GENERAL PRESENTATION

I. DEFINITION OF PROGRAMMING

Programming is the translation of an algorithm developed for problem-solving. In general, the goal of programming is to automate a set of tasks. Programming involves breaking down the task to be automated into a sequence of instructions using appropriate data. A programming language is a formal means of describing processes (i.e., tasks to be performed) in the form of programs (i.e., sequences of high-level instructions and data, understandable by the programmer). There must be a translator that allows the actual execution of programs by a computer. The syntax (rules for writing programs) and semantics (instruction definitions) of a programming language need to be specified precisely, usually in a reference manual.

II. C LANGUAGE

C is a language developed in 1970 by Dennis Ritchie at AT&T's Bell Laboratories. The success in the years that followed and the development of C compilers made it necessary to define an updated and more precise standard. In 1978, the duo of Brian W. Kernighan and Dennis M. Ritchie published the classic definition of the C language (known as the K&R-C standard) in a book titled 'The C Programming Language.' In 1983, the American National Standards Institute (ANSI) commissioned a committee to develop an explicit and machine-independent definition of the C language. The second edition of the book, published in 1988, fully adheres to the ANSI-C standard and has since become the programmer's bible for C.

III. Libraries

Practicing C programming requires the use of function libraries. A library is a set of primitives and/or functions that deal with homogeneous operations. These libraries are available in precompiled form (extension ".LIB"). To use them, header files (extension ".H") must be included in the programs. These files contain prototypes of functions defined in the libraries and create a link between the precompiled functions and the programs. The #include statement inserts the specified header files as arguments into the program's text at compile time. C programming works with different types of files identified by their extensions:

Extension	Description
.C	Source files
.OBJ	Compiled files (object versions)
.EXE	Compiled and linked files (executable versions)
.LIB	Precompiled function libraries
.H	Header files

After compilation, the precompiled library functions will be added to the program to form an executable version.

Examples:

<math.h>: For common mathematical functions. The 1999 version of the C language added many mathematical functions, especially to comply with the IEC 559 standard, also known as IEEE 754.

<stdio.h>: Provides C language input/output functions.
<stdlib.h>: For performing various operations like conversion, generating pseudo-random numbers, memory allocation, process control, environment and signal management, searching, and sorting.
<string.h>: For manipulating strings.
<time.h>: For converting between different date and time formats.

BASIC ELEMENTS OF THE C LANGUAGE

Variables and constants are the fundamental data that can be manipulated by a program. Operators control the actions that data values undergo. To produce new values, variables and constants can be combined using operators in expressions.

I. STRUCTURE OF A C PROGRAM

Like any program in any programming language, a C program must consist of a declarative part and a program body.

Structure

```
[Include statements for library inclusion]
[Definition of constants]
[Declaration of global variables]
main( ) {
[Declaration of local variables]
[Sequence of statements]
return 0;
}
```

with:

- Include is used to call a library that will be used in the program body.
- A constant is a memory space with unchangeable content.
- A variable is a memory space with changeable content.
- Each statement must end with a semicolon.
- The main keyword indicates the beginning of the main program. This is called the main function in C or the entry point. The main function can have no type, in which case it defaults to int.
- The return 0 statement indicates to the environment that the program has completed successfully, without errors or fatal anomalies. This statement is required if the main function has an int type.

Example

Here is a C program that displays the message "Hello, friends."

```
#include <stdio.h>
main( )   {
   printf("Hello, friends ");
   return 0;
}
```

The pseudo-instruction #include <stdio.h> calls the "stdio.h" library, which handles input/output operations.

Note

The main function can be of type void, and in that case, there is no need for a return statement.

Example
Here's the previous example that uses void as the type for the main function.

```
#include <stdio.h>
void main( )   {
   printf("Hello, friends ");
}
```

II. COMMENTS

A comment is an annotation that will not be compiled by the system and is used for potential explanations. It can begin with one of the following symbols:

- '//' to indicate that the rest of the line is used for a comment.
- '/*' and ends with '/*' to indicate that the comment text spans one or more lines.

Examples

```
// This is a single-line comment

/* This is a second
valid comment */

/* This is /* obviously */ forbidden */
```

Note
It is not allowed to use nested comments.

III. SIMPLE TYPES

3.1. Integer Type

Before using a variable, we need to consider two basic characteristics:

- The range of possible values.
- The number of bytes reserved for a variable.

The table below summarizes the characteristics of some C integer types:

Definition	Minimum Range	Maximum Range	Number of Bytes
char	-128	127	1
short	-32768	32767	2
int	-2147483648	2147483647	4
long	-2147483648	2147483647	4

3.2. Real Type

In C, you have a choice between two rational types: float and double. The table below presents the characteristics of each type:

Definition	Number of Bytes
float	**4**
double	**8**

Note
In C, there is no special type for logical variables. Numeric types can be used to express logical operations:
logical false <=> numeric value zero
logical true <=> any value other than zero
Logical operations in C always return int results: 0 for false, 1 for true.

IV. IDENTIFIERS

An identifier can be the name of a variable, constant, or function. An identifier consists of a sequence of letters (uppercase and lowercase), digits, and/or the underscore character ('_'). The first character must be a letter. C distinguishes between uppercase and lowercase letters, so 'VariableName' is different from 'variablename'. The length of identifiers is not limited, but C only distinguishes the first 31 characters.

Note
It is not recommended to use '_' as the first character for an identifier because it is often used to define global variables in the C environment.

Examples

Valid Identifiers	Invalid Identifiers
name1	1name
name_2	name.2
_name_3	-name-3

The declaration of an identifier specifies its type and name.
Example

int a;

This reserves a static memory location relative to the identifier to store its value. Identifiers can be of one of the following three types:

a. Variables
Variables contain values used during program execution. Variable names are valid identifiers.

Example
Here's a C program that calculates and displays the sum of two variables, A (initialized to 20) and B (initialized to 35).

```
#include <stdio.h>
main( )   {
    int A ,B, Sum ;
    A = 20 ;
    B = 35 ;
    Sum = A + B;
    printf("The sum is: %d", Sum);
    return 0;
}
```

Variable declarations allow you to:

- Introduce the variables that will be used.
- Specify their types.
- Specify the allowed operations.
- Reserve the necessary memory space.

Syntax

```
typeBase varName1, varName2, ..., varNameN;
```

Declaring a variable reserves a memory location capable of holding a value whose type is compatible with the declared typeBase. The value of this variable can change during program execution.

