INTRODUCTION TO C++

Sercan Külcü | 21.06.2023

Contents

1 Basic differences between C and C++

In this chapter, we'll explore the basic differences between C and C++. While both languages share a common history and syntax, there are key distinctions that set them apart. By understanding these differences, you'll be able to choose the most suitable language for your programming needs.

Object-Oriented Programming (OOP) Support:

C is a procedural programming language, primarily focused on structured programming concepts. It does not have built-in support for object-oriented programming. On the other hand, C++ is an extension of C that introduces object-oriented programming features, including classes, inheritance, polymorphism, and encapsulation. C++ allows you to write code in both procedural and object-oriented styles, giving you greater flexibility in program design.

Standard Template Library (STL):

One significant advantage of C++ over C is the inclusion of the Standard Template Library (STL). The STL provides a collection of powerful data structures and algorithms, such as vectors, lists, queues, and sorting algorithms. These pre-defined templates save you time and effort by offering ready-to-use solutions for common programming tasks. In C, you would need to implement such data structures and algorithms from scratch or rely on third-party libraries.

Exception Handling:

Exception handling is another notable difference between C and C++. C++ provides built-in exception handling mechanisms using try-catch blocks. This allows you to handle exceptional situations gracefully and recover from errors. In C, error handling is typically done using return codes or global error variables, which require manual checking and propagation throughout the codebase. C++'s exception handling simplifies error management and enhances code readability.

Name Mangling and Function Overloading:

C++ supports function overloading, which means you can define multiple functions with the same name but different parameter lists. The compiler differentiates between these functions based on their parameters, enabling you to write more expressive and reusable code. To achieve this, C++ uses a technique called name mangling, which encodes additional information into the function's name to distinguish it from other functions with the same name. C, on the other hand, does not support function overloading or name mangling.

Standard Libraries:

Both C and C++ have their own standard libraries, but C++'s standard library (known as the C++ Standard Library) is an extension of the C Standard Library. The C++ Standard Library incorporates additional features, such as I/O streams, string manipulation functions, and support for dynamic memory management using the new and delete operators. While C's standard library provides essential functionality, C++ offers a richer set of tools and abstractions.

Compatibility:

C++ is designed to be compatible with C code. You can often use C code in a C++ program by including the appropriate header files and wrapping the C code in an extern "C" block. However, C is not directly compatible with C++ due to the additional features and language constructs introduced in C++. If you have existing C code, it can be easily integrated into a C++ project with some modifications, but the reverse is not always true.

2 Introduction to object-oriented programming

Object-oriented programming is a powerful paradigm that allows you to organize and structure your code in a more modular and intuitive manner. By understanding the core principles and concepts of OOP, you'll be able to leverage its benefits in your C programs.

At its core, object-oriented programming is a programming paradigm that revolves around the concept of objects. An object represents a real-world entity or concept and encapsulates its data and behaviors. It allows you to model complex systems by breaking them down into smaller, self-contained units.

The key principles of OOP include:

a) Encapsulation: Encapsulation is the idea of bundling data and related functions into a single unit, called a class. A class serves as a blueprint for creating objects and defines their attributes (data members) and behaviors (member functions).

b) Inheritance: Inheritance enables the creation of new classes (derived classes) based on existing classes (base classes). Derived classes inherit the properties and behaviors of the base class, allowing code reuse and promoting a hierarchical relationship among classes.

c) Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common base class. This concept enables you to write code that can work with objects of multiple types, providing flexibility and extensibility.

Classes and Objects in C:

While C is not a purely object-oriented language like C++, you can still apply object-oriented principles to some extent. In C, you can define structures to represent objects and functions to manipulate them. By organizing related data and functions together, you create a basic form of encapsulation.

For example, consider a simple structure representing a point:

```
struct Point {
  int x;
```

```
  int y;
};
```

You can then create functions that operate on this structure:

```
void initializePoint(struct Point* point, int x, int y) {
  point->x = x;
  point->y = y;
}

void printPoint(const struct Point* point) {
  printf("(%d, %d)\n", point->x, point->y);
}
```

By encapsulating data and functions within a structure, you create a unit that represents a point object, allowing you to work with points in a more organized manner.

The object-oriented programming paradigm offers several benefits, including:

a) Modularity and Reusability: OOP promotes code modularity by encapsulating data and behaviors within objects. This allows for better organization and makes it easier to reuse code across different parts of a program or in other projects.

b) Maintainability: OOP facilitates code maintenance by providing a clear structure and separation of concerns. Objects encapsulate their own data and behaviors, making it easier to modify or extend specific functionalities without affecting other parts of the program.

c) Flexibility and Extensibility: Through inheritance and polymorphism, OOP enables flexibility and extensibility. Inheritance allows you to create specialized classes based on existing ones, inheriting their properties and behaviors. Polymorphism enables writing code that can work with objects of different types, providing flexibility in handling diverse data.

d) Code Readability: OOP promotes code readability by allowing you to model your program using real-world concepts and entities. Objects, classes, and their interactions mimic real-world relationships, making the code easier to understand and maintain.

