CHAPTER 1

NUMERICAL ALGORITHMS AND ERRORS

SUPPLEMENT No. 1b:

C++ TUTORIAL

prepared for

NUMERICAL METHODS FOR SCIENTISTS AND ENGINEERS With Pseudocodes

By Zekeriya ALTAÇ

October 2024

Supplement No. 1b: The C++ Tutorial

LEARNING OBJECTIVES

The objective of this C++ programming tutorial is to
- present a short summary of the basics of the C++ language;
- describe the implementation of basic programming operations such as loops, accumulators, and conditional constructs;
- explain how to prepare functions or subprograms.

The textbook "*Numerical Methods for Scientists and Engineers: With Pseudocodes*" focuses on implementing the methods in science and engineering applications. Supplemental course materials and resources, including C/C++, Fortran, Visual Basic, Python, Matlab®, and Mathematica®, are provided to assist the instructors in their teaching activities outside the class.

The aim of this short tutorial is to enable students to acquire the knowledge and skills to convert the pseudocodes given in the text into running C++ programming language. It is not intended to be a "complete language reference document." The author assumes that the reader is familiar with programming concepts in general and may also be familiar with the C++ programming language at the elementary level. In this regard, this tutorial illustrates the conversion and implementation of pseudocode statements (such as formatted/unformatted input/output statements, loops, accumulators, control and conditional constructs, creating and using functions, and subprograms, etc.) to the C++ programming language.

1 C++ PROGRAM STRUCTURE

A C++ program consists of preprocessor directives, data type definitions, variable declarations, header files, comments, the `main()` function, and additional functions if required. Every program statement ends in a semicolon (`;`), newlines are not significant except in preprocessor controls, and the blank lines are ignored. All function names, including the main program, which is a function (`main()`), are always followed by parentheses, i.e., (). Curly braces `{ }` contain a group of statements.

1.1 PREPROCESSOR DIRECTIVES

Every C++ program begins with at least one preprocessor directive. The first thing the compiler processes is the preprocessor directive, which provides control instructions from a code referred to as *header file*. The header files, typically having the "`.hpp`" extension, are an important part of C++ programs, which serve as a means to import predefined standard library functions, data types, macros, and other features into the main programs. A list of some of the frequently used C++ libraries and the header files is given in **Table 1.1**. The header files are imported into the code by `#include` preprocessor directive (click to see all available C++ libraries).

1.2 GLOBAL DATA DEFINITIONS

Global data definitions refer to variables and constants that are declared outside of any function, making them accessible from any function within the same file or across multiple files (if declared appropriately).

Table 1.1: Some of the C++ header files and functions.

Header file	Library functions
`iostream`	cin, cout, cerr, wcin, wcout, ...
`iomanip`	setprecision, scientific, setw, ...
`fstream`	fstream, ifstream, ofstream, ...
`cstdlib`	malloc(), calloc(), realloc(), free(), srand(), rand(), ...
`cstring`	strlen(), strcpy(), strcat(), strcmp(), strstr(), ...
`cmath`	sqrt(), sin(), asin(), cos(), acos(), pow(), ceil(), floor(), abs(), log(), log10(), rand(), srand(), ...
`complex`	complex, sqrt(), pow(), ...
`cstdio`	fopen(), printf(), scanf(), fprintf, fscanf(), sprintf(), ... abs(), log(), log10(), rand(), srand(), ...
`ctime`	time(), ctime(), clock(), difftime(), localtime(), ...
`utility`	swap, exchange, forward, ...
`vector`	vector, count, ...

They are stored in the data section of the program's memory. Global variables exist for the entire duration of the program's execution, and they should be used judiciously to avoid issues related to maintainability, debugging, and concurrency.

1.3 FUNCTION DEFINITIONS

Function definitions contain both data definitions and code instructions to be executed when the program runs. All program executable statements are contained within function definitions. Every C++ program should have one function called `main`.

1.4 COMMENTING

Commenting is done to allow *human-readable* descriptions detailing the purpose of some of the statement(s) and/or to create *in situ* documentation. A double forward slash (//) is used for single-line comments; a series of multi-line comments is enclosed within /* and */.

A *single-line* comment is applied to describe an expression (or statement) on the corresponding line. A *block* of comments generally at the beginning of each module (*Header Comments*) is used to describe the purpose of the module, its variables, exceptions, other modules used, etc.

1.5 A C++ CODE EXAMPLE

The basic features of a C++ code (`example.cpp`) are illustrated in the **C++ code 1.1**. The extension `.cpp` is a common extension used for C++ source code.

Lines 1-4, 8, 12, 17, and 20 in `example.cpp` are reserved entirely for comments. Note that in this example, explanatory comments are dispersed throughout the program. The other lines also include comments after the end of the statements.

Lines 6, 9, 11, 16, 19, and 22 contain six blank lines of the code, which are used to break up long sections of a code, making it easier to read and understand. The blank lines are also used to visually separate different logical sections, such as variable declarations, function definitions, major code blocks, etc.

In line 5, the header file (in this case `<iostream>`) is executed first. `<iostream>` (stands for standard input-output stream) is a library file that provides functions and declarations related to input and output operations such as `cin`, `cout`, `clog`, and so on. Another important header file is `<iomanip>`, which provides a set of manipulators used for formatting input and output streams. It is commonly used in conjunction with `<iostream>` for stream-based input and output.

C++ Code 1.1

```
1  /* ===================================================================
2   * Description : An example C++ program calculating the area of a circle.
3   * Written by  : Z. Altac
4   * =================================================================== */
5   #include <iostream>         // Preprocessor directive
6
7   #define PI 3.14159          /* Macro definition */
8   /* A global constant can also be defined as const double PI = 3.14159;*/
9
10  using namespace std;
11
12  /* The main program is contained within { } marks */
13  int main(void) {           /* The main (void) program starts here */
14     double radius, area;    // Local variable definitions
15     radius = 12.0;          // Local data (radius) definitions
16
17     // Statements section
18     area = PI * radius * radius; // Calculate the area of circle
19
20     /* Display the result (area) on the screen using cout/endl */
21     cout << "The area of the circle is: " << area << endl;
22
23     return 0;             // Program is terminated
24  }
```

In line 6, a macro feature of C++ known as *macro definition*, is used. A macro definition is a way to define a fragment of code or a constant value that can be reused throughout your program. Macros are created using the preprocessor directive `#define`. They are processed by the preprocessor before the actual compilation of the code. Here, a global data (`PI` as macro) is defined as `#define PI 3.14159`. The constant `PI` now becomes accessible by the main and (if present) auxiliary functions.

In line 10, the "`using namespace std;`" directive provides names `std::cout` and `std::endl` to display a message on the screen. This "`using namespace std;`" allows us to omit the `std::` prefix and use their shorter names, `cin`, `cout`, and `endl`. This makes the code cleaner and easier to read, but you should be cautious with its use in larger projects.

In lines 12-23, the main code of the program `example` is located. The `main`, which is a function, is the starting point of the program. It is defined as `int main` (`void`) or also expressed as `int main` () or simply `main` (). This representation indicates that the main program (function) does not take an argument.

In line 15, the local variable `radius` is initialized to `12.0`.

In line 18, the area ($A = \pi r^2$) is calculated by the given expression.

In line 21, the calculated result (save on `area`) is printed (on the screen) using the `cout` and `endl` objects of the `<iostream>` library.

In line 23, the program is terminated with `return 0`. The return value of zero signifies successful execution.

In C++, the syntax must be correct. Otherwise, the compiler will generate error messages and will not produce executable code.

