Error Recovery in Operator Precedence Parsing

- **Error Types and Diagnostics**: The parser can detect several types of errors:
  - **Errors Detected During Reduction**:
    - Missing operand.
    - Missing operator.
    - Absence of expression within parentheses.
  - **Errors Detected During Shift/Reduce Actions**:
    - Missing operand.
    - Unbalanced or missing right parenthesis.
    - Missing operators.

Error Recovery in Operator Precedence Parsing

- **Detection Points in Operator Precedence Parsing:**
  - Errors are identifiable at two key stages:
    - When there's no precedence relation between the terminal at the top of the stack and the current input symbol.
    - When a handle is identified, but no corresponding production exists.

Global Correction in Parsing

- **Nature of Global Correction**:
  - Primarily a theoretical concept in the realm of parsing.

- **Implications on Parsing Process**:
  - Utilizing global correction significantly increases both the time and space resources required during the parsing stage.

Error Production

- **Overview of Error Production**:
  - This technique allows a parser to generate relevant error messages while continuing the parsing process.
- **Functionality**:
  - When the parser encounters an incorrect production, it issues an error message and then resumes parsing.
  - E → + E | - E | * A | / A
  - A → E
  - If the parser comes across the production '* A' and deems it incorrect, it can alert the user with a message, perhaps querying whether '*' is intended as a unary operator, before continuing with the parsing.

Phrase-Level Recovery

- This method involves making localized corrections to the input when an error is detected by the parser.
- It involves substituting a part of the input with a string that allows the parser to proceed.
- **Correction Mechanism**:
  - Common corrections include replacing a comma with a semicolon, removing an unnecessary semicolon, or inserting a missing one.
  - while(x>0)
    - y=a+b;
  - Phrase-level recovery might correct this by adding 'do' to form while(x>0) do y=a+b;, enabling the parsing process to continue smoothly.

Basics of Panic Mode Recovery

- A straightforward and commonly used method in various parsing techniques.
- Upon detecting an error, the parser discards input symbols until it encounters a synchronizing token from a predefined set.

- **Characteristics**:
  - Panic mode may bypass large sections of input without further error checking.
  - This approach ensures that the parser does not enter an infinite loop.
  - a = b + c;
  - d = e + f;
  - In panic mode, the parser might skip over the entire line a = b + c; without identifying specific errors in it.

Logical Errors(Run Time Error) in Programming

- **Nature of Logical Errors**:
  - Logical errors are mistakes in a program's logic that the compiler does not catch.
  - These errors occur in programs that are syntactically correct but do not function as intended.
- **Example of a Logical Error**:
  - Consider the code snippet:

```
x = 4;
y = 5;
average = x + y / 2;
```

