Course: C++ language

Ali ZAINOUL

for AELION
December 11, 2023

Table of contents

Handling Exceptions in C++

Listing 1: Example of exception handling

```cpp
#include <iostream>

int main() {
    try {
        // Code that may throw an exception
        int result = 10 / 0;
    } catch (const std::exception& e) {
        // Catching and handling the exception
        std::cerr << "Exception␣caught:␣" << e.what() << std::endl
            ;
    }
    return 0;
}
```

Custom Exceptions in C++

Listing 2: Example of custom exception

```cpp
#include <iostream>
#include <stdexcept>

class CustomException : public std::exception {
public:
    const char* what() const noexcept override {
        return "Custom exception occurred!";
    }
};

int main() {
    try {
        // Code that may throw a custom exception
        throw CustomException();
```

Introduction to Namespaces

- Namespaces in C++ provide a way to organize code by encapsulating identifiers (variables, functions, classes) into a named scope. This helps prevent naming conflicts and makes the code more modular and readable.

Exemple

Listing 3: Example of a namespace

```cpp
// Declaration of a namespace
namespace MyNamespace {
    int variable = 42;
    void myFunction() {
        // Code here
    }
}
// Accessing elements of the namespace with :: resolution operator
int main() {
    std::cout << MyNamespace::variable << std::endl;
    MyNamespace::myFunction();
    return 0;
}
```

Using Directives and Aliases

C++ provides ways to simplify the usage of namespaces using `using` directives and `aliases`.

Listing 4: Using directives

```
// Using directive
using namespace MyNamespace;

int main() {
    // No need for MyNamespace:: prefix
    std::cout << variable << std::endl;
    myFunction();
    return 0;
}
```

Using Directives and Aliases

C++ provides ways to simplify the usage of namespaces using `using` directives and `aliases`.

Listing 5: aliases

```
1
2  // Namespace alias
3  namespace ns = MyNamespace;
4
5  int main() {
6      // Using the alias
7      std::cout << ns::variable << std::endl;
8      ns::myFunction();
9      return 0;
10 }
```

Header Files in C++

- Header files in C++ are used to declare and define elements such as functions, classes, and variables that can be shared across multiple source files. They typically have a `.h` or `.hpp` extension.
- We include header files with the directive: `#include "customHeaderFile.h"` or `#include<headerfile>` if the `headerfile` is part of the standard ones. (e.g. `iostream`, `iomanip`, `algorithm`, `vector` etc.)
- Generally speaking:

```
#include <filename.h> // for files in system/default directory
```

or

```
#include "filename.h" // for files in same directory as src file
```

Header Files in C++ - Example

Listing 6: Example of a header file

```cpp
// ExampleHeader.h
#ifndef EXAMPLEHEADER_H
#define EXAMPLEHEADER_H

#include <iostream>

void myFunction();  // Function declaration
class MyClass {
public:
    void classMethod();
};

#endif // EXAMPLEHEADER_H
```

Compilation Process in C++

- The compilation process in C++ involves translating source code into machine code or an intermediate form. It typically consists of several stages: preprocessing, compilation, assembly, and linking.
  1. **Preprocessing:** Handles directives like `#include` and `#define`, producing a preprocessed source file.
  2. **Compilation:** Converts the preprocessed source file into an assembly file containing low-level machine-like code.
  3. **Assembly:** Converts the assembly code into machine code or an object file.
  4. **Linking:** Combines multiple object files and libraries into an executable program.

Compilation Process in C++ - Example

Listing 7: Compilation commands

```
g++ -E source.cpp -o source.ii    # Preprocessing
g++ -S source.ii -o source.s      # Compilation
as source.s -o source.o           # Assembly
g++ source.o -o executable        # Linking
```

Details of Compilation Process in C++ - 1

1. **Preprocessing:**
   - Purpose: Handle preprocessor directives and produce a preprocessed source file.
   - Command: `g++ -E source.cpp -o source.ii`
   - Example: Replace `#include` directives with actual content, expand macros, and handle conditional compilation.
2. **Compilation:**
   - Purpose: Convert the preprocessed source file into an assembly file containing low-level machine-like code.
   - Command: `g++ -S source.ii -o source.s`
   - Example: Translate high-level C++ code into assembly language instructions.

Details of Compilation Process in C++ - 2

3. Assembly:

- Purpose: Convert the assembly code into machine code or an object file.
- Command: `as source.s -o source.o`
- Example: Translate assembly instructions into machine code specific to the target architecture.

4. Linking:

- Purpose: Combine multiple object files and libraries into an executable program.
- Command: `g++ source.o -o executable`
- Example: Resolve references to functions and variables, combine object files, and link necessary libraries.

Note: These stages are part of the overall compilation process, and each stage contributes to generating the final executable program.