Examples

```
int   x, X, nb_students;
x = 3;
X = 7;
nb_students = 30;
float  x1;
x1 = 2;
double Surface;
Surface = 23.45;
```

Variables can be initialized during their declaration:

```
int   MAX = 1023;
char   TAB = 'a';
float   X = 1.05;
```

b. Constants

In practice, we often use constant values for calculations, to initialize variables, to compare them with variables, etc. In these cases, the computer must assign a type to constant values based on their values. To predict the result and the exact type of calculations, it is important for the programmer to know the rules by which the computer chooses types for constants.

The declaration of a constant reserves a memory location capable of holding a value whose type is deduced from the value assigned to it. The value of this constant cannot change during program execution.

Example

Constant	Assigned Type
a=1	Integer
a=1.2	Real
a=’x’	Character
a="abc"	String

A constant can be defined in two ways:

1. A global constant, defined using the preprocessor directive #define.

Syntax:

#define constName value

Examples:

```
#define PI 3.14
#define result "success"
```

2. A local constant defined within the program body, using the const keyword.

Syntax:

const type constName = value;

By using the const keyword, we can specify that the value of a variable does not change during the execution of a program.

Example:

```
const int MAX = 100;
```

c. Functions

In C, a function, identified by a name, is defined by a block of instructions enclosed within curly braces { }. This concept will be discussed in detail in Chapter : "Functions."

V. STANDARD OPERATORS

5.1. Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on variable values.

Operator	Description	Effect	Example	Result
=	Assignment Operator	Assigns a value to a variable	x=7	Sets the value 7 in variable x
+	Addition Operator	Adds two values	x+3	10
-	Subtraction Operator	Subtracts two values	x-3	4
*	Multiplication Operator	Multiplies two values	x*3	21
/	Division Operator	Divides two values	x/3	2.3333333
%	Modulus Operator	Assigns the remainder of the division to a variable	x%3	1

5.2. Assignment Operators

These operators simplify operations such as adding a value to a variable and storing the result back in the same variable. Instead of writing x = x + 2, you can use assignment operators to write x += 2. With these operators, if the initial value of x is 7, it will be 9 after the operation.

Other assignment operators include:

Operation	Effect
opd1 **+=** opd2	Adds the values of both operands opd1 and opd2 and stores the result in opd1.
opd1 **-=** opd2	Subtracts the value of operand opd2 from opd1 and stores the result in opd1.
opd1 ***=** opd2	Multiplies the values of both operands opd1 and opd2 and stores the result in opd1.
opd1 **/=** opd2	Divides the value of opd1 by opd2 and stores the result in opd1.

5.3. Increment and Decrement Operators

These operators are used to easily increase or decrease a variable's value by one. They are particularly useful for loop structures that require a counter (a variable that increments or decrements by one). The x++ operator can replace cumbersome notations like x = x + 1 or x += 1. These operators, ++ and --, are used in the following cases:

- Incrementing or decrementing a variable. In this case, there is no difference between the prefix notation (++I / --I) and the postfix notation (I++ / I--).

Operator	Description	Effect	Syntax	Résult (with x=7 initially)
++	Increment	Increments the variable by one	x++	8
			++x	8
--	Decrement	Decrements the variable by one	x--	6
			--x	6

- Incrementing or decrementing a variable while simultaneously assigning its value to another variable. In this case, the chosen notation (prefix or postfix) affects the assignment's result. In particular:
  - The prefix notation modifies the value of the variable and then assigns the new value to the second variable. Both operands are modified.
  - The postfix notation assigns the old value of the first variable to the second one and then increments the first variable.

Here, you need to choose between prefix and postfix notation:

Operation	Description	Résult (with x=7)

y = x++	First passes the value of x to y and then increments x.	y=7
		x=8

y = x--	First passes the value of x to y and then decrements x.	y=7
		x=6

y= ++x	Increments x and then passes the new value to y.	x=8
		y=8

y = --x	Decrements x and then passes the new value to y.	x=6
		y=6

5.4. Comparison Operators

Comparison operators are used to compare two operands or an operand with a value. The result of the comparison is numeric: 1 if the comparison is true, and 0 otherwise.

Operator	Description	Effect	Example	Résult (with x = 7)
= =	Equality Operator	Compares two operands for equality	x==3	Returns 1 if x is equal to 3, 0 otherwise
<	Strictly Less Than Operator	Checks if the left operand is strictly less than the right one	x<3	Returns 1 if x is less than 3, 0 otherwise
<=	Less Than or Equal To Operator	Checks if the left operand is less than or equal to the right one	x<=3	Returns 1 if x is less than or equal to 3, 0 otherwise
>	Strictly Greater Than Operator	Checks if the left operand is strictly greater than the right one	x>3	Returns 1 if x is greater than 3, 0 otherwise
>=	Greater Than or Equal To Operator	Checks if the left operand is greater than or equal to the right one	x>=3	Returns 1 if x is greater than or equal to 3, 0 otherwise
!=	Inequality Operator	Checks if the left operand is different from the right one	x!=3	Returns 1 if x is different from 3, 0 otherwise

5.5. Logical Operators (Boolean)

These operators are used to check if one or more conditions are true. A condition can contain one or more sub-conditions separated by one of these operators. The final result of the overall condition is evaluated as true or false.

Operator	Description	Effect	Syntax
||	Logical OR	Checks if at least one of the conditions is true	((condition1)||(condition2))
&&	Logical AND	Checks if all conditions are true	((condition1)&&(condition2))
!	Logical NOT	Returns the inverse of a condition (returns 1 if the condition is 0, and 0 if it's 1) without changing its value	(!condition)

5.6. Type Conversions

The great flexibility of the C language allows for mixing data of different types in the same expression. Before performing calculations, data must be converted to a common type. Most of these conversions happen automatically, without the programmer's intervention. Sometimes, it is necessary to convert data to a different type than what automatic conversion would choose. In such cases, we need to force the conversion using a special operator "( )".

a. Automatic Type Conversions

If an operator has operands of different types, the values of the operands will be automatically converted to a common type. These implicit conversions usually involve converting smaller types to larger types to avoid precision loss.

During an assignment, the data on the right side of the equals sign is converted to the type of the data on the left side.

