1. C++ Program Structure

Every C++ program has a basic structure that you need to follow.

Explanation

A C++ program starts with including header files, followed by a `main()` function. The execution of the program begins at the `main()` function.

Example

```cpp
// 1. Header file for input/output operations
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;

// 2. The main function where program execution begins
int main() {
    // 3. Your code goes here
    cout << "Hello, World!" << endl;

    // 4. Return 0 to indicate successful execution
    return 0;
}
```

Summary

- `#include <iostream>`: Includes the iostream library, which allows you to work with input and output streams (like printing to the console).
- `int main()`: The main function where your program starts.
- `cout`: The standard output stream, used to print text to the console.
- `return 0;`: Indicates that the program has executed successfully.

---

2. Header Files

Header files contain declarations for functions and variables that you can use in your program.

Explanation

You include header files at the beginning of your C++ file using the `#include` directive. This allows you to use pre-written code, saving you time and effort.

Example

```cpp
#include <iostream> // For input/output operations (cin, cout)
#include <string>   // For using the string data type
#include <cmath>    // For mathematical functions (like sqrt, pow)

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    string message = "Hello from a string!";
    cout << message << endl;
    cout << "The square root of 16 is: " << sqrt(16) << endl;
    return 0;
}
```

Summary

- Header files provide access to functions and classes from the C++ Standard Library or other libraries.
- Use `#include <filename>` to include standard library headers.
- Use `#include "filename"` to include your own custom header files.

---

3. Data Types

Data types define the type of data a variable can hold.

Explanation

When you declare a variable, you must specify its data type. This tells the compiler how much memory to allocate for the variable and what kind of data it can store.

Common Data Types

Data Type	Description	Example
`int`	Integers (whole numbers)	`int age = 25;`
`double`	Floating-point numbers	`double pi = 3.14159;`
`char`	Single characters	`char grade = 'A';`
`bool`	Boolean values (true or false)	`bool isStudent = true;`
`string`	Sequence of characters	`string name = "John";`

Example

```cpp
#include <iostream>
#include <string>

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    int age = 30;
    double price = 19.99;
    char initial = 'J';
    bool isActive = false;
    string city = "New York";

    cout << "Age: " << age << endl;
    cout << "Price: " << price << endl;
    cout << "Initial: " << initial << endl;
    cout << "Is Active: " << isActive << endl; // Outputs 0 for false
    cout << "City: " << city << endl;

    return 0;
}
```

Summary

- Data types are essential for declaring variables.
- Choose the data type that best fits the kind of data you need to store.

---

4. Variable Declaration & Initialization

A variable is a named storage location.

Explanation

- **Declaration**: This is where you tell the compiler the variable's name and data type.
- **Initialization**: This is where you assign an initial value to the variable.

You can declare and initialize a variable at the same time.

Example

```cpp
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;
```

```cpp
int main() {
    // Declaration
    int year;

    // Initialization
    year = 2025;

    // Declaration and Initialization in one step
    double temperature = 22.5;

    cout << "Year: " << year << endl;
    cout << "Temperature: " << temperature << endl;

    return 0;
}
```

Summary

- **Declaration**: `dataType variableName;`
- **Initialization**: `variableName = value;`
- It's good practice to initialize variables when you declare them to avoid using variables with unknown values.

---

5. C++ Operators

Operators are symbols that perform operations on variables and values.

Explanation

C++ has various types of operators:

- **Arithmetic Operators**: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus - remainder).
- **Assignment Operators**: `=` (assign), `+=`, `-=`, `*=`, `/=`.
- **Comparison Operators**: `==` (equal to), `!=` (not equal to), `>` (greater than), `<` (less than), `>=` (greater than or equal to), `<=` (less than or equal to).
- **Logical Operators**: `&&` (logical AND), `||` (logical OR), `!` (logical NOT).

Example

```cpp
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;
```

```cpp
int main() {
    int a = 10;
    int b = 4;

    // Arithmetic
    cout << "a + b = " << (a + b) << endl; // 14
    cout << "a % b = " << (a % b) << endl; // 2

    // Comparison
    cout << "a > b is " << (a > b) << endl; // 1 (true)

    // Logical
    bool isSunny = true;
    bool isWarm = true;
    if (isSunny && isWarm) {
        cout << "It's a nice day!" << endl;
    }

    return 0;
}
```

Summary

- Operators are the building blocks for performing calculations and making decisions in your code.
- Understand the difference between the assignment operator (=) and the comparison operator (==).

---

6. C++ Escape Sequences

Escape sequences are special characters used inside strings and character literals.

Explanation

They are preceded by a backslash (\) and are used to represent characters that are difficult or impossible to type directly, like a newline or a tab.

