M[i]=(float*) malloc(4*sizeof(float));</pre>		
		We create three arrays, each representing a row in the matrix. The number of columns is 4, and the type of each column is float. *(M+i) can be used instead of M[i].

Any element of the matrix can be accessed using square brackets `[]` or by using the dereference operator `*` where:

$$M[i][j] \Leftrightarrow *(M[i]+j)$$

$$M[i][j] \Leftrightarrow *((M+i)+j)$$

using typedef

```
typedef float ** matrix;
typedef float * table;
matrix M;
M=(matrix)malloc(3* sizeof(table));
for(int i=0; i<3; i++)
M[i]=(table) malloc(4*sizeof(float));
```

Note : A static array in C is a constant memory address that cannot be changed.

Example:

```
int *p, t[10];
```

```
    p=t;
```

Correct because t is the address of the first element

```
    t=p;
```

Not accepted because t is a constant that cannot be changed.

5. Linked Lists

5.1 Introduction

In programming, to process data of the same type (e.g., student information), we need Arrays. Arrays are an important concept in any programming language because they allow quick access to their elements. However, they have two drawbacks:

- The elements in the array must be contiguous in memory.
- It is not possible to insert or delete items in the table without recreating the table again.

So we need another data structure known as a **Linked list**.

5.2 Definition

Linked lists are a recursive data structure composed of nodes of the same type, connected to each other by pointers. Unlike arrays, these nodes can be in non-contiguous locations in memory. Linked lists are made up of items (records, nodes, or cells), and each item contains one or more fields to store data and a pointer (link) to the next item in the list.

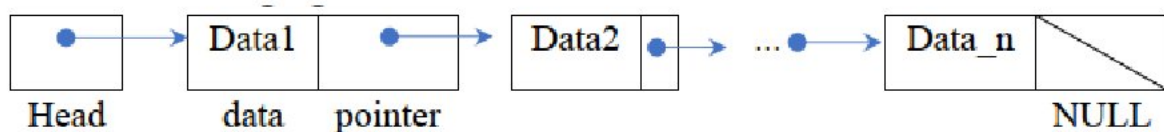
This structure allows you to change its dimension by inserting or removing items from any position in the list. To access any item in the list, you have to start from its header and go through all the items before it, which can take longer than going directly to the items in an array. So, we say that it is a linear data structure as opposed to the array structure that allows random access.

5.3 Representation

In C, a node is represented using "struct" structures, while a header is represented by a pointer.

To simplify the explanation, we use a single integer data field called "data" for all records in the list, instead of using specific data fields for each record type (such as student information, last name, first name, date, etc.).

The following figure illustrates the structure of linked lists:



5.4 The Declaration

Declaring the type of elements or nodes

C	Algorithm
<pre>typedef struct Node { int data; struct Node * next; } Node;</pre>	<pre>Node structure Data : integer next : ^ Node end_structure</pre>

In the structure of linked lists, "data" represents data stored in the list, such as a student's first and last name, the date of an event, and so on. This field can be replaced by any other variable that corresponds to the type of data you want to store in the list.

The "next" field is a pointer that contains the address of the next item in the list, or NULL if it doesn't point to any item. This field is important because it allows the nodes to be linked to each other to form the linked list.

Declaring the Header Type

<code>typedef Node* List;</code>	Type List: ^ Node
----------------------------------	--------------------------

This means that List is the same as Node*.

Example:

<code>List head;</code>	<code>var head: List</code>	A simple pointer-type variable that points to the first element Head <input type="checkbox"/>
<code>Node e1, e2, e3;</code>	<code>e1, e2, e3: Node</code>	3 compound variables of type Node Head <input type="checkbox"/> e1 <input type="checkbox"/> e2 <input type="checkbox"/> e3 <input type="checkbox"/>
<code>e1.data=1;</code> <code>e2.data=2;</code> <code>e3.data=3;</code>	<code>e1.data←1 e2.data←2</code> <code>e3.data←3</code>	Head <input type="checkbox"/> e1 <input type="checkbox"/> 1 e2 <input type="checkbox"/> 2 e3 <input type="checkbox"/> 3
<code>e1.next=&e2;</code> <code>e2.next=&e3;</code>	<code>e1.next←@e2 e2.next←@e3</code>	Head <input type="checkbox"/> e1 <input type="checkbox"/> @e2 → e2 <input type="checkbox"/> @e3 → e3 <input type="checkbox"/> 3
<code>e3.next= NULL;</code> <code>head=&e1;</code>	<code>e3.next← NULL</code> <code>head←@e1</code>	Head <input type="checkbox"/> @e1 → e1 <input type="checkbox"/> 1 @e2 → e2 <input type="checkbox"/> 2 @e3 → e3 <input type="checkbox"/> 3
<code>head->data=4;</code> <code>head->next->data=5;</code>	<code>head^.data←4</code> <code>(head^. next) ^.data←5</code>	Head <input type="checkbox"/> @e1 → e1 <input type="checkbox"/> 4 @e2 → e2 <input type="checkbox"/> 5 @e3 → e3 <input type="checkbox"/> 3
<code>head= head->next;</code> <code>head->data=6;</code>	<code>head← head^.next;</code> <code>head^.data←6;</code>	Head <input type="checkbox"/> @e2 → e1 <input type="checkbox"/> 4 @e2 → e2 <input type="checkbox"/> 6 @e3 → e3 <input type="checkbox"/> 3
<code>head= head->next;</code> <code>head->data=7;</code>	<code>head← head^.next;</code> <code>head^.data←7;</code>	Head <input type="checkbox"/> @e3 → e1 <input type="checkbox"/> 4 @e2 → e2 <input type="checkbox"/> 6 @e3 → e3 <input type="checkbox"/> 7