Semantic Phase Errors

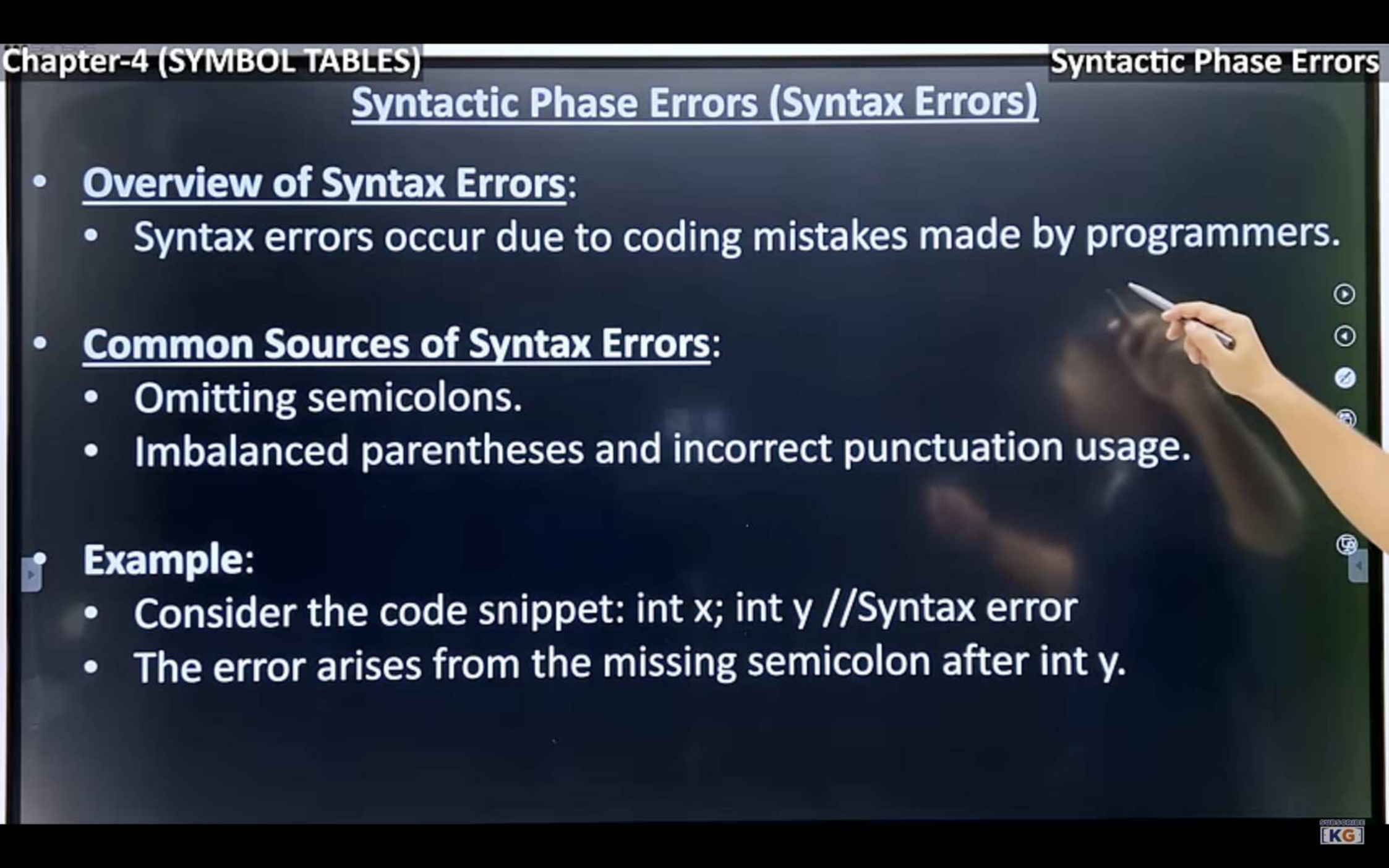
- **Definition of Semantic Errors**:
  - Semantic errors relate to incorrect use of program statements, impacting the program's meaning or logic.
- **Typical Reasons for Semantic Errors**:
  - Using undeclared names.
  - Type mismatches.
  - Inconsistencies between actual and formal arguments in function calls.
- **Example**:
  - In the code scanf("%f%f", a, b);, the error lies in not using the addresses of variables a and b. The correct usage should be scanf("%f%f", &a, &b); to provide the address locations.

Syntactic Phase Errors (Syntax Errors)

- **Overview of Syntax Errors**:
  - Syntax errors occur due to coding mistakes made by programmers.

- **Common Sources of Syntax Errors**:
  - Omitting semicolons.
  - Imbalanced parentheses and incorrect punctuation usage.

- **Example**:
  - Consider the code snippet: int x; int y //Syntax error
  - The error arises from the missing semicolon after int y.