Common Escape Sequences

Sequence	Description
\n	Newline
\t	Horizontal Tab
\\	Backslash
\"	Double Quote
\'	Single Quote

Example

```cpp
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    // Using \n for a new line
    cout << "Hello\nWorld!" << endl;

    // Using \t for a tab
    cout << "Column 1\tColumn 2" << endl;

    // Using \" to print a double quote
    cout << "She said, \"Hello!\"" << endl;

    return 0;
}
```

Summary

- Escape sequences give you more control over how your text output is formatted.
- `\n` is one of the most common escape sequences.

---

7. Type Casting

Type casting is converting a variable from one data type to another.

Explanation

Sometimes you need to treat a variable as a different type. For example, you might need to convert an `int` to a `double` to perform a division with a fractional result.

Example

```cpp
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    int num1 = 10;
    int num2 = 4;

    // Without type casting (integer division)
```

```
    int result1 = num1 / num2;
    cout << "Integer division: " << result1 << endl; // Outputs 2

    // With type casting to double
    double result2 = (double)num1 / num2;
    cout << "Double division: " << result2 << endl; // Outputs 2.5

    return 0;
}
```

Summary

- Type casting allows you to perform operations that require a specific data type.
- Be careful when casting, as you can lose data (e.g., casting a `double` to an `int` truncates the decimal part).

---

8. Switch Statement

The `switch` statement is a control flow statement that allows you to execute different blocks of code based on the value of a variable or expression.

Explanation

It's often used as an alternative to a long chain of `if-else` `if-else` statements. The `switch` statement evaluates an expression, and then compares the result to the values in the `case` labels.

Example

```
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    int day = 4;

    switch (day) {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
```

```
            cout << "Wednesday";
            break;
        case 4:
            cout << "Thursday";
            break;
        case 5:
            cout << "Friday";
            break;
        default:
            cout << "Weekend";
            break;
    }
    cout << endl;

    return 0;
}
```

Summary

- Use a `switch` statement for multi-way branching.
- The `break` statement is crucial; without it, the program will "fall through" and execute the code in the next `case` as well.
- The `default` case is optional and runs if no other `case` matches.

---

9. Error Types

In programming, you'll encounter different types of errors.

Explanation

- **Compile-time Errors (Syntax Errors)**: These are errors found by the compiler before the program runs. They are usually caused by typos or incorrect C++ syntax. The program will not be created until these are fixed.
- **Runtime Errors**: These errors occur while the program is running. They can be caused by things like dividing by zero or trying to access a file that doesn't exist. These errors can crash your program.
- **Logical Errors**: These are the trickiest errors. The program runs without crashing, but it doesn't do what you intended. For example, using `+` instead of `-` in a calculation.

Example

```
#include <iostream>

// Using the standard namespace to make code shorter
```

```cpp
using namespace std;

int main() {
    // 1. Compile-time Error (Syntax Error)
    // The line below is missing a semicolon at the end.
    // cout << "Hello" << endl

    // 2. Runtime Error
    // The line below will cause a "divide by zero" error when the program runs.
    // int result = 10 / 0;

    // 3. Logical Error
    // We want to calculate the average of 5 and 10, but we get the wrong answer.
    // The correct formula is (5 + 10) / 2.
    double average = 5 + 10 / 2.0; // This calculates 5 + 5 = 10, not 7.5
    cout << "Logical Error Example: " << average << endl;

    return 0;
}
```

Summary

- **Compile-time errors** are the easiest to fix because the compiler tells you where they are.
- **Runtime errors** can be harder to find, but they usually point to a specific operation that failed.
- **Logical errors** are the most difficult because the program appears to work correctly. You need to carefully test your code to find them.

---

10. Debugging

Debugging is the process of finding and fixing errors (bugs) in your code.

Explanation

When your program doesn't work as expected, you need to debug it. There are several techniques for debugging.

Debugging Techniques

1. **Printing Values (`cout`)**: The simplest way to debug is to print the values of variables at different points in your program to see how they change. This can help you track down where things go wrong.
2. **Using a Debugger**: A debugger is a tool that lets you run your program step-by-step, inspect the values of variables, and see the flow of execution. This is a much more powerful way to debug than just printing values.

Example (Using `cout` for debugging)

Let's say we have a logical error in a loop.

```cpp
#include <iostream>

// Using the standard namespace to make code shorter
using namespace std;

int main() {
    int sum = 0;
    // We want to sum the numbers from 1 to 5.
    // The loop condition is wrong (i < 5 instead of i <= 5).
    for (int i = 1; i < 5; i++) {
        sum += i;
        // Use cout to see the value of 'sum' in each iteration
        cout << "Current i: " << i << ", Current sum: " << sum << endl;
    }

    // The expected sum is 1+2+3+4+5 = 15, but the output will be 10.
    cout << "Final Sum: " << sum << endl;

    return 0;
}
```

By printing the values inside the loop, we can see that the loop stops when `i` is 4, and we realize our condition should be `i <= 5`.

Summary

- Debugging is a normal part of programming.
- Start with simple techniques like printing values.
- Learn to use a debugger for more complex problems. It will save you a lot of time.