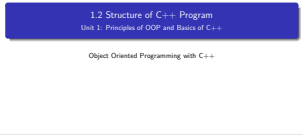
1.2 Structure of C++ Program

Unit 1: Principles of OOP and Basics of C++

Object Oriented Programming with C++



Lecture 2: Structure of a C++ Program

- Unit 1: Principles of OOP and Basics of C++
- Topic 1.2: Structure of C++ Program, Tokens, Keywords, Identifiers, Variables, Data Types, Reference Variables
- Course: DI05011021 - Object Oriented Programming with C++

Lecture 2: Structure of a C++ Program
- Unit 1: Principles of OOP and Basics of C++
- Topic 1.2: Structure of C++ Program, Tokens, Keywords, Identifiers, Variables, Data Types, Reference Variables
- Course: DI05011021 - Object Oriented Programming with C++

Welcome everyone to Lecture 2. Today, we dive into the actual syntax and structure of C++. If our last lecture was about the philosophy of OOP, today is about learning the alphabet and grammar of the C++ language. By the end of this session, you will be able to write and understand the foundational building blocks of any C++ application.

Recap & Learning Objectives

- Recap: Transitioned from Procedural to Object-Oriented Programming (OOP). Discussed OOP pillars: Encapsulation, Inheritance, Polymorphism.
- Objective 1: Identify structural components of a standard C++ program.
- Objective 2: Understand and classify C++ tokens (keywords, identifiers).
- Objective 3: Declare and utilize variables with appropriate primitive data types.
- Objective 4: Explain the concept of reference variables.

Recap & Learning Objectives
- Recap: Transitioned from Procedural to Object-Oriented Programming (OOP). Discussed OOP pillars: Encapsulation, Inheritance, Polymorphism.
- Objective 1: Identify structural components of a standard C++ program.
- Objective 2: Understand and classify C++ tokens (keywords, identifiers).
- Objective 3: Declare and utilize variables with appropriate primitive data types.
- Objective 4: Explain the concept of reference variables.

Let's briefly recall our last class where we contrasted Procedural Programming with OOP. Today, we are putting theory into practice by looking at C++ code. Our goals are simple: understand how a C++ file is structured, learn what tokens are, figure out how to store data using variables and data types, and finally, look at a very special C++ feature called reference variables.

Structure of a C++ Program

- A typical C++ program consists of several sections:
- 1. Documentation Section: Comments describing the program.
- 2. Preprocessor Directives (Link Section): Inclusion of header files (e.g., #include ¡iostream¿).
- 3. Global Declaration Section: Variables and functions used across the entire program.
- 4. The main() Function: The mandatory entry point of the execution.
- 5. Subprograms (Functions): User-defined functions called from main().

Structure of a C++ Program
- A typical C++ program consists of several sections:
- 1. Documentation Section: Comments describing the program.
- 2. Preprocessor Directives (Link Section): Inclusion of header files (e.g., #include ¡iostream¿).
- 3. Global Declaration Section: Variables and functions used across the entire program.
- 4. The main() Function: The mandatory entry point of the execution.
- 5. Subprograms (Functions): User-defined functions called from main().

Think of a C++ program as an essay. It has a specific format. You start with a header or title (Documentation), you gather your references (Preprocessor Directives), you define terms you'll use everywhere (Global Declarations), and then you have your main body paragraphs (the main function). Finally, you might have appendices (Subprograms). Not every section is mandatory except for the main() function, which is where the operating system starts executing your code.

A Simple C++ Program

- // A simple C++ program to print Hello World
- #include ¡iostream¿
- using namespace std;
-
- int main() {
- cout ¡¡ "Hello, World!" ¡¡ endl;
- return 0;
- }

A Simple C++ Program
- // A simple C++ program to print Hello World
- #include ¡iostream¿
- using namespace std;
-
- int main() {
- cout ¡¡ "Hello, World!" ¡¡ endl;
- return 0;
- }

Here is the classic 'Hello, World!' program. Even in these few lines, we see the structure we just talked about. We have a comment at the top for documentation. Then we have '#include ¡iostream¿' which tells the compiler to grab the input/output stream library. 'using namespace std;' allows us to use standard library names directly. Then we have our 'int main()' function, which outputs the text to the screen and returns 0 to indicate the program ended successfully.

