Ministry of Higher Education and Scientific Research
BadjiMokhtar Annaba University
Faculty of Technology

وزارة التعليم العالي و البحث العلمي
جـامعة باجـي مختـار- عنابـة
كليـة الــتكنـولوجيا

Département Electronique.

Polycopié pédagogique

Dossier numéro :.....

Titre

Conception orientée objet

Cours destiné aux étudiants de

Master 1 Automatique et informatique industrielle

Année : 2024

Table of Contents

Contents

1 Chapter 1: Introduction

The adoption of object-oriented technologies in computer science brings numerous advantages that successfully address the major challenges of the new era of computing, notably modularity, reusability, and scalability. Here are some reasons why these technologies are widely favored to tackle these challenges:

1.1 Modularity:

Objects represent autonomous entities that encapsulate data and behaviors. Breaking down a system into objects helps to divide complex functionalities into manageable units. Interfaces between objects define how they interact, promoting a modular design and a clear understanding of the system.

1.2 Reusability:

Classes and objects can be designed to be generic and reusable in different contexts. Object libraries offer pre-designed solutions, allowing developers to save time by using existing components rather than starting from scratch. Reusability enhances code consistency and quality by incorporating tested components from one project to another.

1.3 Scalability:

Objects can be extended or modified without affecting the rest of the system, thanks to mechanisms such as adding new methods or redefining existing ones. The use of encapsulation minimizes the impact of changes by keeping the internal details of an object hidden. Scalability becomes particularly crucial in an environment where needs can change frequently.

1.4 Encapsulation:

Encapsulation bundles data and methods within a single entity. It protects internal data by limiting direct access, following the principle of "private data, public methods." Maintenance is simplified by reducing dependencies and isolating changes within objects.

1.5 Inheritance:

The inheritance mechanism allows a derived class to inherit properties and methods from a base class, promoting code reusability. Class hierarchies organize code logically and structurally.

1.6 Polymorphism:

Polymorphism allows objects of different classes to respond uniformly to identical messages. This facilitates the writing of generic code capable of handling various objects consistently. Polymorphism allows object substitution without modifying the code that uses them, providing flexibility to the system.

1.7 Exception Handling:

Object-oriented languages often integrate exception handling mechanisms, providing structured error management. Exceptions can be raised and captured at the appropriate level, improving code robustness. Exception handling contributes to making the code more reliable and predictable in case of errors.

By combining these concepts, object-oriented technologies provide a robust framework for software design, addressing the requirements of the new computing paradigm focused on modularity, reusability, and scalability. The use of object-based component libraries complements this approach, offering a rich ecosystem of ready-to-use solutions.

The integration of component libraries is a common practice in software development, offering numerous significant advantages. These libraries, gathering pre-written modules or code snippets ready for use, are designed to be reused in various projects. Here are some reasons why the adoption of component libraries is beneficial:

Reusability: Developers can leverage well-tested and well-designed code across different projects, saving time and development efforts.

Time Savings: By using existing components, developers expedite the development process, avoiding the recreation of common functionalities.

Consistency: Component libraries provide standardized solutions, ensuring coherence across the entire software system.

Reliability: Library components are often subjected to rigorous testing, minimizing the risks of errors and bugs in new code.

Ease of Maintenance: Updates or fixes applied to a component library propagate to all projects using it, simplifying maintenance.

Focus on Business Logic: By using pre-existing components for generic tasks, developers can concentrate more on the business logic specific to their application.

Flexibility: Component libraries offer flexibility in choosing tools and technologies tailored to a specific project while benefiting from proven solutions.

Scalability: The integration of modular components makes the software system more scalable, facilitating the addition of new features or adaptation to changes required by evolving needs.

2 Chapter 2 : Basic concepts

C++, designed by Bjarne Stroustrup, is an extension of the C language that incorporates object-oriented programming. Known for its power and flexibility, C++ has become essential in software development due to its versatility and high performance.

2.1 Brief History of C++:

In **1979-1980**, Bjarne Stroustrup commenced the development of C++ at Bell Labs, extending the C language and initially naming it "C with classes." By **1983**, the term "C++" was officially used to designate the language, and in **1985**, the first public version of C++ was published. The year **1990** marked the adoption of the first C++ standard, commonly known as C++98, formally defining the language.

In **2003**, the C++03 standard was adopted, introducing minor corrections. The landscape changed significantly in **2011** with the introduction of the C++11 standard, which brought major features like auto, lambdas, and lambda functions. Subsequently, in **2014**, the C++14 standard was adopted, incorporating minor additions compared to C++11.

The C++ evolution continued in **2017** with the introduction of the C++17 standard, featuring additional improvements, including automatic type deduction for function arguments. In **2020**, the C++20 standard was released, introducing new elements such as concepts, ranges, and notable enhancements.

C++ maintains its momentum with regular standards updates, with the active participation of the developer community in contributing to its ongoing development.

The genesis of the C++ language is shaped by a myriad of influences and inspirations:

Extension of the C Language: C++ was conceived with the objective of enhancing the syntax of the C language, preserving its familiarity while introducing new features.

Object-Oriented Programming (OOP): Fundamental concepts of object-oriented programming, such as class, derived class, and virtual function, were incorporated into C++ to support OOP principles.

Inspirations from Simula67: The language drew inspiration from Simula67, the pioneering object-oriented programming language, particularly in the incorporation of concepts like class, derived class, and virtual function.

Operator Overloading: C++ features for operator overloading and the flexible placement of operators reflect similarities with the Algol68 language.

Inspirations from Generic Languages: The utilization of templates (prototypes) in C++ finds its roots in the generics of Ada and the parameterized modules of Clu.

Exception Handling: The mechanisms for exception handling in C++ were influenced by languages such as Ada, Clu, and ML.

Through the amalgamation of these diverse influences, the C++ language emerged as a versatile and robust programming language. By seamlessly integrating low-level features inherited from C with object-oriented programming concepts and drawing inspiration from various languages, C++ offers developers a powerful and flexible tool for software development.

2.2 Structure of a C++ Program:

The typical structure of a C++ program comprises the following components:

1. **Preprocessor Directives:**

Inclusion of essential libraries, such as <iostream>, to enable input/output operations:

```
#include <iostream>
```

2. **Namespace:**

Facilitates the use of names from the standard library without explicit qualification:

```
using namespace std;
```

3. **Function Declaration (Prototype):**

Recommended when functions are defined after the main function, informing the compiler of their existence: `void myFunction();`

4. **Main Function:**

Entry point of the program where execution commences. Every C++ program must have a main function.

```
int main() {
// Main function code
return 0;
}
```

5. **Variable Declarations: Declaration of variables used in the program:**

```
int age = 25;
double salry = 50000.50;
```

6. **Instructions - Executable Code:**

The core of the program, containing instructions executed sequentially:

```
cout << "Hello, World!" << endl;
```

7. **Return from the Main Function:**

The main function may return a value, typically 0, to indicate normal program exit:

```
return 0;
```

8. **Function Definitions:**

If the program contains functions defined outside of main, their definitions appear after main or at an appropriate location.

```
void myFunction() {
// Function implementation
}
```

This structured approach ensures the clarity and organization of a C++ program, facilitating readability and maintenance.

```
// 1. Preprocessor Directives
#include <iostream>  // Includes the standard input/output library
// 2. Namespace
using namespace std;
// 3. Function Declaration (Prototype) if necessary
// 4. Main Function
int main() {
    // 5. Variable Declarations
    // 6. Instructions - Executable Code
    cout << "Hello, World!" << endl;
    // 7. Return from the Main Function
    return 0;
}
// 8. Function Definitions if necessary
```

2.3 Creating a C++ Program:

Creating a C++ program involves the following four steps:

1. **Text Editor :**

Selecting a text editor involves choosing the platform where you will compose your C++ code. Options include lightweight text editors like Visual Studio Code or Sublime Text, or integrated development environments (IDEs) like Visual Studio, based on personal preferences. Text editors enhance the coding experience by offering features like syntax highlighting, autocomplete, and other tools that contribute to a more efficient development process.

2. **Writing the Code :**

In Step 2, you write the source code for your C++ program using the selected text editor from Step 1. This entails creating a file that contains the program's instructions, including defining variables, crafting executable instructions, and organizing the code into sections like preprocessor directives, namespace, main function, etc. Essentially, this step involves translating your conceptual idea into instructions that the computer can comprehend.

3. **Compiling the Program :**

In Step 3, you compile your C++ source code. Compilation is the crucial process of translating the human-readable code you've written into machine-readable language that the computer can understand. This task is accomplished using a C++ compiler. The compilation process creates an executable file from your source code, making it ready to be executed on your system. It is a vital step that verifies the syntax of your program and generates the corresponding machine code.

4. **Running the Program :**

In Step 4, you run the compiled program. After the source code has been converted into an executable file, you initiate this file to execute the program. This is the phase where your computer interprets the machine instructions produced by the compiler, enabling your program to operate as intended. This step provides the opportunity to observe the outcomes of your efforts and evaluate the behavior of your program.

2.4 Header Files :

Header files in programming, such as <iostream> in C++, contain function declarations, class definitions, constants, and other interface elements. They are employed to share information among different parts of a program or between distinct programs.

Including header files is standard in languages like C and C++. In C++, the "i" prefix is often used, for instance, **<iostream>.**

Key features and uses of header files:

Declarations: These files contain the "declarations" of your program, specifying functions' names and other essential information.

Inclusion with #include Directive: Header files are incorporated into other source files using the « #include » directive. For example: `#include <iostream>`.

Avoid Repetition: Header files prevent redundancy by consolidating information that is used in multiple parts of your program, allowing it to be included wherever necessary.

Clarity and Structure: They enhance the clarity and organization of your program. By segregating "public" parts from more internal components, your code becomes more understandable, aiding those who read it in using it correctly.

2.5 The Preprocessor :

The preprocessor in C++ is a crucial phase in the compilation process that precedes the actual compiler. Its primary tasks involve handling the source file, eliminating comments (both « **/*** » and « ***/** » and those after « **//** »), and examining lines starting with « **#** ». These functionalities enable modifications to the source code, leading to enhanced readability, streamlined code management, and the ability to customize the program's behavior. In essence, the preprocessor plays a pivotal role in increasing the efficiency of software development by automating essential tasks involved in preparing the code for compilation.

2.6 Comments :

In C++, you can incorporate single-line comments by using //. Any content following // on the same line is treated as a comment and does not impact program execution. These comments are employed to annotate the code, offering explanations about its functionality or temporarily deactivating a line of code during the development process.

```
// This is a single-line comment
int variable = 42; // Another comment on the same line
```

Multi-Line Comments (Block Comments) :

In C++, multi-line comments are initiated with /* and concluded with */. These block comments are valuable for offering comprehensive explanations about code segments, intricate algorithms, or design decisions. They are not processed by the compiler and are solely intended for enhancing the readability and comprehension of the code.

```
/*
This is a
multiline comment.
*/
int anotherVariable = 17 ;
```

2.7 Variables:

In programming, variables are designated storage locations linked to values. They are used to store data temporarily or represent information within a program.

2.7.1 Declaration and Value Assignment:

In C++, declaring and assigning a value to a variable can be done as follows:

```
// Declaration of an integer variable
int number;
// Assignment of a value to the variable
number = 42;
// Simultaneous declaration and assignment
double pi = 3.14159;
```

Reserving memory space for the variable (int number;), then assigning a value to this variable (number = 42;). You can also combine these two actions in a single line, declaring and assigning the value at the same time (double pi = 3.14159;).

2.7.2 Variable Types:

In C++, variables have various types, each designed to store different kinds of data. Here is a compilation of fundamental data types in C++ along with a brief explanation:

1. **Integers (Integer Types):**

Int: Signed integer (typically 32 bits).

Short: Signed short integer (16 bits).

Long: Signed long integer (at least 32 bits).

long long: Signed very long integer (at least 64 bits).

2. **Decimals (Floating-Point Types):**

Float: Single-precision floating-point number.

Double: Double-precision floating-point number.

long double: Extended precision floating-point number.

3. **Characters (Character Types):**

Char: Character (8 bits).

wchar_t: Wide character (at least 16 bits).

Boolean (Boolean Type) : Bool : True (true) or false (false).