Example: Consider the following calculation:

```
int I = 8;
float X = 12.5;
float Y;
Y = I * X;
```

To be multiplied with X, the value of I is converted to a float (the widest type between int and float). The result of the multiplication is of type float. We get the result: Y = 100.000000.

In this case, there may be a loss of precision if the destination type is narrower than the source type.

Example: Consider the following calculation:

```
int I = 3;
float X = 12.5;
int Y;
Y = I * X;
```

We get the result: Y = 37 instead of 37.5.

Rules for Automatic Conversions

Automatic conversions used during arithmetic operations can follow one of the following rules:

- Char and short types are converted to int.

  char → int

  short → int
- Integer type is converted to a floating-point type.

  int → float
- For real numbers, the computer chooses the wider of the two types following this hierarchy:

  float → double
- During an assignment, the result is always converted to the type of the destination. If this type is narrower, there can be a loss of precision.

Example: Observe the necessary conversions during a simple division:

```
int X;
float A = 12.48;
char B = 4;
X = A / B;
```

A is a float, and B is an int. A/B should result in a float after converting B to a float. However, when the float result is assigned to the variable X, which is of type int, the digits after the decimal point are truncated, leading to the result X = 3.

b. Forced Type Conversions

This type of conversion is also called "casting" or "typecasting." It is possible to explicitly convert a value to any type by forcing the transformation using the ( ) operator.

Syntax

(Type) Expression;

Example

If we want to divide two integer variables and obtain a real (float) result, we need to convert the value of one of the operands to float. Automatically, C will convert the value of the other operand to float and perform a real division:

```
char A = 3;
int B = 4;
float C;
C = (float)A / B;
```

The value of A is explicitly converted to float. The value of B is automatically converted to float. The result of the division (a floating-point type with a value of 0.75) is assigned to C.

Result: C = 0.75

Note

The types of A and B remain unchanged; only the values used in the calculations are converted.

VI. PRIORITY OF OPERATORS

When multiple operators are used in the same expression, it's important for the system to know the order in which to process them. Therefore, here are the priorities of all operators in descending (predefined) order:

Priority	Operators					
+++++++++	()					
++++++++	--	++	!			
+++++++	*	/	%			
++++++	+	-				
+++++	<	<=	>=	>		
++++	==	!=				
+++	&&					
++	||					
+	=	+=	-=	*=	/=	%=

In each priority class, operators have the same priority. If there is a sequence of binary operators in the same class, evaluation occurs from left to right in the expression.

For unary operators (!, ++, and --) and assignment operators (=, +=, -=, *=, /=, %=), evaluation occurs from right to left in the expression.

VII. INPUT/OUTPUT OPERATIONS

The standard library <stdio.h> contains a set of functions that facilitate communication between the computer and the external world.

7.1. Formatted Output of Data

The printf function is used to transfer text, variable values, or expression results to the screen.

Syntax

```
printf("format", Expr1, Expr2, ..., ExprN);
```

Where:

format: the format of representation.

Expr1, Expr2, ..., ExprN: variables and expressions whose values are to be displayed.

The "format" part is actually a string of characters that can contain:

- text,
- escape sequences (\n, \t, \a, \b),
- and/or format specifiers (%d, %f, ...).

The format specifiers for printf are summarized in the following table:

Symbol	Type	Description
%d	int	signed integer
%u	int	unsigned integer
%o	int	octal integer
%x	int	hexadecimal integer
%c	int or char	character
%f	double	floating-point number in decimal notation
%e	double	floating-point number in scientific notation
%s	char*	string of characters

For each expression to be evaluated and displayed, there must be a corresponding format specifier. The format specifier should be inserted into the format string where you want to display the result of the corresponding expression.
The format can be written in one of the following three cases:

- Displaying a message.

Example: printf("Here is my first line");

- Displaying a message and a variable.

Example: printf("The result is %d", R);

- Displaying multiple elements (messages, variables, and expressions).

Example: printf("The sum of %d and %d is equal to %d", A, B, A + B);
Notes:

- The expression "printf("%10f\n", x);" indicates that there will be display of a float with 10 characters, including the decimal point.
- The expression "printf("%10.3f\n", x);" indicates that there will be display of a float with 10 characters, including 3 digits after the decimal point and the decimal point itself.

7.2. Formatted Data Input

The scanf function is the counterpart of printf and is used for inputting data values. It provides us with nearly the same format specifiers as printf but in reverse.
Syntax

scanf("format", var1, var2, ..., varN);

With:

- format: the format for reading the data.
- vari: variables to which the data will be assigned.

The scanf function receives data from the keyboard. The format string determines how the received data should be interpreted. The data received correctly is stored successively in the order of the variables specified by vari.
The format specifiers for scanf are the same as those used in the printf function.
Example 1

```
char month[10];
scanf("%s", month);
```

If the variable to be read is numeric, you must precede its name with the & operator to specify its address.
Example 2
Consider the following two statements:

```
int DAY, MONTH, YEAR;
scanf("%d %d %d", &DAY, &MONTH, &YEAR);
```

The scanf statement reads three integer values, which are assigned to the variables DAY, MONTH, and YEAR, respectively. To input the three values, the user can use the Enter key after entering each value. The values can also be separated by tabs or spaces.

CONDITIONAL STRUCTURES

I. ALTERNATIVE STRUCTURES

1.1. Simple Alternative Structures

Syntax of the reduced form

```
if ( expression )
Block of Instructions
```

- If the expression yields a non-zero value, then the Block of Instructions is executed.
- If the expression yields a value equal to zero, then the Block of Instructions will not be executed.

Syntax of the complete form

```
if ( expression ) {
Block of Instructions1
}
else {
Block of Instructions2
}
```

- If the expression yields a non-zero value, then Block of Instructions1 is executed.
- If the expression yields a value equal to zero, then Block of Instructions2 is executed.

The expression part can refer to:

- A variable of a numeric type,
- Or an expression that yields a numeric result.

The Block of Instructions part can refer to:

- A block of instructions enclosed in curly braces, with each instruction terminated by a semicolon,
- Or a single instruction terminated by a semicolon.

Examples:

1-

```
if (N > 0)
      printf("N is positive \n");
```

2-

```
if (a > b)
     max = a;
else
     max = b;
```

3-

```
if (N != 0)
    printf("%d is not zero \n", N);
else
   printf("%d is zero \n", N);
```

This instruction is equivalent to:

```
if (N)
printf("%d is not zero \n", N);
else
printf("%d is zero \n", N);
```

1.2. Nested Alternative Structures

The Block of Instructions1 and/or Block of Instructions2 can, in turn, be an alternative structure.