Tokens in C++

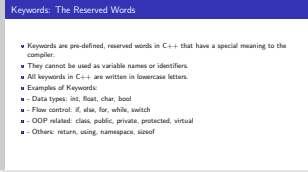
- Tokens are the smallest individual units in a C++ program.
- The compiler breaks the source code into these tokens.
- There are 5 main types of tokens in C++:
- 1. Keywords (e.g., int, return)
- 2. Identifiers (e.g., main, myVariable)
- 3. Literals / Constants (e.g., 10, "Hello")
- 4. Operators (e.g., +, -, =)
- 5. Punctuators / Separators (e.g., ;, {}, ())

Tokens in C++
- Tokens are the smallest individual units in a C++ program.
- The compiler breaks the source code into these tokens.
- There are 5 main types of tokens in C++:
- 1. Keywords (e.g., int, return)
- 2. Identifiers (e.g., main, myVariable)
- 3. Literals / Constants (e.g., 10, "Hello")
- 4. Operators (e.g., +, -, =)
- 5. Punctuators / Separators (e.g., ;, {}, ())

When the C++ compiler reads your code, it doesn't read it like a human reads a novel. It breaks the text down into the smallest meaningful chunks, called tokens. Just like a sentence is made of words, punctuation, and numbers, a C++ program is made of tokens. If you write 'int x = 5;', 'int' is a keyword, 'x' is an identifier, '=' is an operator, '5' is a literal, and the semicolon is a punctuator.

Keywords: The Reserved Words

- Keywords are pre-defined, reserved words in C++ that have a special meaning to the compiler.
- They cannot be used as variable names or identifiers.
- All keywords in C++ are written in lowercase letters.
- Examples of Keywords:
- - Data types: int, float, char, bool
- - Flow control: if, else, for, while, switch
- - OOP related: class, public, private, protected, virtual
- - Others: return, using, namespace, sizeof



Keywords are vocabulary reserved specifically for the C++ language itself. Because the compiler uses these words to understand the structure and actions of your program, you are forbidden from using them for your own variables. For instance, you cannot name a variable 'int' or 'return'. Remember, C++ is strictly case-sensitive, so 'Return' with a capital R is technically not a keyword, though it's bad practice to use it as a name.

Identifiers: Naming Your Entities

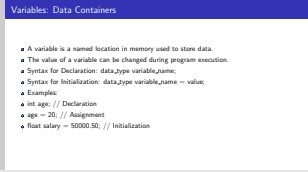
- Identifiers are user-defined names given to variables, functions, arrays, classes, etc.
- Rules for naming Identifiers:
- 1. Can consist of letters (A-Z, a-z), digits (0-9), and underscores (_).
- 2. MUST begin with a letter or an underscore, NOT a digit.
- 3. Cannot be a keyword.
- 4. C++ is case-sensitive (e.g., 'Age' and 'age' are different).
- Examples of Valid: studentName, _count, sum20
- Examples of Invalid: 1stNumber, int, my-var

Identifiers: Naming Your Entities
- Identifiers are user-defined names given to variables, functions, arrays, classes, etc.
- Rules for naming Identifiers:
- 1. Can consist of letters (A-Z, a-z), digits (0-9), and underscores (_).
- 2. MUST begin with a letter or an underscore, NOT a digit.
- 3. Cannot be a keyword.
- 4. C++ is case-sensitive (e.g., 'Age' and 'age' are different).
- Examples of Valid: studentName, _count, sum20
- Examples of Invalid: 1stNumber, int, my-var

Identifiers are the names YOU give to things in your code. While keywords belong to C++, identifiers belong to you. However, you must follow the rules. You can use letters, numbers, and underscores, but you can never start with a number. 'student1' is fine, '1student' is a syntax error. Also, avoid using special characters like hyphens or at-symbols. Because C++ is case-sensitive, 'Total', 'total', and 'TOTAL' would be considered three completely different identifiers.

Variables: Data Containers

- A variable is a named location in memory used to store data.
- The value of a variable can be changed during program execution.
- Syntax for Declaration: data_type variable_name;
- Syntax for Initialization: data_type variable_name = value;
- Examples:
- int age; // Declaration
- age = 20; // Assignment
- float salary = 50000.50; // Initialization



Think of a variable as a labeled box in your computer's RAM. The 'data type' dictates what size the box is and what kind of items can be put inside. The 'identifier' is the label on the box. 'Declaration' is when you tell the compiler to create the box. 'Initialization' is when you put the first item into that box. You can change the contents of the box later, which is why it's called a 'variable'.