4. **Unsigned Types :**

unsigned int : Unsigned integer (typically 32 bits).

unsigned short : Unsigned short integer (16 bits).

unsigned long : Unsigned long integer (at least 32 bits).

unsigned long long : Unsigned very long integer (at least 64 bits).

```
int age = 25;              // Integer
double pi = 3.14159;       // Floating-point number
char letter = 'A';         // Character
bool isTrue = true;         // Boolean
std::string name = "John";   // String
std::vector<int> array = {1, 2, 3};  // Dynamic array
```

2.8 Rules for Writing Identifiers:

Identifiers in C++ are employed to label variables, functions, classes, objects, and other elements within the code. The rules governing the writing of identifiers in C++ are as follows:

1. **Allowed Characters**:

Identifiers can commence with a letter (uppercase or lowercase), an underscore (_), or a special character based on the encoding (as in UTF-8).

Subsequent characters can include letters, digits, or underscores.

2. **Case Sensitivity**:

Identifiers exhibit case sensitivity, signifying that "variableName" and "VariableName" are distinct identifiers.

3. **Reserved Words (Keywords)** :

Usage of reserved words or keywords of the language as identifiers is prohibited. Words like int, double, class, etc., fall into this category.

4. **Identifier Length** :

While the length of identifiers is typically unrestricted, it is advisable to maintain a reasonable length for enhanced readability.

2.9 Constants :

In programming, constants are immutable values that remain unchanged throughout the execution of the program. They are used to represent fixed data and are defined once and for all in the code.

The basic syntax for declaring constants in C++ is as follows:

```
const data_type CONSTANT_NAME = value;
```

1. **Definition of a Constant Using the Preprocessor:**

In C++, you can define a constant using the preprocessor with the « const » directive.

```
#include <iostream>
// Definition of a constant using `const`
const double INTEREST_RATE = 0.05;
int main() {
    // Using the constant
    double balance = 1000.0;
    double interest = balance * INTEREST_RATE;
    std::cout << "The interest is: " << interest << std::endl;
    return 0;
}
```

2. **Enumerated Constants:**

In C++, the utilization of enumerations (enum) allows you to establish enumerated constants. This is advantageous for depicting sets of constant values and enhancing code readability by substituting literal values with meaningful symbolic names.

```
#include <iostream>
// Declaration of an enumeration
enum Days {
    MONDAY,    // 0
    TUESDAY,   // 1
    WEDNESDAY, // 2
    THURSDAY,  // 3
    FRIDAY,    // 4
    SATURDAY,  // 5
    SUNDAY     // 6
};
int main() {
    // Using the enumeration
    Days currentDay = WEDNESDAY;
    // Comparison with an enumerated constant
    if (currentDay == WEDNESDAY) {
```

```
        std::cout << "It's Wednesday!" << std::endl;
    }
    // Displaying the numeric value associated with an enumerated constant
    std::cout << "The current day is: " << currentDay << std::endl;
    return 0;
}
```

In this example, the enumeration Days is defined with the days of the week. The enumerated constants (MONDAY, TUESDAY, etc.) are automatically associated with integer values (0, 1, 2, ...). You can use these enumerated constants in your code to make the program more readable.

2.10 Operators:

In C++, operators are distinctive symbols essential for executing a variety of operations on variables and values.

1. Arithmetic Operators:

« + » : Addition.

« - » : Subtraction.

« * » : Multiplication.

« / » : Division.

« % » : Modulo (remainder of division).

```
// Arithmetic operators syntax
int a = 5, b = 3;
int sum = a + b;        // Addition
int difference = a - b; // Subtraction
int product = a * b;    // Multiplication
int quotient = a / b;   // Division
int remainder = a % b;  // Modulo (remainder of division)
```

2. Assignment Operators:

« = » : Assignment (assigns a value to a variable).

« += », « -= », « *= », « /= », « %= » : Compound operators (add, subtract, multiply, divide, take modulo and assign the result to the variable).

```
// Assignment operators syntax
int a = 5, b = 3;
// Basic assignment
int c = a;      // c is assigned the value of a
// Compound assignment
a += b;         // Equivalent to: a = a + b;
a -= b;         // Equivalent to: a = a - b;
a *= b;         // Equivalent to: a = a * b;
a /= b;         // Equivalent to: a = a / b;
a %= b;         // Equivalent to: a = a % b;
```

3. Comparison Operators:

« == » : Equal to.

« != » : Not equal to.

« < », « <= » : Less than, Less than or equal to.

« > », « >= » : Greater than, Greater than or equal to.

```
// Comparison operators syntax
int a = 5, b = 3;
// Equality
bool isEqual = (a == b);     // false
// Inequality
bool isNotEqual = (a != b);  // true
// Less than
bool isLessThan = (a < b);   // false
// Less than or equal to
bool isLessThanOrEqual = (a <= b);  // false
// Greater than
bool isGreaterThan = (a > b);      // true
// Greater than or equal to
bool isGreaterThanOrEqual = (a >= b);  // true
```

4. **Logical Operators:**

« && » : Logical AND.

« || » : Logical OR.

« ! » : Logical NOT.

```
// Logical operators syntax
bool condition1 = true, condition2 = false;
// Logical AND
bool logicalAND = (condition1 && condition2);  // false
// Logical OR
bool logicalOR = (condition1 || condition2);   // true
// Logical NOT
bool logicalNOT_condition1 = !condition1;       // false
bool logicalNOT_condition2 = !condition2;       // true
```

5. **Increment and Decrement Operators:**

« ++ » : Increment.

« -- » : Decrement.

```
// Increment and Decrement Operators
int a = 5;
// Increment (adds 1 to the value)
a++;  // a becomes 6
// Decrement (subtracts 1 from the value)
a--;  // a goes back to 5
```

6. **Binary Operators (Bitwise):**

« & » : Bitwise AND.

« | » : Bitwise OR.

« ^ » : Bitwise XOR.

« ~ » : Bitwise NOT.

« << » : Left shift.

« >> » : Right shift.

```
// Binary Operators (Bitwise)
int e = 5;  // binary: 0101
int f = 3;  // binary: 0011
// Binary AND
int bitwiseAnd = e & f;  // binary: 0001 (1 in decimal)
// Binary OR
int bitwiseOr = e | f;   // binary: 0111 (7 in decimal)
// Binary XOR (Exclusive OR)
int bitwiseXor = e ^ f;  // binary: 0110 (6 in decimal)
// Binary NOT (bitwise inversion)
int bitwiseNot = ~e;     // binary: 1111 (inverted)
```

7. **Pointer Operators:**

« & » : Address of a variable.

« * » : Indirection (value pointed to by a pointer).

```
int number = 42;
int* pointerToNumber = &number; // Address of the variable
int pointedValue = *pointerToNumber; // Value pointed to by the pointer
```

8. **Other Operators:**

« Sizeof **» :** Size of a type or variable.

« ? » and **« : » :** Ternary operator (condition ? value_if_true : value_if_false).

« , » : Comma operator (separates expressions in a list).

2.11 Input-Output:

In C++, input and output (I/O) operations are handled by stream objects in the standard library (<iostream>). Streams offer functionalities for reading from a source (like the keyboard or a file) and writing to a destination (such as the screen or a file). Here's a fundamental overview of I/O in C++ :

2.11.1 Input (cin):

« **cin** » is used to read a number entered by the user.

```
#include <iostream>
int main() {
    // Declare variables to store user input
    int number1, number2;
    // Prompt the user for the first number
    std::cout << "Enter the first number: ";
    std::cin >> number1;
    // Prompt the user for the second number
    std::cout << "Enter the second number: ";
    std::cin >> number2;
    // Display the sum of the two numbers
    std::cout << "Sum: " << number1 + number2 << std::endl;
    return 0;
}
```

2.11.2 Output (cout):

« **cout** » is used to display

```
#include <iostream>
int main() {
    // Use cout to display a message
    std::cout << "Hello, World!" << std::endl;
```

```
    return 0;
}
```

2.12 Control Structures :

2.12.1 The if-else statement :

The if-else statement in C++ is a control flow structure enabling the execution of distinct code blocks depending on a Boolean condition.

```
if (condition) {
    // Code block to execute if the condition is true
} else {
    // Code block to execute if the condition is false
}
```

- If the expression within the parentheses of the if statement is true, the code enclosed within the first set of curly braces is executed.

- Conversely, if the expression evaluates to false, the code enclosed within the braces following the else keyword is executed.

```
int age = 20;

if (age >= 18) {
    // This block is executed if age is greater than or equal to 18
    std::cout << "You are an adult." << std::endl;
} else {
    // This block is executed if age is less than 18
    std::cout << "You are a minor." << std::endl;
}
```

2.12.2 The while loop (while)

The while loop in programming functions as a control structure, permitting the execution of a code block as long as a specified condition holds true.

```
while (condition) {
    // Code block to execute as long as the condition is true
}
```

- The condition within the parentheses is assessed before each iteration of the loop.

- If the condition holds true, the code within the curly braces is executed, and subsequently, the condition undergoes re-evaluation.

- The loop persists in execution as long as the condition remains true. Once the condition turns false, the loop concludes, and the program proceeds to the subsequent statement following the loop.

```
int counter = 0;

while (counter < 5) {
    // This block is executed as long as the counter is less than 5
    std::cout << "Counter: " << counter << std::endl;
    counter++;
```

2.12.3 The do-while loop

The do-while loop, akin to the while loop, presents another looping structure in programming. Its key distinction lies in ensuring that the code block within the loop executes at least once, irrespective of whether the specified condition is true or false.

```
do {
    // Code block to execute
} while (condition);
```

- The code block within the do is the initial execution.
- Subsequent to the block's execution, the condition in the while is examined.
- If the condition is true, the loop persists in execution. If it evaluates to false, the loop concludes, and the program proceeds to the subsequent statement post the loop.

```
int counter = 0;
do {
    // This block is executed at least once
    std::cout << "Counter: " << counter << std::endl;
    counter++;
} while (counter < 5);
```

2.12.4 The for loop

The for loop, a control flow structure in programming, facilitates the repetitive execution of a code block for a predefined number of iterations. It is commonly employed when the count of iterations is predetermined.

```
for (initialization; condition; update) {
    // Code block to execute
}
```

1. **Initialization** :

The initialization statement is executed once before the commencement of the loop. Typically, it is employed to set the initial value of a loop control variable.

2. **Condition** :

The condition is assessed before each iteration of the loop. If the condition holds true, the loop persists; otherwise, it terminates.

3. **Update :**

The update statement is executed after each iteration of the loop. It is commonly utilized to modify the loop control variable.

```
for (int i = 0; i < 5; i++) {
    // This block is executed five times
    std::cout << "Iteration: " << i << std::endl;
}
```

2.12.5 The switch-case statement

The switch statement, found in various programming languages, including C++, serves as a control flow structure enabling the examination of a variable against a set of predefined values. It offers a mechanism to execute distinct code blocks based on the variable's value, with each value corresponding to a specific code block termed a case.

```
switch (expression) {
    case value1:
        // Code to execute if expression equals value1
        break;
```

```
    case value2:
        // Code to execute if expression equals value2
        break;
    // Additional cases as needed
    default:
        // Code to execute if none of the cases match
}
```

- The expression undergoes evaluation, and its value is juxtaposed against each case.

- Upon finding a case that aligns with the expression's value, the associated code block is executed.
- The break statement is deployed to exit the switch statement. In the absence of a break, the program will persist in executing code in subsequent case statements until a break is encountered or the switch statement concludes.

- The default case is discretionary, supplying code to execute when none of the cases finds a match.

```
int day = 3;
switch (day) {
    case 1:
        std::cout << "Monday" << std::endl;
        break;
    case 2:
        std::cout << "Tuesday" << std::endl;
        break;
    case 3:
        std::cout << "Wednesday" << std::endl;
        break;
    default:
        std::cout << "Unknown day" << std::endl;
}
```

2.12.6 The break statement

The break statement is a control flow element employed in various programming languages, including C++. It is primarily linked with loops (for, while, do-while) and the "switch" statement. The break statement serves the purpose of concluding the execution of the nearest enclosing loop or switch statement.

2.12.7 The continue statement

The continue statement is utilized within loops (do-while, for, or while) to transition to the next iteration without executing the remaining code within the loop block for that particular iteration.

2.12.8 The exit statement

The « **exit ( )** » function in C++ is employed to abruptly terminate the program. Here are the key points:

1. The « **exit ( )** » function concludes the program instantaneously.
2. It accepts an integer argument representing the exit status.
3. **Using EXIT_SUCCESS** (or exit(0)) signifies a successful program termination.
4. **EXIT_FAILURE** is commonly used to indicate program termination due to an error.
5. Cleanup operations and functions registered with atexit() are not guaranteed to execute.

2.13 Arrays, pointers, structures :

2.13.1 Arrays:

In C++, an array is a grouping of elements of the same data type stored in consecutive memory locations. Access to the array elements is facilitated using their respective indices.