Syntaxes:

```
if ( expression1 )
         if ( expression2 ) {
               Block of Instructions11
         }
        [else {
               Block of Instructions12
         }
        ]
[else {
        Block of Instructions2
}
]
```

or

```
if ( expression1 ) {
         Block of Instructions1
}
[else
         if ( expression2 ) {
                  Block of Instructions21
         }
         [else {
                 Block of Instructions22
         }
         ]
]
```

Example 1:

Consider the two parts of the following programs that display the sign of a number N. Both parts are equivalent.

```
if (N >= 0)
if (N > 0)
printf("%d is positive \n", N);
else
printf("%d is zero \n", N);
else
printf("%d is negative \n", N);
```

```
if (N > 0)
printf("%d is positive \n", N);
else
if (N < 0)
printf("%d is negative \n", N);
else
printf("%d is zero \n", N);
```

Example 2:

Consider the following part of a program that tests if a given value N is both divisible by 2 and 5.

```
if (N % 2 == 0 && N % 5 == 0)
printf("The number is both divisible by 2 and 5 ");
else
printf("The number is not both divisible by 2 and 5 ");
```

or

```
if (N % 2 == 0) {
if (N % 5 == 0)
printf("The number is both divisible by 2 and 5 ");
}
else
printf("The number is not both divisible by 2 and 5 ");
```

with the requirement of using curly braces for the inner if to properly associate the else with the first if.

Note:
In C, an else part is always associated with the last if that does not have an else part. To avoid confusion and to enforce a certain interpretation of an expression, it is recommended to use curly braces { }.

II. SELECTIVE STRUCTURES

The switch statement defines a branching mechanism that allows jumping to a label based on the value of an expression.
Syntax:

```
switch (expression) {
    [case label1: list of instructions1; break;]
    [case label2: list of instructions2; break;]
    …
   [case labeln: list of instructionsn; break;]
   [default: list of instructions]
}
```

With:

- Case labels labeli must be scalar-type expressions (integer, character, logical, enumeration).
- Once the branching is executed, execution continues by default until the end of the switch block. The break statement is used to force an exit from the block.
- The default block is not mandatory. It will be executed if the input value does not match any of the cases.

Example:
Consider the following C program that displays the name of the day based on the number given by the user:

```
#include <stdio.h>
int main() {
   int day;
   printf("Enter a day number: ");
   scanf("%d", &day);
   switch (day) {
      case 1: printf("Monday\n"); break;
      case 2: printf("Tuesday\n");
      case 3: printf("Wednesday\n"); break;
      case 4: printf("Thursday\n");
      case 5: printf("Friday\n");
      case 6: printf("Saturday\n");
      case 7: printf("Sunday\n");
      default: printf("Invalid day\n");
   }
   return 0;
}
```

REPETITIVE STRUCTURES

In theory, repetitive structures are interchangeable. However, it is strongly recommended to always choose the structure that best suits the case at hand.

I. WHILE Structure

The while structure corresponds perfectly to the "tant que" structure in algorithmic language.

Syntax:

```
while (expression) {
   statementBlock
}
```

With:

- As long as the expression yields a non-zero value, the statementBlock is executed.
- If the expression yields zero, execution continues with the instruction following the statementBlock (after the closing curly brace).
- The statementBlock can be executed zero or multiple times.

The expression part can refer to:

- A variable of a numeric type,
- An expression that provides a numeric result.

The statementBlock part can refer to:

- A block of instructions enclosed in curly braces, with each instruction ending with a semicolon,
- A single instruction terminated by a semicolon.

Example:

Consider a program that displays numbers from 0 to 9 as follows:

```
#include <stdio.h>
main() {
   int I = 0;
   while (I < 10) {
      printf("%d \n", I);
      I++;
   }
   return 0;
}
```

This same program can be written as follows:

```
#include <stdio.h>
main() {
   int I;
   I = 0;
   while (I < 10)
      printf("%d \n", I++);
   return 0;
}
```

II. DO-WHILE Structure

The do-while structure is similar to the "Répéter .. Jusqu'à" structure in algorithmic language. The do-while structure evaluates the condition after executing the block of instructions.

Syntax:

```
do {
   statementBlock
} while (expression);
```

The statementBlock is executed at least once and as long as the expression yields a non-zero value.

A typical application of do-while is for input validation where a data entry must satisfy a certain condition.

Example:

Write a program that allows entering an integer value between 1 and 10 as follows:

```
#include <stdio.h>
main() {
   int N;
   do {
      printf("Enter a number between 1 and 10: ");
      scanf("%d", &N);
   } while (N < 1 || N > 10);
}
```

III. FOR Structure

The for structure in algorithmic language is used to simplify the programming of counting loops. The for structure in C is more general and much more powerful.

Syntax:

```
for (expr1; expr2; expr3) {
   statementBlock
}
```

With:

- expr1 is evaluated once before entering the loop. It is used to initialize loop data.
- expr2 is evaluated before each re-execution of the loop. It is used to decide whether the loop will be executed or not.
- expr3 is evaluated at the end of each execution of the loop. It is used to reset one or more loop data.

If the statement block contains only one instruction, the curly braces can be omitted.

Example 1:

Write a program that displays the squares of positive integers less than 10 as follows:

```
#include <stdio.h>
main() {
   int I;
   for (I = 0; I < 10; I++)
      printf("The square of %d is %d \n", I, I * I);
   return 0;
}
```

Note:
The expr1, expr2, and expr3 parts can contain multiple sub-instructions separated by commas.

Example 2:
Write a program that displays the sum of positive integers less than or equal to 100 as follows:

```
#include <stdio.h>
main() {
    int n, sum;
    for (sum = 0, n = 1; n <= 100; n++)
        sum += n;
    printf("The sum of numbers from 1 to 100 is %d\n", sum);
    return 0;
}
```

Note:
You can declare the variable inside the for statement at the time of its initialization.
Example:
The statement "for (int i = 0; i <= 10; i++) { }" is equivalent to the following two statements: "int i; for (i = 0; i <= 10; i++) { }".

Functions

Each program includes at least one function, which is the main function. A function is a grouping of variables and instructions identified by a name and having a specific purpose. Functions in C correspond to functions and procedures in algorithmic language.