The -> operation in C language

Since the 'head' pointer points to the 'e1' element, the variable pointed to by 'head' and 'e1' are equivalent, so the expression '(*head).next' can be used to access the 'next' field of the 'head' point element instead of 'e1.next'

In C, we use the '->' operator instead of '(*).' to access the fields of the structure pointed to by 'head'. The expression 'head->next' is therefore equivalent to '(*head).next' to access the 'next'

field of the element pointed to by 'head'

```
e1.next ⇔ (*head).next ⇔ head->next
```

```
e1.data=5; ⇔ (*head).data=5; ⇔ head->data=5;
```

Note that head.data is incorrect because head is a pointer, not a structure.

The 'head', 'e1.next', 'e2.next' and 'e3.next' pointers are all of the same type, which means that it is possible to make assignments between them.

The Last Element

The last item in the list has no next item, so its 'next' pointer is assigned to NULL. When traversing the list, NULL is used to check whether the last item has been reached or not.

```
e3->next= NULL;
```

```
head= NULL; It's an empty list
```

The traversing of a linked list

The following example shows how to navigate through items in a linked list.

Let's say we have the following list:

Since "e1.next" points to "e2", we can make "head" point to "e2" by doing the following:

```
"head =e1.next;"
```

Now that "head" points to "e2", then "head->next" is equivalent to "e2.next".

```
head= &e2 ⇔ head= e1.next ⇔ head= head->next
```

<pre>while (head!= NULL){ do something head = head->next; }</pre>	<pre>while (head#NULL) do Do a southings head←head^.Next end while</pre>
--	--

So, to switch from one node to another, we use "head=head->next"

To access all the items in the list, we repeat the process until head takes the value of next from the last node, which is NULL.

Observation:



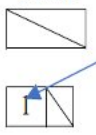
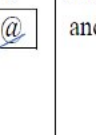
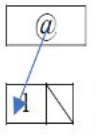
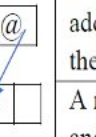

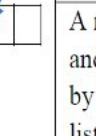
```
while (head!= NULL) ⇔ while (head)
```

5.5 The Creation

To create a linked list, memory is dynamically reserved from a simple pointer variable.

Suppose we have an empty list with head=NULL; To create a new item, we use the

malloc()dynamic memory allocation function.

List e, head= NULL;	var e, head: List head← NULL	Head  e 	Two lists are created (Node* pointer)
e = malloc(sizeof(Node)); e->data=1; e->next= NULL;	Allocation(e,1) e^.data←1 e^.next←NULL	Head  e 	A new "e" element has been created and its fields have been initialized.
head = e;	head ←e	Head  e 	Here, "e and head" have the same address, which means they refer to the same element.
e = malloc(sizeof(Node)); e->data=2;	Allocation(e,1) e^.data←2	Head  e 	A new "e" element has been created and can be added to the top of the list by linking it to the first item in the list with its next field. e->next=head; head=e; or at the end head->next=e

Note in C++

```
e = malloc( sizeof(Node) ); ⇔ e = new Node;
```

6. Operations on Linked Lists

Now we'll create a set of subroutines to manage lists, such as adding or removing an item, displaying all items in the list, searching the list, and so on. It is recommended that you combine all of these functions in a dedicated list library.

Observation

There are several ways to create functions to add or remove an item from the list:

- By using functions that take a list as a parameter and return a list. In this case, the list can be passed by value.
- By using procedures and an auxiliary element (sentry) to avoid passing by address. In this case, the list can be passed by value.
- Using unaided procedures. In this case, the list must be passed by address.