Data Types Overview

- Data types specify the size and type of values that can be stored in a variable.
- C++ Data Types are divided into three categories:
- 1. Built-in / Primitive Types: int, char, float, double, bool, void.
- 2. Derived Types: array, pointer, function, reference.
- 3. User-defined Types: struct, union, enum, class.

Data Types Overview
- Data types specify the size and type of values that can be stored in a variable.
- C++ Data Types are divided into three categories:
- 1. Built-in / Primitive Types: int, char, float, double, bool, void.
- 2. Derived Types: array, pointer, function, reference.
- 3. User-defined Types: struct, union, enum, class.

C++ is a strongly typed language, which means before you can use a variable, you must explicitly declare its data type. This helps the compiler allocate the correct amount of memory and prevents you from accidentally trying to multiply a word by a number. Today, we are focusing on the built-in primitive types, which form the foundation of all other data types in C++.

Primitive Data Types: Integers

- Integer types store whole numbers without decimal points.
- Type: int (usually 4 bytes, range approx. -2 billion to +2 billion)
- Modifiers can alter the size or sign of integers:
- - short int (usually 2 bytes)
- - long int (usually 4 or 8 bytes)
- - unsigned int (only positive numbers, doubles the positive range)
- - signed int (default, includes negative numbers)
- Example: unsigned int distance = 4000000000;

Primitive Data Types: Integers
- Integer types store whole numbers without decimal points.
- Type: int (usually 4 bytes, range approx. -2 billion to +2 billion)
- Modifiers can alter the size or sign of integers:
- - short int (usually 2 bytes)
- - long int (usually 4 or 8 bytes)
- - unsigned int (only positive numbers, doubles the positive range)
- - signed int (default, includes negative numbers)
- Example: unsigned int distance = 4000000000;

When you need to count things, use integers. The standard 'int' usually takes 4 bytes of memory, which is enough to store numbers up to about 2.1 billion. If you are dealing with very small numbers, you might use 'short' to save memory. If you know a value can never be negative—like a person's age or a distance—you can use the 'unsigned' modifier, which shifts the entire range to the positive side, allowing you to store much larger positive numbers in the same amount of memory.

Primitive Data Types: Floating-Point

- Floating-point types store real numbers with decimal points.
- 1. float: Single precision, typically 4 bytes.
- - Precision: ˜7 decimal digits.
- 2. double: Double precision, typically 8 bytes.
- - Precision: ˜15 decimal digits.
- 3. long double: Extended precision, typically 12 or 16 bytes.
- Example:
- float pi = 3.14f; // 'f' suffix indicates float literal
- double exactPi = 3.141592653589793;

Primitive Data Types: Floating-Point
- Floating-point types store real numbers with decimal points.
- 1. float: Single precision, typically 4 bytes.
- - Precision: ˜7 decimal digits.
- 2. double: Double precision, typically 8 bytes.
- - Precision: ˜15 decimal digits.
- 3. long double: Extended precision, typically 12 or 16 bytes.
- Example:
- float pi = 3.14f; // 'f' suffix indicates float literal
- double exactPi = 3.141592653589793;

When your math requires fractions or decimals, you use floating-point types. 'float' provides standard precision and is good for general use, like graphics programming where memory speed is key. 'double' provides double the precision and is standard for scientific calculations or financial applications where accuracy is critical. Note that in C++, decimal literals are treated as 'double' by default unless you append an 'f' at the end.

Primitive Data Types: Character and Boolean

- Character (char):
- - Typically 1 byte.
- - Stores a single character enclosed in single quotes ('A', 'z', '5', '?').
- - Internally stored as an integer based on its ASCII value.
- Boolean (bool):
- - Typically 1 byte.
- - Can only hold one of two values: true (1) or false (0).
- - Used extensively in conditional logic and loops.
- Example: char grade = 'A'; bool passed = true;

Primitive Data Types: Character and Boolean
- Character (char):
- - Typically 1 byte.
- - Stores a single character enclosed in single quotes ('A', 'z', '5', '?').
- - Internally stored as an integer based on its ASCII value.
- Boolean (bool):
- - Typically 1 byte.
- - Can only hold one of two values: true (1) or false (0).
- - Used extensively in conditional logic and loops.
- Example: char grade = 'A'; bool passed = true;

Characters in C++ are stored using the 'char' type. You must use single quotes for characters; double quotes are for strings. Under the hood, a 'char' is actually just a small integer representing an ASCII code—for example, capital 'A' is 65. The 'bool' type, short for Boolean, is the simplest data type. It only represents truth: true or false. Booleans are the backbone of decision-making in your code, like 'if' statements.

