2. Basic of C Programming (Part-2)

Syeda Jannatul Naim
Lecturer | Dept. of CSE
World University of Bangladesh (WUB)

January, 2025

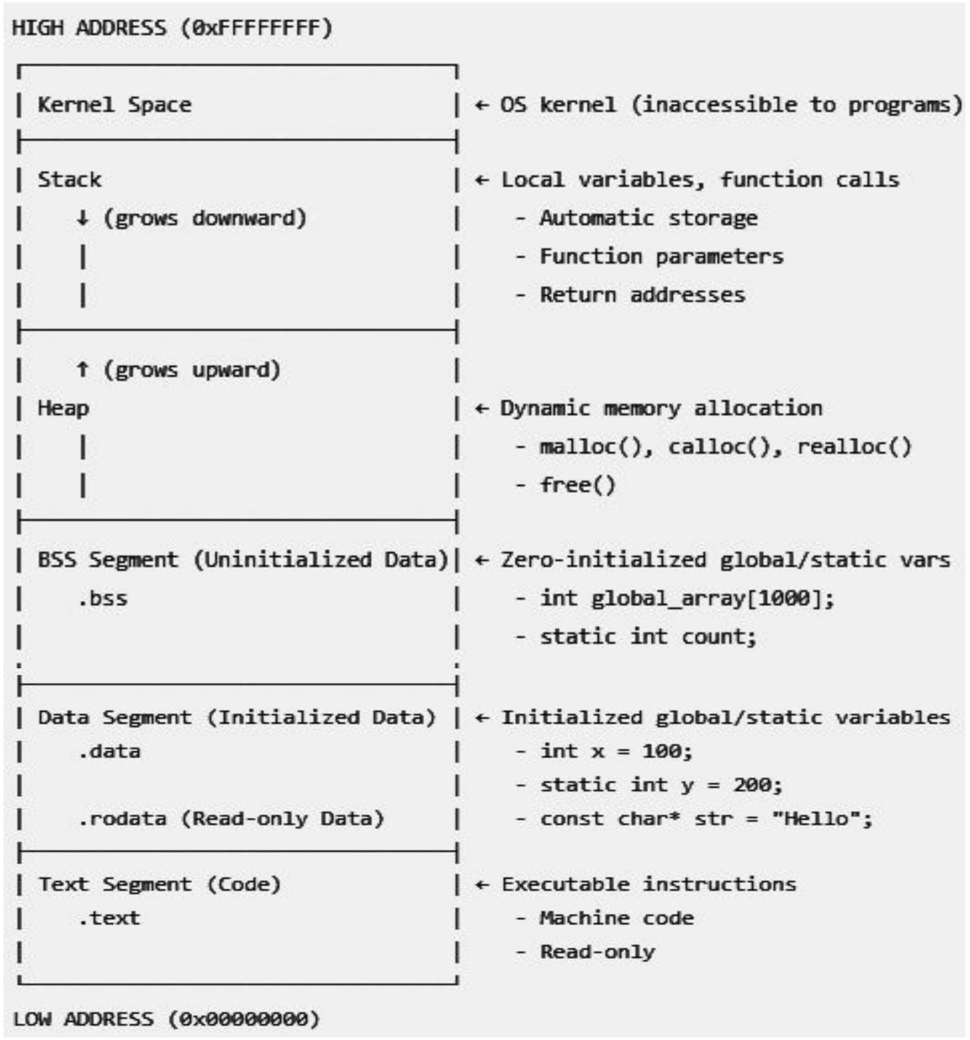
Content

1. Memory Layout of C Programs

- ❑ The **memory layout of a program** shows how its data is stored in memory during execution. It helps developers understand and manage memory efficiently.
  - Memory is divided into sections such as code, data, heap, and stack.
  - Knowing the memory layout is useful for optimizing performance, debugging and prevent errors like segmentation fault and memory leak.

- ❑ When a C program runs, the operating system divides its **virtual memory** into several **segments**. Each segment has a **specific purpose**, **different lifetime**, and **different access rules**.

- ❑ The following diagram shows memory from **low address** → **high address**.



```
HIGH ADDRESS (0xFFFFFFFF)

┌─────────────────────────────────┐
│ Kernel Space                    │ ← OS kernel (inaccessible to programs)
├─────────────────────────────────┤
│ Stack                           │ ← Local variables, function calls
│    ↓ (grows downward)           │   - Automatic storage
│    │                            │   - Function parameters
│    │                            │   - Return addresses
├─────────────────────────────────┤
│    ↑ (grows upward)             │
│ Heap                            │ ← Dynamic memory allocation
│    │                            │   - malloc(), calloc(), realloc()
│    │                            │   - free()
├─────────────────────────────────┤
│ BSS Segment (Uninitialized Data)│ ← Zero-initialized global/static vars
│    .bss                         │   - int global_array[1000];
│                                 │   - static int count;
├─────────────────────────────────┤
│ Data Segment (Initialized Data) │ ← Initialized global/static variables
│    .data                        │   - int x = 100;
│                                 │   - static int y = 200;
│    .rodata (Read-only Data)     │   - const char* str = "Hello";
├─────────────────────────────────┤
│ Text Segment (Code)             │ ← Executable instructions
│    .text                        │   - Machine code
│                                 │   - Read-only
└─────────────────────────────────┘

LOW ADDRESS (0x00000000)
```