I. Definition and Declaration

1.1 Function Definition

The definition of a function determines the function's header and the instructions that compose it. The header includes the type of the result returned, the identifier or the function's name, and the declaration of parameters within parentheses.

Syntax:

```
ReturnType functionName(paramType param1, paramType param2, ...) {
    // Local variable declarations
    // Instructions
    [return result;]
}
```

Where:

- returnType is the type of the result returned by the function, which can be void or another simple type.
- functionName is the unique identifier or name of the function.
- paramType param are formal parameters, which are variables used for communication between the function and the calling program. They are only visible and known within the function.
- return result is an instruction that only exists if the function's type is not void. The returned result must have the same type as the function.

Example 1:

Here's a function named square that calculates the square of a given real number:

```
float square(float a) {
    float result;
    result = a * a;
    return result;
}
```

Example 2:

Here's a function named sum that calculates the sum of two given real numbers:

```
float sum(float a, float b) {
    float s = a + b;
    return s;
}
```

<u>a. Local Variables</u>

A variable is considered local if it is defined within a function. It is only accessible within the scope of its declaration. Different functions can use variables with the same name, and they do not refer to the same variable.

b. Global Variables
Global variables are declared at the beginning of the file, outside of all functions, and are accessible to all functions in the program. Usually, global variables are declared immediately after including library headers.

c. Formal Parameters
Formal parameters (or dummy parameters) are the variables declared in the function's header to be defined later. They are used for communication between the calling function and the called function. These parameters are created when the function is called and are removed from memory when the function has completed its processing.

d. Actual Parameters
Actual parameters are the variables declared in the calling function and contain values that are sent to the function when it is called. Actual parameters are transferred in the same order as the formal parameters and must be of the same type as the formal parameters.

Example 1:
For the function that calculates the square of a real number, you could have the following main function:

```
int main() {
   float y;
   scanf("%f", &y);
   float result = square(y);
   printf("The square of %f is: %f", y, result);
   return 0;
}
```

Example 2:
For the function that calculates the sum of two real numbers, you could have the following main function:

```
int main() {
   float x, y;
   scanf("%f", &x);
   scanf("%f", &y);
   float result = sum(x, y);
   printf("The sum of %f and %f is: %f", x, y, result);
   return 0;
}
```

1.2 Function Declaration

The declaration of a function only determines the return type, the function's name, and the types of parameters within parentheses. The declaration does not include the function's body.

Declarations are used when a function is defined after the main function or the calling function.

The function declaration can be:
- Local, placed just before the calling function.
- Global, placed before all functions.

Syntax:

```
returnType functionName(paramType param1, paramType param2, ...);
```

The function declaration is an independent statement and must end with a semicolon.

1.3 Function Types

A function can have one of the following types:

- void if it corresponds to the concept of a procedure in algorithmic. In this case, the function does not return any result.

Example:

Here's a function named display that displays the square of any number:

```
void display(float x) {
    printf("The square of %f is %f", x, x * x);
}
```

- A simple type if it corresponds to the concept of a function in algorithmic. In this case, the function must return a result that is compatible with its type.

Example:

Here's a function named factorial that calculates the factorial of an integer:

```
int factorial(int n) {
    int i, f = 1;
    for (i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}
```

II. Function Call

A function can be called in the main function or another function.

2.1 Calling a void Function

Calling a void function is done by using the function name followed by actual parameters in parentheses.

Syntax:

```
functionName(actualParam1, actualParam2, ...);
```

Example:

Write a C program that calls the display function:
#include <stdio.h>

```
int main() {
    float a;
    scanf("%f", &a);
    display(a);
    return 0;
}
```

2.2 Calling a Function with a Simple Return Type

Functions with a simple return type can be called in one of three ways, depending on the need:

- In an assignment.
- In a display.
- In an expression.

a. Function Call in an Assignment

This way of calling a function allows you to assign the result returned by the function to a variable with the same or compatible type.

Example:
Write a C program that calls the factorial function and assigns its result to a variable:

```
int main() {
    int a, factA;
    scanf("%d", &a);
    factA = factorial(a);
    printf("The factorial of %d is %d", a, factA);
    return 0;
}
```

b. Function Call in a Display
This way of calling a function allows you to call the function within a display statement. The result returned by the function will be printed. The format specifier used in the display must be of the same or compatible type as the function's result.
Example:
Write a C program that calls the factorial function within a display statement:

```
int main() {
    int a;
    scanf("%d", &a);
    printf("The factorial of %d is %d", a, factorial(a));
    return 0;
}
```

c. Function Call in an Expression
In this type of call, the result returned by the function is evaluated within an expression.
Example:
Write a C program to calculate the combination of p choose n:

```
int main() {
    int n, p;
    float cnp;
    do {
        printf("Enter two integer values: ");
        scanf("%d %d", &n, &p);
    } while (p > n || p < 0);
    cnp = factorial(n) / (factorial(p) * factorial(n - p));
    printf("The combination of %d choose %d is %f", n, p, cnp);
    return 0;
}
```

III. Parameter Passing

A function with formal parameters needs corresponding actual values to execute. These values are transmitted to the function at the time of the call through actual parameters, which is called passing parameters by value. This method involves copying the content of the actual parameters into the corresponding formal parameters. In this case, the formal parameters contain a copy of the actual parameters. Passing by value does not allow the called function to directly modify the value of the actual parameter. It can only modify its own temporary copy.
Example:
Write a function "exchange" that swaps the values of two integers. Write the C program that calls this function.

```
#include <stdio.h>
int main() {
    // Declaration of the exchange function
    void exchange(int, int);
    int n = 10, p = 20;
    printf("Before call: %d %d\n", n, p);
    exchange(n, p);
    printf("After call: %d %d\n", n, p);
    return 0;
}
// Definition of the exchange function
void exchange(int a, int b) {
    int c;
    printf("Exchange start: %d %d\n", a, b);
    c = a;
    a = b;
    b = c;
    printf("Exchange end: %d %d\n", a, b);
}
```

If you need to return multiple values or modify the values of the actual parameters of the calling function, you should use the concept of passing parameters by address. In this case, the function's parameters should be used as addresses rather than values. The calling function provides the address of its parameter so that the called function can work on the same memory location of the variable. The function's parameters are then pointers.
Example: The "exchange" function, which swaps two values, was defined with parameters passed by value. It cannot modify the "n" and "p" parameters of the calling program because it only exchanges copies of "n" and "p. To achieve the desired result, the calling program should pass pointers to the values to be modified.

```
#include <stdio.h>

int main() {
    void exchange(int *, int *);
    int n = 10, p = 20;
    printf("Before call: %d %d\n", n, p);
    exchange(&n, &p);
    printf("After call: %d %d\n", n, p);
    return 0;
}
// Definition of the exchange function
void exchange(int *a, int *b) {
    int c;
    printf("Exchange start: %d %d\n", *a, *b);
    c = *a;
    *a = *b;
    *b = c;
    printf("Exchange end: %d %d\n", *a, *b);
}
```

IV. Variable Scope

A block of instructions is enclosed in curly braces and consists of two parts:

- Local variable declarations within the block.
- A set of instructions forming the body of the block.
- Variables declared within a block of instructions are only visible within that block. They are considered local variables.