2 VARIABLES, CONSTANTS, AND INITIALIZATION

2.1 IDENTIFIERS AND DATA TYPES

Identifiers (i.e., symbolic names for variables, functions, etc.) are represented symbolically with letters or combinations of letters and numbers (i.e., a, b, ax, xy, a1, tol, . . .). The first character of an identifier must be a letter, which may also include an underscore (_). The C++ language has 95 keywords, such as **`int`**, **`do`**, **`while`**, etc., reserved for specific tasks and cannot be used as identifiers.

The C++ supports a wide variety of built-in data types: signed integer type (**`int`**, **`short`**, **`long`**), real types (**`float`**, **`double`**, **`long double`**), and void type (*see* full listing).

Integer (`int`): Integers are whole numbers that can hold positive or negative whole values; e.g., 0, -1223, or 140. C++ provides a rich set of integer types suitable for various applications. Understanding their sizes and ranges helps in selecting the appropriate type for your specific needs. The 4-byte integer, `int` or `long` type, is the most commonly used integer type and ranges from -2,147,483,648 to 2,147,483,647. A **`short`** integer, usually 2 bytes, ranges from -32,768 to 32,767.

Character (`char`): Character variables are used to store single characters and are represented by the **`char`** data type. The **`char`** data type is typically used to represent a single character, such as letters, digits, punctuation marks, etc. A single character of data holds 1 byte of information. A sequence of characters can be stored in an array of **`char`**, often used to represent strings in C++.

Real numbers (`float`, `double` or `long double`): Real numbers with a decimal point can be represented; e. g., -2.3, 0.14, 1.2e5, and so on. **`float`**, **`double`**, and **`long double`** require 32, 64, and 80 bits. A **`float`** typically represents a single-precision floating-point number. It represents a range of values approximately from 1.2×10^{-38} to 3.4×10^{38} with about 6-7 decimal digits of precision. The suffix **`f`** is used for **`float`** literals. By default, decimal literals are considered **`double`**.

Void (`void`): It is an incomplete type, which implies "*nothing*" or "*no type*". It is used as a (i) function return type, (ii) void pointer, and (iii) function parameter, which indicates that a function does not take parameters.

Signed/Unsigned (`signed/unsigned`): These are type modifiers. As **`signed`** allows the storage of both positive and negative numbers, the **`unsigned`** can store only positive values but has a larger positive range.

2.2 TYPE DECLARATION OR INITIALIZATION OF VARIABLES AND CONSTANTS

A *variable* is a symbolic representation of the address in memory space where values are stored, accessed, and updated. A *variable declaration* is to specify the *type* of the variable (i.e., the *identifier*) as **`char`**, **`int`**, **`float`**, and so on, but a value to the variable has yet to be assigned. The value of a variable changes during the execution of a computer program, while the value of a **`constant`** does not. All variables (or constants) used in a program or subprogram, as well as data types, must be declared before they can be used in any expression.

The type declaration syntax for variables or constants is as follows:

```
type  identifier-1, identifier-2, ..., identifier-n ;
type  constant-identifier = type-compatible value ;
```

<cstdlib> (stands for C Standard Library) defines several general purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetic ops, searching, sorting, and converting.

> In C++, the identifiers are case-sensitive, where the lower- and uppercase letters are treated as different. For example, `Name`, `NAME`, or `name` denote three different variables. In general, lowercase letters are used for variables, and uppercase letters are used for constants (i.e., macro definitions).

The **type** declaration is carried out at the beginning of the **main()** function. A variable can also be initialized while being type declared by assigning a value to it, typically with the assignment operator **"="**. Below is a number of declarations and declarations with initialization.

```
int iter, maxit, i, j;    // Declare integer variables
float radius, area;       // Declare float (real) variables
int p=0, m = 99;          // p and m are declared and initialized
float PI = 3.14159;       // pi is declared and initialized
double zd=2.0*PI          // zd is declared and initialized
char s='*', no;           // no and s is declared, s is initialized
char lines[72];           // Declare a string array of size 72
```

In this example, **m**, **PI**, **zd**, and **s** are not only type-declared but also initialized at the same time. The order of declaration can be important. For instance, in order to initialize **zd**, **PI** must be declared and initialized beforehand.

> In general, a variable does not have a default value. Using the value of an uninitialized variable in operations may lead to unpredictable results, or the program may crash in some compilers.

3 INPUT/OUTPUT (I/O) FUNCTIONS

An inevitable element in any program is the communication of the input and output data with the main program or its sub-programs. This is done through `cin` and `cout` functions of the `iostream` standard library, which are used to read initial values of variables into the program from a keyboard or write the intermediate or final values of variables to a screen. The **cin** and **cout** are the most important and useful objects to display and read data on or from input and output devices. For formatted output operations, **cout** is used together with the insertion operator '**<<**'.

The C++ language also provides several built-in functions to display the output in formatted form. These built-in functions are available in the header file `iomanip` and `ios` class of header file `iostream`. There are two ways to carry out a formatted IO operation: (i) using the member functions of the **ios** class and (ii) using the manipulators defined in **iomanip**.

The syntax for the `cin` and `cout` functions is as follows:

```
cin  >> variable-1 >> variable-2 >> ... >> variable-n;
cout << variable-1 << variable-2 << ... << variable-n;
```

where the insertion **>>** and extraction **<<** operators have been used for formatted input and output, respectively. In the standard setting, a floating-point is displayed to six digits by default. Decimal numbers are printed without a decimal point unless there are digits after the decimal point by default. On the other hand, very small or large numbers are displayed in exponential form.

Table 1.2: Manipulators to perform formatted I/O in `iomanip` and `ios` files.

Function	Description
`endl`	move cursor position to a newline
`setw(int)`	set width in number of characters for the immediate output data
`width(int)`	set width in number of characters for the immediate output data
`left, right`	set left or right alignment flags
`setfill(char)`	fill blank spaces in output with given character
`setprecision(int)`	set number of digits of precision
`fixed`	display output in fixed point notation
`scientific`	display output in scientific notation
`fill(char)`	fill blank spaces in output with given character
`precision(int)`	set number of decimal point to a float value
`setf(fmtflg)`	set formatting output flag like `showpos, oct, hex`, etc
`unsetf(fmtflg)`	clear format flag setting
`ends`	print a blank space (null character)
`flush`	flush the output stream
`setbase(int)`	set number base
`dec, oct, hex`	set decimal, octal, hexadecimal flags
`showbase, noshowbase`	set `show` or `noshowbase` flags.
`showpos, noshowpos`	set `show` or `noshowpos` flags.

Consider the following code segment:

```
int n1, n2;
 cout << "Enter two integer numbers: ";
 cin >> n1 >> n2;
 cout << "You entered " << n1 << " and " << n2 << endl;
 cout << "Their squares are " << n1*n1 << " and " << n2*n2;
```

For the input values of `n1=3` and `n2=5`, the code generates the following output:

```
You entered 3 and 5
Their squares are 9 and 25
```

3.1 FORMAT SPECIFICATIONS

Formatting a program output is an essential part of any serious computer application. In C++, manipulators are special functions that are used to modify the formatting of input and output streams.

Manipulators in C++ are powerful tools for controlling how data is formatted in I/O streams (*see* **Table 1.2**). They allow control over how data is displayed or interpreted in console I/O operations, typically with the `iostream` library. By using the manipulators in conjunction with the insertion and extraction operators, you can customize the output to suit your needs on stream objects.

The standard C++ output manipulators are `endl` and `flush`. The `endl` is used to generate a line feed at the end of a line and flush the stream. This is identical to placing '`\n`' on the output stream. The `flush` is used to flush the output buffer without inserting a new line.