- By using functions that take a list as a parameter and return a boolean value (bool) to tell us whether the operation was successful (true) or not (false). In this case, the list must be passed by address. We will use the latter method.

6.1 Dend iflaying a list

<pre>void dend iflay_list(List head) { while (head != NULL) { printf("%d->", head->data); head = head->next; } printf("fin\n"); }</pre>	<pre>procedure dend iflay_list(Listhead) Begin while (head ≠ NULL) do write(head->data, "->"); head ← head^.next; end while printf("end"); end</pre>
<pre>void dend iflay_list(List head) { if (head){ printf("%d->", head->data); dend iflay_list(head->next); } else printf("fin\n"); }</pre>	

We iterate through each item in the list and dend iflay the associated data. We note here that the list has been passed by value, and so the head of the original list won't be changed if we change the value of head, so we use it to browse the list safely.

6.2 List size

Go through the list and add 1 until we get to NULL

<pre>int size_list(List head) { int n=0; while (head != NULL) { head= head->next; n++; } return n; }</pre>	<pre>int size_list(List head) { if (!head) return 0; return 1+ size_list(head->next); }</pre>	<pre>function size_list(List head): integer var n:integer; Begin n←0 while (head ≠ NULL) do n←n+1; head ← head^.next; end while size_list←n; end</pre>
---	--	--

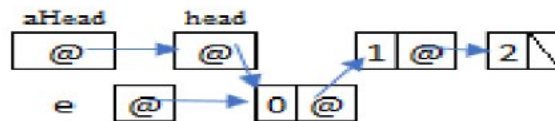
6.3 Add an item to the list

The process of adding an item to a linked list takes place in 3 steps:

1. Create and initialize a node
2. Determine the location of the node.
3. Add the node to the list by reassigning the pointers.

Add an item to the top of the list (at the top)

1. Adding an item to the beginning of the list requires changing the head of the list, so it's important to pass the list by address so that this change is visible in the caller.
2. Create a new item, and if it fails, return false to the caller
3. Initialize Item
4. Replace "next" with "e" to point to the first item in the list
5. Change the head of the list to point to the new item, "aHead and e" are two local variables that are removed immediately after the procedure is executed



<code>int add_head(List* aHead, int d){</code>	<code>function add_head(aHead:^List, d:integer):bool</code>	1
<code> var e:List;</code>	<code> Begin</code>	
<code> List e = malloc(sizeof(Node));</code>	<code> allocate(e,1);</code>	2
<code> if (e == NULL) {</code>	<code> if (e = NULL) then</code>	
<code> return 0;</code>	<code> add_head ←false;</code>	
<code> }</code>	<code> else</code>	
<code> e-> data = d;</code>	<code> e^.data←d;</code>	3
<code> e-> next = *aHead;</code>	<code> e^.next←aHead^;</code>	4
<code> *aHead=e;</code>	<code> aHead^←e;</code>	
<code> return 1;</code>	<code> add_head ←true;</code>	5
<code>}</code>	<code> endif</code>	
	<code> end</code>	

Add an item at the end

1. It's possible that we'll add an element in the header, so we need to pass it by address.
Create anew item

2. Initialize the element and set NULL to the "next" as this will be the last element
3. If the list is empty, we'll insert it in the head
4. If the list contains at least one item, look for the last item
5. Insert Item Last

<pre>int append_end(List*aHead, int d){ List t; List e = malloc(sizeof(Node)); if (e == NULL) { return 0; } </pre>	<pre>function append_end(aHead:^List, d:integer): bool var e, t : List; Begin allocate(e,1); if (e = NULL) then append_end←false; else </pre>	1
<pre>e->data = d; e->next = NULL; </pre>	<pre>e^.data←d; e^.next← NULL; </pre>	2
<pre>if (*aHead == NULL) *aHead = e; </pre>	<pre>if (aHead^=NULL) then aHead^←e; </pre>	3
<pre>else { t= *aHead; while (t-> next != NULL) t= t-> next; </pre>	<pre>else t←aHead^; while (t^.next≠NULL) do t←t^.next; end while </pre>	4
<pre>t-> next=e; } return 1; }</pre>	<pre>t^.next←e; endif append_end←True; endif end</pre>	5

6.4 Remove an item from the list

The process of removing a node from a list takes place in 4 steps:

1. Determine the previous node of the node you want to delete.
2. Keep the address of the node to be deleted in a variable
3. Connect the previous node to the next node of the node you want to delete.
4. Flush the memory reserved by the node you want to delete.

So there are 3 cases, either the list is empty, contains a single item, or contains more than one item.