1. Memory Layout of C Programs

1. Text Segment (Code Segment)

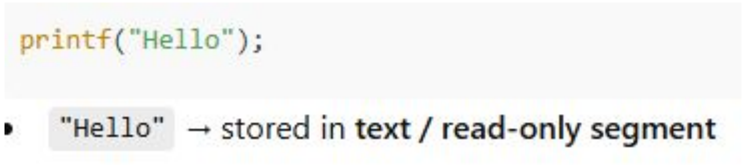
Lowest address

What it contains

- Compiled **machine instructions** of the program
- Read-only data

Properties

- **Read-only** (to prevent accidental modification)
- **Shared** among processes running the same program
- Loaded from the executable file



```
printf("Hello");
```

- `"Hello"` → stored in **text / read-only segment**

2. Initialized Data Segment

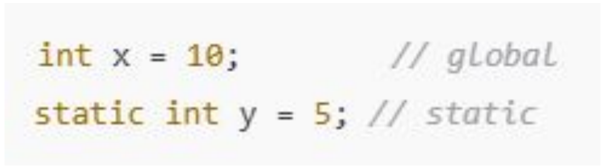
Above text segment

What it contains

- **Global and static variables** that are **initialized**

Properties

- Read/write
- Initialized before `main()` starts



```
int x = 10;          // global
static int y = 5; // static
```

1. Memory Layout of C Programs

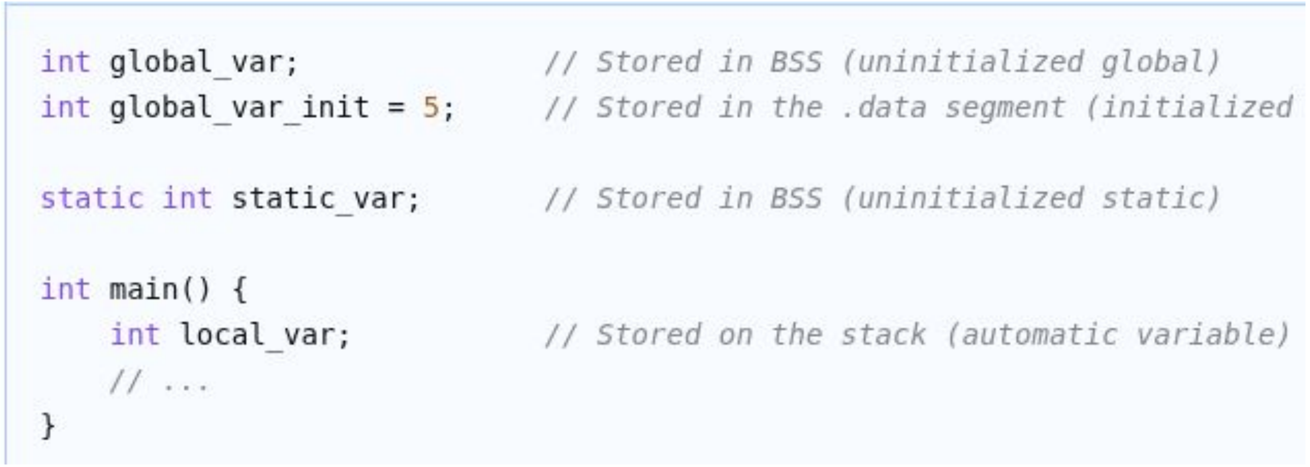
3. Uninitialized Data Segment (BSS=block started by symbol) *Above initialized data*

What it contains

- **Global and static variables** that are **not initialized**
- Automatically initialized to **0**

Properties

- Saves space in executable file size
- Initialized at runtime



```c
int global_var;                // Stored in BSS (uninitialized global)
int global_var_init = 5;       // Stored in the .data segment (initialized

static int static_var;         // Stored in BSS (uninitialized static)

int main() {
    int local_var;             // Stored on the stack (automatic variable)
    // ...
}
```