The following code illustrates displaying different types of data:

```cpp
/* C++ program to show input and output    */
#include <iostream>
#include <string>

using namespace std;

int main()
{   // declarations of variables
    int int_val;
    char ch_val;
    float f_val;
    string strng;
    // Read an integer value
    cout << "Enter an integer value : " << endl;
    cin >> int_val;  // Assign input data to int_val
    // Read a float value
    cout << "Enter a float value : " << endl;
    cin >> f_val;   // Assign input data to f_val
    // Read a character value
    cout << "Enter a character value : " << endl;
    cin >> ch_val; // Assign input data to ch_val
    // Read a string value
    cout << "Enter a string value : " << endl;
    cin >> strng;  // Assign input data to strng

    // Print the input values
    cout << "Integer input value is: " << int_val << endl;
    cout << "Float input value is: " << f_val << endl;
    cout << "Character input value is: " << ch_val << endl;
    cout << "String input value is: " << strng << endl;
    return 0;
}
```

The displayed output will apply default format settings. However, the output can be formatted according to the user specifications if needed.

The `printf` and `scanf` are part of the C standard library and also work in programs written in C++, even though they were not designed for this purpose. It is difficult to generate portable codes with When **`printf`** and **`scanf`** because they can behave slightly differently on different compilers; however, **`cout`** and **`cin`** work consistently across all compilers.

In C++, *format specifications* are used to control the output format of data types when using **`iostream`** and related functions. A *format specification* is a placeholder denoting a value to be filled in during the printing, which is particularly useful for formatting numbers, strings, and other data types. The **`iomanip`** library is a powerful tool for formatting input and output, making it easier to present data in a user-friendly manner. The **`iomanip`** header file provides several useful I/O manipulators (**`setw`**, **`setprecision`**, **`setfill`**, etc.) that help format the output of streams, particularly **`std::cout`** and **`std::cin`**. The `setw(width)` adjusts the field `width` (an integer value) for the item about to be printed. Note that this manipulator affects only the next value to be printed. If the content is shorter than **`width`**, the output will be padded with spaces (or a fill character if set with **`setfill`**) on the left by default. Otherwise, the output with the entire content will be displayed without truncation.

The argument **width** is the minimum width. If an output value requires more space, the manipulator will use as much as is needed. Thus, no *overflow* error is encountered. The default vaue is **width=0**, meaning that if the width is not specified, the output will not be padded and will occupy only as much space as necessary to display the value.

By default, the output values are right justified in their fields. However, the values can be justified in their fields by using **left**, **right**, or **internal** manipulators. The **internal** manipulator separates the sign of a number from its digits, displaying the sign left-justified and the digits right-justified. By default, the fill character is a space. However, the **setfill** manipulator can be used to specify a fill character for the output padding throughout the program.

There are three floating point formats: **general**, **fixed**, and **scientific**. The fixed format always has a number, decimal point, and fraction part, no matter how big the number gets, i.e., the value 1.23e+10 would be displayed as 12300000000 instead of 1.23e+10. But the scientific format always displays a number in scientific notation, i.e., the value of `0.012345` would be displayed as `1.234500e-02`. The general, which is the default format, automatically chooses **fixed** and **scientific** formats based on the value. The **scientific** manipulator forces all subsequent floating-point values output to the stream to be displayed in scientific notation. In other words, when this manipulator is applied to a stream like **cout**, all floating-point numbers displayed after that will be shown in **scientific** format until the format is changed either by using **fixed**, **defaultfloat**, or another manipulator.

The **setprecision** manipulator is used to specify the number of digits to be displayed in floating-point output. An important point about floating point output is the precision, which is roughly the number of digits displayed for the number. The exact definition of the precision depends on which output format is currently being used. In **general**, the precision is the maximum number of digits displayed, including digits before and after the decimal point by excluding the decimal point itself. In **scientific**, the digits in the exponent are not included. In **fixed** and **scientific** formats, the precision is the number of digits after the decimal point.

The following code illustrates some key features of manipulators available in the **iomanip** library:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    cout << "set width to 10 chars: " << endl;
    cout << setw(10) << 1234 << endl;
    cout << "set width to 10 chars and fill with 'b' " << endl;
    cout << setw(10) << setfill('_') << 1234 << endl;
    cout << left << setw(10) << setfill('b') << 1234 << endl;
    cout << right << setw(10) << setfill('*') << 1234 << endl;
    cout << "set precision to 6 " << endl;
    cout << setprecision(6) << 123.4567890 << endl;
    cout << right << setw(10) << setprecision(6) << 123.4567890 << endl;
    cout << setfill(' ');    // undo setfill
    cout << right << setw(10) << setprecision(6) << 123.4567890 << endl;
    cout << "showpos flag " << endl;
    cout << showpos << 999 << endl;
    return 0;
}
```

The code output looks as follows:

```
         1         2         3         4
1234567890123456789012345678901234567890
set width to 10 chars:
      1234    // width=10, right-justified
set width to 10 chars and fill with 'b'
______1234    // width=10, right-justified, left-padded with blank spaces
1234bbbbbb    // width=10, left-justified, right-padded with 'b's
******1234    // width=10, left-justified, right-padded with '*'s
set precision to 6
123.457       // displays 6 digits
***123.457    // width=10, right-justified, left-padded with '*'s
   123.457    // width=10, fill characters is restored to blank
showpos flag
+999          // displays the + sign for the positive values
```

Note that the comments that followed a double backslash are not part of the output but are provided for line-by-line explanation purposes.

For example, the following code segment reserves 5 characters (`w =5`) for the integer and 8 characters (`w=8` with `p=5`) for the float variable.

```
float fval=1.23450;
int nval=12;
cout << "nval " << setfill('b') << setw(5) << nval << endl;
cout << "fval " << setw(8) << setprecision (5) << fval ;
```

The output is as follows:

```
         1         2
12345678901234567890
      nval_bbb12
      fval_b1.23450
```

In this output, the blank space in the format string is denoted by '_', and the blank spaces due to format are filled with '`b`'.

4 SEQUENTIAL STATEMENTS

Frequently, a sequence of statements (with no imposed conditions) are used to perform a specific task. These statements are executed in the order they are specified in the program. Using a semicolon (;) as a separator, multiple expressions can be placed in a single line. For instance, the following code segment illustrates that the first and second statements written on a single line using a semicolon are executed sequentially.

```
a = b * b + c * c ;  d = sqrt(a) // Statement-1 and Statement-2
x1 = d / ( b + c )               // Statement-3
x2 = d / ( b - c )               // Statement-4
y = x1 + x2                      // Statement-5
```

5 ASSIGNMENT OPERATORS AND OPERATIONS

Arithmetic operations involve plus (`+`), minus (`-`), multiplication (`*`), division (`/`), and the modulus (`%`) operators. These operations (excluding the modulus operator) can be used with integer or floating-point types. The modulus operator involving an integer division truncates any fractional part, e.g., `15/3`, `16/3`, and `17/3` all yield 5. The modulus operator (`x%y`) produces the remainder from the division `x/y`, e.g., `15% 3=0`, `16%3=1`, and `17%3=2`.

Assignment operations are used to assign values or computed results of an expression to variables. A simple *assignment* denoted by ← (a left-arrow) in the pseudocode notation is replaced by the "`=`" sign, syntax as shown below:

```
variable_name = value ;      or
variable_name = expression ;
```

The *expression* on the right-hand side of the "=" sign is evaluated first, and its value is placed at the allocated memory location of the *variable* (i.e., `variable_name`) on the left-hand side. Any recently computed value of a *variable* replaces its previous value in memory.

In C++, several other operators for more complex assignments are also provided. These operators help simplify a code by combining an operation with an assignment, reducing redundancy. For instance, a variable may appear on both the left- and right-hand sides of an expression, as in `x = x + y` or `x = x - y`. These expression can be compressed as `x += y` or `x -= y`, where the operators `+=` and `-=` are referred to as *compound assignment operators*. In this regard, compound assignment operators combine arithmetic operators (`+`, `-`, `*`, `/`, and `%`) with the assignment operator (`=`) to get `+=`, `-=`, `*=`, `/=`) and `%=` (*see* **Table 1.3**).

Another set of important arithmetic operators are the increment `++` and decrement `--` operators, respectively, which can be used as prefix (`++x`) or postfix (twx++) with different characteristics. They are applicable to integer and floating-point type variables. Prefix increment, `++x`, increases the value of the variable `x` by 1, and then returns the updated value. But postfix increment `x++` returns the current value of the variable `x` by 1, and then increases the value of `x`. The decrement operator is employed in a similar manner. These operators are commonly used in loops and repetitive constructions to efficiently modify the variable values.

Consider the following C++ code segment:

```
1    int X = 0;  // X is initialized by zero
2    X++;        // Equivalent to X = X +1, increment X by 1, X becomes 1
3    X+=2;       // Equivalent to X = X +2, increment X by 2, X becomes 3
4    X= X + 4;   // Increment X by 4, X becomes 7
5    X--;        // Equivalent to X = X -1, decrement X by 1, X becomes 6
6    X-=3;       // Equivalent to X = X -3, decrement X by 3, X becomes 3
```

Here, in this code segment, the comments on each line describe the tasks performed in that line. In line 1, `X` is defined as an integer and initialized to zero at the same time. In line 2, the memory value of `X` is substituted in the rhs, which updates the value of `X` as 1 but displays zero. In lines 3 and 4, `X` is incremented by 2 and 4, respectively. In lines 5 and 6, `X` is decremented by 1 and 3, respectively.

> In C++, *exponentiation operator* (raising a number to a power) is not directly supported by a built-in operator as in some other programming languages. The operation is generally carried out with the `exp(a*ln(b))` or `pow(base, exponent)` function of the `<math.h>` library.

6 RELATIONAL AND LOGICAL OPERATORS

Branching in a computer program causes a computer to execute a different block of instructions, deviating from its default behavior of executing instructions sequentially. **Table 1.3** summarizes the arithmetic, relational, and logical operators with illustrative examples.

The *relational operators* (`<, >, <=, >=, ==, !=`) are used to compare two values or expressions and give a boolean value: `true` or `false`. These operators are commonly used in conditional and control constructions to control the flow of a program based on comparisons. The relational operators can be used with various data types, including integers, floats, and characters. However, comparing different types may lead to implicit type conversion. While relational operators do not short-circuit like logical operators, they can be combined in logical expressions for more complex conditions.

The *logical operators*, logical AND (`&&`), logical OR (`||`), logical NOT (`!`), are used to combine or negate boolean expressions. They are essential in controlling the flow of programs, especially in conditional statements and loops. These operators are frequently used in conditional constructions, such as `if`-constructs, to evaluate multiple conditions or control the flow of loops based on complex logical expressions.

Logical_expression can be a combination of logical constants, logical variables, and logical operators. A logical operator is defined as an operator on numeric, character, or logical data that yields a logical result. There are two basic types of logical operators: relational operators and combinational (logical) operators.

Branching structures are controlled by *logical variables* and *logical operations*. Logical operators evaluate relational expressions to either 1 (`true`) or 0 (`false`). Logical operators are typically used with boolean operands. The logical AND operator (&&) and the logical OR operator (`||`) are both binary in nature (require two operands). The logical NOT operator (`!`) negates the value of a Boolean operand, and it is a unary operator.

Logical operators are used in a program together with relational operators to control the flow of the program. The `&&` and `||` operators connect pairs of conditional expressions. Let L_1 and L_2 be two logical prepositions. In order for L_1`&&`L_2 to be true, both L_1 and L_2 must be true. In order for L_1`||`L_2 to be true, it is sufficient to have either L_1 or L_2 to be true. When using `!`, a unary operator, in any logical statement, the logic value is changed to true when it is false or changed to false when it is true. These operators can be used to combine multiple expressions. For given `x=5`, `y=9`, `a=18`, and `b=3`, we can construct the following logical expressions:

```
(x < y && y < a && a > x)            //  1, i.e., true
(x < y && y > a && a >= b)           //  0, i.e., false
((x < y && y < a) || a < b)          //  1, i.e., true
((x > y || y > a) || a < b)          //  0, i.e., false
(!(x > 6) && !(y < 5)) && a > x)     //  1, i.e., true
```

The order of evaluation of `&&` and `||` is from left to right.

In C++, *exponentiation operator* (raising a number to a power) is not directly supported by a built-in operator as in some other programming languages. The operation is generally carried out with the `exp(a*ln(b))` or `std::pow(base, exponent)` function of the `cmath` library.

7 CONDITIONAL CONSTRUCTIONS

Branching control and conditional structures allow the execution flow to jump to a different part of the program. *Conditional* statements create branches in the execution path based on the evaluation of a condition. When a control statement is reached, a *condition* is evaluated, and a path is selected depending on the result

Table 1.3: Arithmetic, relational, logical, and compound assignment operators in C++.

Operator	Description	Example
ARITHMETIC OPERATORS		
±	Addition and subtractions	`a ± b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	finding the remainder (modulo).	`5 % 2`
RELATIONAL OPERATORS		
`==`	compares the operands to determine equality	`a == b`
`!=`	compares the operands to determine unequality	`a != b`
`>`	determines if first operand greater	`a > b`
`<`	determines if first operand smaller	`a > b`
`<=`	determines if first operand smaller than or equal to	`a > b`
`>=`	determines if first operand greater and equal to	`a > b`
LOGICAL OPERATORS		
`&&`	Logical AND operator	`a && b`
`||`	Logical OR operator	`a || b`
`!`	Logical NOT operator	`!(a)`
COMPOUND OPERATORS		
±=	Addition assignment (`p = p ± q`)	`p ±= q`
`*=`	Multiplication assignment (`p = p * q`)	`p *= q`
`/=`	Division assignment (`p = p /q`)	`p /= q`
`++`	Increment operator, which increments the value of the operand by 1	`i++` or `++i`
`--`	Decrement operator, which decrements the value of the operand by 1	`i--` or `--i`

of the condition.

For this purpose, C++ provides `if`, `if-else`, `if-else-if`, and `switch` constructions, which check a condition and execute certain parts of the code if <*condition*> (logical expression) is true.

7.1 `if`, `if-else`, `if-else if` CONSTRUCTIONS

The basic `if` construct shown below is the most common and simplest of the conditional constructs. It executes a block of statements *if and only if* a logical expression (i.e., *condition*) is true. This structure corresponds to the **If-Then-End If** structure of the pseudocode convention.