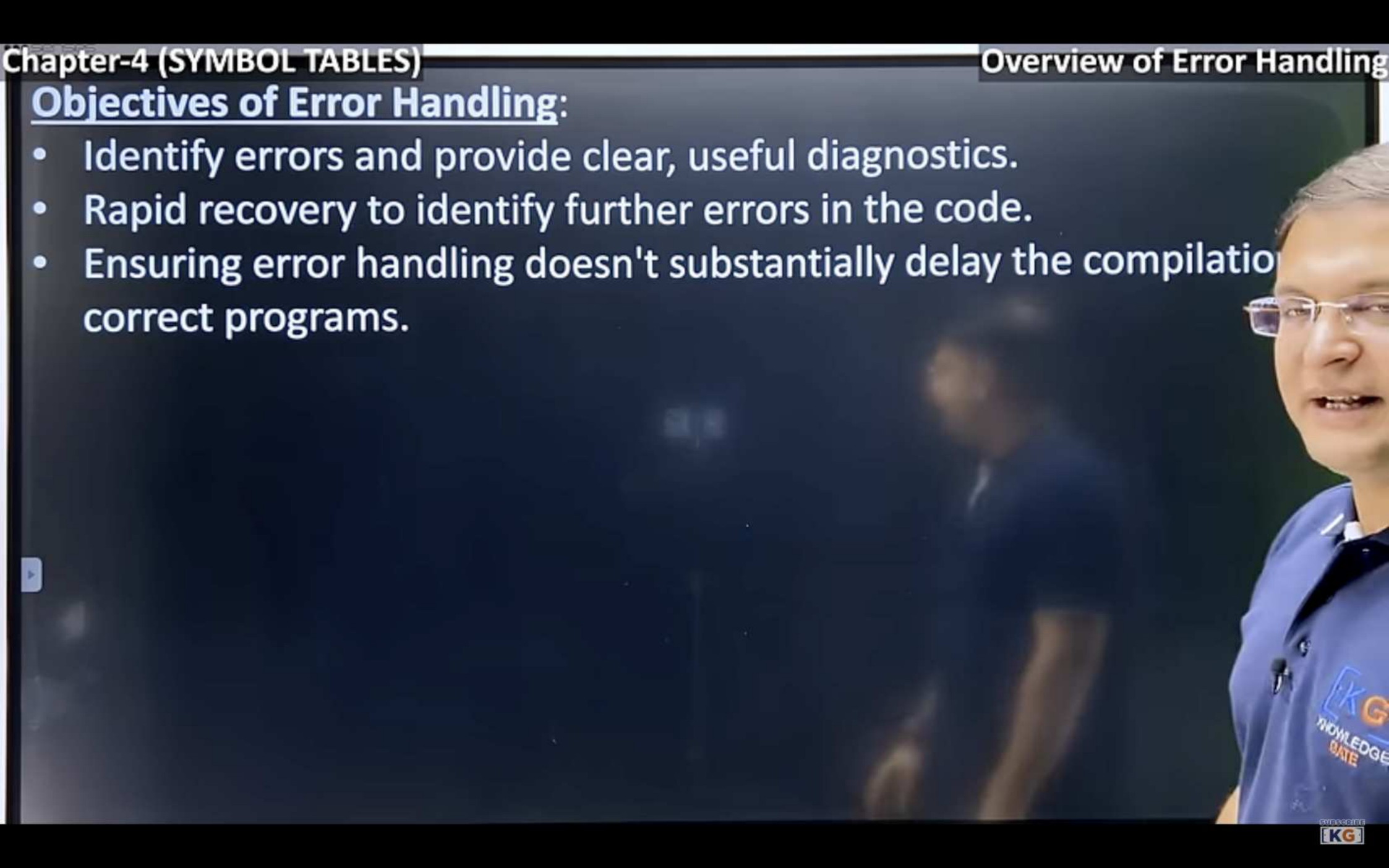


Lexical Phase Errors in Compiling

- **Definition of Lexical Phase Error**:
  - Occurs when a sequence of characters doesn't form a recognizable token during the source program scanning, preventing valid token generation.
- **Common Causes of Lexical Errors**:
  - Adding an unnecessary character.
  - Omitting a required character.
  - Substituting a character incorrectly.
  - Swapping two characters.
- **Examples of Lexical Errors**:
  - In Fortran, identifiers exceeding 7 characters are considered lexical errors.
  - The presence of characters like ~, &, and @ in a Pascal program constitutes a lexical error.

Objectives of Error Handling:

- Identify errors and provide clear, useful diagnostics.
- Rapid recovery to identify further errors in the code.
- Ensuring error handling doesn't substantially delay the compilatio
  correct programs.



Overview of Error Recovery in Compilers

Error Recovery Significance:

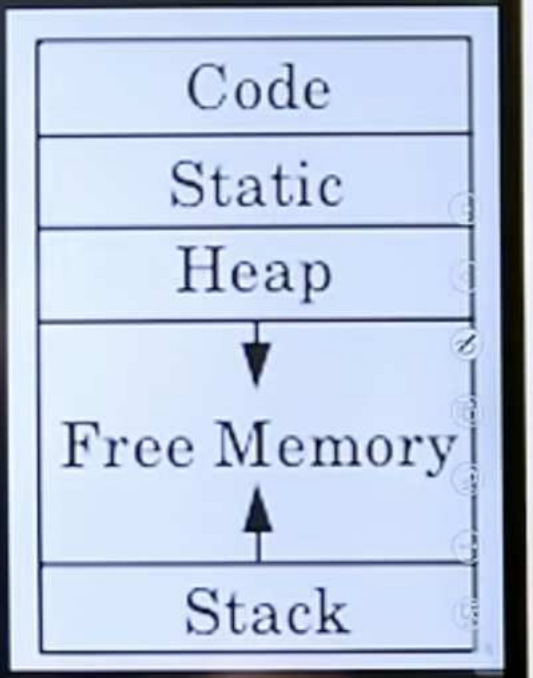
- Essential for compilers to process and execute programs even with errors.

Error Message Characteristics:

- Messages should relate to the original source code, not its internal representation.
- Simplicity is key in error messaging.
- Precision in pinpointing and fixing errors is crucial.
- Avoid repetition of the same error message.

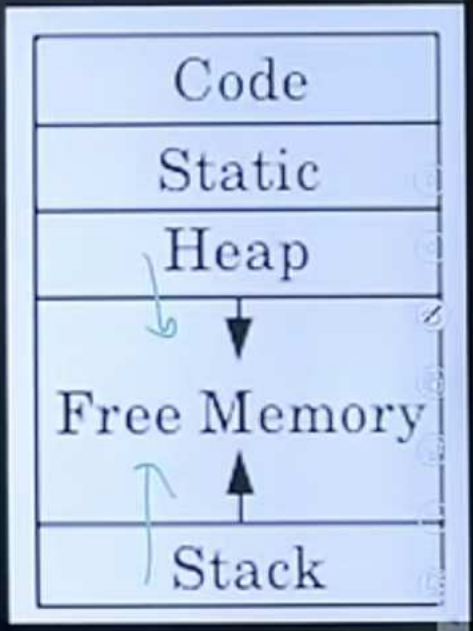
Overview of Memory Allocation Methods in Compilation

- **Stack Allocation:**
  - Manages data structures known as activation records.
  - Activation records are pushed onto the stack at call time and popped when the call ends.
  - Local variables for each procedure call are stored in the respective activation record, ensuring fresh storage for each call.
  - Local values are removed when the procedure call completes.