Delete the element from the beginning (the head)

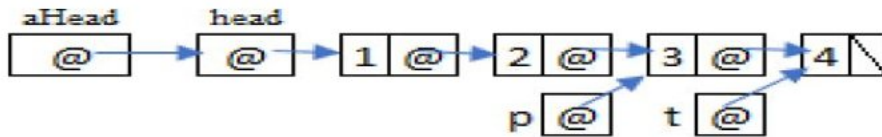
1. It's possible that we'll remove an item from the header, so we need to pass the list by address
2. If the list is empty, there is no item to delete, so we return false.
3. Stores the address of the first item to be deleted in t
4. Connecting the head with the second element
5. Remove the memory reserved by the first item

<code>int delete_head(List*aHead) { List t;</code>	<code>function delete_head(aHead:^List): bool var t:List; Begin</code>	1
<code>if (*aHead== NULL) return 0;</code>	<code>if (aHead^ ==NULL) then delete_head← false; else</code>	2
<code>t = *aHead;</code>	<code>t←aHead^;</code>	3
<code>*aHead =t-> next;</code>	<code>aHead← t^.next;</code>	4
<code>free(t); return 1; }</code>	<code>deallocate (t); delete_head←True; endif end</code>	5

Remove an item from the end

1. It's possible that we're removing an item from the head, so we need to pass the list by address. t is the last element and p is the second-to-last element
2. If the list is empty, there is no item to delete, so we return false
3. If there is only one item in the list, remove it directly from the head

4. If the list contains more than one element, we look for the last element *t* and the second-to-last *p*
5. We set NULL to "next" of the penultimate *p*, because it has become the last, and we remove the last *t* from memory.



<pre>int delete_end (List*aHead) { List t, p; </pre>	<pre>function delete_end(aHead:^List): bool var t, p:List; Begin </pre>	1
<pre> if (*aHead== NULL) return 0; </pre>	<pre> if (aHead^ ==NULL) then delete_end ←false; else </pre>	2
<pre> if ((*aHead)->next ==NULL) { free(*aHead); *aHead = NULL; } </pre>	<pre> if (aHead^.next =NULL) then dealdeal(*aHead); *aHead=NULL; </pre>	3
<pre> else { t = *aHead; while (t->next != NULL) {p=t; t= t->next; } </pre>	<pre> else t←aHead^; while (t^.next ≠ NULL) do p←t t←t^.next; endwhile </pre>	4
<pre> p->next=NULL; free (t); } return 1; }</pre>	<pre> p^.next=NULL; deallocate (t); endif delete_end ←true; endif end</pre>	5

6.5 Delete list

1. We remove from the header until the list becomes empty
2. Or by using the `delete_head` function until it returns false

<pre> void delete_list(List*aHead) { List t; while(*aHead!= NULL) { t = *aHead; *aHead =t-> next; free(t); } } </pre>	<pre> procedure delete_ list(aHead:^List) var t:List; Begin while (aHead^ ≠NULL) do t←aHead^; aHead^←t^.next; deallocate (t); end while end </pre>	1
<pre> void delete_list(List*aHead) { while (delete_head(aHead)); } </pre>	<pre> procedure delete_ list(aHead:^List) Begin while (aHead^ ≠NULL) do delete_head(aHead); end while end </pre>	2

6.6 Main Program (Use)

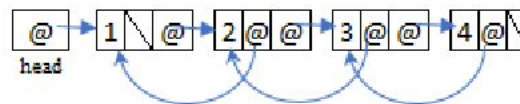
<pre> int main() { List head =NULL; add_head(&head, 3); add_head(&head, 2); append_end(&head, 4); add_head(&head, 1); append_end(&head, 5); printf("size=%d\n", size_list(head)); dend iflay_list(head); delete_head(&head); delete_end(&head); printf("size=%d\n", size_list(head)); dend iflay_list(head); delete_list(&head); printf("size=%d\n", size_list(head)); dend iflay_list(head); return 0; } </pre>	<pre> Begin add_head(@head, 3); add_head(@head, 2); append_end(@head, 4); add_head(@head, 1); append_end(@head, 5); write("size=", size_list(head)); dend iflay_list(head); delete_head(@head); delete_end(@head); write("size=", size_list(head)); dend iflay_list(head); delete_list(@head); write("size=", size_list(head)); dend iflay_list(head); end </pre>
--	---

- The program will print iflay
size=5
1->2->3->4->5->end
- Then it will print iflay
size=3
2->3->4->end
- At the end it will print iflay
size=0
end

7. Double linked list

In addition to the data and the pointer that points to the next item, a doubly linked list contains another pointer, usually called a "prev," that points to the previous item. This pointer makes it easier to navigate through the list in both directions, simplifying the process of removing or inserting an item before the selected one.