Syntax:

```
{
   // Local declarations
   // Instructions
}
```

Thus, a block of instructions in a function or in conditional commands like "if," "while," "do," or "for" can contain local variable declarations and even functions.

Example: Write a function "sum" that displays the sum of the first "n" positive numbers.

```
#include <stdio.h>

void sum(int n) {
   if (n > 0) {
      int i;
      int s = 0;
      for (i = 1; i <= n; i++)
         s += i;
      printf("The sum is: %d", s);
   } else {
      printf("The value of n is negative.");
   }
}
```

The variables "s" and "i" are declared within a conditional block. If the condition is not met, they will not be defined, and they will disappear at the end of the conditional block.

Note:

Local variables within functions are a good example to explain their visibility concerning the entire program. A local variable is only visible within the body of the function. Beyond that, it does not exist and is not recognized.

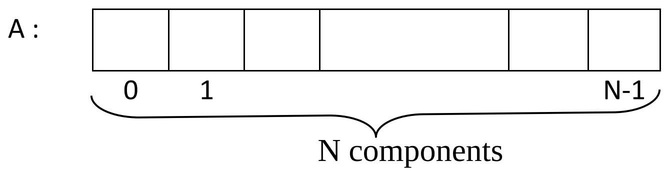
Arrays

I. One-Dimensional Arrays

1.1 Definition

A one-dimensional array A is a structured variable composed of a number of simple variables of the same type, called the array's components. The number of components N is referred to as the array's dimension.

Schematic Representation:



In the C language, array indices start at 0. The last component has an index of N-1.

1.2 Declaration and Storage

a. Declaring Arrays in C

Declaring an array requires specifying the type of elements it will contain and the number of those elements (dimension).

Syntax:

DataType ArrayName[Dimension];

Examples:

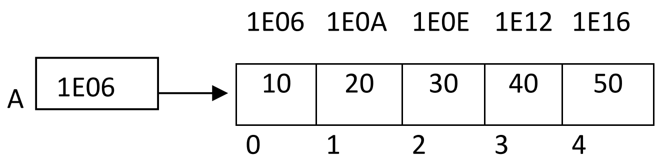
```
int A[10];
float B[100];
char C[30], D[100];
```

b. Storage

In C, any array occupies contiguous memory space capable of storing all its elements. The array's name always represents the address of the first element. The addresses of the other components are automatically calculated relative to this address using their indices.

Example:

Suppose we have an array A containing the following integer values: 10, 20, 30, 40, 50.



To access the value in the cell with index 3, its address is calculated as (1E06 + 3 * size of an integer).

1.3 Initialization and Automatic Reservation

a. Initialization

When declaring an array, you can initialize the array's components by providing the respective values enclosed in curly braces.

Syntax:

```
DataType ArrayName[N] = {val1, val2, ..., valN};
```

Examples:

```
int A[5] = {10, 20, 30, 40, 50};
float B[4] = {-1.05, 3.33, 8.5, -12.3};
int C[10] = {1, 0, 0, 1, 1, 1, 0, 1, 0, 1};
```

Make sure the number of values in the list matches the array's dimension. If there aren't enough values in the list for all the components, the remaining components are automatically initialized with the default value for the array's data type. If the number of elements in the list exceeds the array's dimension, the system will generate an error.

b. Automatic Reservation

If the dimension is not explicitly stated during initialization, the computer will automatically reserve the necessary number of bytes to contain all the values enclosed in curly braces.

Syntax:

```
DataType ArrayName[] = {val1, val2, ..., valN};
```

Examples:

```
int A[] = {10, 20, 30, 40, 50};
// Reserves 5 cells containing integers.
float B[] = {-1.05, 3.33, 8.5, -12.3};
// Reserves 4 cells containing real numbers.
```

1.4 Accessing Components

Accessing an element in an array is done using the indexing operation, where you specify the array's name followed by the index of the element to reference inside square brackets.

Syntax:

```
ArrayName[index]
```

Example:

Suppose an array A is declared as follows:

int A[5] = {1200, 2300, 3400, 4500, 5600};

- A[0] refers to the content of the first cell in the array, with an index of 0.
- A[1] refers to the content of the second cell in the array, with an index of 1.
- ...
- A[4] refers to the content of the last cell in the array, with an index of 4.

1.5 Display and Input

The for loop structure is particularly well-suited for working with arrays. Most applications can be implemented by modifying the typical examples for displaying and assigning values.

Example 1:

Write a C program that allows you to input the elements of an array of 5 integers.

```
#include <stdio.h>
int main() {
   int A[5];
   int i; // Counter
   for (i = 0; i < 5; i++) {
      scanf("%d", &A[i]);
   }
   return 0;
}
```

Example 2:
Write a C program that displays the elements of an array of 5 integers filled with the values {10, 20, 30, 40, 50}.

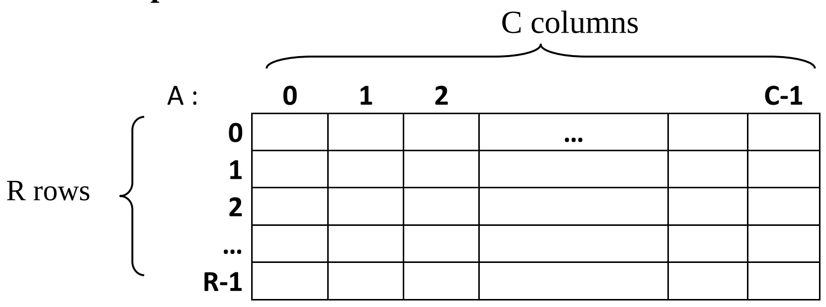
```
#include <stdio.h>
int main() {
   int A[5] = {10, 20, 30, 40, 50};
   int i; // Counter
   for (i = 0; i < 5; i++) {
      printf("%7d\t", A[i]);
   }
   return 0;
}
```

II. Two-Dimensional Arrays

2.1 Definition

A two-dimensional array A corresponds to a matrix with R rows and C columns. R and C are the two dimensions of the array, and a two-dimensional array contains R * C components. A two-dimensional array is considered square when R is equal to C.