C++ Containers - Part 1

C++ provides a variety of containers, each designed for specific use cases. Here are some commonly used containers:

- **std::vector**: Dynamic array implementation.
  - cppreference: std::vector
- **std::list**: Doubly-linked list.
  - cppreference: std::list
- **std::deque**: Double-ended queue.
  - cppreference: std::deque
- **std::array**: Fixed-size array.
  - cppreference: std::array
- **std::map**: Associative container with key-value pairs (sorted by key).
  - cppreference: std::map

C++ Containers - Part 2

- **std::set**: Associative container with sorted unique elements.
  - cppreference: std::set
- **std::unordered_map**: Associative container with key-value pairs (unordered).
  - cppreference: std::unordered_map
- **std::unordered_set**: Associative container with unordered unique elements.
  - cppreference: std::unordered_set
- **std::queue**: Queue adapter.
  - cppreference: std::queue
- **std::stack**: Stack adapter.
  - cppreference: std::stack

C++ Algorithms - `<algorithm>` Header

The C++ Standard Template Library (STL) provides a powerful set of algorithms in the `<algorithm>` header. These algorithms operate on various types of containers and perform operations such as sorting, searching, and transformations.

- Sorting algorithms: `std::sort`, `std::stable_sort`, etc.
- Searching algorithms: `std::find`, `std::binary_search`, etc.
- Numeric algorithms: `std::accumulate`, `std::transform`, etc.
- Set operations: `std::set_union`, `std::set_intersection`, etc.
- and many more...
- **cppreference:** C++ Algorithms

Lambda Functions in C++

- Lambda functions, introduced in C++11, provide a concise way to define anonymous functions in-place.
- Lambda functions can capture variables from their surrounding scope, making them flexible and powerful.

Lambda Functions in C++ - Example

Listing 8: Example of a Lambda Function

```cpp
// Lambda function to add two integers
auto add = [](int a, int b) { return a + b; };

// Usage of the lambda function
int result = add(3, 4);  // result is 7
```

Pointers to Functions in C++

- Pointers to functions allow you to create variables that can store the address of a function.
- Pointers to functions are particularly useful when functions need to be passed as arguments or stored in data structures.

Pointers to Functions in C++ - Example

Listing 9: Example of a Pointer to Function

```cpp
#include <iostream>
// Function to square an integer
int square(int x) { return x * x; }
int main() {
    // Pointer to the square function
    int (*ptrSquare)(int) = square;
    // Using the pointer to call the function
    std::cout << "Square␣of␣5:␣" << ptrSquare(5) << std::endl;  // Output: 25
    return 0;
}
```

Functors in C++

- Functors, short for function objects, are objects that can be invoked as if they were functions.
- Functors are often used in scenarios where a function object with state is needed, providing more flexibility than regular functions.

Functors in C++ - Example

Listing 10: Example of a Functor

```cpp
#include <iostream>
// Functor to square an integer
struct Square {
    int operator()(int x) const { return x * x; }
};
int main() {
    // Using the functor
    Square squareFunctor;
    std::cout << "Square␣of␣6:␣" << squareFunctor(6) << std::endl;
         // Output: 36
    return 0;
}
```

Diamond Problem in Multiple Inheritance

- In C++, the **Diamond Problem** occurs when a class inherits from two classes that share a common base class. This creates ambiguity in situations where the derived class accesses members or methods from the shared base class through multiple paths.
  To address the Diamond Problem, two common solutions are:
  - **Virtual Inheritance:** Use the `virtual` keyword when inheriting from the common base class. This ensures a single instance of the shared base class in the hierarchy, resolving ambiguity.
  - **Operator Resolution:** Explicitly specify the path to the method or member using the scope resolution operator `::`. This resolves ambiguity by indicating which version of the method/ member to use.
- For more details and examples, you can explore the concept of multiple inheritance in C++ on GeeksforGeeks:
  Multiple Inheritance in C++.

C++ Standards: C++11, C++14, C++17, C++20

C++ evolves over time with the introduction of new standards, each bringing new features and improvements:

- **C++11:** Released in 2011, C++11 introduced features like auto keyword, lambda expressions, and smart pointers.
- **C++14:** Released in 2014, C++14 builds upon C++11 with enhancements but doesn't introduce major new features.
- **C++17:** Released in 2017, C++17 added features such as std::optional, std::variant, and parallel algorithms.
- **C++20:** Released in 2020, C++20 brought concepts, ranges, and improvements to coroutines and modules.

Explore the standards and their features on cppreference: C++11, C++14, C++17, C++20

Common C++ Keywords

Understanding C++ keywords is crucial for writing effective and readable code. Here are some commonly used keywords:

- **explicit:** Specifies that a constructor or conversion function should not be implicitly invoked.
- **const:** Indicates that a variable's value cannot be modified.
- **constexpr:** Specifies that a variable or function can be evaluated at compile-time.
- **static:** Specifies that a variable or function is shared among all instances of a class.
- **inline:** Suggests the compiler to insert the function code directly where it is called, potentially improving performance.

Learn more about these keywords and others on cppreference:
C++ Keywords