2.13.1.1 One-dimensional arrays:

The syntax for declaring an array in C++ is as follows:

```
data_type array_name[size];
```

- **Declaring One-Dimensional Arrays**

In C++, the declaration of one-dimensional arrays involves specifying the data type of the elements, followed by the array name and the size of the array enclosed in square brackets

([ ]).

```
data_type array_name[array_size];
int numbers[5];  // Array of integers with size 5
```

- **Initializing One-Dimensional Arrays**

During declaration in C++, arrays can be initialized by providing initial values enclosed in curly braces ({ }).

If the size is not explicitly specified, the compiler infers it from the number of elements.

```
int numbers[] = {1, 2, 3, 4, 5};  // Compiler infers size as 5
```

- **Accessing Éléments**

In C++, accessing elements in a one-dimensional array is achieved by using the array index. The array index signifies the position of an element in the array, commencing from zero.

```
#include <iostream>
int main() {
    // Declare and initialize an integer array
    int numbers[] = {10, 20, 30, 40, 50};
    // Accessing elements using array indices
    std::cout << "Element at index 0: " << numbers[0] << std::endl;
    std::cout << "Element at index 1: " << numbers[1] << std::endl;
    std::cout << "Element at index 2: " << numbers[2] << std::endl;
    std::cout << "Element at index 3: " << numbers[3] << std::endl;
    std::cout << "Element at index 4: " << numbers[4] << std::endl;
    return 0;
}
```

- **Key Points to Remember** :

  Array indices commence from zero (0).

  Trying to access an element beyond the array size can lead to undefined behavior or runtime errors.

  It is imperative to ensure that the index used for accessing elements falls within the bounds of the array

- **Determining the Size**

To ascertain the size of an array in C++, the « **sizeof** » operator can be employed. The overall size of the array is obtained by dividing the total size of the array by the size of one individual element.

```
int sizeOfArray = sizeof(numbers) / sizeof(numbers[0]);
```

2.13.1.2 Multidimensional arrays :

In C++, a multidimensional array is characterized by having more than one dimension. While the two-dimensional array is the most common, C++ also supports arrays with more than two dimensions. This concept can be extended to create three-dimensional arrays, where each element is identified by three indices (e.g., **array[x][y][z]**). Similarly, arrays with more dimensions can be formed by adding additional indices.

```
#include <iostream>
int main() {
    // Declare and initialize a 2D array (matrix)
    int matrix[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    // Accessing elements in a 2D array
    std::cout << "Element at row 1, column 2: " << matrix[0][1] << std::endl;
    std::cout << "Element at row 2, column 3: " << matrix[1][2] << std::endl;
    std::cout << "Element at row 3, column 4: " << matrix[2][3] << std::endl;
    return 0;
}
int threeDArray[2][3][4];  // Example of a three-dimensional array
```

2.13.1.3 Character strings :

In programming, a "**string**" refers to a sequence of characters, commonly employed to represent text and considered a fundamental data type in numerous programming languages. In C++, strings can be represented using the std::string class from the C++ Standard Library.

- **Declaration of a character string**

When declaring a character string in C++, you have two primary approaches: utilizing a list of character constants or employing a literal string (string literal).

1. By a List of Character Constants (C-Style String)

```
#include <iostream>
int main() {
    // Declaration by a list of character constants (C-style string)
    const char myCString[] = {'H', 'e', 'l', 'l', 'o', '\0'};
    // Display the C-style string
    std::cout << myCString << std::endl;
    return 0;
}
```

For instance, in the provided example, myCString is a C-style string declared as an array of characters. It is initialized with a list of character constants, and the string is terminated with the null character '\0'.

2. By a literal chain

```
#include <iostream>
int main() {
    // Declaration by a literal string
    const char* myLiteralString = "Hello";
    // Display the literal string
    std::cout << myLiteralString << std::endl;
    return 0;
}
```

2.13.2 Pointers:

In C++, a pointer is a variable designed to store the memory address of another variable. Pointers play a crucial role in low-level memory operations and are commonly employed for tasks such as dynamic memory allocation, array manipulation, and interaction with functions that operate on memory addresses.

1. **Declaration syntax :**

The syntax for declaring pointers in C++ entails specifying the data type of the variable the pointer will point to, followed by an asterisk (*), and then the name of the pointer variable.

```
data_type *pointer_name;
```

Data_type: The data type of the variable that the pointer will reference.

« ***** » : The asterisk signifies that the variable being declared is a pointer.

Pointer_name: The name assigned to the pointer variable.

```
int main() {
    int x = 10;      // Integer variable
    int *ptr;        // Pointer to an integer
    ptr = &x;        // Assigning the address of x to the pointer
    return 0;
}
```

2. **Using pointers :**

In programming, there are two primary methods to access a variable:

By its name (identifier):

Direct access to a variable using its name.

By its address, using a pointer:

Pointers, which enable access to a variable using its memory address. A pointer is a variable whose value is the memory address of another variable.

3. **Dereference Operator (*)**

Usage: Used to access the value stored at the memory address pointed to by a pointer.

```
int x = 10;
int *ptr = &x;   // Pointer declaration and initialization
int value = *ptr; // Dereferencing the pointer to access the value
```

4. **Address-of Operator (&)**

Usage: Used to obtain the memory address of a variable.

```
int y = 20;
int *ptr = &y;   // Assigning the address of y to the pointer
```

2.13.3 Pointer arithmetic

Pointer arithmetic in C++ encompasses the manipulation of addresses stored in pointers to navigate through memory locations. This process is contingent on the size of the data type, ensuring precise navigation through memory.

In the realm of pointer arithmetic, you can increment or decrement pointers to traverse to the subsequent or preceding memory location, considering the size of the data type they reference. This aspect becomes notably crucial when dealing with arrays or dynamically allocated memory.

As an illustration, if you possess a pointer « **int *ptr** » pointing to an integer array, incrementing « **ptr** » by 1 would shift it to the next integer in the array, not merely the next memory address.

```
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr;
// Incrementing by 1 moves to the next integer in the array
int secondElement = *(ptr + 1); // Equivalent to arr[1]
```

2.13.3.1 Pointers and arrays

Pointers and arrays are closely related concepts in C++. Understanding their relationship is fundamental to efficient memory management and array manipulation. Here are key points:

1. **Array Name as a Pointer**: In C++, the name of an array can be used as a pointer to its first element.

```
int numbers[] = {1, 2, 3, 4, 5};
int *ptr = numbers;  // Equivalent to &numbers[0]
```

2. **Pointer Arithmetic with Arrays**: Pointers can be incremented or decremented to navigate through array elements.

```
int numbers[] = {1, 2, 3, 4, 5};
int *ptr = numbers;
// Incrementing the pointer moves to the next element in the array
int secondElement = *(ptr + 1);  // Equivalent to numbers[1]
```

3. **Array Access using Pointers**: Array elements can be accessed using pointers and array subscript notation.

```
int numbers[] = {1, 2, 3, 4, 5};
int *ptr = numbers;
// Using pointer arithmetic to access array elements
int thirdElement = *(ptr + 2);  // Equivalent to numbers[2]
```

4. **Pointer to an Array**: Pointers can also point to entire arrays.

```
int numbers[] = {1, 2, 3, 4, 5};
int (*ptrToArray)[5] = &numbers;  // Pointer to an array of 5 integers
```

2.13.3.2 Pointers and strings

Strings are commonly depicted as arrays of characters, terminated by a null character ('\0').

In C++, strings are usually portrayed as arrays of characters.

A pointer to a character « **(char*)** » can be employed for handling strings.

```
char str[] = "Hello";     // Character array representing a string
char* ptr = str;           // Pointer to the first character of the array
```

2.13.3.3 Array of Pointers

In C++, it is possible to create an array where each element is a pointer. These pointers have the flexibility to point to different data types, including other arrays, variables, or dynamically allocated memory. This capability is advantageous for effectively managing collections of data, such as strings or arrays of varying sizes.

```cpp
#include <iostream>
int main() {
    // Array of pointers to integers
    int num1 = 10, num2 = 20, num3 = 30;
    int* numbers[] = {&num1, &num2, &num3};
    // Accessing elements in the array
    std::cout << "First number: " << *numbers[0] << std::endl;
    std::cout << "Second number: " << *numbers[1] << std::endl;
    std::cout << "Third number: " << *numbers[2] << std::endl;
    return 0;
}
```

2.13.3.4 Pointers to pointers

Pointers to pointers, also known as double pointers, are variables designed to store the memory address of another pointer. They prove beneficial in situations where dynamic manipulation of pointers or the creation of dynamic multi-dimensional arrays is necessary.

Here's a brief example in C++ to illustrate pointers to pointers:

```cpp
#include <iostream>
int main() {
    int value = 42;
    // Pointer to an integer
    int* ptr1 = &value;
    // Pointer to a pointer (double pointer)
    int** ptr2 = &ptr1;
    // Accessing the value using a double pointer
    std::cout << "Value using double pointer: " << **ptr2 << std::endl;
    return 0;
}
```

2.14 Dynamic memory allocation :

Dynamic memory allocation in C++ entails the runtime management of memory using pointers. This process involves employing the « **new** » and « **delete** » operators to allocate and deallocate memory on the heap.

2.14.1 The new operator

In C++, the new operator is utilized for dynamic memory allocation. It allocates memory on the heap during program execution and returns a pointer to the allocated memory. This functionality enables the creation of objects or arrays whose size can be determined at runtime.

```cpp
#include <iostream>
int main() {
    // Dynamic memory allocation for a single integer
    int* dynamicInt = new int;
    // Dynamic memory allocation for a single double
    double* dynamicDouble = new double;
    // Assigning values to the dynamically allocated variables
    *dynamicInt = 42;
    *dynamicDouble = 3.14;
    // Accessing and printing the values
    std::cout << "Dynamically allocated integer: " << *dynamicInt << std::endl;
    std::cout << "Dynamically allocated double: " << *dynamicDouble << std::endl;
    // Deallocating the dynamically allocated memory
    delete dynamicInt;
    delete dynamicDouble;
    return 0;
}
```

2.14.2 The delete operator

The delete operator in C++ is employed to deallocate memory that was previously allocated using the new operator. This step is crucial in preventing memory leaks in your program.

```
#include <iostream>
int main() {
    // Dynamic memory allocation for a single integer
    int* dynamicInt = new int;
    // Assigning a value to the dynamically allocated integer
    *dynamicInt = 42;
    // Accessing and printing the value
    std::cout << "Dynamically allocated integer: " << *dynamicInt << std::endl;
    // Deallocating the dynamically allocated memory
    delete dynamicInt;
    return 0;
}
```

2.15 The structures:

In C++, a structure is a user-defined data type that allows grouping variables of different data types under a single name using the "struct" keyword.

2.15.1 Declaration of a structure

In C++, you typically use the « **struct** » keyword to declare a structure.

```
#include <iostream>
// Declaration of a structure named 'Point'
struct Point {
    int x;
    int y;
};
int main() {
    // Creating an instance of the 'Point' structure
    Point p1;
    // Initializing the members of the structure
    p1.x = 3;
    p1.y = 7;
    // Displaying the values
    std::cout << "Point coordinates: (" << p1.x << ", " << p1.y << ")" << std::endl;
    return 0;
}
```

2.15.2 Manipulating a structure

Manipulating a structure entails carrying out operations or modifications on its members.

Accessing a field in a structure is achieved by using a dot (.) followed by the structure variable's name and then the field's name. Once accessed, this field can be treated like any other variable of its type.

```
#include <iostream>
// Definition of a structure named 'Person'
struct Person {
    std::string name;
    int age;
};
int main() {
    // Creating an instance of the 'Person' structure
    Person alice {"Alice", 25};
    // Displaying the age before the birthday
    std::cout << "Before birthday: " << alice.age << std::endl;
    // Celebrating the birthday by incrementing the age
    ++alice.age;
```

```
    // Displaying the updated age
    std::cout << "After birthday: " << alice.age << std::endl;
    return 0;
}
```

2.15.3 Global use of a structure:

The global use of a structure requires declaring it at the global scope, enabling accessibility throughout the entire program.