1. Memory Layout of C Programs

4. Heap

*Grows **upward** ↑*

What it contains

- Dynamically allocated memory

Managed by

- Programmer (`malloc`, `calloc`, `free`)
- Memory exists until explicitly freed

Problems if misused

- Memory leak
- Dangling pointer
- Fragmentation

5. Stack

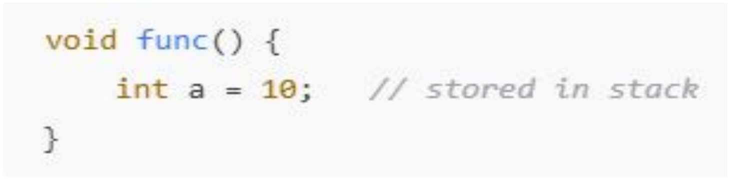
*Grows **downward** ↓*

What it contains

- Local variables
- Function parameters
- Return addresses

Properties

- Automatically managed
- Memory freed when function returns
- Very fast access



```c
void func() {
    int a = 10;    // stored in stack
}
```

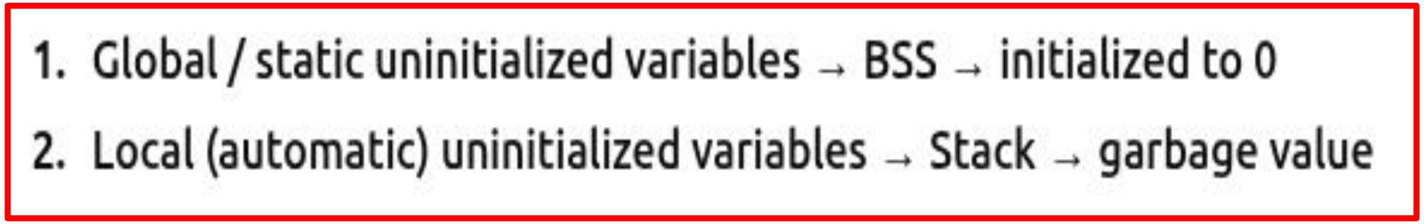
1. Memory Layout of C Programs

6. Command-line Arguments & Environment Variables

Highest address

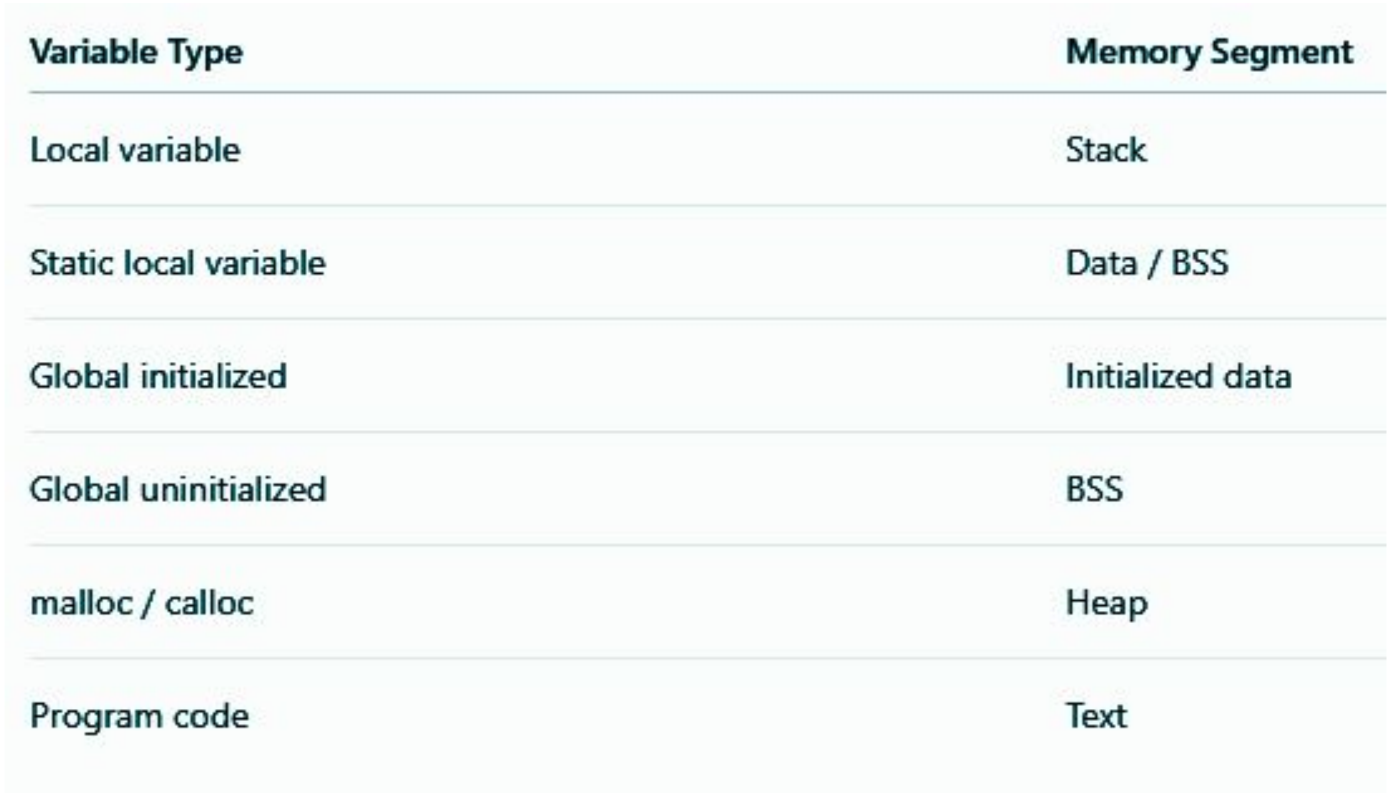
What it contains

- `argc`, `argv`
- Environment variables (`PATH`, `HOME`, etc.)