Schematic Representation:



2.2 Declaration and Storage

a. Declaring Two-Dimensional Arrays in C

Syntax:

```
DataType ArrayName[R][C];
```

Here, R is the number of rows, and C is the number of columns.

Examples:

int A[10][10];
float B[2][20];
char C[15][40];

b. Storage

In the C language, a two-dimensional array A is interpreted in memory as a one-dimensional array with R dimensions, with each component being a one-dimensional array with C dimensions.

Example:

Storing a two-dimensional array:
int A[3][2] = {{1, 2}, {10, 20}, {100, 200}};
Memory Representation:

A :	1	2	10	20	100	200
1E06						

2.3 Initialization and Automatic Reservation

a. Initialization

When declaring a two-dimensional array, you can initialize its components by specifying the values in lists enclosed in curly braces. Within each list, the components of each row of the array are enclosed in curly braces for better program readability.

Syntax:

```
DataType ArrayName[R][C] = {{Row1Values},
                            {Row2Values},
                              …
                            {LastRowValues}
                           };
```

Examples:

```
int A[3][10] = {
    {0, 10, 20, 30, 40, 50, 60, 70, 80, 90},
    {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
    {1, 12, 23, 34, 45, 56, 67, 78, 89, 90}
};

float B[3][2] = {
    {-1.05, -1.10},
    {86e-5, 87e-5},
    {-12.5E4, -12.3E4}
};
```

Ensure that

- the number of values in each row list does not exceed the number of columns,
- and the number of row lists does not exceed the number of rows.

If there aren't enough values in the lists for all the components, the remaining components are automatically initialized with the default value for the array's data type. If the number of elements in the list exceeds the array's dimension, an error is generated.

More Examples:

int C [4][4] = {{1, 0, 0, 0 }, {1, 1, 0, 0 }, {1, 1, 1, 0 }, {1, 1, 1, 1 }} ;	C : $\begin{array}{\|c\|c\|c\|c\|}\hline 1 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ \hline\end{array}$
int C [4][4] = {{1, 1, 1, 1 }} ;	C : $\begin{array}{\|c\|c\|c\|c\|}\hline 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline\end{array}$
int C [4][4] = {{1},{1},{1},{1}} ;	C : $\begin{array}{\|c\|c\|c\|c\|}\hline 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline\end{array}$
int C [4][4] = {{1, 1, 1, 1, 1 }} ;	erreur

b. Automatic Reservation

With automatic reservation for two-dimensional arrays, you can omit specifying the number of rows but not the number of columns.

If the number of rows R is not explicitly specified, the computer will automatically reserve the necessary number of bytes based on the number of row lists.

Example:

```
int A[][10] = {
   {0, 10, 20, 30, 40, 50, 60, 70, 80, 90},
   {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
   {1, 12, 23, 34, 45, 56, 67, 78, 89, 90}
};
// Reserves 3 rows, each containing 10 elements.
```

If the number of columns C is not explicitly specified, the system generates an error.

Example:

```
int A[3][] = {
   {0, 10, 20, 30, 40, 50, 60, 70, 80, 90},
   {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
   {1, 12, 23, 34, 45, 56, 67, 78, 89, 90}
};
// Error.
```

2.4 Accessing Components

Accessing an element in a two-dimensional array is done through indexing, specifying the array's name followed by the indices of the element in square brackets.

Syntax:

```
ArrayName[Row][Column]
```

The elements of a two-dimensional array with dimensions R and C are structured as follows:

A[0][0]	A[0][1]	A[0][2]	. . .	A[0][C-1]
A[1][0]	A[1][1]	A[1][2]	. . .	A[1][C-1]
A[2][0]	A[2][1]	A[2][2]	. . .	A[2][C-1]
. . .	. . .	. . .	. . .	. . .
A[R-1][0]	A[R-1][1]	A[R-1][2]	. . .	A[R-1][C-1]

The array indices range from 0 to R-1 for rows and from 0 to C-1 for columns. The component of the N-th row and M-th column is denoted as A[N-1][M-1] with $0 \le N < R$ and $0 \le M < C$.

2.5 Display and Input

When working with two-dimensional arrays, you need two indices to traverse the rows and columns of the arrays.

Example 1:

Write a C program that allows you to input the elements of a 5x10 integer array.

```
#include <stdio.h>
int main() {
    int A[5][10];
    int i, j;  // Counters
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 10; j++) {
            printf("Enter an element: ");
            scanf("%d", &A[i][j]);
        }
    }
    return 0;
}
```

Example 2:
Write a C program that displays the elements of a 5x10 integer array assumed to be filled with the following values: {{1, 2, 3, 4}, {6, 7, 8}}.

```
#include <stdio.h>
int main() {
    int A[5][10] = {{1, 2, 3, 4}, {6, 7, 8}};
    int i, j;  // Counters
    for (i = 0; i < 5; i++) {
        for (j = 0; j < 10; j++) {
            printf("%7d\t", A[i][j]);
        }
        printf("\n");  // New line
    }
    return 0;
}
```

Homework 1
(Simple Data Types, Operators, and Expressions)

Exercise 1
Write a C program that allows you to read the two sides (length and width) of a rectangle and display its perimeter and area.

Exercise 2
Write a program that reads three integer values and displays their sum, product, and average.

Exercise 3
Write a program that reads the values of a student's grades N1, N2, N3 and their respective coefficients C1, C2, C3, and displays:

a. The arithmetic average of the grades.
b. The weighted average of the grades.

Exercise 4
Write a C program that allows you to read the electricity consumption and the unit price and display the amount to be paid on the bill.

Exercise 5
Write a C program that allows you to read an integer representing time in seconds and convert it into hours, minutes, and seconds.