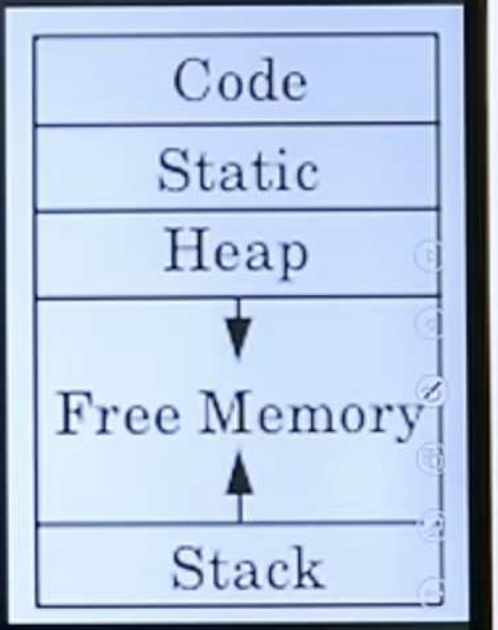
Overview of Memory Allocation Methods in Compilation

- **Heap Allocation Methods**:
  - **Garbage Collection**:
    - Handles objects that persist after losing all access paths.
    - Reclaims object space for reuse.
    - Garbage objects are identified by a 'garbage collection bit' and returned to free space.
  - **Reference Counting**:
    - Reclaims heap storage elements as soon as they become inaccessible.
    - Each heap cell has a counter tracking the number of references to it.
    - The counter is adjusted as references are made or removed.



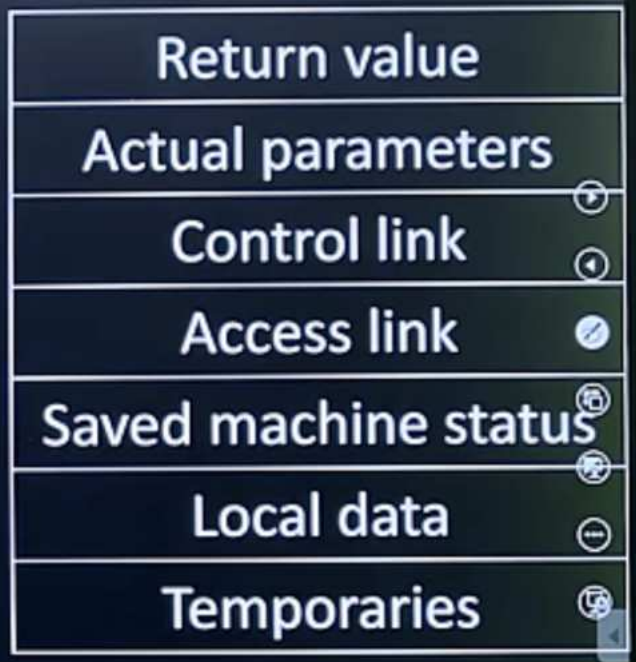
Overview of Memory Allocation Methods in Compilation

- **Code Storage**:
  - Contains fixed-size, unchanging executable target code during compilation.
- **Static Allocation**:
  - Allocates storage for all data objects at compile time.
  - Object sizes are known at compile time.
  - Object names are bound to storage locations during compilation.
  - The compiler calculates the required storage for each object, simplifying address determination in the activation record.
  - Compiler sets up addresses for the target code to access data at compile time.



Activation record fields include:

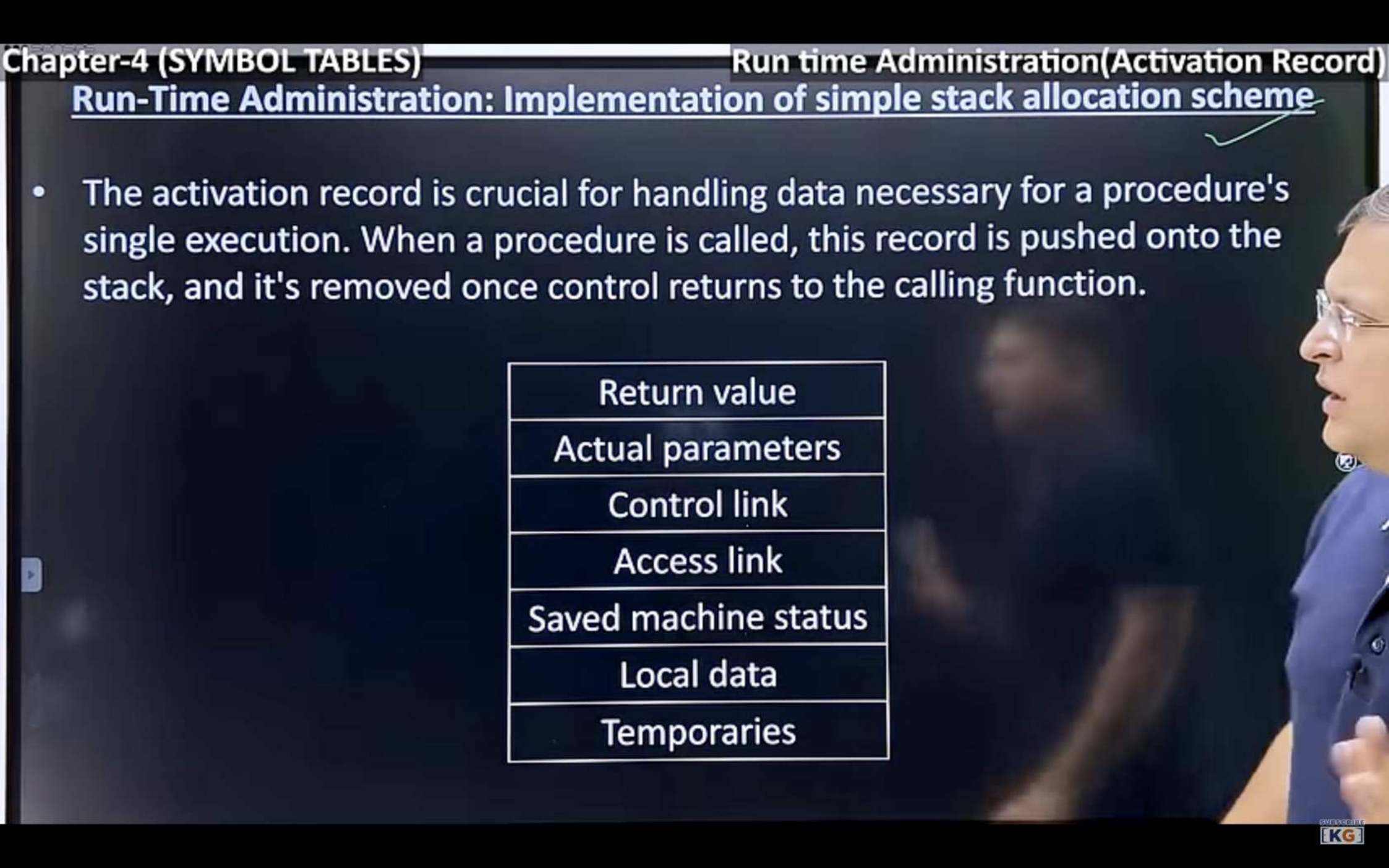
- **Return Value**: Allows the called procedure to return a value to the caller.
- **Actual Parameters**: Used by the caller to provide parameters to the called procedure.
- **Control Link**: Connects to the caller's activation record.
- **Access Link**: References non-local data in other activation records.
- **Saved Machine Status**: Preserves machine state prior to the procedure call.
- **Local Data**: Contains data specific to the procedure's execution.
- **Temporaries**: Stores interim values during expression evaluation.



Run-Time Administration: Implementation of simple stack allocation scheme

- The activation record is crucial for handling data necessary for a procedure's single execution. When a procedure is called, this record is pushed onto the stack, and it's removed once control returns to the calling function.

Return value
Actual parameters
Control link
Access link
Saved machine status
Local data
Temporaries



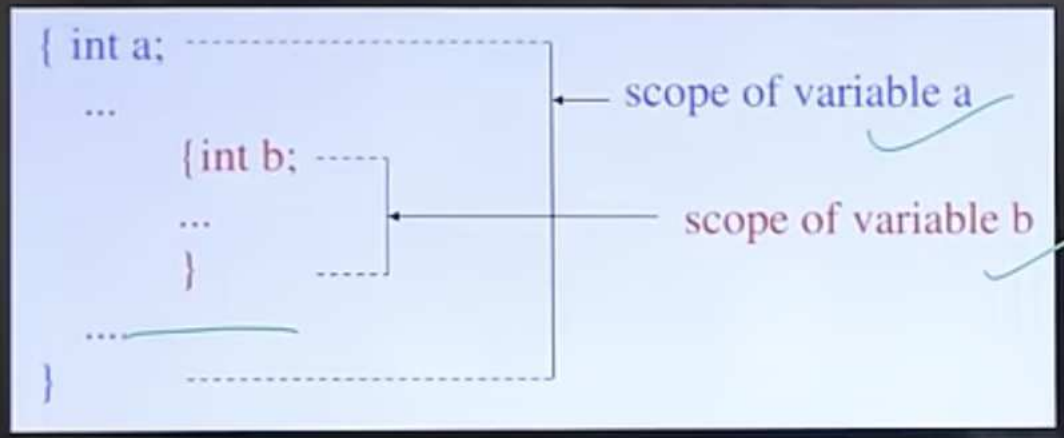
Aspect	Lexical Scoping	Dynamic Scoping
Determination Time	At compile time.	At runtime.
Scope Basis	Location in source code.	Calling sequence of functions.
Predictability	High; scope is clear from code structure.	Low; depends on program's execution path.
Variable Accessibility	Determined by code blocks and functions.	Influenced by function call order.
Common Usage	Preferred in most modern languages.	Less common; found in older languages.