The general syntax for performing a global assignment of a structure in C++ is as follows :

```
// Structure declaration
struct StructureName {
    // Structure members
    dataType member1;
    dataType member2;
    // ...
};
// Declaration of a structure instance
StructureName structureInstance1;
// Global assignment of the structure
structureInstance1 = {value1, value2 /*, ... */};
```

2.15.4 Initializing a structure

Initializing a structure involves providing initial values for its members. In C++, you can perform this initialization at the time of declaration.

```
#include <iostream>
#include <string>
// Definition of a 'Person' structure
struct Person {
    std::string name;
    int age;
};
int main() {
    // Initializing a 'Person' structure at the time of declaration
    Person alice = {"Alice", 30};
    // Displaying the initialized values
    std::cout << "Name: " << alice.name << std::endl;
    std::cout << "Age: " << alice.age << std::endl;
    return 0;
}
```

3 Chapter 3: Classes and objects

3.1 Class and Object

Within the realm of programming, especially in the context of object-oriented programming (OOP), the term "object classes" commonly denotes the blueprints or templates employed in the creation of objects. Let's delve into the definitions associated with object classes:

Class: A class serves as a blueprint or template, delineating the properties (attributes) and behaviors (methods) inherent to objects belonging to that class.
Object: An object is a concrete instance of a class, embodying a distinct entity within the program and having the capacity to engage with other objects.
Data Members: Data members within a class are variables that specifically pertain to that class, delineating the attributes or properties inherent in objects instantiated from that class. Essentially, these data members encapsulate the state of an object.

Member Functions: Member functions are functions associated with a class, designed to manipulate the data members of that class. These functions specify the behaviors or actions that objects of the class can execute, thereby defining the operational aspects of the class.

3.1.1 Declaration of a class:

The keyword **« class »** in C++ signifies the initiation of a class declaration. Declaring a class in C++ entails outlining the structure of the class, encompassing its data members and member functions, yet refraining from divulging the implementation specifics.

```
class ClassName {
public:
    // Data members (attributes)
    DataType1 member1;
    DataType2 member2;

    // Member functions (methods)
    ReturnType method1(ParameterType1 param1);
    ReturnType method2(ParameterType2 param2);
    // ... additional methods
};
```

1. **Definition of member functions:**

In C++, member functions represent the behaviors or actions linked to a class, dictating how objects of that class engage with and manipulate their internal data. These functions encapsulate the behavior of objects and are tasked with executing specific operations on the data members of a class.

2. **Inline definition of member functions:**

In C++, member functions can be defined inline, meaning their implementation is directly included within the class declaration. This approach is convenient, especially for short and straightforward functions.

```
class Rectangle {
public:
    double length;
    double width;
    // Inline definition of a member function to calculate the area
    double calculateArea() {
        return length * width;
    }
};
```

3. **Deferred definition of member functions:**

In C++, the deferred definition of member functions involves declaring functions inside a class, specifying their return types, names, and parameters..

The actual implementation of these functions is provided outside the class. This method helps maintain a concise class interface, concentrating on essential details in the declaration, while allowing for a more extensive, modular, and organized implementation outside the class.

This deferred definition approach is particularly beneficial for larger or more complex functions, as it enhances code readability and supports separate compilation in larger projects.

```
#include <iostream>
// Class declaration with member function declaration
class Calculator {
public:
    // Declaration of a member function
    int add(int a, int b);

    // Declaration of another member function
    int multiply(int a, int b);
};
// Definition of the member functions outside the class
int Calculator::add(int a, int b) {
    return a + b;
}
int Calculator::multiply(int a, int b) {
    return a * b;
}
int main() {
    // Creating an object of the Calculator class
    Calculator myCalculator;
    // Using the deferred-defined member functions
    int sum = myCalculator.add(5, 3);
    int product = myCalculator.multiply(4, 2);
    // Displaying the results
    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Product: " << product << std::endl;
    return 0;
}
```

3.1.2 Declaration of objects:

In C++, declaring objects is a fundamental process that entails defining the type of the object and assigning it a name. Objects, which serve as instances of classes, are pivotal in the creation and utilization of user-defined types.

The typical syntax for declaring an object entails indicating the class type followed by the chosen object name.

```
ClassName objectName;
```

3.2 Access to class members:

When accessing the members, which include both data members and member functions, of a class in C++, one employs the dot **(« . »)** operator for objects. The accessibility to class members is contingent on the access specifiers (« **publi c », « private »**) specified within the class.

```
#include <iostream>
#include <string>
class Person {
public:
```

```
    // Data member
    std::string name;

    // Member function
    void displayName() {
        std::cout << "Name: " << name << std::endl;
    }
};
int main() {
    // Declaration of an object of the Person class
    Person person1;
    // Access and modification of a data member
    person1.name = "John Doe";
    // Calling a member function
    person1.displayName();
    return 0;
}
```

In C++, the distinction between private and public members is crucial for managing access to a class's elements. These access modifiers play a pivotal role in achieving encapsulation and ensuring the integrity of data.

3.2.1 Public Members :

- Members, including both data members and member functions, declared as public, are accessible from outside the class.
- They constitute the interface of the class, offering an external perspective on its functionalities.
- Public members are employed for interacting with objects of the class.

```
class Example {
public:
    // Public data member
    int publicVar;

    // Public member function
    void publicFunction() {
        // Implementation
    }
};
```

3.2.2 Private Members :

- Members declared as private are only accessible within the class.
- They remain concealed from external access, fostering encapsulation and data hiding.
- Private members are typically utilized for internal operations and data storage..

```
class Example {
private:
    // Private data member
    int privateVar;
    // Private member function
    void privateFunction() {
        // Implementation
    }
};
```

3.3 Encapsulation:

Encapsulation, a fundamental concept in object-oriented programming (OOP), involves consolidating both data (attributes or properties) and methods (functions or procedures) that operate on the data into a unified entity known as a class. The encapsulation principle strives to

confine direct access to certain components of an object and expose only what is essential for the object's functionality.

Key aspects of encapsulation include:

- **Data Hiding** :

Internal details of the class, such as data members, are kept private, preventing external code from direct access or modification.

- **Access Control** :

Internal components are regulated by access specifiers (public, private, protected). Public members facilitate interaction, while private ones are accessible solely within the class.

- **Security and Integrity** :

Enhances security by limiting direct access, ensuring data integrity through controlled interactions.

3.4 The current object and the « this » keyword:

In C++, the term "current object" pertains to the instance of a class currently undergoing operations, especially within member functions. The "this" keyword, functioning as a pointer, contains the memory address of the current object and is implicitly accessible in non-static member functions. This keyword proves particularly valuable in differentiating between class data members and function parameters, particularly when they share identical names. By leveraging "this," developers can enhance code clarity, avert naming conflicts, and ensure unequivocal access to the current object's members. Such utilization of the "this" keyword contributes to writing cleaner and more readable code, particularly in the context of object-oriented programming. It facilitates maintaining a distinct separation between the internal state of the object and its external interactions.

3.5 Assignment of Objects:

In C++, object assignment is achieved through the use of the assignment operator **(« = »)**. This operator copies the content of one object into another object of the same type. If a class does not provide a custom assignment operator, the compiler generates a member-wise copy by default, copying each data member from the source object to the destination object.

However, classes have the option to define custom assignment operators to perform specialized copies, especially when dealing with dynamic memory or requiring additional actions during assignment. Furthermore, the utilization of copy constructors is common in object assignment, ensuring the proper initialization of a new object with the content of an existing one.

It's crucial to note that when assigning objects of derived classes to objects of base classes, object slicing may occur, resulting in the loss of derived class-specific features..

```
#include <iostream>
// Class with dynamic memory and a custom assignment operator
class CustomAssignmentExample {
public:
    int* dynamicData;
    // Constructor
    CustomAssignmentExample(int value) : dynamicData(new int(value)) {}
    // Custom assignment operator
    CustomAssignmentExample& operator=(const CustomAssignmentExample& other) {
        if (this != &other) {
```

```
            delete dynamicData;
            dynamicData = new int(*(other.dynamicData));
        }
        return *this;
    }
    // Destructor to release dynamic memory
    ~CustomAssignmentExample() {
        delete dynamicData;
    }
};
int main() {
    // Creating objects and displaying values
    CustomAssignmentExample obj1(42);
    CustomAssignmentExample obj2(100);
    std::cout << "Before assignment: " << *(obj1.dynamicData) << " | " <<
*(obj2.dynamicData) << std::endl;
    // Custom assignment and displaying values
    obj2 = obj1;
    std::cout << "After assignment: " << *(obj1.dynamicData) << " | " <<
*(obj2.dynamicData) << std::endl;
    return 0;
}
```

3.6 Constructors and Destructors:

3.6.1 Constructors:

In C++, a constructor is a special class member function with the following key features:

Purpose :

- Constructors serve as special member functions responsible for initializing the object upon creation.
- Their primary goal is to establish the object in a valid and usable state.

Syntax :

- The constructor shares the same name as the class and lacks a return type.
- Multiple constructors with different parameter lists can be defined, allowing overloading.

Initialization :

- Constructors undertake the initialization of data members within the class, along with resource allocation and necessary setup.

Automatic Invocation :

- Constructors are automatically triggered when an object of the class is created.
- While explicit invocation is possible, it is often implicitly activated during object instantiation..

3.6.2 Destructors:

Like the constructor, the destructor is a special class member function with the following properties :

Purpose :

- Destructors assume the responsibility of cleaning up resources, releasing memory, and executing essential tasks before the object is destroyed.

- Their role is crucial for ensuring proper resource management and preventing memory leaks.

Syntax :

- The destructor bears the same name as the class, preceded by a tilde (~), and lacks parameters or a return type.
- Similar to constructors, destructors can be overloaded.

Automatic Invocation :

- Destructors are automatically invoked when an object goes out of scope or when « **delete** » is used for dynamically allocated objects.

Order of Execution :

- Destructors are executed in the reverse order of object creation, adhering to a last-in, first-out sequence..

```
#include <iostream>
class Example {
public:
    // Constructor
    Example() {
        std::cout << "Constructor called" << std::endl;
    }
    // Destructor
    ~Example() {
        std::cout << "Destructor called" << std::endl;
    }
    // Member function
    void DisplayMessage() {
        std::cout << "Hello from Example class!" << std::endl;
    }
};
int main() {
    // Object instantiation - Constructor is called
    Example obj;
    // Member function call
    obj.DisplayMessage();
    // Destructor is called at the end of the program
    return 0;
}
```

3.7 Classes, variables and special functions

3.7.1 Static Data Members:

We will start by providing a general definition of static data outside the concept of a class. Subsequently, this notion will be addressed in the case of an object class.

3.7.2 Static Local Variables:

In C++, static local variables are variables declared within a function that preserve their values across consecutive calls. Employing the "static" keyword, these variables are initialized just once and persist throughout the entire program. Despite being globally allocated in memory, they are confined in scope to the declaring function. Unlike ordinary local variables, static local variables endure beyond the function's execution on the stack, and their values persist between various calls to the function. This functionality is commonly utilized in scenarios such as

counters, where the variable's state needs to endure across multiple calls within the same function.

```
void exampleFunction() {
    static int localVar = 0;
    // ...
}
```

3.7.3 Static Member Variables:

In C++, static member variables represent class-level variables that are shared across all instances of the class. Introduced with the « **static** » keyword within the class definition, these variables possess a singular instance linked to the entire class, rather than being specific to individual objects. Memory allocation for static member variables is shared among all class instances and typically occurs outside the class declaration. Accessible using the class name, they do not depend on a particular class instance for access. Widely utilized for preserving shared information or functioning as counters, static member variables serve as a mechanism to store data common to all objects of a class.

```
class ClassName {
public:
    // Other member declarations...
    // Static member variable declaration
    static DataType staticMemberVariable;
};
// Initialization of the static member variable (usually done outside the class)
DataType ClassName::staticMemberVariable = initialValue;
```

Initialization of Static Member Variables

In C++, static member variables are incorporated into a class by employing the "static" keyword. The initialization process involves two steps: initially, the variable is declared within the class, and subsequently, the actual initialization is conducted outside the « **class—typically** » in the source file connected with the class. This method guarantees that static member variables receive a clearly defined initial value and are shared across all instances of the class. The initialization occurs only once, and access to the variable is facilitated using the class name.

```
#include <iostream>
class Example {
public:
    // Declaration of static member variable
    static int staticMemberVar;
};
// Initialization of the static member variable (usually in the source file)
int Example::staticMemberVar = 42;
int main() {
    // Accessing the static member variable using the class name
    std::cout << "Static Member Variable: " << Example::staticMemberVar << std::endl;
    return 0;
}
```

3.7.3.1 Static Member Functions:

In C++, static member functions are distinct functions linked with a class, functioning independently of specific instances. Utilizing the static keyword within the class, these functions operate without direct access to instance-specific data and are invoked using the class

name. Particularly beneficial for tasks associated with the class as a whole, irrespective of individual objects, static member functions lack a "this" pointer and do not demand an instance. This makes them well-suited for utility functions or operations involving only the static members of the class.

```
class ClassName {
public:
    // Other member declarations...
    // Declaration of static member function
    static ReturnType staticMemberFunction(ParameterType1 param1, ParameterType2
param2, ...);
};
```

3.8 The functions :

In C++, the incorporation of functions serves as a pivotal element in shaping code structure, fostering reusability, promoting abstraction, and bolstering encapsulation. This, in turn, contributes to an overall enhancement of software clarity, flexibility, and maintainability.