```
if (condition)
    { STATEMENT(s) }      // if condition is true
```

Note that the *condition* is enclosed with round brackets. The `if`-construct can also command multiple statements. By wrapping them in braces, the block of statements is made syntactically equivalent to a single statement.

A more general `if-else` construct (presented below) provides an alternative block for the statements to be executed in the case the *condition* is false. It is equivalent to the **If-Then-Else-End If** structure of the pseudocode convention. The **if-else** construct has the following form:

```
if (condition)
    { STATEMENT(s) }      // if condition is true
else
    { STATEMENT(s) }      // if condition is false
```

One may chain multiple conditions using `else if` to test multiple possibilities. Nesting `if` statements within other `if` statements is allowed. A more complicated `if-else if` construct can be devised as follows:

```
 if (condition1)
    { STATEMENTS-1 }      // if condition1 is true
 else if (condition2)
    { STATEMENTS-2 }      // if condition2 is true
 else if (condition3)
    { STATEMENTS-3 }      // if condition3 is true
 else
    { STATEMENTS-4 }      // if condition3 is false
```

This chain can be continued indefinitely by repeating the last statement with another `if-else` statement.

7.2 TERNARY CONDITIONAL OPERATORS

A conditional expression can be constructed with the ternary operator `"?:"`, which provides an alternate way to write `if-else` or similar conditional constructs. The syntax is given as

```
condition ? val_if_true : val_if_false
```

This statement evaluates the *condition* first. If *condition* is *true*, `val_if_true` is executed; otherwise, `val_if_false` is evaluated. Note that `val_if_true` and `val_if_false` must be of the same type, and they must be simple expressions rather than full statements. The following example, which determines the maximum of a pair of integers, illustrates the use of a ternary operator:

```
int a = 10, b = 20, c, d;

 c = (a > b) ? a : b;              // c = max(a, b)
 std::cout << c << std::endl;      // c becomes 20
 d = (a > 6 ? (b <= 25 ? 13 : 25) : 100);
 std::cout << d << std::endl;      // d becomes 13
```

In evaluating `c`, the condition `(a > b)` is evaluated, and since `a<b`, the value of `c` is set to `b`, i.e., 20. In evaluating `d`, the condition `(b <= 25)` is evaluated, which yields `13` since the condition is `true`. Then `(a > 6)` is evaluated to give `13` since this condition is also `true`.

7.3 `switch` CONSTRUCTION

The `switch` construction is an alternative to the `if-else-if` ladder. A `switch` construction allows a multi-decision to be executed based on the value of a switch variable (`int`, `char`, or `enum`). It is a cleaner alternative to using multiple `if` statements when you have many conditions based on a single variable.

Using `switch` can improve the clarity and maintainability of the code when dealing with multiple conditions based on a single variable. The general form of the `switch` statement is as follows:

```
switch (expression) {

    case value1:          //  case of expression = value1
       { STATEMENTS-1 }   //  statements to be executed
       break;             //  exit the construction

    case value2:
       { STATEMENTS-2 }
       break;

    case value3:
       { STATEMENTS-3 }
       break;
        ....
    default:
       { STATEMENTS-d }
}
```

where **value1**, **value2**, and so on are integers. The expression in the switch must evaluate to an **int**, **char**, or **enum**. Upon evaluating **expression**, if and when it matches one of the available cases, the switch branches to the matching case and executes the statements from that point onwards. Otherwise, it branches to **default**, which is optional, and executes its statements.

The following C++ code segment uses **year** to execute **switch** construct. For the case of **year=1**, **Freshman** is displayed; for **year=2**, **year=3**, and **year=4**, **Sophomore**, **Junior**, and **senior** are displayed, respectively. If **year** corresponds to none of the above, the message **Graduated** is displayed.

```
int year=3;        // year is initialized
switch (year) {

    case 1:
        cout << "Freshman";
        break;
    case 2:
        cout << "Sophomore";
        break;
    case 3:
        cout << "Junior";      // Output is  Junior
        break;
    case 4:
        cout << "Senior";
        break;
    default:
        cout << "Graduated";
        break;
}
```

Invoking `break` at the end of each case block exits the `switch` construction. If omitted, the execution will "fall through" to the next case, which can be useful in certain scenarios.

8 CONTROL CONSTRUCTIONS

Control (loop) constructions are used when a program needs to execute a block of instructions repeatedly until a *condition* is met, at which time the loop is terminated. There are three control statements in most programming languages that behave in the same way: `while`, `do-while` (equivalent to `Repeat-Until` loop of pseudocode convention), and `for`-constructs.

8.1 `while` CONSTRUCTION

A `while`-construction, which works the same way as the **While**-loop in pseudocode, has the following form:

```
while (condition)
   { STATEMENT(s) }    // if  condition is true
```

In `while` construction, *condition* is evaluated before the block statements. If *condition* is `true`, the *block of statements* inside the loop will be executed. If *condition* is initially false, the statement block will be skipped.

For successful implementation of `while` constructions, make sure that (i) the loop variable is initialized before the loop starts; (ii) the loop variable is updated within the loop or an infinite loop might occur; and (iii) be cautious of conditions that might always evaluate to true, leading to infinite loops.

8.2 `do-while` CONSTRUCTION

The `do-while` construction is equivalent to the **Repeat-Until** loop in the pseudocode. The test *condition* is at the bottom of the loop. It is similar to `while` construction in that a code block (statements) is executed as long as the *condition* is false.

The `do-while` construction has the form

```
do
    { STATEMENT(s)  }   // if condition is false
while (condition)
```

Note that the *block of statements* is executed at least once even if *condition* is `false` from the beginning. This loop is useful when the initial execution of the block statements is necessary, such as prompting for user input.

8.3 `for` CONSTRUCTION

The `for`-construction is a control flow statement used when a block of code needs to be executed a specified number of times. It is equivalent to **For**-loop in the pseudocode. It is particularly useful when the number of iterations is known beforehand. A `for`-construction has the form

```
for (init; condition; update)
    {  STATEMENT(s)  }  // statements in the block are executed
```

where *init* denotes declaration and initialization of the loop index or loop counter, *condition* is the loop-continuation condition, and *update* denotes updating the loop index by incrementing or decrementing it. The most common way to update the loop index in unity steps is by using increment (`++`) or decrement (`--`) operators.

Consider the following loops:

```
int i, j, k, m; float x;
   for (i=2; i<10 ; i++) {
       STATEMENT(s)    // The block is executed for i=2, 3,..., 9
   }
   for (j=1; j<9 ; j+=2) {
       STATEMENT(s)    // The block is executed for j=1, 3, 5, 7
   }
   for (k=5; k>2 ; k--) {
       STATEMENT(s)    // The block is executed for k=5, 4, 3
   }
   for (m=6; m>0 ; m-=2) {
       STATEMENT(s)    // The block is executedfor m=6, 4, 2
   }
   for (i=5; i<=20 ; i+=5) {
       STATEMENT(s);   // The block is executedfor i=5, 10, 15, 20
   }
   for (x = 0; x <= 1; x+=0.2) {
       STATEMENT(s);   // The block is executedfor x=0, 0.25, 0.5, 0.75, 1
   }
```

The values that the loop index takes are specified in the comment lines above. Here, we can also see that incrementing or decrementing the loop index using values other than one is accomplished through the use of compound assignments, as in `j+=2`, `m-=2`, or `i+=5`. In these examples, the part of the `for`-loop that updates the loop variable, `i += n` or `i -= n`, allows one to specify how much to change the index with each iteration. The value of `n` can be any integer (or even a floating-point number) to control the loop's behavior as desired. Additionally, multiple variables can be declared in the initialization section, separated by commas, for example: `for (int i = 2, j = 99; i < j; i++, j-)`.

8.4 break AND continue STATEMENTS

As shown before, `break` was used to branch out of a `switch` statement. Likewise, it could also be used to branch out of any of the control constructs. A `break` statement can be used to terminate the execution of `switch`, `while`, `do-while`, or `for`. It is important to realize that a `break` will only branch out of an innermost enclosing block and transfer program flow to the first statement following the block. For example, consider the following nested `do`-loops in C++ with invoking "using namespace std":

```
int i, d;
for (d=2; d<4; d++) {                            // Output is
   for (i=d; i<= 6; i++) {                       //  d=2 i=2
       cout << "d=" << d << " i=" << i << endl;  //  d=2 i=3
       if (i==4) {                               //  d=2 i=4
           break;                                //  d=3 i=3
       };                                        //  d=3 i=4
   };
}
```

The `break` branches out of the innermost `for`-loop when `i` becomes `i=4` as the outermost `for`-loop continues to run over all available index values.

The `continue`-statement operates on `while`, `do-while`, or `for` loops but not on `switch`. In `while` and `do-while` constructs, the loop-continuation test is evaluated immediately after the `continue` statement executes.

In the `for` loop of the following code segment, the loop control variable is initialized at `n=1`. Then the loop condition (Is `n<=5`?) is evaluated. When the loop variable becomes 3 (i.e., `n=3`), `continue` statement skips the rest of the statements and `cout` and takes the iteration to top of the loop.