The following figure shows the structure of a doubly linked list



7.1 Declaration

<pre> typedef struct Node { int data; struct Node* next, * prev; } Node; </pre>	<pre> Node structure Data: integer next, prev : ^Node end_structure </pre>
--	--

"data" represents the data stored in the list. "Next" is a pointer that contains the address of the next item, while "Prev" is a pointer that contains the address of the previous item.

The add and remove operations are as follows

7.2 Add an element at the beginning (header)

<code>e-> next = *aHead;</code>	<code>e^.next←aHead^</code>	1
<code>e-> prev = NULL;</code> <code>if(*aHead!= NULL)</code> <code>(*aHead)->prev = e;</code>	<code>e^.prev←NULL</code> <code>if(*aHead ≠ NULL) then</code> <code> aHead^.prev←e</code> <code>END IF</code>	2
<code>*aHead=e;</code>	<code>aHead^←e</code>	3

1. Change "next" from "e" to point to the first element
2. It points to NULL (first element)The "prev" of the first element, if it exists, points to the new element.
3. Change the head of the list to point to the new item

7.3 Add an item at the end

<code>e-> next = NULL;</code>	<code>e^.next← NULL;</code>	1
<code>if (*aHead == NULL) {</code> <code>e-> prev = NULL;</code> <code> *aHead = e;</code> <code>}</code>	<code>if (aHead^=NULL) then</code> <code> e^.prev←NULL;</code> <code> aHead^←e;</code>	2
<code>else {</code> <code> t= *aHead;</code> <code> while (t-> next != NULL)</code> <code> t= t-> next;</code>	<code>else</code> <code> t←aHead^;</code> <code> WHILE (t^.next≠NULL) do</code> <code> t←t^.next;</code> <code> END WHILE</code>	3
<code>E-> prev = t;</code> <code>t-> next=e;</code> <code>}</code>	<code>e^.prev←t;</code> <code>t^.next←e;</code> <code>END IF</code>	4

1. NULL because it will be the last element
2. In case the list is empty, it is appended in the header, while prev says NULL
3. If the list contains at least one item, the last item is searched for
4. The prev of the new item refers to the last item in the list. Insert Item Last

7.4 Delete the element from the beginning (the head)

1. Stores the address of the first item to be deleted
2. Bind with the second element. If the list is not empty, the "prev" of the first item must be NULL

3. Flush the memory reserved by the first item

<code>t = *aHead;</code>	<code>t←aHead^;</code>	1
<code>*aHead =t-> next;</code> <code>if(*aHead!= NULL)</code> <code>(*aHead)->prev = NULL;</code>	<code>aHead← t^.next;</code> <code>if(*aHead ≠ NULL) then</code> <code> aHead^.prev← NULL;</code> <code>END IF</code>	2
<code>free(t);</code>	<code>deallocate (t);</code>	3

7.4 Remove an item from the last

1. In case the list contains more than one item, the last item t is searched, and there is no need to save the second-to-last one because it is accessible.
2. We get the second-to-last one by means of "prev" of the last t. We set "next" to NULL of the second-to-last because it has become the last. We remove the last t from memory

<code>t = *aHead->next;</code> <code>while (t->next!= NULL)</code> <code>t= t->next;</code>	<code>t←aHead^;</code> <code>WHILE (t^.next ≠ NULL) do</code> <code> t←t^.next;</code> <code>END WHILE</code>	1
<code>p=t->prev;</code> <code>p ->next=NULL; free(t);</code>	<code>p←t^.prev;</code> <code>p^.next←NULL;</code> <code>deallocate (t);</code>	2

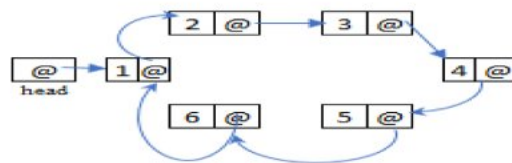
Remark:

The "prev" of the first item can be used to refer to the last item in the list, speeding up the process of accessing the last item for addition or deletion.

8 Special Linked Lists

In addition to linked single and double lists, there are linked single and double circular lists

The circular list is a normal linked list, except that the last item is not NULL, but refers to the first item in the list, as is the case in the double circular list, where the "next" of the last item refers to the first item and the "prev" of the first item refers to the last item.