Scoping in Programming Languages

- Scoping refers to the rules that govern the visibility and lifespan of variables and functions within different parts of a program. It defines the context in which an identifier, such as a variable or function name, is valid and accessible.

Representing Scope Information

- Scope of variables in statement blocks



- Scope of formal arguments of functions

```
int factorial(int n) {
    ...
}
```

← scope of argument n

Representing Scope Information

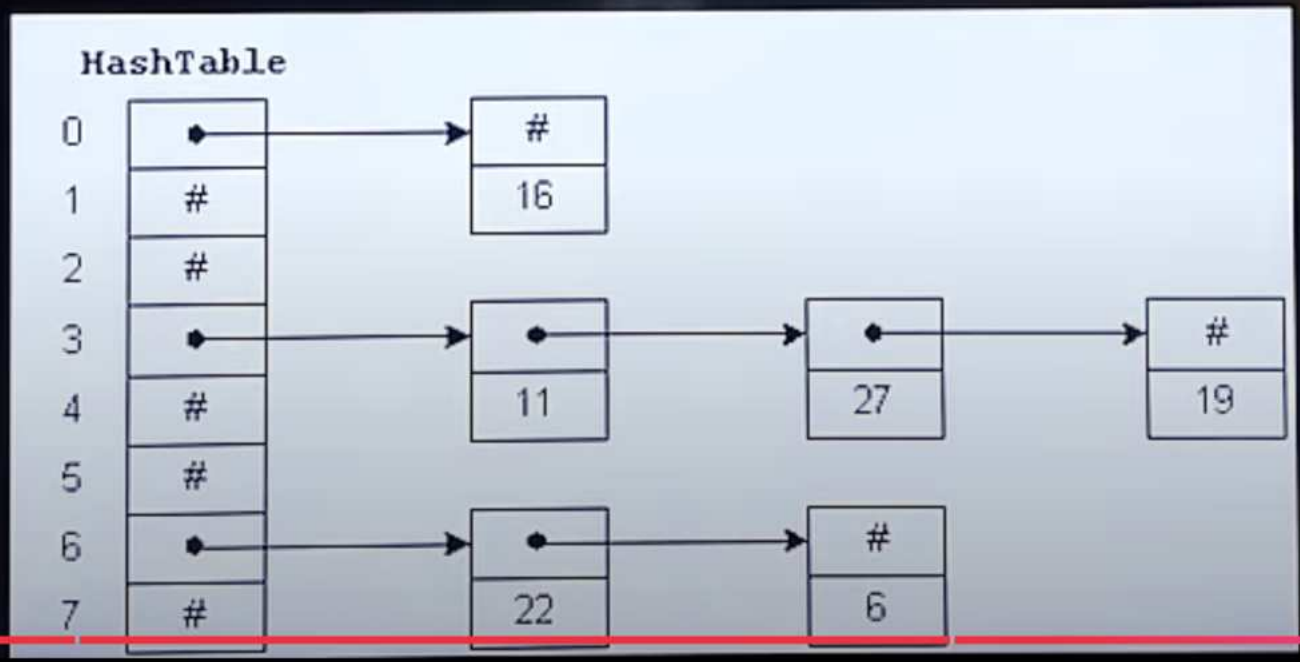
- **Scope of Identifiers**: Scope defines where in a program an identifier is valid and accessible.
- **Language-Dependent Scope**: Different programming languages have varying scopes. For instance, in FORTRAN, a name's scope is limited to a single subroutine, while in ALGOL, it's confined to the section or procedure where it's declared.
- **Multiple Declarations**: An identifier can be declared multiple times in different scopes as distinct entities, each with unique attributes and storage locations.
- **Symbol Table's Role**: The symbol table maintains the uniqueness of each identifier's declarations.
- **Unique Identification**: Each program element is assigned a unique number, which helps in differentiating local data in different scopes.
- **Scope Determination**: Semantic rules from the program's structure are utilized to identify the active scope, especially in subprograms.
- **Scope-Related Semantic Rules**: a. An identifier's usage is confined to its defined scope. b. Identifiers with the same name and type cannot coexist within the same lexical scope.