1. Global / static uninitialized variables → BSS → initialized to 0
2. Local (automatic) uninitialized variables → Stack → garbage value

1. Memory Layout of C Programs



Variable Type	Memory Segment
Local variable	Stack
Static local variable	Data / BSS
Global initialized	Initialized data
Global uninitialized	BSS
malloc / calloc	Heap
Program code	Text

2. Memory Leak

A **memory leak** happens when a program **allocates memory dynamically** (from the heap) but **does not release it** after use.
That memory stays reserved and **cannot be reused**, even though the program no longer needs it.

Why Is This a Problem?

- Memory usage keeps increasing (heap memory)
- Program becomes slow
- Long-running programs may crash
- System resources get wasted

3. Data Overflow

Data overflow occurs when a variable is assigned a value **greater than the maximum value** that its data type can store. Because memory size is limited, the value **wraps around** or becomes **incorrect**.

Why Data Overflow Happens

- Using a **small data type**
- Ignoring **data type limits**
- Arithmetic operations beyond range
- Large calculations without checks

Effects of Data Overflow

- Wrong calculations
- Logical errors
- Program crashes
- Security vulnerabilities (critical systems)

<u>4. Data Underflow</u>

Data underflow occurs when a variable’s value **becomes smaller than the minimum value** that its data type can represent.

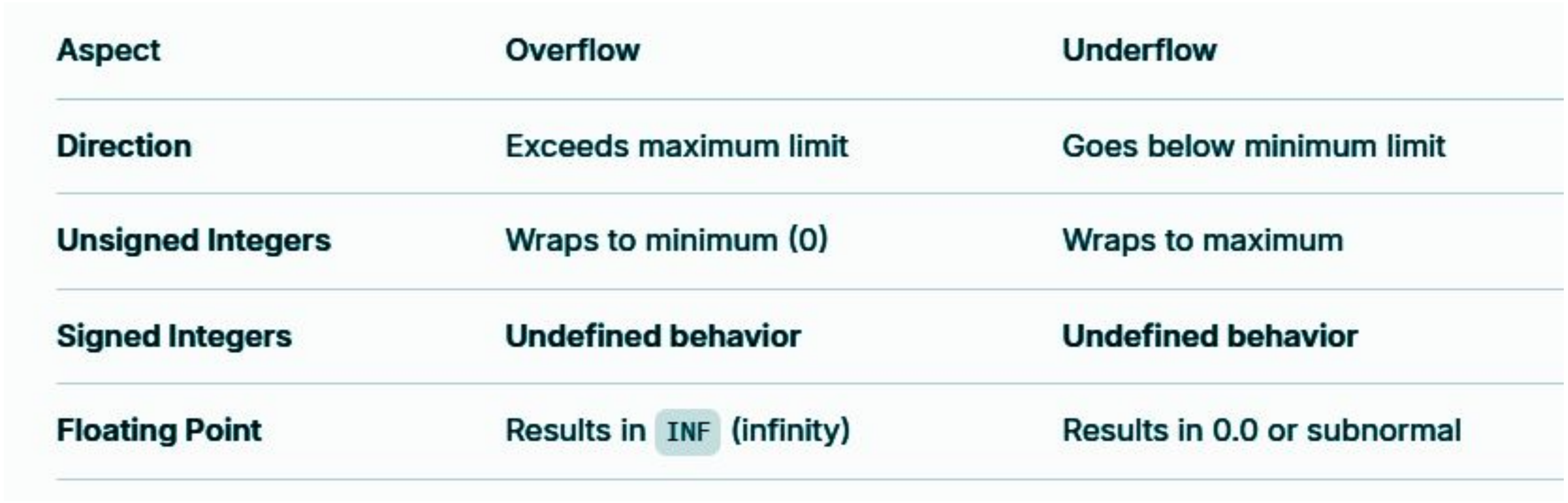
When Data Overflow Happens

- Subtracting numbers from a variable already at its minimum
- Decrementing a loop counter past its lower limit
- Floating-point numbers approaching **zero**

Why Data Underflow Is Dangerous

- Loss of precision
- Wrong calculations
- Infinite loops
- Incorrect comparisons (x == 0 becomes true)

Data Overflow Vs. Data Underflow



Aspect	Overflow	Underflow
Direction	Exceeds maximum limit	Goes below minimum limit
Unsigned Integers	Wraps to minimum (0)	Wraps to maximum
Signed Integers	**Undefined behavior**	**Undefined behavior**
Floating Point	Results in `INF` (infinity)	Results in 0.0 or subnormal