3.8.1 Definition and advantages of functions:

In the realm of programming, functions embody self-contained code blocks characterized by a specified name, return type, and optional parameters. Their multifaceted benefits encompass modularity, which aids in organizing code; reusability, a key factor in minimizing redundancy; abstraction, allowing the concealment of implementation intricacies; parameter passing, enabling adaptability; return values, facilitating data flow; readability, which enhances comprehension of the code; simplified testing and debugging, easing error isolation; and encapsulation, fortifying code security. Collectively, functions play a central role in the development of modular, efficient, and easily maintainable software.

3.8.2 Characteristics of functions:

In C++, a function is characterized by several defining features:

Location: It may reside either within the main program file (main()) or in a separate file.

Name: Each function is distinguished by a unique identifier, allowing invocation from main(), other functions, or even through recursion.

Arguments: During invocation, a function receives data referred to as arguments, influencing its behavior or computation.

Return Value Type: If a function produces a result, its return type specifies the data type of that result; if no result is returned, the return type is declared as void.

Local Variables: Variables declared within a function are confined to its scope, remaining accessible only within that specific function.

These attributes collectively contribute to the clarity, modularity, and efficiency of C++ code. They facilitate organized code structures and promote effective reuse of code, enhancing the overall quality and maintainability of the software.

3.8.3 Declaration of a function, prototype:

Prototype of a function:

In C++, a function prototype serves as a succinct glimpse into the structure of a function. It outlines essential details such as the return type, name, and parameters without disclosing the full implementation. Positioned prior to the main code, the function prototype plays a crucial role in allowing the compiler to verify the correct usage of the function, thereby improving code organization and clarity.

Key Roles:
Return Type: Clearly defines the type of value that the function is expected to return.

Function Name: Provides a recognizable identifier for the function, facilitating later use within the code.

Parameters: Specifies the types and order of input data that the function anticipates, guiding users on how to appropriately invoke the function.

Declaration syntax:

```
int add(int a, int b);
```

Function call:
A "function call" in programming refers to the action of invoking or executing a specific function. It involves using the function's name and providing any required parameters or arguments, if applicable. When a function is called, the program temporarily transfers control to the function's code, executes the instructions within the function, and then returns to the point in the program immediately after the function call.

```
return_type result = function_name(argument1, argument2, ...);
```

3.8.4 Definition of a function:
A function in programming is a self-contained and reusable block of code.

Components:

Name: Identified by a unique name.

Return Type: Specifies the type of value the function produces, or void if none.

Parameters: Takes a set of inputs, each with a specified type and name.

Body: Contains the code to be executed when the function is called.

3.8.5 Syntax of defining a function:
The syntax for defining a function in C++ is as follows:

```
return_type function_name(parameter_type1 parameter_name1, parameter_type2
parameter_name2, ...) {
    // Function body: code to be executed
    // ...
    // Return statement (if applicable)
    return return_value;
}
```

3.9 Silent (fictitious) parameters and actual parameters:
In C++, when we talk about "dummy (formal) parameters" and "actual parameters" (or arguments), we are referring to the following concepts:

3.9.1 Dummy (Formal) Parameters :

Definition: Parameters declared in the function definition and used in the function body.
Purpose: Act as placeholders for values that will be passed to the function when it is called.
Example:

```
void exampleFunction(int dummyParameter) {
    // 'dummyParameter' is a formal parameter
    // Code using 'dummyParameter'
}
```

3.9.2 Actual Parameters (Arguments) :

Definition: Values or expressions provided in the function call.
Purpose: Supply specific data to the function for its execution.
Example :

```
int main() {
    int actualArgument = 42;
    exampleFunction(actualArgument); // 'actualArgument' is an actual parameter
    return 0;
}
```

3.10 Type of Variables:

3.10.1 Local variables

In programming, local variables are containers for data that are specifically defined within a designated block or function. Their scope is confined to the execution of the block or function where they are declared, and they cease to exist and be accessible beyond this boundary. The lifespan of local variables is linked to the duration of the enclosing code, and they are generally inaccessible outside of this context. Initialization of local variables can occur upon declaration, and they serve as repositories for temporary data essential to a particular code segment. This approach not only supports code organization but also mitigates the potential for unintended interactions, adhering to the fundamental principle of encapsulation.

3.10.2 Global Variables:

In programming, global variables are storage entities declared outside of specific functions or blocks, granting accessibility across the entirety of the program. Possessing a global scope, they remain accessible from any part of the program, enduring throughout its entire execution. Global variables are frequently employed for storing shared data required across various functions or code segments. Despite their utility in facilitating data sharing, it is crucial to exercise caution to preserve data integrity and uphold code maintainability.

```
#include <iostream>
// Global variable
int globalVariable = 10;
int main() {
    // Accessing the global variable
    std::cout << "Global Variable: " << globalVariable << std::endl;
    // Modifying the global variable
    globalVariable += 5;
    // Accessing the modified global variable
    std::cout << "Modified Global Variable: " << globalVariable << std::endl;
    return 0;
}
```

3.11 Passing arguments to functions:

In programming, the fundamental process of furnishing values or expressions when invoking functions is crucial and is known as passing arguments. This dynamic mechanism enables functions to receive essential data necessary for their execution.

Arguments exhibit diverse forms, contributing to the versatility of function calls. They can manifest as constants, representing fixed values; variables, holding dynamic data; expressions, encapsulating mathematical or logical calculations; or even as the results of other functions.

There exist three common methods or mechanisms for passing arguments to a function in programming, typically categorized as follows:

3.11.1 Passing by value:

In programming, the passing by value is an approach to transmitting arguments to a function, wherein the actual value of the argument is duplicated into the function's parameter. A replica of the argument's value is generated and supplied to the function, and any modifications to the parameter within the function do not impact the original value in the calling code.

```
void exampleFunction(int value) {
    // 'value' is a parameter passed by value
    // Changes made here do not affect the original value
}
```

3.11.2 Passing by pointer:

In programming, passing by pointer is a technique for conveying arguments to a function by furnishing the memory address (pointer) of the argument. Rather than transmitting the literal value, the function receives the memory address of the argument, granting it access to and the ability to modify the original data in the calling code. In the function declaration, parameters are defined as pointers, and during the function call, the memory address of the argument is passed.

```
void exampleFunction(int *pointer) {
    // 'pointer' is a parameter receiving the memory address of an integer
    // Changes made here affect the original data in the calling code
}
```

3.11.3 Passing by reference:

In programming, passing by reference is an approach to transmitting arguments to a function by supplying a direct reference or alias to the original data in the calling code. The function interacts directly with the original data, and any alterations made to the parameter within the function impact the original value in the calling code. In the function declaration, parameters are defined as references, and during the function call, the actual variables are passed.

```
void exampleFunction(int &reference) {
    // 'reference' is a parameter receiving a reference to an integer
    // Changes made here directly affect the original data in the calling code
}
```

3.12 Passing and returning an array to a function:

A function has the capability to accept arrays as arguments and return them.

3.12.1 Passing an array to a function:

In programming, to pass an array to a function, you need to provide the array as an argument during the function call.

- **Declaration syntax:**

The declaration syntax for a function that takes an array as a parameter in C++ usually requires specifying the array type and size (if applicable) within the parameter list. Here's an example:

```
// Declaration of a function that receives an array as a parameter
void functionName(dataType arrayName[], int size);
```

- **Call syntax:**

When calling a function that receives an array as a parameter, the syntax involves passing the array name along with its size (if required) within the parentheses. Here's an example:

```
// Call syntax for a function that receives an array as a parameter
functionName(arrayName, arraySize);
```

3.12.2 Returning an array from a function:

In programming, the process of returning an array from a function entails creating a function that either generates or manipulates an array, and designating the return type of the function as an array.

- **Declaration syntax:**

To declare a function that returns an array, you specify the array type as the return type in the function declaration.

The return type is commonly represented as a pointer pointing to the first element of the array.

```
// Function declaration returning an array
returnType* functionName(parameters);
```

- **Call syntax:**

When invoking a function that returns an array, the approach involves assigning the result to a pointer variable of the corresponding array type.

```
// Call syntax for a function returning an array
returnType* resultArray = functionName(arguments);
```

3.13 Passing a structure to a function:

In programming, incorporating a structure as an argument in a function involves supplying an instance of the structure during the function call.

When passing a structure to a function in C++, there are two prevalent approaches: passing by value or passing by reference using a pointer.

```
#include <iostream>
// Structure definition
struct Point {
    int x;
    int y;
};
// Function taking a structure as a parameter
void printPoint(Point p) {
```

```
    std::cout << "Point: (" << p.x << ", " << p.y << ")" << std::endl;
}
int main() {
    // Creating a structure instance
    Point myPoint = {3, 7};
    // Passing the structure to the function
    printPoint(myPoint);
    return 0;
}
```

3.14 Pointer to a function:

In C++, a pointer to a function is a variable that stores the memory address of a function, facilitating the indirect invocation of the function through the pointer.

The syntax for declaring a pointer to a function in C++ is outlined as follows:

```
returnType (*pointerName)(parameterTypes);
```

returnType : The return type of the function.

pointerName : The name of the pointer to the function.

parameterTypes : The types of parameters the function takes, enclosed in parentheses.

```
#include <iostream>
// Function definition
int add(int a, int b) {
    return a + b;
}
int main() {
    // Declaration of a pointer to a function
    int (*sumPointer)(int, int);
    // Assignment of the function's address to the pointer
    sumPointer = &add;
    // Function invocation through the pointer
    int result = sumPointer(3, 5);
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

3.15 Scope of a Function:

The scope of a function in programming delineates where the function is defined and can be utilized. Functions commonly exhibit local scope when declared within a specific code block, like inside another function. This implies that variables declared within the function are typically accessible only within the confines of that function. Conversely, functions can also possess global scope when defined outside any specific block, enabling access from any part of the program. The scope of a function determines the accessibility and lifespan of variables, influencing code organization and mitigating naming conflicts. Proficiency in understanding function scope is imperative for crafting modular and easily maintainable code.

```
#include <iostream>
int globalVar = 10;
void globalFunction() {
    std::cout << "Global function. Global variable: " << globalVar << std::endl;
}
int main() {
    int localVar = 5;
    void localFunction() {
```

```
        std::cout << "Local function. Local variable: " << localVar << std::endl;
    }
    globalFunction();
    localFunction();
    return 0;
}
```

3.16 Friend Functions:

In C++, a friend function is a non-member function bestowed with special access privileges to the private and protected members of a class. Declared within the class using the "friend" keyword, these functions are not considered members of the class but possess the capability to access its private members as if they were. Friend functions find utility when external functions require specific access to class internals without being formally included in the class definition. They offer a mechanism to uphold encapsulation while enabling designated external functions to closely interact with the class's private members.

```
class ClassName {
private:
    // Private members...
public:
    // Declaration of friend function
    friend ReturnType friendFunction(ParameterType1 param1, ParameterType2 param2,
...);
};
```

3.17 Friend Classes:

In C++, a friend class is a class endowed with special access to the private and protected members of another class. Declared within the class using the "friend" keyword, this designation enables the friend class to closely interact with the members of the declaring class. Friend classes provide a regulated means for sharing private members between classes, promoting collaboration without sacrificing encapsulation. Nevertheless, exercising caution in the usage of friend classes is crucial to sustain a clear and well-structured design.

```
class FriendClass {
public:
    // Member functions...
};
class MyClass {
private:
    int privateMember;
public:
    friend class FriendClass;
};
```

The « **FriendClass »** is designated as a friend class within the « **MyClass** » definition. This implies that « **FriendClass** » is granted special access privileges to the private and protected members of « **MyClass** ». In this particular instance, **« FriendClass** » possesses the capability to access the private member « **privateMember** » of « **MyClass** ».

4 Chapter 4: Inheritance

In C++, inheritance is a foundational concept in object-oriented programming that enables a class to acquire properties and behaviors from another class. It is a crucial feature that promotes code reuse and modularity. The process involves two classes: a base class comprising shared attributes and behaviors, and a derived class inheriting those features. The visibility of inherited members is regulated through access modifiers, with public inheritance being the most prevalent.

4.1 Declaration of a Derived Class:

In C++, a derived class is crafted to inherit properties and behaviors from a base class. The declaration syntax for a derived class entails specifying the derived class name, followed by a colon (« **:** ») and the access specifier, succeeded by the name of the base class.