Exercise 6
Write a C program that allows you to read an integer and display its square, cube, and square root.
Example:
```
Enter a number: 9
The square of 9 is 81
The cube of 9 is 729
The square root of 9 is 3
```

Exercise 7
Write a C program that allows you to read a list of five numbers. Calculate and display the sum of the squares of these numbers.
Example:
```
List of numbers: 5 2 3 1 4
The sum of the squares is 55
```

Homework 2
Control Flow

Exercise 1

Write a program that calculates the year-end bonus according to the following scale:

- 7% of the monthly salary if the employe's index is less than 250.
- 15 D per dependent child.
- 12 D per year of seniority starting from the 3rd year elapsed: any started year counts as a complete year.

Exercise 2

The profit B of a commercial representative in a company is calculated based on the turnover CA they generate. Specifically, B = TX * CA, and:

If CA < 1500, then TX = 1%.
If 1500 <= CA < 2000, then TX = 1.5%.
If CA >= 2000, then TX = 2%.
Write a program that allows entering the turnover achieved by the representative and determining their profit.

Exercise 3

Write a program that allows reading two integer values (A and B) from the keyboard and displays the sign of the product of A and B without performing the multiplication.

Exercise 4

Write a program that calculates the real solutions of a second-degree equation ax^2 + bx + c = 0.

Exercise 5

Write a program that allows reading a positive integer and checks if it is cubic or not. A number is cubic if it is equal to the sum of the cubes of its constituent digits.
Example: 153 = 1^3 + 5^3 + 3^3.

Exercise 6

Write a program that allows reading three integer values (A, B, and C) from the keyboard and displays the smallest of the three values.

Exercise 7

Write a program that allows entering a month number and displays the corresponding month and quarter in full words.

Exercise 8

Write a program that allows adding, subtracting, multiplying, or dividing two integer values based on the operator chosen by the user.

Exercise 9

Write a program that reads N integer numbers from the keyboard and displays their sum, product, and average. Choose an appropriate data type for the values to be displayed. The number N is to be entered from the keyboard. Solve this problem:
a) using a while loop,

b) using a do-while loop,
c) using a for loop.
d) Which of the three variations is the most natural for this problem?

Exercise 10
Extend the best of the three versions from Exercise 1:
Repeat the input of the number N until N has a value between 1 and 15. Which looping structure do you use, and why?

Exercise 11
Calculate the integer quotient and remainder of the integer division of two integers entered from the keyboard using successive subtractions.

Exercise 12
Calculate the factorial N! = 1 * 2 * 3 * ... * (N-1) * N of a natural number N while respecting that 0! = 1.
a) Use while,
b) Use for.

Exercise 13
Calculate the result X^N of two natural integers X and N entered from the keyboard using successive multiplications.

Exercise 14
Display the multiplication table for N ranging from 1 to 10.

```
X*Y    0   1   2   3   4   5   6   7   8   9  10
--------------------------------------------------
  0      0   0   0   0   0   0   0   0   0   0   0
  1      0   1   2   3   4   5   6   7   8   9  10
  2      0   2   4   6   8  10  12  14  16  18  20
  3      0   3   6   9  12  15  18  21  24  27  30
  4      0   4   8  12  16  20  24  28  32  36  40
  5      0   5  10  15  20  25  30  35  40  45  50
  6      0   6  12  18  24  30  36  42  48  54  60
  7      0   7  14  21  28  35  42  49  56  63  70
  8      0   8  16  24  32  40  48  56  64  72  80
  9      0   9  18  27  36  45  54  63  72  81  90
 10      0  10  20  30  40  50  60  70  80  90 100
```

Homework 3 Functions

Exercise 1

Comment the following declarative lines:

- int f1 (float);
- void f2 (int, float);
- void f3 (int a);
- f4 (char);

Exercise 2

Write the declaration:

- Of a function named f5 that accepts a real number and returns nothing.
- Of a function named f6 that accepts a real number and returns a real number.

Exercise 3

Describe the result produced by the following program:

```
#include <stdio.h>
int a = 100, b = 200;
void main() {
   int counter, c;
   int fonc1(int c);
   for (counter = 1; counter <= 5; ++ counter) {
      c = 4 * counter * counter;
      printf("%d ", fonc1(c));
   }
}
fonc1(int x) {
   int c;
   c = (x < 50) ? (a + x) : (b - x);
   return (c);
}
```

Exercise 4

Write a function "Read_num" that reads two positive non-zero integers and calls a function "length" that returns the number of digits in a number. The "Read_num" function returns to the main:

- 1 if the first entered number is longer than the second.
- 2 if the second entered number is longer than the first.
- 0 if both numbers have the same length.

Exercise 5

Write a function that checks if a number is prime. This function is called by the main function.

Exercise 6

Write a function that returns the value of the largest digit in a positive integer.

Exercise 7

Write a function that checks whether an integer is symmetric or not.

Exercise 8
a) Write a function "max_two" that determines the maximum between two integers.
b) Write a function "max_three" that determines the maximum between three integers using the "max_dtwo" function.

Exercise 9
Write a function to calculate the power of two integers without using the "power" function.

Exercise 10
Write a recursive function to convert a decimal number N to base B.

Homework 3
Arrays

Exercise 1
Write a program that reads the dimension N of an array T of type int (maximum dimension: 50 components), fills the array with values entered from the keyboard, and displays the array. Calculate and display the sum of the array elements. Use functions of your choice.

Exercise 2
Write a program that reads the dimension N of an array T of type int (maximum dimension: 50 components), fills the array with values entered from the keyboard, and displays the array. Then, rearrange the elements of the array T in reverse order without using an auxiliary array. Display the resulting array. Use functions of your choice.

Exercise 3
Write a program that calculates the scalar product of two integer vectors U and V (of the same dimension). Use functions of your choice.

Exercise 4
Write a program that determines the largest and smallest values in an array of integers A. Then, display the value and position of the maximum and minimum values. If the array contains multiple maxima or minima, the program should record the position of the first maximum or minimum encountered. Use functions of your choice.

Exercise 5
An array A of dimension N+1 contains N integer values sorted in ascending order, and the (N+1)th value is undefined. Insert a value VAL entered from the keyboard into array A in a way that results in a sorted array of N+1 values. Use functions of your choice.

Exercise 6
Write a program that sets the elements on the main diagonal of a given square matrix A to zero. Use functions of your choice.

Exercise 7
Write a program that performs the addition of two matrices A and B of the same dimensions N and M. Use functions of your choice.