5. Storage class: scope and lifetime of variables

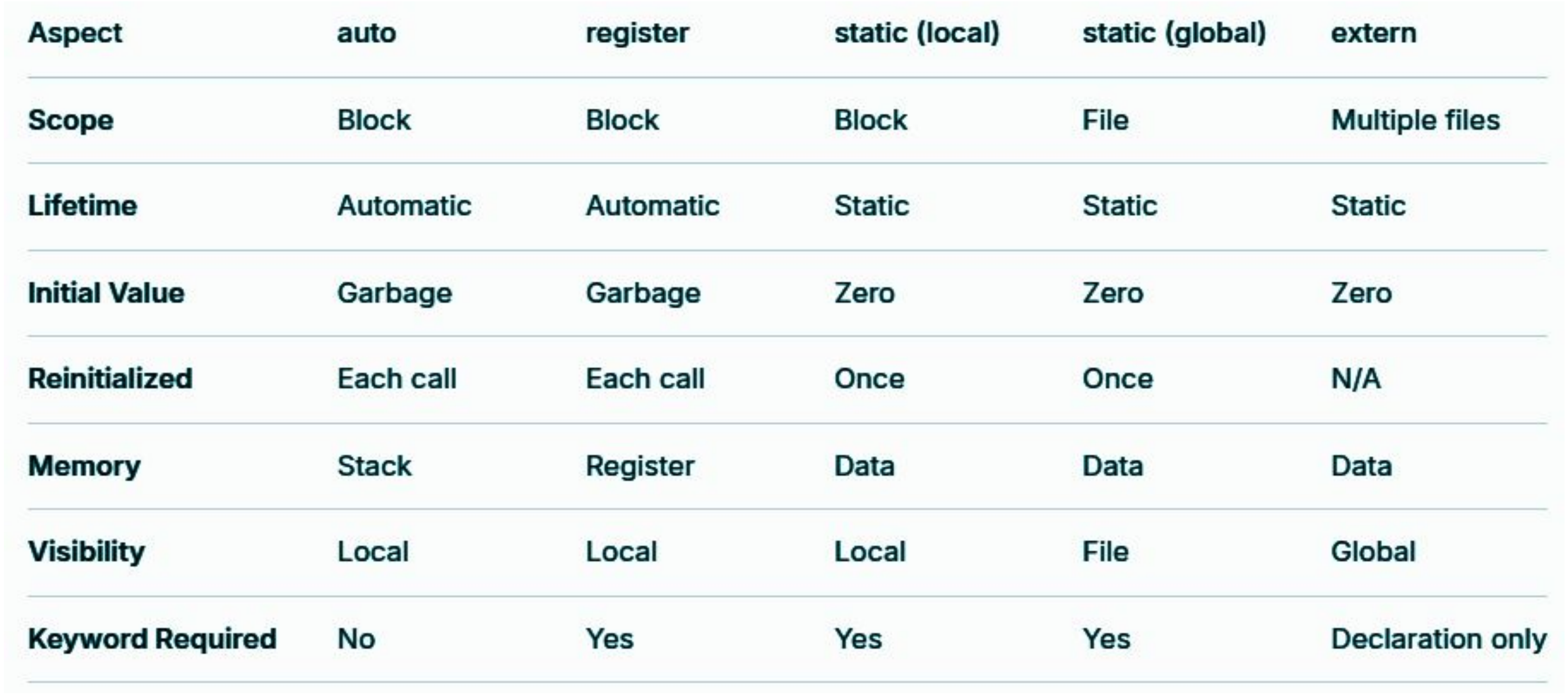
In C, a **storage class** defines **where a variable is stored**, **its scope (visibility)**, and **its lifetime (how long it exists in memory)**.

- **Scope** → Where the variable can be accessed in the program
- **Lifetime** → How long the variable exists in memory