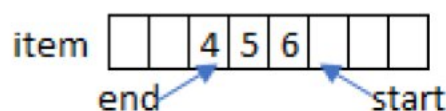
**8.1 Queues**

The **queue** is an abstract data structure that is used to store a set of records of the same type. It offers two essential operations: the addition of a new element, also known as an enqueue, in French: enfileur, and the deletion of an element, known as a deletion (in English: dequeue, in French: défileur). This structure respects the FIFO (First In First Out) property, which means that the first item added is the first item to be deleted. In other words, the output order is the same as the input order.

Example: list of events, queue, list of files sent to the printer, etc.

A queue can be implemented using an array using two indices to track the position of the head (or "start") and the "end" (or "end" in English). When an item is added to the queue, it is placed at the queue

position and the queue index is incremented. Similarly, when an item is removed from the queue, it is removed from the head position and the end hint is also incremented. If the end of the array is reached, you can go back to the beginning of the array to continue adding items if slots are available, or you can allocate a new, larger array and copy the existing items to it.



8.1.1 Using the arrays

1. Declaration: A structure is created that contains a dynamically allocated table of items in memory, a start location "start" for adding, an "end" location for deletion, and "capacity" that contains the number of items that can be added to the table.
2. init(): The table is created and set to -1 to start and end to indicate that the queue is empty. If the creation process fails, the function returns false.
3. isEmpty(): The queue is empty if "start" and "end" are -1.
4. isFull(): The queue is full if the value of "start+1" is the same as the value of "end", and we use

mod "%" if we reach the end of the table to bring it back to the beginning.

5. enqueue(): Makes sure the queue is not full, then adds 1 to start and adds x to the table.
6. dequeue(): Returns the first element of the array to which end points, and adds 1 to end. In case the queue is empty, it informs the user.

Note:

Normally, when a function encounters an error or unexpected behavior situation, it is not expected to return a value that could be erroneous or misinterpreted by the calling function. Instead, it must throw an exception (error) that will be caught and handled in the calling function or in another function in the call stack. By explicitly flagging the error, the exception helps identify the source of the problem and makes it easier to find and resolve the problem.

In C++ you can write

```
if(isEmpty(Q)) throw -1;
```

<pre>typedef struct Queue{ int *item; int start, end, capacity; }Queue;</pre>	<pre>Structure Queue item:^integer; start, end, capacity: integer; end_structure</pre>	1
<pre>bool init(Queue *Q, int capacity) { Q->start = -1; Q->end = -1; Q->capacity= capacity; Q->item= (int*)malloc(sizeof(int)*capacity); return Q-> item != NULL; }</pre>	<pre>function init(Q:^Queue,capacity:integer):bool Begin Q^.start ←-1Q^.end ←-1; Q^.capacity = capacity; allocate(Q^.item, capacity); init← Q^.item ≠ NULL; end</pre>	2
<pre>bool isEmpty(Queue Q){ return Q.start ==-1 && Q.end==-1; }</pre>	<pre>function isEmpty(Q: Queue):bool Begin isEmpty← Q.start ==-1 and Q.end=-1 end</pre>	3
<pre>bool isFull(Queue Q){ return (Q.start+1)% Q.capacity == Q.end; }</pre>	<pre>function isFull(Q:Queue):bool Begin isFull←(Q.start+1) mod Q.capacity = Q.end end</pre>	4
<pre>void enqueue (Queue Q,int x){ if(isFull(Q)){ printf("error: Queue is full"); return; } if(isEmpty(Q)) Q.start=Q.end=0; else Q.start= (Q.start +1)% Q.capacity; Q.item[Q.start]=x; }</pre>	<pre>Procedure enqueue(Q:Queue,x:integer) Begin if(isFull(Q))then write("error: Queue is full"); else if isEmpty(Q) then Q.start←0 Q.end←0; else Q.start←(Q.start+1) mod Q.capacity END IF Q.item[Q.start] ←x; END IF end</pre>	5

<pre> int deQueue(Queue Q) { if(isEmpty(Q)) { printf("error: Queue is empty"); return; } int x= Q.item[Q.end]; if (Q.start == Q.end) Q.start = Q.end= -1; else Q.end = (Q.end+1)% Q.capacity; return x; } </pre>	<pre> Function deQueue (Q: Queue): Integer Begin if(isEmpty(Q)) then write("error: Queue is empty"); deQueue ← -1; else deQueue ← Q.item[Q.end]; if Q.start = Q.end then Q.start ← -1; Q.end ← -1; else Q.end ← (Q.end+1) mod Q.capacity END IF END IF end </pre>	6
---	--	---

8.1.2 Using Linked Lists:

A simple queue implementation using arrays can cause performance issues if the queue is large or heavily used, as each insertion or deletion may require moving any remaining items in the array to maintain FIFO ownership. To avoid these problems, it's best to use higher-performance data structures such as linked lists.