```
        for (int n = 1; n <=6; ++n) {
            if (n == 3)
            {   // skip loop if n=3
                continue;   // skip remaining code in loop body
                };
            int n2=n*n;
            int n3=n*n2;
            cout << "n= " << n << " " << n2 << " " << n3 << endl;
        };
```

The code output is as follows:

```
n= 1 1 1
n= 2 4 8
n= 4 16 64
n= 5 25 125
n= 6 36 216
```

8.5 EXITING A LOOP

It is sometimes required to exit a loop other than by the *condition* at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do-while`, just as from `switch`.

In the C++ library `cstdlib`, the `exit()` function provides the user to *terminate* or *exit* the calling process immediately. The `exit` function can be used anywhere in the program, subprograms, or loops. When the `exit()` function is invoked, it closes all open file descriptors belonging to the process and any children of the process inherited.

```
    int i, d;
    for (d=2; d<4; d++) {
       for (i=d; i<= 8; i++) {
           printf ("d=%d i=%d\n", d,i);
           if (i==5) {
               exit(0);   // program is terminated
               };
        };
    };                    // all statements after the 'exit' are skipped
    cout << "d=" << d << " i=" << i << endl;
```

where `<iomanip>` header file and "`using namespace std`" should be invoked at the beginning of this code.

The code output look as follows:

```
d=2 i=2
d=2 i=3
d=2 i=4
d=2 i=5
```

9 ARRAYS

In C++, an array, as a special case of a variable, is a collection of elements of the same type stored in contiguous memory locations. Arrays can have one, two, or more dimensions, i.e., a_1, a_2, a_3, $b_{1,1}$, $b_{2,1}$, $c_{1,1,3}$, and so on. The subscripts are always *integers*.

Arrays must always be explicitly declared at the beginning of each program because the range and length of an array are critical in programming. To create an array, the name of the array followed by square brackets `[ ]` is specified after defining the data type. For example, in the following code segment, the first expression creates a one-dimensional array `arr`, having six integer elements. Each element can be accessed or referred to by a subscript in brackets; i.e., `a[0]`, `a[1]`, ..., `a[5]`. The second expression creates a two-dimensional array `b` having four elements: `b[0][0]`, `b[0][1]`, `b[1][0]`, and `b[1][1]`.

```
int arr[6];        // one-dimensional integer array of length 6
float b[2][2];     // two-dimensional float array of length 4
```

In C++, the index of an array starts at 0 (*zero*). The valid indices for an array of size n are 0 to $n-1$. Thus, to access the last element of a one-dimensional array, the index value of $n-1$ is used.

Arrays can be initialized during the declaration as follows:

```
type  array-name [size] = { list of values } ;
```

Consider the following array initializations:

```
int a[6] ={5, 2, -4, 1, 7, -8};          // initialization of 1-d array
float b[2][2]={{4.2, 3.7},{1.9, 8.3}}; // initialization of 2-d array
int c[] ={8, 3, 6, 1, 7, 1, 4};          // initialization of 1-d array
char car[3] = {'a', 'b', 'c'};           // initialization of 1-d array
int arrB[9] = {9,-2};                    // Initialize first two elements
float d[31] = {0.0};                     // initialization with zeros
```

Here, the initialization of `a` gives `a[0]`=5, `a[2]`=2, `a[3]`=-4,......, `a[5]`=-8. The two-dimensional array `b` is initialized as follows: `b[0][0]`=4.2, `b[0][1]`=3.7, `b[1][0]`=1.9, and `b[1][1]`=8.3. In the initialization of `c`, the size of an array is not specified, in which case the compiler automatically assigns a size equal to the number of elements with which the array is initialized. The character type array **`car`** has a length of 3 and is initialized as `char[0]='a'`, `char[1]='b'`, and `char[2]='c'`. In the integer array **`arrB`**, the first two elements are initialized: `arrB[0]=9` and `arrB[1]=-2`; others are set to zero. We often need to initialize a large array with a specific value. This can be simply done as `array-name[size]=value`. Note that the initialization of `d` with zero is carried out by matching the type and size of the array.

As for one-dimensional arrays, multi-dimensional arrays may be defined without a specific size. However, only the left-most subscript (i.e., *the number of rows*) can be omitted; all other dimensions must be specified. The compiler determines the size of the omitted dimension based on the initializer list. Accessing elements is done using multiple indices. For instance, accessing the second diagonal element of **matE** is shown below:

```
int dia2 = matE[2,2];  // accesses the 2nd diagonal element of matrix matE
matE[3,2] = 93;        // changes the element at (3,2) position to 93
```

Looping through arrays is permitted. For example, the following **for** loop displays each element of **arrA**:

```
for (int i = 0; i < 5; i++) {
    std::cout << arrA[i] << std::endl;    // displays each element
    }
```

> The C++ language does not support *whole array operations*, like FORTRAN 90-95, MATLAB, R, or PYTHON with **NumPy**. However, the pseudocode convention adopts *whole array arithmetic*, and array names are treated as if they were scalars. To adapt it to C++ programming, **For** loops need to be used for such operations.

10 CLASS TEMPLATE vector

The C++ Standard Library also includes a class template **vector**, similar to class template **array**, but denotes dynamic arrays. A **vector** is a collection of indexed same-type objects. A **vector** is also referred to as a *container* since it "contains" other objects.

The **vector** class supports dynamic resizing meaning that a vector can be expanded or shrunk at run-time and provides random access to elements using the subscript operator (**[]**) and the at () method, which includes bounds checking. It automatically handles memory allocation or deallocation, reducing the risk of memory leaks. It also offers many member functions for managing data, such as **push_back()**, **pop_back()**, **insert()**, **erase()**, and more. This template provides fast access to elements and efficient use of memory, although inserting or deleting elements in the middle can be slower due to shifting.

10.1 DECLARING AND INITIALIZING A vector

To use a **vector**, you need to include the appropriate header. If you do not wish to use **std::vector** each time you invoke a vector operation, then you should also invoke **using namespace**, as shown below:

```
#include <vector>
using namespace std;
```

Note that a **vector** is a *template*, not a *type*. Types that are generated from **vector** should include the element type, as in **vector<int>**. A **vector** must be declared starting with the **vector** keyword as follows:

```
vector <data_type> vector_name
std:: vector <data_type> vector_name // if std namespace not used
```

The data type is specified by angle brackets containing the type of data the vector will accept. Note that if **'using namespace std'** is not invoked, then **vector** needs to be specified with the namespace **std**, as shown below.

