PRIYANKA P (ID:24007)
SOURCE: THE C PROGRAMMING LANGUAGE
AUTHOR: DENNIM.M.RITCHIE & BRIAN.W.KERNIGHAN
DATE: 19.2.25

CHAPTER 4:

Problem:
1. return the line with a specific pattern “ould”
2. atof() - if a fn is not explicitly declared with its return type then, it is implicitly defined as int as it’s return type.
3. Reverse polish calculator 5 4 + =(5+4)

Storage classes:

Auto:
Inside a fn, or fn parameters or local variables
Not access outside its scope.

Extern:
extern int count;
Only one definition of an external variable should exist in the entire program.
Multiple declarations can exist in different files to allow access.
It can share variables across multiple files and functions

Scope Rules:
The scope of an identifier refers to where in the program var/fn can be accessed.
Definition: Allocates memory for the variable and optionally initializes it.
Declaration: Informs the compiler about the existence and type of a variable but does not allocate memory.

Header files:
declaration of variables and function prototype is present.
Uses functions from other files via #include "calc.h".

Why Split into Multiple Files?
Modularity – Makes code more manageable and easier to debug.
Code Reusability – Functions can be reused in other programs.
Faster Compilation – Only modified files need to be recompiled.
Better Organization – Different responsibilities are separated into different files.

Eg:-
```
// calc.h
#ifndef CALC_H
#define CALC_H
```

```
#define NUMBER '0'  // Signal that a number was found
#define MAXVAL 100  // Maximum depth of val stack

// External variable declarations
extern int sp;
extern double val[];

// Function prototypes
void push(double);
double pop(void);
int getop(char[]);
int getch(void);
void ungetch(int);

#endif  // CALC_H
```

Why STATIC ?
In this, sp is declared as extern, in order to restrict the variable to be accessed within its file we have declared that as a static variable in its source file. So that access is restricted for that var outside the file while handling multiple files.

It can be accessed only within its scope.
It is initialized only once.
It can be both global and local.

Never use static for global variables in a header file because each .c file that includes it will get its own separate copy.
Use static inside .c file to restrict scope to that file.

Register:
The register keyword is used to inform the compiler that a variable will be used frequently, suggesting that it be stored in a CPU register rather than in RAM memory. This can improve execution speed, as accessing registers is much faster than accessing RAM.
compilers are free to ignore this request, and modern optimizing compilers often determine the best usage of registers automatically.
Cannot take address (&x) and only a few variables can be in registers.
Can only be used for local variables and function parameters.

Block structure:
C is not a block-structured language like Pascal
because functions cannot be defined inside other functions.
However, variables can be declared in a block-structured manner within functions.
Use meaningful variable names to reduce conflicts.
Avoid using the same variable name in multiple scopes, especially with different types.
Use function parameters wisely to avoid hiding global variables unnecessarily.

Initialization:
Global and static variables default to 0,
automatic and register variables contain garbage values.
Automatic variables can be initialized with expressions;
global and static variables require constant expressions.
Array size can be inferred if not explicitly provided.
Character strings automatically include \0 if initialized with string literals.

Preprocessor directive:

File Inclusion:
File inclusion allows you to include external files, typically header files, in your program.
Why use #include?
Ensures all source files use the same definitions and declarations.
Prevents duplicate code by defining macros and constants in one place.
When an included file changes, all files that depend on it must be recompiled.

Macro: Macro with arguments
To remove a macro definition, use #undef.
The # operator **converts macro arguments into string literals**.
The ## operator **merges tokens** into a single identifier.

Conditional compilation- #if, #ifndef ,#elif, #else ,#ifdef ,#endif.

#define is a preprocessor directive.
it is handled before compilation.
No memory is allocated. It performs a text replacement wherever PI appears in the code.
It does not have a type, so the compiler does not check for type safety.

Cannot be debugged easily since it does not exist in the compiled code

Const vs macro?
const is generally better than #define because:
- Type safety: Prevents accidental misuse of the constant.
- Debugging support: Exists in compiled code, making debugging easier.

Const:
Type checked as it is compiled. Cant change its value