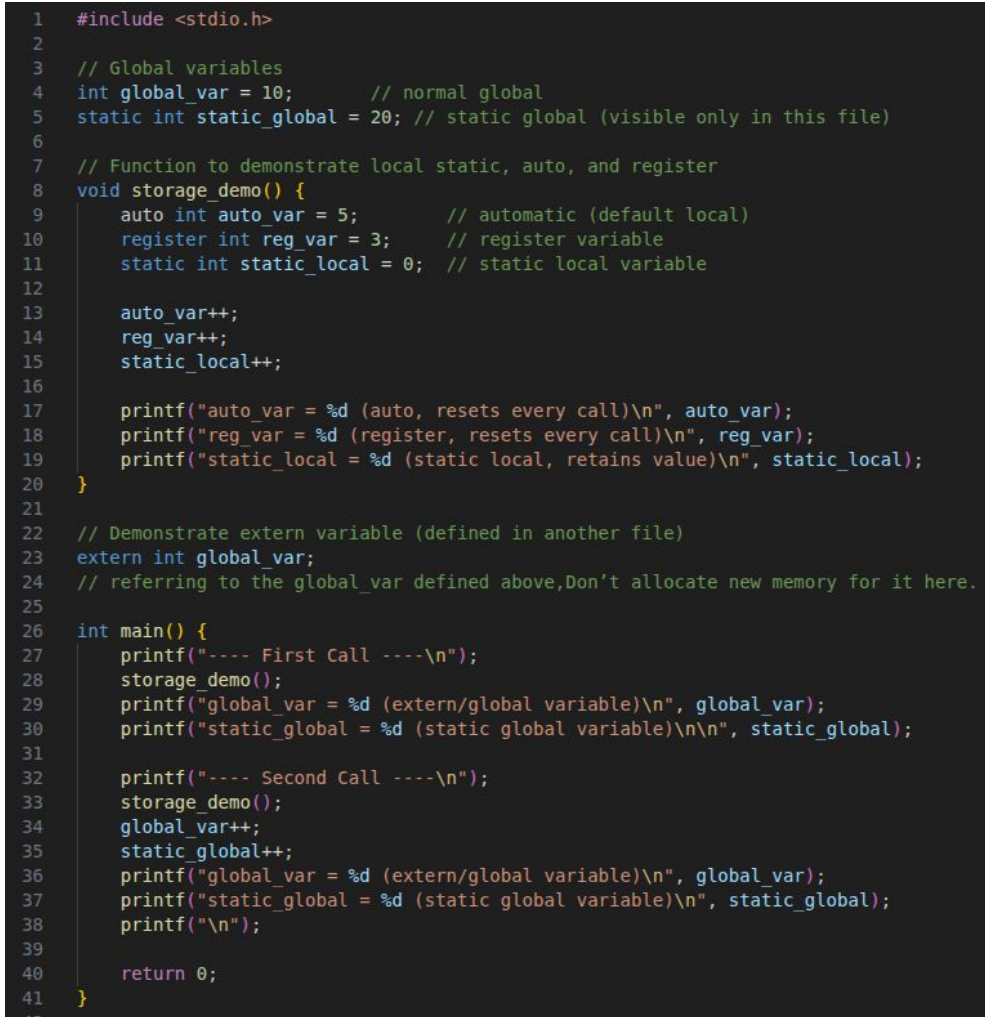
C has **four main storage classes**:

1. auto
2. register
3. static
   a. static local
   b. static global
4. extern

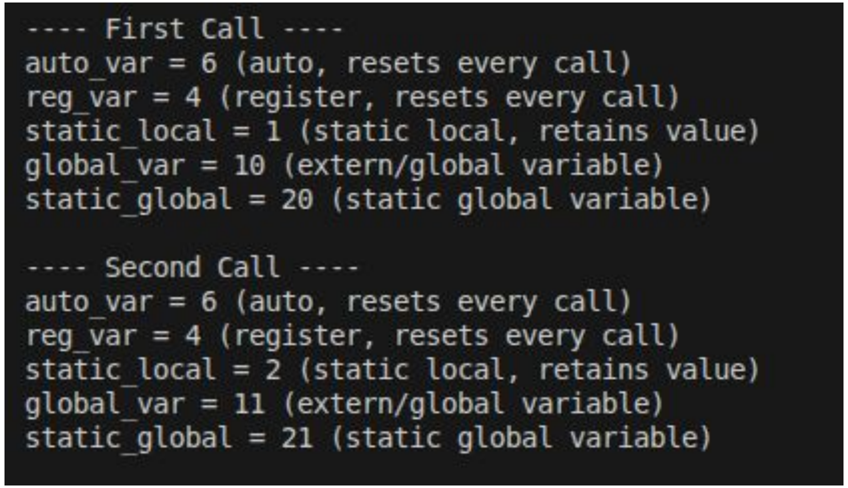
5. Storage class: scope and lifetime of variables



Aspect	auto	register	static (local)	static (global)	extern
Scope	Block	Block	Block	File	Multiple files
Lifetime	Automatic	Automatic	Static	Static	Static
Initial Value	Garbage	Garbage	Zero	Zero	Zero
Reinitialized	Each call	Each call	Once	Once	N/A
Memory	Stack	Register	Data	Data	Data
Visibility	Local	Local	Local	File	Global
Keyword Required	No	Yes	Yes	Yes	Declaration only



```c
#include <stdio.h>

// Global variables
int global_var = 10;       // normal global
static int static_global = 20; // static global (visible only in this file)

// Function to demonstrate local static, auto, and register
void storage_demo() {
    auto int auto_var = 5;        // automatic (default local)
    register int reg_var = 3;     // register variable
    static int static_local = 0;  // static local variable

    auto_var++;
    reg_var++;
    static_local++;

    printf("auto_var = %d (auto, resets every call)\n", auto_var);
    printf("reg_var = %d (register, resets every call)\n", reg_var);
    printf("static_local = %d (static local, retains value)\n", static_local);
}

// Demonstrate extern variable (defined in another file)
extern int global_var;
// referring to the global_var defined above,Don’t allocate new memory for it here.

int main() {
    printf("---- First Call ----\n");
    storage_demo();
    printf("global_var = %d (extern/global variable)\n", global_var);
    printf("static_global = %d (static global variable)\n\n", static_global);

    printf("---- Second Call ----\n");
    storage_demo();
    global_var++;
    static_global++;
    printf("global_var = %d (extern/global variable)\n", global_var);
    printf("static_global = %d (static global variable)\n", static_global);
    printf("\n");

    return 0;
}
```