Symbol table and its entries

- **Variables**:
  - Identifiers with changeable values during and between program executions.
  - Represent memory location contents.
  - Symbol table tracks their names and runtime storage allocation.
- **Constants**:
  - Identifiers for immutable values.
  - Runtime storage allocation is unnecessary.
  - Compiler embeds them directly into the code during compilation.
- **User-Defined Types**:
  - Combinations of existing types defined by the user.
  - Accessed by name, referring to a specific type structure.
- **Classes**:
  - Abstract data types offering controlled access and language-level polymorphism.
  - Include information about constructors, destructors, and virtual function tables.
- **Records**:
  - Collections of named, possibly heterogeneous members.
  - Symbol table likely records each member within a record.

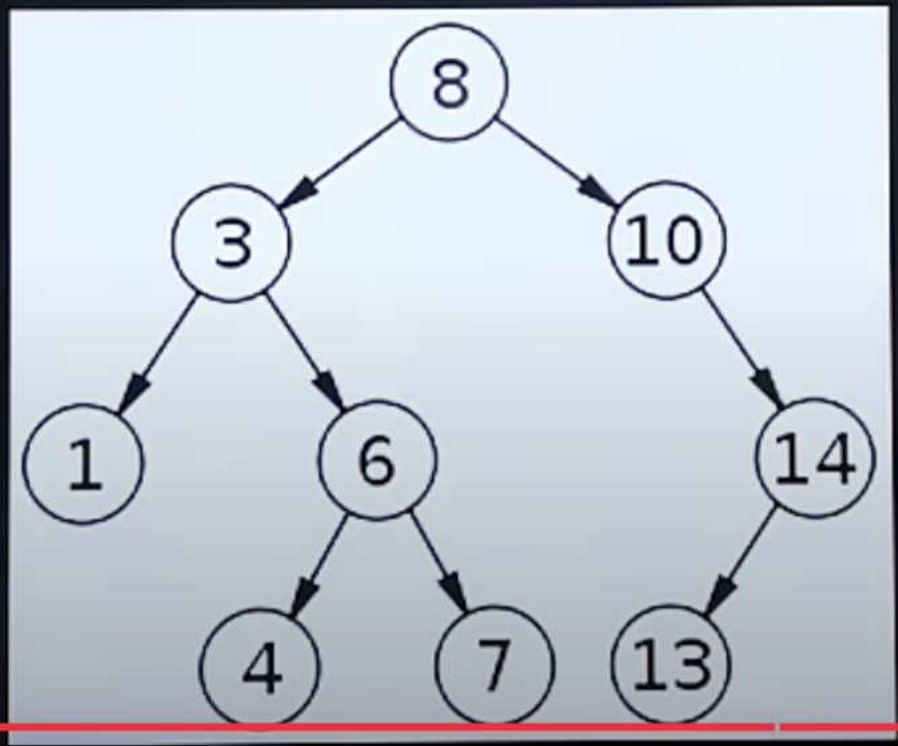
Different data structures used in implementing a symbol table

- **Hash Tables and Hash Functions:**
  - Hash tables map elements to a fixed range of values (hash values) using hash functions.
  - They minimize element movement within the symbol table.
  - Hash functions ensure a uniform distribution of names.



Different data structures used in implementing a symbol table

- **Search Tree:**
  - Offers logarithmic time for operations and lookup.
  - Balancing is achieved through AVL or Red-black tree algorithms.



Different data structures used in implementing a symbol table

- **Ordered List:**
  - Binary search on a sorted array enables faster search $O(\log$
  - Insertion is slower $O(n)$ due to sorting.
  - Beneficial for fixed sets of names, like reserved words.

Variable Name	Data Type
age	int
height	float
name	string
score	double
width	float

Different data structures used in implementing a symbol table

- **Unordered List:**
  - Easy to implement using arrays or linked lists.
  - Linked lists allow dynamic growth, avoiding fixed size constraints.
  - Quick insertion $O(1)$ time, but slower lookup $O(n)$ for larger tables.