To simulate a queue using lists, it is necessary to add and remove items at two different ends of the list. For example, you can add new items at the end of the list and remove items at the beginning of the list. This approach can also be reversed, by adding items at the beginning of the list and removing items at the end. In both cases, the list structure allows for quick and efficient insertions and deletions, without the need for costly item moves as with the array implementation.

1. Declaration: We create a structure that contains two fields, the first referring to the first item in the list and the second referring to the last item in the list.
2. init(): by assigning a null to first and last.
3. isEmpty(): An empty Queue is an empty list.
4. enqueue(): is the same as the "append_end" function, and to avoid going through all the items in the list to get to the last item, we always store the address of the last item in Q.last. In the case where the list is empty, we add the new element to the start and last, but if it is not empty, we paste the new element with the last element, and then change last so that it points to the new element.
5. dequeue(): is the same as the "delete_head" function, except that the dequeue function returns the element that was deleted. So before we delete element t, let's save t-> data in

x, and then delete it and return the value of x.

<pre>typedef struct Queue{ struct Node* first, *last; int size; } Queue;</pre>	<pre>Structure Queue first, last:^Node; Size :Integer; end_structure</pre>	1
<pre>Queue * init (){ Queue *Q=new Queue; Q->first =NULL; Q->last =NULL; Q->size =0; return Q }</pre>	<pre>procedure init (Q: ^Queue) Begin Q^.first← NULL; Q^.last← NULL; end</pre>	2
<pre>bool isEmpty(Queue *Q){ return Q->size==0; }</pre>	<pre>function isEmpty(Q: Queue):bool Begin isEmpty← Q.first= NULL; end</pre>	3
<pre>void enqueue(Queue *Q, int x) { Node *e = malloc(sizeof(Node)); e->data = x; e->next = NULL; if(isEmpty(Q)) Q->first =e; else Q->last->next=e; Q->last =e; Q->size++; }</pre>	<pre>procedure enqueue(var Q: Queue, x:integer) var e:^Node; Begin allocate(e,1); e^.data← x; e^.next← NULL; if (isEmpty(Q)) then Q.first←e; else Q.last^.next←e; END IF Q.last←e; end</pre>	4
<pre>int dequeue(Queue *Q) { Node* t; int x; if(isEmpty(Q)){ printf("error: Queue is empty"); exit(1); } t = Q->first; x = t ->data; Q->first = t-> next; free(t);</pre>	<pre>function ofQueue(var Q: Queue): integer var t:^Node; Begin if (isEmpty(Q)) then printf("error: Queue is empty"); deQueue←-1; else t← Q.first; deQueue ← t^.data;</pre>	5

<pre>return x; }</pre>	<pre>Q.first← t^.next ; deallocate (t); END IF end</pre>	
------------------------	---	--

8.2 Stacks:

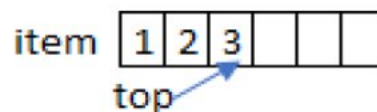
It is an abstract data structure consisting of a set of records of the same type, in which only two operations can be performed: the addition of a new element, this process is called push, and the removal of an element from the group, this process is known as pop, and these operations take place at a single end of the group called the top. This data structure has the characteristic of LIFO (Last In First Out), i.e., the last item added is the first to be removed, and the output order is therefore the opposite of the input order.

Example:

Stacks are commonly used in situations where data needs to be processed in reverse order to the order in which it was received, such as:

- Memory Management in Computer Systems
- The web browser saves the list of visited pages in a stack.
- The list of operations in Word, for example, is stored in a stack and is used to undo changes.

Stacks can be implemented using arrays or linked lists, but linked lists are often preferred because they provide more predictable performance when adding or removing items.



8.2.1 Using the table

1. Declaration: A structure is created that contains a dynamically allocated item element table, and top the add or remove location and capacity that represents the size.
2. init(): The table is created and set to top to 0 to indicate that the stack is empty.
3. isEmpty(): The stack is empty if the top value is 0.
4. isFull(): If the array is full, top equals capacity.
5. 1. Pop(): The Pop function allows you to decrement 'top' and return the last element it points to.
6. Push (): The Push function allows you to add the x element to the table and increment

the 'top' pointer by 1. You need to make sure that the stack (table) is not full.