```
vector<int> nvec;              // nvec holds objects of type int
vector<char> cvec;             // cvec holds objects of type char
std::vector<int> n2v;          // nvec holds objects of type int
std::vector<float> f2v;        // fvec holds objects of type float
```

In the following code, the vector template is included in line 2. The length of the vector is set to 5 in line 6, i.e., `vitem = 5`. Then the vector **aVec** is declared in line 7. The first parameter of **aVec** is the maximum number of items, and the second one is the value to be stored in the vector. This results in assigning the value of "1" to all available items, which gives 1 for all five items.

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int main() {
6    int vitem = 5;
7    vector<int> aVec(vitem, 1);
8            for (int j : aVec)      // the output looks like as
9                cout << j << " ";   // 1 1 1 1 1
10 }
```

11 FUNCTION CONSTRUCTIONS

Declaring and using functions is a fundamental part of writing structured and modular code. Functions help organize code into reusable blocks, making it easier to read, maintain, and debug. A small sample of the C++ cmath library functions are presented in **Table 1.4**, where the variables x and y are of type **double**. Always refer to the official documentation or resources specific to your compiler for any additional functions or variations that might be available.

Table 1.4: Some of the available `<cmath>` library functions.

Functiom	**Description**
`fabs(x)`	absolute value of x
`sqrt(x)`	square root of x
`round(x)`	rounds x to nearest integer
`trunc(x)`	truncates x to nearest integer part
`ceil(x)`	the smallest integer greater than or equal to x
`floor(x)`	the largest integer less than or equal to x
`fmod(x, y)`	the remainder of x divided by y
`pow(x, a)`	**x** raised to the power of **a**
`exp(x)`	the exponential value of e^x
`log(x)`	the natural logarithm (base e) of x
`log10(x)`	the logarithm (base 10) of x
`log2(x)`	the logarithm (base 2) of x
`sin(x)`	the sine of x (angle in radians)
`asin(x)`	the arcsine of x (result in radians)
`atan(x)`	the arctangent of x (result in radians)
`atan2(x)`	the arctangent of x, taking into account the signs of both arguments
`sinh(x)`	the hyperbolic sine of x

The C++ main program itself is a *function* designed to perform a specific task (i.e., the aim of the program). A program often requires repetitive tasks. On the other hand, in practice, it is impractical to write a complete *program* from A to Z that includes everything within a **main** program. Such programs would not only be too long but also too complicated. To avoid repetition, **function**s are utilized to perform specific tasks. In this regard, functions can be viewed in two categories: (i) those provided with the standard C++ library and (ii) those prepared by the user (i.e., user-defined functions). The C++ standard library provides a collection of functions to perform common mathematical calculations, string, character, or input/output manipulations, etc. On the other hand, the programmer does also need to prepare *user-defined functions* to perform specific tasks that are not available in standard libraries, which may be used once or numerous times in the program.

11.1 FUNCTION DECLARATION

The general structure of a **function** is given below:

```
return-type function-name ( p1, p2, ..., pn )
     {   code block               // function body
         return value;            // set function-name = value
     }
```

where **p1**, **p2**, ... , **pn** are the input parameters separated with commas, **value** is the returned function value, and the **function-name** is a valid identifier that is descriptive of what it does. In addition to constants and variable declarations, the function body, **code block**, includes commands and expressions to be executed to carry out a specific task when the program runs. The **return-type** is the data type of the result returned to the caller. A **void** return-type indicates that a function does not return a value. The return-type, function-name, and argument list together are often referred to as the *function header*.

The local variables should also be declared before they can be used. The declarations and statements within braces form the function block. There are two ways to return (as illustrated below) from a called function to the point at which it was invoked.

```
return value (or expression);   // in return-type functions
return;                         // in void-return-type functions
```

A void return-type function does not return a value, while a return-type function returns a result computed from the **expression**. The return-type of a function is the type of value that the function returns. The **main(void)** function (or **main()**) however returns zero (i.e., **return 0;**), and the word **void** as argument (or no argument) indicates that the **main** has no arguments.

The following is an example of a void function that generates no computed value or return data.

```
void print_welcome( ) // a void function!
{
   printf("Welcome to the game!");
   return;            // A void function can be terminated by 'return'
}
```

A function can also have its own internal (*local*) variables that are accessible only internally, i.e., its content is invisible to other functions. A function is only prepared once, and it is accessed and executed from the **main** function or any other **function** whenever needed. Once a return type function completes the specific task it is supposed to perform, at least one output value is returned to the calling function. There is no restriction on the number of functions. A function can be used with any other relevant programs and can be invoked or accessed many times in the same program.

The following (return-value) function finds the maximum of two real numbers **a** and **b**.

```
float                  // this is the 'return type'
findmax( float a, float b) // function name 'max' is followed by parameters
{ if (a > b)
      return a;        // set return value to 'a'
  else
      return b;        // set return value to 'b'
   }
```

Note that this **float**-type function returns a **float** value under any circumstances.

Functions are called by naming the function along with their entire parameter (or agument) list (i.e., referred to as function call) to which information is passed. Consider the following main program involving the use of **findmax**.

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  float findmax( float a, float b) // A float function 'findmax'
6   { if (a > b)          // It has two (a and b) input parameters
7        return a;        // set return value to 'a', a float type
8      else
9        return b;        // set return value to 'b', a float type
10      }                 // end of function
11 int main() {
12     float a, b, c, d, s1, s2;
13     cout << "Enter four numbers\n";
14     cin >> a >> b >> c >> d;
15     s1 = findmax(a, b);
16     s2 = findmax(c, d);
17     cout << "The max. value is " << fixed
18     << setprecision(1) << findmax(s1, s2) << endl;
19     return 0;
20 }
```

Here, in this program, **findmax** is invoked in lines 16, 17, and 19. In line 16, **s1** is set to the maximum of **(a,b)**. In line 17, **s2** is set to the maximum of **(c,d)**. In line 19, the maximum of (**s1**,**s2**) is directly evaluated and printed. In this C++ code, **findmax** is placed before **main**. So when the compiler encounters **findmax**, it does *not* have any information about **findmax**, its arguments, and so on. However, if **function findmax** is placed after the main code, the compiler will *not* know what **findmax** is. To avoid such problems, each function should be placed before first use. However, as a general and better practice, the function prototypes are placed at the top of the program before **main()**, and function definitions after the **main()**. This provides the compiler with a brief glimpse of a function whose full definition will appear later, as shown below:

```
return-type function-name-1( p11, p12, ..., p1n ) ;
return-type function-name-2( p21, p22, ..., p2n ) ;
       ...
int main(void) {
```

```
        Declearations and Statements
         }
    return-type function-name-1( p11, p12, ..., p1n ) {
        Declearations and Statements
        return value-1;
         }
    return-type function-name-2( p21, p22, ..., p2n ) {
        Declearations and Statements
       return value-2;
         }
         ...
```

The placement of the functions, however, can be changed. The only requirement is that a function cannot be called before it has been declared or defined.

> In most computer programming languages, there are two ways to pass arguments: *pass-by-value* and *pass-by-reference*. When arguments are 'passed by value', changes to the argument do not affect an original value of the variable in the caller. When an argument is 'passed by reference', the called function can modify the original value of the variable. In C++, all arguments are 'passed by value'. Through the use of pointers, it is possible to achieve pass-by-reference.

In the following program, the function declaration (the first line of a function) is placed in line 3 before **`main()`** at line 5. The complete function definition is placed after **`main()`** in lines 15-20.