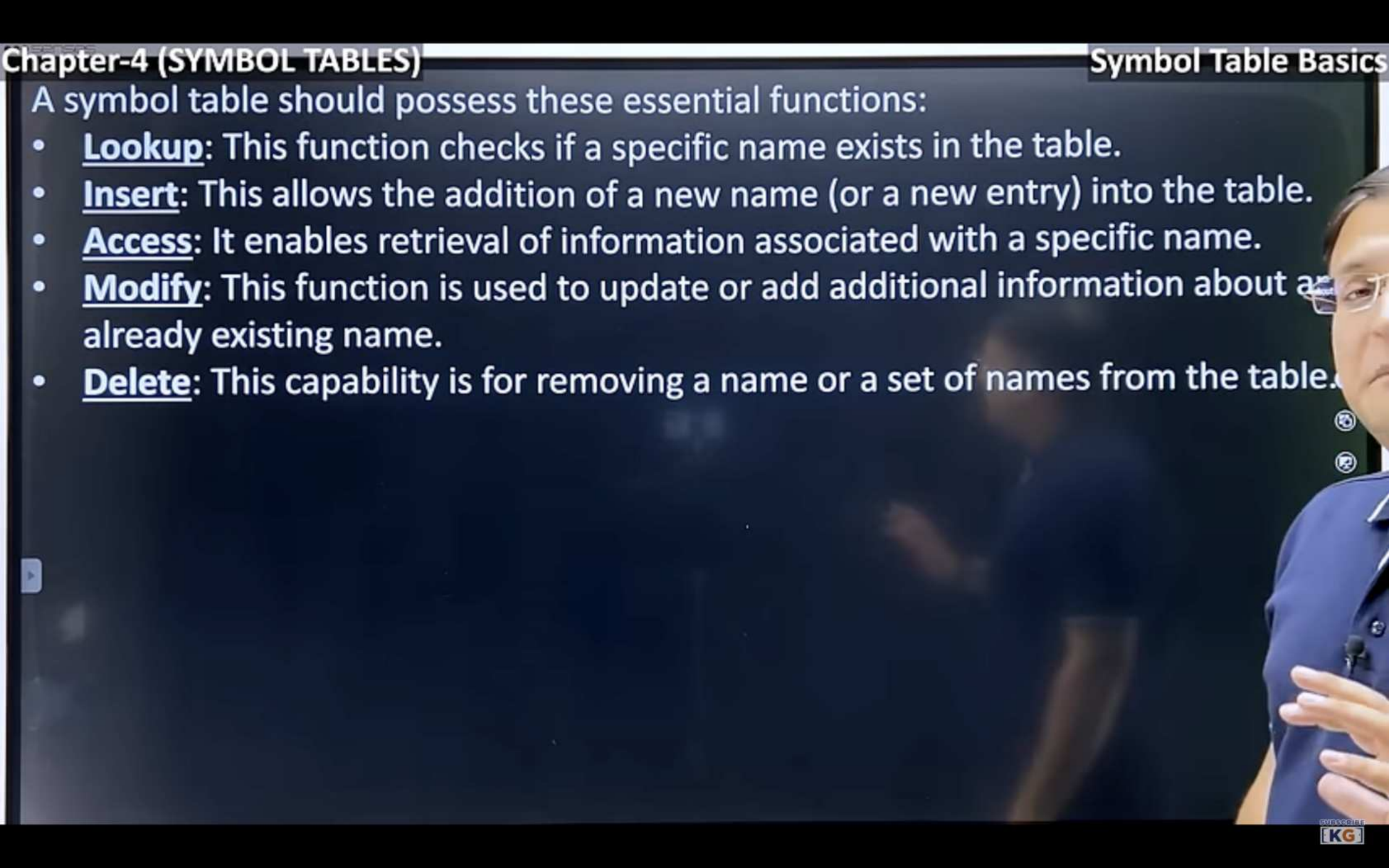
Student Name	Age
John	20
Maria	19
Alex	21
Sarah	18
Brian	22

Important symbol table requirements

- **Adaptive Structure**: Entries must be comprehensive, reflecting each identifier's specific use.
- **Quick Lookup/Search**: High-speed search functionality is essential, dependent on the symbol table's design.
- **Space-Efficient**: The table should dynamically adjust in size for optimal space use.
- **Language Feature Accommodation**: It needs to support language-specific aspects like scoping and implicit declarations.

A symbol table should possess these essential functions:

- **Lookup**: This function checks if a specific name exists in the table.
- **Insert**: This allows the addition of a new name (or a new entry) into the table.
- **Access**: It enables retrieval of information associated with a specific name.
- **Modify**: This function is used to update or add additional information about a already existing name.
- **Delete**: This capability is is for removing a name or a set of names from the table.



- **Information used by compiler from symbol table**
  - Data type and name.
  - Declaring procedure.
  - Pointer to structure table or record.
  - Parameter passing by value or by reference.
  - No and types of argument passed to function.
  - Base address.

Chapter-4 (SYMBOL TABLES)

Symbol Table Basics

- The information is collected by the analysis phases of the compiler and used by the synthesis phases of the compiler.
  - Lexical Analysis:- Creates new table entries
  - Syntax Analysis:- add information about attributes types, scope and use in the table.
  - Semantic Analysis:- to check expression are semantically correct and type checking.
  - Intermediate Code Generation:-symbol table helps in adding temporary variable information in code.
  - Code Optimization:- use symbol table in machine dependent optimization.
  - Code Generation:- uses address information of identifier present in the table for code generation.

Symbol Table Basics

- **Definition**:- The symbol table is a data structure used by a compiler to store information about the source program's variables, functions, constants, user-defined types, and other identifiers.
- **Need**:- It helps the compiler track the scope, life, and attributes (like type, size, value) of each identifier. It is essential for semantic analysis, type checking, and code generation.