Introduction to Reference Variables

- A reference variable is an alias, or an alternate name, for an existing variable.
- Unlike standard variables, references DO NOT occupy their own independent memory.
- They share the exact same memory address as the original variable.
- Syntax: data_type &ref_name = original_variable;
- Rules:
- 1. A reference must be initialized at the time of declaration.
- 2. Once initialized, it cannot be re-assigned to refer to a different variable.

Introduction to Reference Variables
- A reference variable is an alias, or an alternate name, for an existing variable.
- Unlike standard variables, references DO NOT occupy their own independent memory.
- They share the exact same memory address as the original variable.
- Syntax: data_type &ref_name = original_variable;
- Rules:
- 1. A reference must be initialized at the time of declaration.
- 2. Once initialized, it cannot be re-assigned to refer to a different variable.

Now we move to a feature unique to C++ (and not found in C): reference variables. Think of a reference as a nickname for a person. If 'Robert' is the original variable, 'Bob' could be the reference. They are the same person; they occupy the same space in the room. If Bob gets a haircut, Robert has a haircut. In code, this means any change made to the reference directly modifies the original variable.

Reference Variables in Action

- int score = 100;
- int &refScore = score; // refScore is an alias for score
-
- cout ¡¡ "score: " ¡¡ score ¡¡ endl; // Output: 100
- cout ¡¡ "refScore: " ¡¡ refScore ¡¡ endl; // Output: 100
-
- refScore = 150; // Modifying through the reference
-
- cout ¡¡ "score: " ¡¡ score ¡¡ endl; // Output: 150
- cout ¡¡ "refScore: " ¡¡ refScore ¡¡ endl; // Output: 150

Reference Variables in Action
int score = 100;
int &refScore = score; // refScore is an alias for score
cout ¡¡ "score: " ¡¡ score ¡¡ endl; // Output: 100
cout ¡¡ "refScore: " ¡¡ refScore ¡¡ endl; // Output: 100
refScore = 150; // Modifying through the reference
cout ¡¡ "score: " ¡¡ score ¡¡ endl; // Output: 150
cout ¡¡ "refScore: " ¡¡ refScore ¡¡ endl; // Output: 150

Let's look at this code. We create a variable 'score' and set it to 100. Then we create a reference variable '&refScore' and assign 'score' to it. When we print both, they are 100. Then, we change 'refScore' to 150. We didn't explicitly touch 'score', but when we print 'score' again, it is also 150. This proves they are two names for the exact same memory location.

Summary

- C++ programs have a structured layout, always requiring a main() function.
- Tokens (keywords, identifiers, literals, operators, punctuators) are the fundamental vocabulary.
- Variables must be declared with a specific data type before use.
- Primitive types include int, float, double, char, and bool, each with specific memory sizes.
- Reference variables provide a secure, non-reassignable alias to existing memory locations.

Summary
- C++ programs have a structured layout, always requiring a main() function.
- Tokens (keywords, identifiers, literals, operators, punctuators) are the fundamental vocabulary.
- Variables must be declared with a specific data type before use.
- Primitive types include int, float, double, char, and bool, each with specific memory sizes.
- Reference variables provide a secure, non-reassignable alias to existing memory locations.

To wrap up today's lecture, we've laid the groundwork for writing C++ code. You now know the basic structure of a program, the rules for naming things via identifiers, the reserved words or keywords you can't use for names, and how to allocate memory using data types and variables. We also introduced reference variables, a powerful tool you will use extensively when we start writing complex functions and classes.

Q&A and Next Lecture Preview

- Questions?
- Up Next: Lecture 1.3 - Operators and Expressions
- - Arithmetic, Relational, and Logical Operators
- - Bitwise and Assignment Operators
- - Operator Precedence and Associativity
- - Type Casting (Implicit and Explicit)

Q&A and Next Lecture Preview
- Questions?
- Up Next: Lecture 1.3 - Operators and Expressions
- - Arithmetic, Relational, and Logical Operators
- - Bitwise and Assignment Operators
- - Operator Precedence and Associativity
- - Type Casting (Implicit and Explicit)

We've covered a lot of syntax today. Does anyone have any questions about data types, variable initialization, or how reference variables work? ... If there are no questions, please read ahead in the textbook on Operators. In the next lecture, we will learn how to manipulate the variables we learned to create today using mathematical and logical expressions. Thank you, and see you next time!