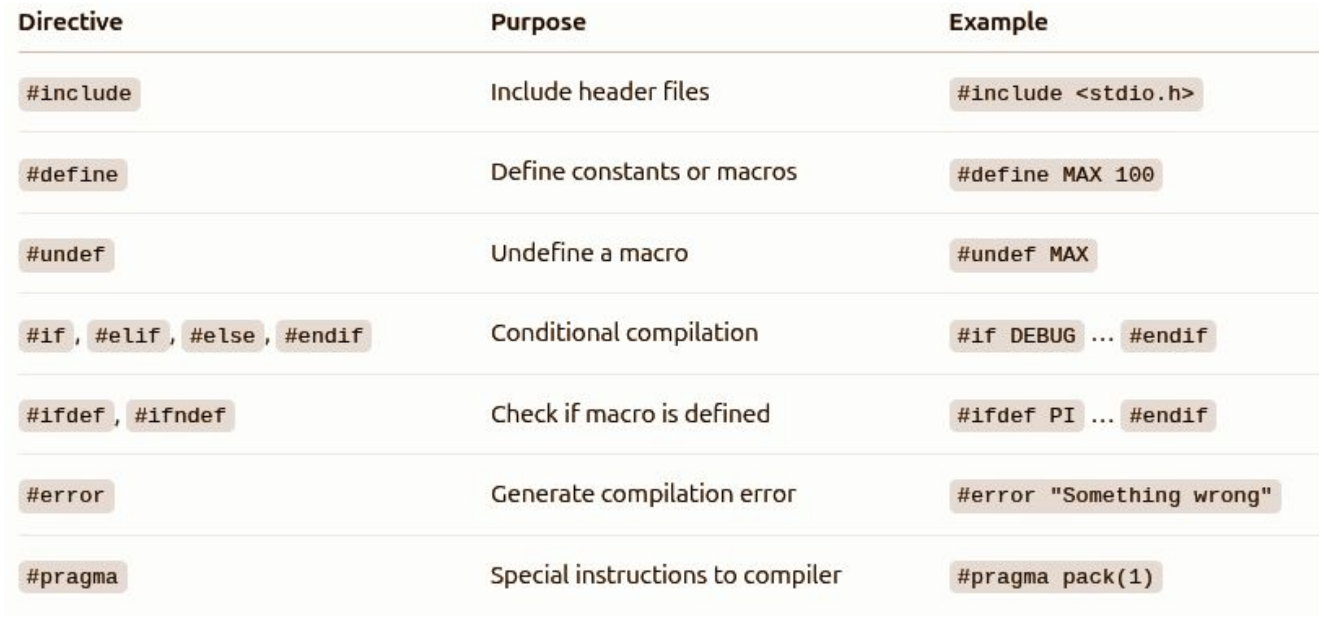


```
---- First Call ----
auto_var = 6 (auto, resets every call)
reg_var = 4 (register, resets every call)
static_local = 1 (static local, retains value)
global_var = 10 (extern/global variable)
static_global = 20 (static global variable)

---- Second Call ----
auto_var = 6 (auto, resets every call)
reg_var = 4 (register, resets every call)
static_local = 2 (static local, retains value)
global_var = 11 (extern/global variable)
static_global = 21 (static global variable)
```

6. Preprocessor Directive

A **preprocessor directive** is a **command that is executed by the preprocessor before compilation** i.e. modifies source code before compilation.