```
1  #include <iostream>
2
3     float findmax( float a, float b); // function declaration
4
5     int main()  {  // main function starts here
6         float a, b, c, d, s1, s2;
7             printf("Enter four numbers\n");
8             scanf("%f %f %f %f", &a, &b, &c, &d);
9             s1 = findmax(a, b);
10            s2 = findmax(c, d);
11            printf("The max. value is %.3f\n", findmax(s1, s2));
12        return 0;
13    }
14
15    // complete function definition placed after 'main'
16    float  findmax( float a, float b) {
17         if (a > b)
18            return a;           // return-value
19         else
20            return b;           // return-value
21    }     // findmax ends here
```

An efficient program generally consists of one or more independent modules. The modular programming approach is also easier to conceptualize and write a program as a whole.

11.2 RECURSIVE FUNCTION

In C++, a function in a program is allowed to call itself multiple times. A recursive function is a function that can call itself repeatedly until a certain condition or task is met. This approach is often used in cases where a task can be broken down into smaller, similar sub-problems or navigating data structures like trees.

Consider the recursive function $f_n(x) \leftarrow n + x * f_{n-1}(x)$ with $f_0(x) = x$. This function can be designed as a recursive function as follows:

```
1  #include <iostream>
2
3  float FX(int n, float x) {
4     float f;
5     if(n==0)
6          f = x;
7      else
8          f = (float)n + x * FX( n - 1, x);
9      return f;
10 }
11
12 int main() {
13     int n = 5;
14     float x = 1.1;
15     std::cout << "FX of " << n << " is " << FX(n,x) << std::endl;
16     return 0;
17 }
```

A local intermediate variable is defined (**`float f`**) and used in lines 6 and 8 to store intermediate results. The self-calling takes place in line 8 as **`FX( n - 1, x)`**. On function name **`FX`** takes the value of **`f`** on return.

11.3 FUNCTION AS A SUBPROGRAM

A subroutine, method, procedure, or subprogram in other languages allows multiple returns (i.e., outputs). However, C++ a **`function`** can only return one value directly. To achieve multiple returns in C++ using functions can be achieved using **`std::pair`**, **`std::tuple`**, **`pointers`**, or **`references`**.

If the function has exactly two outputs, **`std::tuple`** can be used. The following is an example implementing **`std::pair`**:

```
#include <iostream>
#include <utility>     // to use function 'pair'
using namespace std;
pair<int, int> minmax(int a, int b) {
    if (a>b)           // check condition
       return make_pair(b, a);
    else
       return make_pair(a, b);}
```

```

int main() {
    auto result = minmax(6, 9);
    cout << "Min: " << result.first << ", Max: " << result.second << endl;
    return 0;
}
```

Here, the **minmax** is defined as **pair** function with two integer input parameters (and integer outputs). It uses **std::make_pair**, a utility function, that creates a **std::pair** object, which is a simple data structure that holds two values. This function is a part of the C++ Standard Library and is defined in the **<utility>** header. The **auto** keyword is a type specifier that allows the compiler to automatically deduce the type of a variable based on its initializer.

The function parameters can be passed by reference or as pointers, allowing the function to modify the values of those parameters directly.

```
#include <iostream>
using namespace std;

void input(float &x, float &y, float &z) {
    cout << "Enter three numbers => ";
    cin >> x >> y >> z;
}
void calc(float a, float b, float c, float &sum, float &ssqr) {
    sum = a + b + c;
    ssqr = a * a + b * b + c * c;
}
void output(float sums, float ssqr) {
    cout << "The sum of the numbers entered : " << sums << endl;
    cout << "The sum of the squares of the numbers : " << ssqr << endl;
}

int main() {
    float a, b, c, sums, ssqr;
    input(a, b, c);
    calc(a, b, c, sums, ssqr);
    output(sums, ssqr);
    return 0;
}
```

Note that all functions are **void** type. The function **input** has three outputs (**x**, **y**, and **z**), which are passed to the caller program with the & symbol, which allows the arguments to be passed by reference. The function **calc** has three inputs (**a**, **b**, and **c**) and two outputs: the sum of the numbers (**sums**) and the sum of the squares of the numbers (**ssqr**), which are passed to the caller by the & symbol. The **output** function, printing out the results, has only input parameters.

The code output for the input values of 2, 3, and 4 reads as follows:

```
Enter three numbers => 2 3 4
The sum of the numbers entered : 9
The sum of the squares of the numbers : 29
```

11.4 ARRAY ARGUMENTS

Arguments can often be arrays. When an argument is a one-dimensional array, the length of the argument may not need to be specified.

There are basically two ways to pass an array to a function: *call by value* or *call by reference*. When using the call by value method, the argument needs to be an initialized array, or an array of fixed size equal to the size of the array to be passed. In the call by reference method, the argument is a pointer to the array.

In the following code, the `main()` function has an array of integers. A user-defined function `max_of_array` is called by passing array `arr` to it. The `max_of_array` function receives the array and searches for the largest element using a `for` loop. After `maks` is set to `arr[0]`, whenever an array element with a value greater than `maks` is found, it is set to `maks`. By the time the end of the loop is reached, `maks` gives the largest value in the array `arr`.

```
1  #include <iostream>
2  #include <array>
3
4  using namespace std;
5
6  int max_of_array(const array<int, 5>& arr); // declare function
7
8  int main() {                        //  main function
9     array<int, 5> arr = {21, 47, 93, 38, 25}; //initialize array
10
11    cout << "Max value is " << max_of_array(arr) << endl; // print maximum
12 }                                   // end of main
13
14 int max_of_array(const std::array<int, 5>& arr) {  //  define function
15    int maks = arr[0];
16    for (size_t k = 1; k < arr.size(); k++) {
17       if (arr[k] > maks) {
18           maks = arr[k];
19       }
20    }
21    return maks;
22 }
```

Note that the length of `arr` is specified in lines 6 and 14 as `5` in the declaration and definition of the function `max_of_array`. In lines 9 and 17, the length of `arr` is compatible with the data.

In the following version, the `max_of_array` function is defined with two arguments, an uninitialized array without any size specification. The length of the array declared in the `main()` function is obtained by using the `sizeof` function, which gives the size (in bytes) that a data type occupies in the computer's memory. To find the number of integer values, the total memory size of the array (`sizeof(arr)`) is divided by the size of the single integer data type, i.e., `sizeof(int)`.

```
#include <iostream>
#include <vector>

using namespace std;

```

```
int max_of_array(int n, const vector<int>& arr); // declare function

int main() {
    vector<int> arr = {21, 47, 93, 38, 25}; // initialize array
    int n = arr.size();
    cout << "Max value is " << max_of_array(n, arr) << endl; // print max. val
}

int max_of_array(int n, const vector<int>& arr) { // define the function
    int maks = arr[0];
    for (int k = 1; k < n; k++) {
        if (arr[k] > maks) {
            maks = arr[k];
        }
    }
    return maks;
}
```

This version of the function uses the `vector` template and is more flexible, allowing it to handle array variables of different lengths.

Bibliography

[1] Bronson, G. J., BORSE, G. J. *C++ for Engineers and Scientists*. Cengage Learning, 2009.

[2] DAVIS, S. R.. *C++ For Dummies*. Wiley, 2014.

[3] DEITEL, P., DEITEL, H. C., *C How to Program: with an introduction to C++*, 8th Ed.. Pearson Education, 2016.

[4] GADDIS, T. *Starting Out with C++ from Control Structures to Objects*. Pearson Education, 2017.

[5] GREGORIRE, M., *Professional C++*. Wiley, 2021.

[6] LOSPINOSO, J., *C++ Crash Course*, A Fast-Paced Introduction. No Starch Press, 2019.

[7] LIPPMAN, S. B., LAJOIE, S., MOO, B. E., *C++ Primer*, 5th Ed. Addison-Wesley, 2013.

[8] STROUSTRUP, B. *The C++ Programming Language*. Pearson Education, 2013.

[9] www.tutorialspoint.com

[10] www.w3schools.com

[11] https://www.learn-cpp.org

[12] www.programiz.com

[13] www.codechef.com

[14] ww.gnu.org

[15] ww.gisocpp.org