OOP in C++:

While C supports some object-oriented principles, C++ is designed as a fully object-oriented language. It provides native support for classes, objects, inheritance, polymorphism, and other OOP features.

In C++, you can define classes directly, specifying their data members and member functions within the class definition. This makes it easier to encapsulate related data and behaviors into cohesive units. Additionally, C++ provides access specifiers (public, private, protected) to control the visibility and accessibility of class members.

C++ also introduces constructors and destructors, which are special member functions used for object initialization and cleanup, respectively. Constructors are automatically called when objects are created, and destructors are called when objects are destroyed.

3 Classes, objects, and constructors

In this chapter, we'll dive deeper into the world of object-oriented programming (OOP) and explore the concepts of classes, objects, and constructors. These fundamental elements of OOP allow you to create structured and modular code by encapsulating data and behaviors into reusable units. Let's get started!

In object-oriented programming, a class is a blueprint or template that defines the structure and behavior of objects. It encapsulates data (attributes) and functions (methods) that operate on that data. Classes provide a way to create user-defined data types in C that can have their own properties and behaviors.

An object, on the other hand, is an instance of a class. It represents a specific entity or concept and contains its own set of data and functions. You can create multiple objects from a single class, each with its own unique state and behavior.

Defining a Class in C:

In C, you can simulate classes using structures. Define a structure to represent the data members of the class and functions to operate on that structure. This approach provides a basic level of encapsulation and allows you to work with objects in a structured manner.

For example, let's define a class called Rectangle that represents a rectangle:

```
struct Rectangle {
  int length;
  int width;
};
```

You can then create functions to operate on this structure:

```
void initializeRectangle(struct Rectangle* rect, int length, int width) {
  rect->length = length;
  rect->width = width;
}
```

```
int calculateArea(const struct Rectangle* rect) {
  return rect->length * rect->width;
}

void printRectangle(const struct Rectangle* rect) {
  printf("Length: %d, Width: %d\n", rect->length, rect->width);
}
```

To create an object of a class, you declare a variable of the structure type and initialize it using the class's initialization function. This process allocates memory for the object and sets its initial state.

For example, to create a rectangle object and perform operations on it:

```
struct Rectangle myRectangle;
initializeRectangle(&myRectangle, 5, 3);
printRectangle(&myRectangle);
int area = calculateArea(&myRectangle);
printf("Area: %d\n", area);
```

Constructors and destructors are special member functions that provide object initialization and cleanup, respectively. While C does not have native support for constructors and destructors like C++, you can simulate their functionality.

To simulate a constructor, create a function that initializes the object and returns it. This function acts as a constructor by allocating memory and setting initial values.

For example:

```
struct Rectangle* createRectangle(int length, int width) {
  struct Rectangle* rect = malloc(sizeof(struct Rectangle));
```

```
    rect->length = length;
    rect->width = width;
    return rect;
}
```

To simulate a destructor, create a function that frees the memory allocated for the object. This function acts as a destructor by performing necessary cleanup operations.

```
void destroyRectangle(struct Rectangle* rect) {
    free(rect);
}
```

4 Basic input/output in C++

In this chapter, we'll dive into the basics of input/output (I/O) operations in C++. Input/output is an essential aspect of programming as it allows interaction with users, reading data from files, and displaying results. We'll explore various techniques to perform I/O operations in C++. Let's get started!

C++ provides the standard input/output library, iostream, which offers convenient ways to perform I/O operations. The iostream library includes two main classes: cin and cout.

- cin is used for reading input from the user or from another source.
- cout is used for displaying output to the user or writing to a file.

You can use cout to display output to the user or write to a file. It is as simple as using the << operator to send data to the output stream.

For example:

```
#include <iostream>


int main() {
  int age = 25;
  std::cout << "My age is: " << age << std::endl;
  return 0;
}
```

In this example, we use cout to display the message "My age is: " followed by the value of the variable age. The << operator is used to concatenate the output, and std::endl is used to insert a newline.

You can use cin to read input from the user or from a file. It is as simple as using the >> operator to extract data from the input stream.

For example:

```
#include <iostream>
```

```
int main() {
  int number;
  std::cout << "Enter a number: ";
  std::cin >> number;
  std::cout << "You entered: " << number << std::endl;
  return o;
}
```

In this example, we use cin to read an integer value entered by the user. The >> operator is used to extract the input and store it in the variable number. We then display the entered number using cout.

In addition to standard input and output, C++ provides support for reading from and writing to files. To perform file I/O, you need to include the <fstream> library.

For example:

```
#include <iostream>
#include <fstream>

int main() {
  std::ofstream outputFile("output.txt");
  if (outputFile.is_open()) {
    outputFile << "Hello, File I/O!";
    outputFile.close();
    std::cout << "File written successfully." << std::endl;
  } else {
    std::cout << "Unable to open the file." << std::endl;
```

```
    }

    return 0;

}
```

In this example, we create an output file stream (ofstream) and open a file called "output.txt". We then use the << operator to write the message "Hello, File I/O!" to the file. Finally, we close the file. If the file is successfully opened, we display a success message; otherwise, we display an error message.