```
class BaseClass {
    // Base class members...
};
class DerivedClass : public BaseClass {
    // Derived class members...
};
```

In this syntax:

« **BaseClass** » is the designation of the base class.

« **DerivedClass** » is the appellation of the derived class.

« **: public BaseClass** » signifies that « **DerivedClass** » publicly inherits from « **BaseClass** ».

The « **public** » access specifier dictates the visibility of the base class members within the derived class. It specifies that the public members of the base class retain their public status in the derived class. Various access specifiers, such as « **protected** » and « **private,** » can be employed to manage the visibility of inherited members accordingly.

4.2 Inheritance and Protection:

In C++, the concepts of inheritance and protection are intricately connected, revolving around the visibility and accessibility of base class members within derived classes. The access specifiers—public, protected, and private—play a pivotal role in determining how inherited members manifest in the derived class.

4.2.1 Public Inheritance :

In the case of public inheritance, where a derived class inherits publicly from a base class, all public members of the base class retain their public status in the derived class. Meanwhile, protected members transition to a protected status, and private members remain inaccessible.

4.2.2 Protected Inheritance :

Alternatively, with protected inheritance, the public and protected members of the base class assume a protected status in the derived class, while private members persist as inaccessible.

4.2.3 Private Inheritance :

In scenarios of private inheritance, both public and protected members of the base class assume a private status within the derived class, rendering private members inaccessible. This encapsulation ensures a more restricted visibility and access to the inherited elements.

```
#include <iostream>
class Base {
public:
    void publicFunction() {
        std::cout << "Public function in Base\n";
    }
       protected:
    void protectedFunction() {
        std::cout << "Protected function in Base\n";
    }
       private:
    void privateFunction() {
        std::cout << "Private function in Base\n";
    }
};
class Derived : public Base {
public:
    void useBaseMembers() {
        publicFunction();    // Accessible, as it's public
        protectedFunction(); // Accessible, as it's protected
        // privateFunction(); // Not accessible, as it's private in Base
    }
};
int main() {
    Derived derivedObject;
    derivedObject.useBaseMembers();
           return 0;
}
```

In this example :

« **Derived** » publicly inherits from « **Base** ».

« **publicFunction ( )** » becomes accessible in « **Derived** ».

« **protectedFunction ( )** » remains accessible in « **Derived** ».

« **privateFunction ( )** » is not accessible in « **Derived** ».

This demonstrates the impact of public inheritance on the visibility of base class members in the derived class.

4.3 Specification of Inheritance:

In C++, defining inheritance involves specifying the access level for the derived class concerning its base class. The access specifiers ("public," "protected," and "private") play a pivotal role in determining how the members of the base class are accessible in the derived class.

4.3.1 Public Inheritance :

Openly exposes the public features of the base class, serving as a method to transparently extend and specialize existing functionality. This approach enables the derived class to openly utilize and customize the public features of the base class.

4.3.2 Protected Inheritance :

Restricts external entry to public and protected features, a choice made when the derived class is implementing functionality but wishes to keep it hidden from direct exposure, ensuring a controlled access to the implemented features.

4.3.3 Private Inheritance :

Maintains the privacy of inherited features, a strategy employed when the derived class aims to utilize the base class's functionality discreetly, avoiding any implied direct relationship and without revealing these features explicitly.

4.4 Calling Constructors and Destructors:

In C++, constructors and destructors hold special significance as member functions dedicated to the instantiation and termination of objects, respectively.

4.4.1 Constructors :

When an object of a class is created, these functions are invoked. Constructors play a crucial role in initializing the object's state, allocating resources, and executing any required setup procedures. Notably, constructors share the same name as the class and lack a return type.

4.4.2 Destructors :

Conversely, when an object goes out of scope or is explicitly deleted, destructors come into play. Destructors are responsible for releasing resources, performing cleanup operations, and handling any necessary finalization tasks. Destructors bear the same name as the class, preceded by a tilde ("~"), and do not take any parameters.

The automatic invocation of constructors occurs during the creation of objects, ensuring proper initialization. Likewise, when objects go out of scope or are explicitly deleted, the corresponding destructors are called.

Example:

```
int main() {
    MyClass obj1;  // Constructor is called when obj1 is created
           {
        MyClass obj2;  // Constructor is called when obj2 is created
                  }  // Destructor is called when obj2 goes out of scope
                  // Destructor is called when obj1 goes out of scope or program exits
                  return 0;
}
```

Illustratively, in the provided example, constructors are invoked during the creation of « **obj1** » and « **obj2**» while destructors are called when these objects go out of scope.

4.5 Derivation of a class with a parameterized constructor:

Creating a derived class with a parameterized constructor involves establishing a new class (the derived class) that inherits from an existing class (the base class), wherein the base class is equipped with a constructor capable of receiving parameters.

4.5.1 Base Class:

The BaseClass is equipped with a constructor that accepts an int parameter (value).

This constructor is parameterized, indicating that it necessitates an argument when creating an object of the class.

4.5.2 Derived Class:

The DerivedClass is derived from the BaseClass through the syntax : public BaseClass.

When creating an object of the derived class, it is essential to initialize the base class's constructor with the necessary argument (valueBase in this instance).

The constructor of the derived class may include its own parameters (valueDerived), and it initializes the base class constructor using the member initializer list(: BaseClass(valueBase)).

```
#include <iostream>
// Base class with a parameterized constructor
class BaseClass {
public:
    BaseClass(int value) : baseValue(value) {
        std::cout << "BaseClass Constructor: " << baseValue << std::endl;
    }
    void displayBaseValue() {
        std::cout << "BaseClass Value: " << baseValue << std::endl;
    }
private:
    int baseValue;
};
// Derived class inheriting from BaseClass
class DerivedClass : public BaseClass {
public:
    DerivedClass(int derivedValue, int baseValue) : BaseClass(baseValue),
derivedValue(derivedValue) {
        std::cout << "DerivedClass Constructor: " << derivedValue << ", " << baseValue
<< std::endl;
    }
    void displayDerivedValue() {
        std::cout << "DerivedClass Value: " << derivedValue << std::endl;
    }
private:
    int derivedValue;
};
int main() {
    DerivedClass derivedObject(42, 24);
    derivedObject.displayBaseValue();
    derivedObject.displayDerivedValue();
           return 0;
}
```

In this example:

BaseClass has a parameterized constructor that initializes a private member baseValue.

DerivedClass inherits from BaseClass and has its own parameterized constructor, which initializes both the base class and derived class members.

The main function creates an object of the derived class and calls member functions from both the base and derived classes.

When you run this program, you'll see output indicating the construction and values of both the base and derived classes.

4.5.3 Some rules about constructors:

1. In the absence of an explicit constructor declaration in a class, the compiler automatically generates a default constructor, which lacks parameters and carries out no actions.
2. Upon explicit declaration of a constructor, the compiler refrains from generating a default constructor.
3. If there is a necessity to utilize the default constructor of the base class, it must be explicitly defined in the base class, typically with an empty body.
4. A default constructor should be defined under the following circumstances:
   a. When it is employed for creating an object of the derived class. In such instances, a default constructor should be explicitly defined in the base class.
   b. In situations where a constructor has already been explicitly declared, and there is a desire to declare objects in the program without initializing them at the time of their declaration.

```
#include <iostream>
// Base class with a parameterized constructor
class BaseClass {
public:
    BaseClass(int value) : baseValue(value) {
        std::cout << "BaseClass Constructor: " << baseValue << std::endl;
    }
    // Default constructor
    BaseClass() {
        std::cout << "BaseClass Default Constructor" << std::endl;
    }
private:
    int baseValue;
};
// Derived class inheriting from BaseClass
class DerivedClass : public BaseClass {
public:
    // Parameterized constructor calling the base class constructor
    DerivedClass(int derivedValue, int baseValue) : BaseClass(baseValue),
derivedValue(derivedValue) {
        std::cout << "DerivedClass Constructor: " << derivedValue << ", " <<
baseValue << std::endl;
    }
    // Default constructor calling the base class default constructor
    DerivedClass() : BaseClass() {
        std::cout << "DerivedClass Default Constructor" << std::endl;
    }
private:
    int derivedValue;
};
int main() {
    // Creating objects of the derived class with different constructors
    DerivedClass derivedObj1(42, 24);
    DerivedClass derivedObj2;
    return 0;
}
```

This example demonstrates a base class BaseClass with a parameterized constructor and a

default constructor. The derived class DerivedClass inherits from BaseClass and has its own parameterized constructor and default constructor. The main function creates objects of the derived class using both constructors.

4.6 Polymorphism, Virtual Functions

4.6.1 Polymorphism:

Linguistically, polymorphic means the ability to take on different forms. Polymorphism is one of the foundations of object-oriented programming.

Although the polymorphism technique can be used with functions that do not belong to classes, it is a fundamental complement to the inheritance technique.

It allows for the redefinition of methods/data inherited from the parent class. In general, in a derived class, three types of members (methods/data) are distinguished:

1. Methods (data) specific to the new class: this is referred to as extension.
   a. These methods (data) can have a name not used in the parent class.
   b. They can also have a name used in the parent class with parameters in a different number and/or type, constituting an overload of the method.
2. Methods/data inherited from the parent class: this is then inheritance. When using an inherited method on an object of the derived class, it is implicitly converted into an object of the parent class. By default, all methods/data of the parent class are inherited.
3. Methods/data that redefine existing methods in the parent class: this is masking. These methods have the same name and the same prototype (declaration) as those they redefine. It is then said that the method of the derived class specializes that of the parent class.

4.6.2 Virtual Functions:

If a pointer to a base class is declared, it becomes possible to instantiate objects from the entire hierarchy of classes derived from that base class. However, if a method in the base class is hidden in the derived classes (meaning the function retains exactly the same signature), calling it on one of the objects of the derived class will always invoke the method from the same prototype in the base class. Thanks to virtual functions, polymorphism allows two objects from different classes to respond differently to the same call (function with the same name and signature).

4.6.2.1 Definition of a virtual function:

A virtual function, also known as a polymorphic function, is a member function invoked based on the object's type rather than the pointer's type. To declare a virtual function in the base class, the "virtual" keyword is used before the function prototype. This approach allows defining specific versions of the function for each derived class, enabling the calling of the appropriate function based on the object's type.

When using an external definition of a virtual function, it is unnecessary to repeat the "virtual" keyword before the function body. Simply using the "virtual" keyword before the base class

method is sufficient, without repeating it for derived class methods. However, it is recommended to include this keyword for all derived classes to avoid traversing the entire class hierarchy in search of virtual functions.

When calling a virtual function, the program first looks for the function in the object's class. If not found, it ascends the class hierarchy. The choice of the virtual function to execute occurs at runtime rather than compile time, unlike other functions. This technique is known as dynamic binding.

Consider the following example, which revisits the "affiche" method of the "vehicle" and "car" classes defined earlier. The distinction here lies in using a pointer "pv" to the base class "vehicle," which is first referenced with an instance of the "vehicle" class and then with an instance of the "voiture" class. The "affiche()" method is called using this same pointer.