<pre>typedef struct Stack{ int *item; int top, capacity; } Stack;</pre>	<pre>structure Stack item:^integer; Top, size:integer; end_structure</pre>	1
<pre>bool init(Stack *s, int capacity) { s->top = 0; s-> capacity = capacity; s->item=malloc(sizeof(int)*size); return s->item != NULL; }</pre>	<pre>function init(s:^ Stack, capacity:integer):bool Begin s^.top←0; s^. capacity ← capacity; allocate(s^.item, capacity); init← s^.item ≠ NULL; end</pre>	2
<pre>bool isEmpty(Stack s){ return s.top==0; }</pre>	<pre>function isEmpty(s:Stack):bool Begin init←s.top=0; end</pre>	3
<pre>bool isFull(Stack s){ return s.top==s.capacity; }</pre>	<pre>function isFull(s:Stack):bool Begin isFull←s.top= s.capacity; end</pre>	4
<pre>int Pop(Stack s){ int x; if(isEmpty(s)){ printf("error: Stack is empty"); exit(1); } s.top--; x=s.item[s.top]; return x; }</pre>	<pre>function Pop(s: Stack): integer Begin if(isEmpty(s)) then write("error: Stack is empty"); Pop←-1; else s.top← s.top -1; Pop←s.item[s.top]; END IF end</pre>	5
<pre>void Push(Stack s,int x){ if(isFull(s)){ printf("error: Stack is full"); exit(1); } s.item[s.top]=x; s.top++; }</pre>	<pre>procedure Push(s: Stack, x: integer) Begin if (isFull(s)) then write("error: Stack is full"); else s.item[s.top] ←x; s.top← s.top +1; END IF end</pre>	6

8.2.2 Using Linked Lists:

To simulate a stack using lists, the addition and deletion must be done on the same side (at the beginning or at the end).

1. isEmpty(): An empty stack is an empty list.
2. Pop(): The pop function is the same as the delete_head function except that the pop function returns the item that was deleted. So before we delete the first element t, we save t-> data in x, and then delete it and return the value of x.
3. Push(): The push function is the same as the add_head function

<pre>bool isEmpty(List head){ return head==NULL; }</pre>	<pre>function isEmpty(head: List):bool Begin isEmpty ← head= NULL; end</pre>	1
<pre>int Pop(List*aHead) { List t; int x; if(*aHead==NULL){ printf("error: Stack is empty"); exit(1); } t = *aHead; *aHead =t-> next; x=t->data; free(t); return x; }</pre>	<pre>function Pop(aHead:^List): integer var t:List; Begin if(isEmpty(aHead^)) then write("error: Stack is empty"); Pop←-1; else t←aHead^; aHead← t^.next; Pop← t^.data; deallocate (t); END IF end</pre>	2
<pre>void Push(List* aHead, int x) { List e = malloc(sizeof(Node)); e-> data = x; e-> next = *aHead; *aHead=e; }</pre>	<pre>procedure Push(aHead:^List,x:integer) var e:List; Begin allocate(e,1); e^.data← x; e^.next←aHead^; aHead^←e; end</pre>	3

9. Conclusion

Pointers, dynamic memory management, linked lists, operations on linked lists, doubly linked lists, and special linked lists were all covered in this chapter.

References

1. Thomas H. Cormen, Algorithmes Notions de base Collection : Sciences Sup, Dunod, EAN EBOOK : 9782100702909, 2013.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest Algorithmique - 3ème édition - Cours avec 957 exercices et 158 problèmes Broché, Dunod, 2010, ISBN-10 : 2100545264.
3. Damien Berthet, Vincent Labatut. Algorithmique & programmation en langage C - vol.1. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.232. cel-01176119v1
4. Damien Berthet et Vincent Labatut. Algorithmique & programmation en langage C - vol.2 : Sujets de travaux pratiques. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.258. <cel-01176120>
5. <https://elearning.univ-msila.dz/moodle/course/view.php?id=10217&lang=fr>
6. Damien Berthet et Vincent Labatut. Algorithmique & programmation en langage C - vol.3 : Corrigés de travaux pratiques. Licence. Algorithmique et Programmation, Istanbul, Turquie. 2014, pp.217. <cel-01176121>
7. John Henry Mueller, Luca Massaron. Les algorithmes pour les Nuls grand format, Pour les nuls, 2017, ISBN-13978-2412025901.
8. Luca Massaron, John Paul Mueller, Jean-Pierre Cano, Marc Rozenbaum. Les Algorithmes pour les Nuls - 2e édition, Pour les nuls, 2024, ISBN-13978-2412094860.
9. Alan Grid, La Pogramation pour les Débutants Absolus, Independently published, 2023, ISBN-13979-8378538997.
10. Rémy Malgouyres, Rita Zrouer et Fabien Feschet. Initiation à l'algorithmique et à la programmation en C : cours avec 129 exercices corrigés. 2ième Edition. Dunod, Paris, 2011. ISBN : 978-2-10-055703-5.