```
# include < iostream >
using namespace std ;
class vehicu
le
{
protected :
int License ;
float power ;
public :
vehicule (int n , float m)
{
License =n ;
power =m ;
}
virtual void affiche ()
{
cout << " License : " << License << "\ n";
cout << " Power : " << power << "\ n";
}
};
class car : public vehicle
{
int nbdoor ;
public :
car (int n , float m ,int p )
: vehicule (n ,m)
{

nbdoor = p;
}
virtual void affiche ()
{
vehicule :: affiche ();
cout << " number of doors: " << nbdoor << "\n ";
}
};
int main ()
{
int n;
float m ;
int p;
vehicule * pv ;
cout <<" \n License Plate and Car Power and number of doors " ;
cin >>n >>m >> p ;
pv =new car (n ,m ,p );
pv -> affiche ();
```

```
cout <<" \n License Plate and Car Power ";
cin >>n >> m;
pv =new vehicle (n , m );
pv -> affiche ();
return 0;
}
```

During the execution of the program, the following text is displayed:

We can clearly see that with the same syntax of the call "pv->affiche()", the program invokes different methods depending on the object type. In the first case, it calls the method of the derived class "voiture", and in the second case, it invokes the method of the base class "vehicule".

4.6.2.2 Role of a Virtual Function:

A virtual function can be defined in a derived class for several reasons:

1. **Modification of the Base Class's Standard Processing** : Redefining a virtual function allows for the modification of the standard processing defined in the base class. This provides the flexibility to adapt the function's behavior according to the specific needs of the derived class.

2. **Completion of the Base Class's Standard Processing** : A virtual function can also be used to complement the standard processing of the base class. In this case, the function in the base class is called by the corresponding function in the derived class, allowing for the addition of additional features without completely rewriting the base class's code.

3. **Cancellation of the Base Class's Standard Processing** : In certain situations, it may be necessary to completely cancel the standard processing defined in the base class. To achieve this, the virtual function can be redefined with an empty body, thus canceling any actions defined in the base class.

4.6.2.3 Pure Virtual Functions and Abstract Classes:

The declaration of a pure virtual function is done by adding "=0" at the end of its prototype. Unlike regular virtual functions, every pure virtual function must be redefined in the derived classes.

An abstract class is a class that contains at least one pure virtual function. An abstract class cannot be instantiated directly, meaning objects of this class cannot be created.

Abstract classes are typically used to consolidate common members among derived classes and to enforce a standard behavior (an interface) that all derived classes must implement consistently. This approach aims to make applications more robust: even if changes are made to the processing in the derived classes, no modifications are required in the programs using these classes.

5 Chapter 5: STL

5.1 The C++ Standard Library:

The C++ Standard Library (STL) is an essential part of the C++ programming language. It provides a set of classes and functions that extend the language's capabilities, making the development of robust and efficient software easier. The STL is based on templates, enabling generic programming, which means creating code independent of data types. Using the C++ Standard Library allows programmers to benefit from a robust and portable implementation of basic features while promoting modern and efficient programming practices.

5.2 Concept of containers:

Containers in programming refer to data structures that facilitate the storage, organization, and access of homogeneous collections of elements. These structures provide structured means to manage and manipulate data. In C++, the Standard Template Library (STL) offers a variety of containers such as vectors, lists, and sets, each tailored to specific programming needs. Containers enable sequential, associative, as well as stack and queue-based storage approaches, providing significant flexibility in software development.

5.3 The concept of complexity:

The concept of complexity refers to the analysis of algorithm efficiency and the associated resource requirements during their execution. Complexity can be categorized into temporal complexity and spatial complexity.

5.3.1 Temporal Complexity :

This facet of complexity assesses the amount of time an algorithm requires to complete based on the size of the input data. It helps understand how the algorithm's execution time increases with an increase in input size. Temporal complexity is often expressed using "big O" notation, providing an upper bound on the growth rate of the algorithm's execution time.

5.3.2 Spatial Complexity :

Spatial complexity involves the analysis of the amount of memory space an algorithm requires concerning the input size. It evaluates the algorithm's memory usage during its execution. Similar to temporal complexity, spatial complexity is expressed using "big O" notation, offering an upper limit on the growth of the algorithm's memory usage.

5.4 The dynamic array (vector):

The dynamic array, also known as "vector" in the Standard Template Library (STL) in C++, is an adaptable data structure that allows for the dynamic storage of a set of elements of the same type. Here are some detailed explanations regarding this dynamic array (vector) :

1. **Size Flexibility**: Unlike static arrays in C++, the size of a vector can be dynamically adjusted during program execution. This means that you can add or remove elements without needing to specify the initial size of the array.

2. **Random Access**: Elements in a vector can be accessed randomly using indices, providing fast access to any element in the array.

3. **Automatic Memory Management**: The vector automatically manages the memory required to store its elements. It allocates additional memory automatically when new elements are added, simplifying memory management.

4. **Useful Member Functions**: The vector offers various useful member functions, such as « **push_back** » to add an element to the end, « **pop_back** » to remove the last element, and functions for resizing and reserving memory.

5. **Iterator Support**: Vectors support the use of iterators, making it easy to traverse elements through loops.

Example :

```
#include <iostream>
#include <vector>
int main() {
    // Creating a vector of integers
    std::vector<int> myVector;
    // Adding elements to the vector
    myVector.push_back(1);
    myVector.push_back(2);
    myVector.push_back(3);
    // Accessing elements using indices
    std::cout << "First element: " << myVector[0] << std::endl;
    // Iterating through elements using an iterator
    std::cout << "Vector elements: ";
    for (auto it = myVector.begin(); it != myVector.end(); ++it) {
        std::cout << *it << " ";
    }
}
```

The concept of iterators in programming refers to objects that provide a means to traverse or iterate through the elements of a container, such as an array, a list, or other data structures. Iterators act as a bridge between algorithms and data structures, enabling sequential access to the elements of a container.

Here are some important details about the concept of iterators:

1. **Sequential Access**: Iterators enable sequential access to the elements of a container. They provide a mechanism to move from one element to the next in a well-defined order.
2. **Abstraction of Container Navigation**: Iterators abstract the process of navigating through a container, allowing algorithms to work with different types of containers without needing to know their internal structure.
3. **Standardized Interface**: Iterators typically offer a standardized interface, including operations such as advancing to the next element, accessing the current element, and checking for the end of the container.

Example :

```
#include <iostream>
#include <vector>
int main() {
    // Creating a vector of integers
    std::vector<int> myVector = {1, 2, 3, 4, 5};
    // Using an iterator to traverse the vector
    std::cout << "Vector elements using iterator: ";
    for (std::vector<int>::iterator it = myVector.begin(); it != myVector.end();
++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

This program will output:

```
Vector elements using iterator: 1 2 3 4 5
```

This demonstrates how iterators provide a way to sequentially access the elements of a container, in this case, a vector.

5.5 The list:

The list in C++ refers to a doubly linked data structure in the Standard Template Library (STL). Here are some specific details about this list :

1. **Doubly Linked** : A list is structured as a doubly linked data structure, meaning that each element in the list has links to the previous and next elements. This feature facilitates fast operations for inserting and deleting elements at various positions in the list.

2. **Non-Random Access**: Unlike dynamic arrays (vectors), access to elements in a list is not done randomly using indices. Instead, elements are traversed sequentially using iterators.
3. **Efficient Insertion and Deletion**: Operations for inserting and deleting elements are efficiently performed in a list, especially when done in the middle of the list. This makes it a preferred choice in situations requiring frequent modifications to the data structure.
4. **Automatic Memory Management**: Similar to other containers in the STL, the list ensures automatic management of the memory required to store its elements.
5. **Iterators**: Lists support the use of iterators, simplifying the sequential traversal of elements.

Here is a simple example illustrating the use of a list in C++:

```
#include <iostream>
#include <list>
int main() {
    // Creating a list of integers
    std::list<int> myList = {1, 2, 3, 4, 5};
    // Adding an element to the end of the list
```

```
    myList.push_back(6);
    // Iterating through elements using an iterator
    std::cout << "List elements: ";
    for (auto it = myList.begin(); it != myList.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

5.6 The associative array (map):

The associative array, commonly referred to as "map" in C++, is a data structure in the Standard Template Library (STL) designed to store key-value pairs. Here are some details about the associative array (map) :

1. **Key and Value** : Each element in the map is associated with a unique key and a corresponding value. The key serves as an identifier, enabling fast access to the associated value.
2. **Sorted by Key** : By default, a map organizes its elements in ascending order based on keys. This facilitates efficient searching for a specific key.
3. **Logarithmic Complexity** : Operations such as search, insertion, and deletion in a map have logarithmic complexity concerning the number of elements, making it efficient for larger datasets.
4. **No Duplicate Keys** : Each key within a map is unique. If an attempt is made to insert an already-existing key, the value associated with that key is updated.
5. **Use of Bracket Operator** : To access the value associated with a key, the bracket operator (« **[ ]** ») is used. For example, map["key"] will return the value associated with the key "key."
6. **Iterator Support** : Maps support the use of iterators, facilitating sequential traversal of elements.

Here is a simple example illustrating the use of a map in C++:

```
#include <iostream>
#include <map>
int main() {
    // Creating a map with key-value pairs
    std::map<std::string, int> myMap;
    myMap["one"] = 1;
    myMap["two"] = 2;
    myMap["three"] = 3;
    // Accessing elements using the bracket operator
    std::cout << "Value associated with key 'two': " << myMap["two"] << std::endl;
    // Iterating through elements using an iterator
    std::cout << "Elements of the map: ";
    for (auto it = myMap.begin(); it != myMap.end(); ++it) {
        std::cout << "(" << it->first << ", " << it->second << ") ";
    }
    return 0;
}
```

5.7 A pair of elements (pair):

In programming, a pair of elements, typically defined by the "pair" data type, represents a structure that allows associating two elements of different types as a single entity. In C++, the Standard Template Library (STL) provides the std::pair template class to implement this concept. Here is some information regarding the use of a pair of elements:

1. **Pair Composition:** The pair consists of two elements commonly referred to as "first" and "second." These terms do not necessarily reveal the nature of the elements but distinguish them.
2. **Element Types**: The two elements of a pair can be of different types. For example, a pair may include a first element of integer type and a second element of string type.
3. **Pair Instantiation:** To instantiate a pair, you can use the constructor of the "std::pair" class or use the utility function "std::make_pair."
4. **Access to Elements:** Access to the pair's elements is done through the members first and second.
5. **Common Usage:** Pairs are frequently used when two values need to be associated without requiring a more complex structure.
6. **Usage with Other Structures:** Pairs are also employed in conjunction with other data structures such as maps, where each element in the map is a key-value pair.

```
#include <iostream>
#include <utility>
int main() {
    // Creating a pair of integer and string
    std::pair<int, std::string> myPair(42, "Bonjour");
    // Accessing and displaying the elements of the pair
    std::cout << "First element: " << myPair.first << std::endl;
    std::cout << "Second element: " << myPair.second << std::endl;
    return 0;
}
```

5.8 A set of elements:

In programming, a set of elements, commonly referred to as a "set," is a data structure within the C++ Standard Template Library (STL). Here are some details about the use of a set of elements :

Unordered Nature : A set is an unordered collection of unique elements. Elements are not stored in any specific order, and each element is unique, meaning that duplicates are not allowed.

Automatic Sorting : Elements in a set are automatically sorted during insertion. This facilitates efficient search and retrieval operations.

Use of Iterators : Iterators can be used to sequentially traverse the elements of a set.

```
#include <iostream>
#include <set>
int main() {
    // Creating a set of integers
    std::set<int> mySet;
    // Inserting elements into the set
    mySet.insert(3);
    mySet.insert(1);
    mySet.insert(4);
    mySet.insert(2);
```

```
    // Traversing elements using an iterator
    std::cout << "Elements of the set: ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }
    return 0;
}
```

5.9 The stack:

In programming, a stack, often referred to as a "stack," is a data structure that follows the Last In, First Out (LIFO) principle. Here are some considerations regarding the use of a stack:

1. **LIFO Principle**: The stack adheres to the Last In, First Out principle, meaning that the last element added is the first to be removed. This characteristic sets it apart from other structures such as queues, which follow the First In, First Out (FIFO) principle.
2. **Basic Operations**: Fundamental operations on a stack include adding an element to the top (push) and removing the element from the top (pop). These operations occur exclusively at the top of the stack.
3. **Common Use**: Stacks are frequently used to track the execution of functions in a program (handling function calls and recursive calls), to reverse the order of elements, or to perform undo operations in applications.
4. **Implementation with Containers**: In C++, the stack can be implemented using the std::stack class from the Standard Template Library (STL). This class utilizes other containers, such as vectors or deques, as the underlying support.

Fundamental operations when manipulating a stack include:

- **Push**: Adding an element to the top of the stack.
- **Pop**: Removing the element from the top of the stack and returning it.
- **Is the stack empty? (empty)**: Returning true if the stack is empty (no elements present), otherwise, returning false.
- **Is the stack full? (full)**: In some implementations, especially those with a fixed size, this function may return true if the stack is full, indicating that no more elements can be added.
- **Number of elements in the stack (size)**: Returning the current number of elements in the stack.
- **Stack size (capacity)**: In some implementations with a fixed size, this function returns the maximum number of elements that can be stored in the stack.
- **What is the top element (top)?**: Returning the element at the top of the stack without removing it.

```
#include <iostream>
#include <stack>
int main() {
    // Creating a stack of integers
    std::stack<int> myStack;
```

```
    // Adding elements to the stack
    myStack.push(3);
    myStack.push(1);
    myStack.push(4);
    // Removing and displaying elements from the stack
    std::cout << "Elements of the stack: ";
    while (!myStack.empty()) {
        std::cout << myStack.top() << " ";
        myStack.pop();
    }
    return 0;
}
```

5.10 The queue:

In programming, a queue is a data structure that adheres to the First In, First Out (FIFO) principle. Here are detailed explanations about the use of a queue:

- **FIFO Principle**: The queue follows the FIFO principle, meaning that the first element added is also the first one to be removed. This approach differs from a stack, which follows the Last In, First Out (LIFO) principle.
- **Basic Operations**: Fundamental operations on a queue include adding an element to the back (« **enqueue** ») and removing the element from the front (« **dequeue »**). Elements are added to the back of the queue and removed from the front.
- **Common Use**: Queues are frequently used for tasks such as managing tasks in a print queue, processing jobs in operating systems, or modeling scenarios where elements enter a system and are processed in the order of arrival.
- **Implementation with Containers**: In C++, the queue can be implemented with the « **std::queue** » class from the Standard Template Library (STL). This class often uses other containers, such as deques, as underlying support.

Here is a simple example of using a queue in C++:

```
#include <iostream>
#include <queue>
int main() {
    // Creating a queue of integers
    std::queue<int> myQueue;
    // Adding elements to the queue
    myQueue.push(3);
    myQueue.push(1);
    myQueue.push(4);
    // Removing and displaying elements from the queue
    std::cout << "Elements of the queue: ";
    while (!myQueue.empty()) {
        std::cout << myQueue.front() << " ";
        myQueue.pop();
    }
    return 0;
}
```

5.11 Containers instructions :

5.11.1 Sorting a container (sort):

Sorting a container, often referred to as "sort" in C++, is a fundamental operation that organizes the elements of the container according to a specified order. This operation is widely used to structure data for efficient searching and retrieval.

The use of « **std::sort »** allows the application of a generic sorting operation to a variety of containers such as arrays, vectors, lists, and more. A particularly powerful aspect of « **std::sort »** lies in its customization capabilities. This is achieved by providing a custom comparison function or a lambda expression. This customization provides the flexibility needed to sort elements based on specific criteria. For example, it is possible to sort a list of complex objects based on one of their properties.

Here is a summary of customizing sorting with « **std::sort »** using a lambda expression:

```
std::sort(myVector.begin(), myVector.end(), [](int a, int b) {
    // Sorting in descending order
    return a > b;
});
```

It is also important to note that, when sorting containers containing custom types, the « < » (less than) operator must be defined for those types. If not, a custom comparison function must be provided.

5.11.2 Deletion of elements from a container (clear, erase, and remove):

The removal of elements in a container can be done using several operations, with three main ones : « **clear** », « **erase** », and « **remove** ». Here are detailed explanations for each of these operations:

1. **Clear** : The clear method allows completely emptying the contents of a container by removing all its elements « `container.clear( )`».

Example :

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};
    myVector.clear();
    std::cout << "Size after clear: " << myVector.size() << std::endl;
    return 0;
}
```

2. **Erase** : The erase function is used to delete one or more specific elements from the container based on their position or value.

By position: « `container.erase(iterator)` »;

By range: « `container.erase(startIterator, endIterator)` »;

Example :

```
#include <iostream>
#include <list>
int main() {
    std::list<int> myList = {1, 2, 3, 4, 5};
```

```
    auto it = myList.begin();
    ++it; // Move the iterator to the second element
    myList.erase(it);
    for (int num : myList) {
        std::cout << num << " ";
    }
    return 0;
}
```

3. **Remove** : The remove function is used to delete all elements equal to a certain value from the container « `container.remove(value)` ».

Example :

```
#include <iostream>
#include <forward_list>
int main() {
    std::forward_list<int> myForwardList = {1, 2, 3, 4, 3, 5};
    myForwardList.remove(3);
    for (int num : myForwardList) {
        std::cout << num << " ";
    }
    return 0;
}
```

4. **The search for elements (find):** The « **std::find »** function in C++ is a function used to search for the first occurrence of a specified value within a range defined by two iterators. Here are some specific details regarding its usage :

The signature of the « **std::find »** function is as follows:

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

« First »: Iterator pointing to the beginning of the range.

« Last »: Iterator pointing to the end of the range.

« Value »: Value to search for within the range.

The function returns an iterator pointing to the first occurrence of the searched value in the range. If the value is not found, it returns the last iterator, indicating the end of the range.

Example: Here's a simple example with a vector of integers:

```
#include <iostream>
#include <vector>
#include <algorithm>
int main() {
    std::vector<int> myVector = {1, 2, 3, 4, 5};
     auto it = std::find(myVector.begin(), myVector.end(), 3);
    if (it != myVector.end()) {
        std::cout << "Value 3 was found at position: " <<
std::distance(myVector.begin(), it) << std::endl;
    } else {
        std::cout << "Value 3 was not found in the vector." << std::endl;
    }
    return 0;
}
```

In this example, the « **std::find »** function is used to search for the value 3 in the vector.

Complexity: The complexity of « **std::find »** depends on the type of container used. For vectors, the search has linear complexity as it traverses the elements sequentially.

6 Chapter 6: Advanced concepts

6.1 C++ and Exception Handling

6.1.1 Standard Library « exception »:

It is common for people to use the « **exception** » standard library in C++. While not mandatory, it contains all the relevant details for using standard exceptions in C++. For this initial example, we will use the « **std::exception** » type to report an issue, but note that this is not the best approach (reporting an issue without describing its nature has limited practical utility). Our program now looks as follows; at this stage, it still does not handle any exception cases.

```
#include <iostream>
#include <stdexcept>
class MyException : public std::exception {
public:
    const char* what() const noexcept override {
        return "This is a custom exception.";
    }
};
int main() {
    try {
        throw MyException();
    } catch (const std::exception& e) {
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

6.1.2 Throw an exception:

The called subroutine is responsible for determining if it is in an exceptional situation. This requires a thorough analysis on the part of the subroutine programmers, similar to what is needed for any error handling. When the subroutine identifies an exception, its task is to inform the calling subroutine. It is then said that the called subroutine « **throws** » or « **raises** » an exception to the calling subroutine, using the throw statement (literally: "**throw!**"). Note the syntax : an exception is thrown, to which (at least in this example) a descriptive message is attached – the exception message.

```
#include <iostream>
#include <stdexcept>
void myFunction(int value) {
    try {
        if (value == 0) {
            throw std::runtime_error("Value cannot be zero.");
        }
        // Your code here
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }
}
int main() {
    int userInput;
    std::cout << "Enter a value: ";
    std::cin >> userInput;
    myFunction(userInput);
    return 0;
}
```

6.1.3 Catch an Exception:

```
#include <iostream>
#include <stdexcept>
int main() {
    try {
        // Code that might throw an exception
        throw std::runtime_error("This is an example exception.");
    } catch (const std::exception& e) {
        // Code to handle the exception
        std::cerr << "Caught exception: " << e.what() << std::endl;
    }
    return 0;
}
```

The try block contains the code that may generate an exception. If an exception is indeed thrown (in this case, a std::runtime_error), the program moves to the catch block. The type of the exception to be caught is specified in the catch block (in this case, const std::exception& e), and the exception is caught by reference. Then, the what() function of the exception is used to obtain a description of the error, which is subsequently printed to the standard error stream (std::cerr). You have the option to customize the type of exception you catch based on your specific needs.

6.1.4 Exceptions and Classes :

In C++, the joint use of classes and exceptions is common to enhance error management and strengthen code robustness. Here is an example illustrating how to design a custom exception class in C++:

```
#include <iostream>
#include <stdexcept>
class MyException : public std::exception {
public:
    MyException(const char* message) : message_(message) {}

    const char* what() const noexcept override {
        return message_.c_str();
    }
private:
    std::string message_;
};
class Calculator {
public:
    static int divide(int numerator, int denominator) {
        if (denominator == 0) {
            throw MyException("Division by zero is not allowed.");
        }
        return numerator / denominator;
    }
};
int main() {
    try {
        int result = Calculator::divide(10, 0);
        std::cout << "Result: " << result << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Exception caught: " << e.what() << std::endl;
    }
    return 0;
}
```

6.2 Templates in C++

In C++, the template feature provides a way to reuse source code.

6.2.1 Objectives:

Templates allow for the creation of functions and classes by parameterizing the type of certain components (such as the type of parameters or the return type for a function, or the type of elements for a collection class, for example). They enable the writing of generic code, which can be used for a family of functions or classes that only differ in the value of these parameters. In summary, the use of templates allows for "parameterizing" the type of manipulated data, providing increased flexibility and code reusability.

6.2.2 Declaration of a template:

```
template <class|typename name[=type] [, class|typename name[=type][...]>
```

where "name" is the name assigned to the generic type in this declaration. The keyword "class" here has the same meaning as "type" and can be interchangeably replaced by the keyword "typename". An arbitrary number of generic types can be declared, separating them by commas.

```
template <type parameter[=value][, ...]>
```

where "type" is the type of the constant parameter, "parameter" is the parameter name, and "value" is its default value.

6.2.3 Function and Class Template:

After specifying one or more "template" parameters, it is common to proceed with the declaration or definition of a "template" function or class. Within this definition, generic types can be used just like normal types.

```
template<class T> T findMinimum(T a, T b);
template<class T> class List { T element; };
```

It's worth noting that a template function is often referred to as an algorithm, as is the case with most function templates in the C++ standard library. This concept simply outlines how to accomplish a particular operation.

6.2.4 Example : Template Function

The goal is to create a « **mini()** » function that takes two arguments and returns the smaller of the two. To avoid writing a separate function for each type to handle, we will use a template. The compiler will automatically generate the necessary code for each used type, such as int and float, in this example.

```
#include <iostream>
using namespace std;
template<class T>
T mini(T a, T b) {
    if (a < b) return a;
```

```
    else return b;
}
template<class T>
T mini(T a, T b, T c) {
    return mini(mini(a, b), c);
}
int main() {
    int n = 12, p = 15, q = 2;
    float x = 3.5, y = 4.25, z = 0.25;
    cout << "mini(n, p) -> " << mini(n, p) << endl; // implicit
    cout << "mini(n, p, q) -> " << mini(n, p, q) << endl;
    cout << "mini(x, y) -> " << mini(x, y) << endl;
    cout << "mini(x, y, z) -> " << mini(x, y, z) << endl;
    cout << "mini(n, q) -> " << mini<float>(n, q) << endl; // explicit
    return 0;
}
```

In this example, the « **mini()** » function is defined with a template, allowing it to handle various types. The function is then used with different types of arguments, showcasing the flexibility provided by templates.

7 Bibliography

[1] Y. Gerometta and J. L. Corre, C++ Le Guide Complet. Micro Applications, 2008.

[2] G. Willms, C++. PC Poche, Micro Applications, 2000.

[3] C. Delannoy, Programmer en C++. Eyrolles, 2006.

[4] C. Delannoy, Apprendre le C++. Eyrolles, 2008.

[5] S. Dupin, Le Langage C++. Le tout en poche, Campus Press, 2005.

[6] M. Nebra and M. Shcaller, Programmez avec le Langage C++. Le Livre du Z´ero, 2009.

[7] N. Benabadji and N. Bechari, Guide Pratique de Programmation en C++. Houma INFO, Editions Houma, 2009.

[8] S. Graine, Le Langage C++. L'Abeille, 2004.

[9] C. Delannoy, Exercices en Langage C++. Chihab-Eyrolles, 1993.

[10] B. Stroustrup, The C++ Programming Language. Addison-Wesley, 3rd ed., 1997.

[11] J. R. Hubbard, Theory and Problems of Programming with C++. Schaum's Outline Series, McGraw-Hill, 2nd ed., 2000.

[12] B. Meyer, Conception et Programmation orientées objet, Eyrolles, 2000.

[13] F. Barbier, Conception orientée objet en Java et C++: Une approche comparative, Pearson Education, 2009.

[14] E. Yourdon, P. Coad, Conception orientée objet, Dunod, 1997.

[15] H. Bersini, La programmation orientée objet. Cours et exercices UML 2 avec Java, C#, C++, Python, PHP et LINQ, Eyrolles; 6e édition, 2013.

[16] C. Delannoy, S'initier à la programmation et à l'orienté objet : Avec des exemples en C, C++, C#, Python, Java et PHP, Eyrolles; 2e édition, 2016.

[17] L. GERVAIS, Apprendre la Programmation Orientée Objet avec le langage C# (2e édition), Editions ENI; 2e édition, 2016.

[18] T. GROUSSARD Luc GERVAIS, Java 8 - Apprendre la Programmation Orientée Objet et maîtrisez le langage (avec exercices et corrigés), Editions ENI, 2015.

[19] L. GERVAIS, Apprendre la Programmation Orientée Objet avec le langage Java, ENI, 2014.