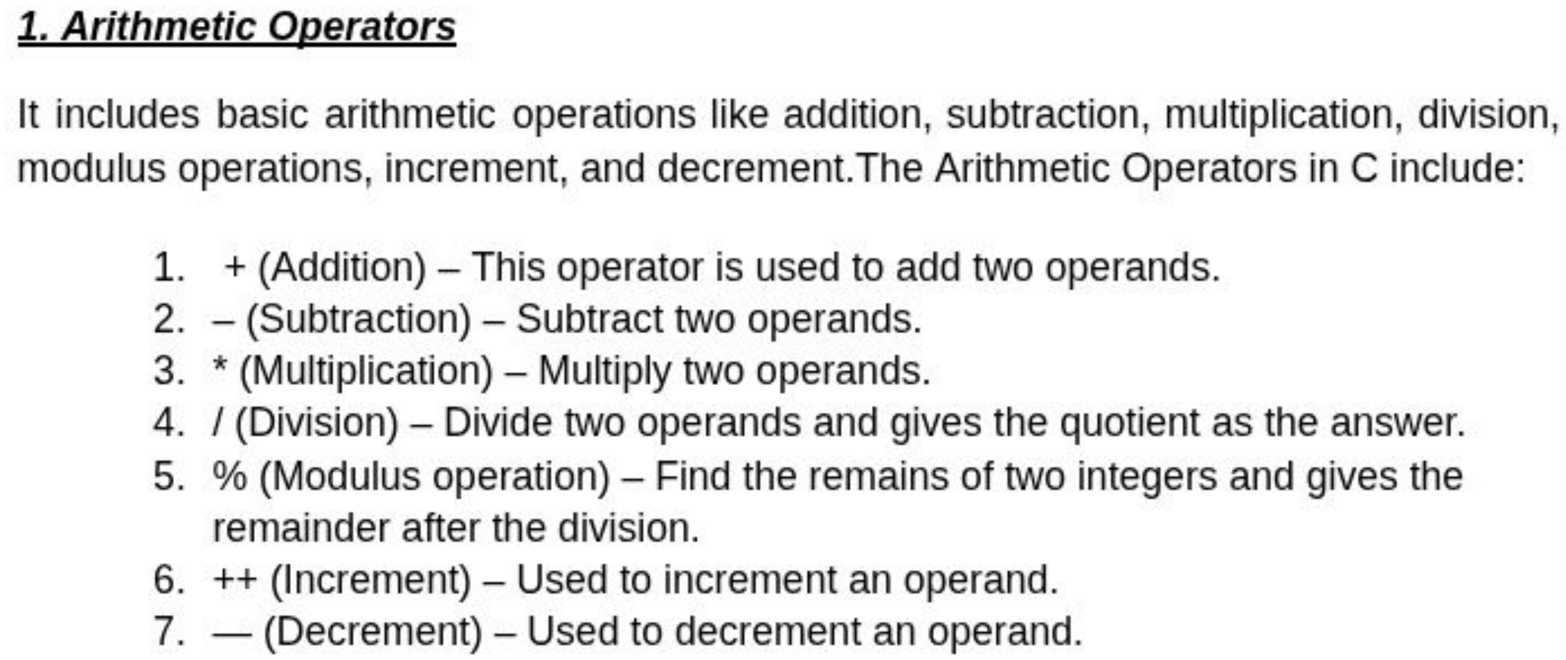
- It starts with #
- It **instructs the compiler** to do something **before the actual compilation begins**



Directive	Purpose	Example
`#include`	Include header files	`#include <stdio.h>`
`#define`	Define constants or macros	`#define MAX 100`
`#undef`	Undefine a macro	`#undef MAX`
`#if`, `#elif`, `#else`, `#endif`	Conditional compilation	`#if DEBUG` … `#endif`
`#ifdef`, `#ifndef`	Check if macro is defined	`#ifdef PI` … `#endif`
`#error`	Generate compilation error	`#error "Something wrong"`
`#pragma`	Special instructions to compiler	`#pragma pack(1)`

7. Operator

Operators can be defined as basic symbols that help us work on **logical and mathematical operations**. Operators in C, are tools or symbols that are used to perform mathematical operations concerning **arithmetic, logical, conditional and, bitwise operations.**



1. Arithmetic Operators

It includes basic arithmetic operations like addition, subtraction, multiplication, division, modulus operations, increment, and decrement.The Arithmetic Operators in C include:

1. + (Addition) – This operator is used to add two operands.
2. – (Subtraction) – Subtract two operands.
3. * (Multiplication) – Multiply two operands.
4. / (Division) – Divide two operands and gives the quotient as the answer.
5. % (Modulus operation) – Find the remains of two integers and gives the remainder after the division.
6. ++ (Increment) – Used to increment an operand.
7. — (Decrement) – Used to decrement an operand.

7. Operator

2. Relational Operators

It is used to compare two numbers by checking whether they are equal or not, less than, less than or equal to, greater than, greater than or equal to.

1. == (Equal to)– This operator is used to check if both operands are equal.
2. != (Not equal to)– Can check if both operands are not equal.
3. > (Greater than)– Can check if the first operand is greater than the second.
4. < (Less than)- Can check if the first operand is lesser than the second.
5. >= (Greater than equal to)– Check if the first operand is greater than or equal to the second.
6. <= (Less than equal to)– Check if the first operand is lesser than or equal to the second

If the relational statement is satisfied (it is true), then the program will return the value 1, otherwise, if the relational statement is not satisfied (it is false), the program will return the value 0.

7. Operator

3. Logical Operators

It refers to the boolean values which can be expressed as:

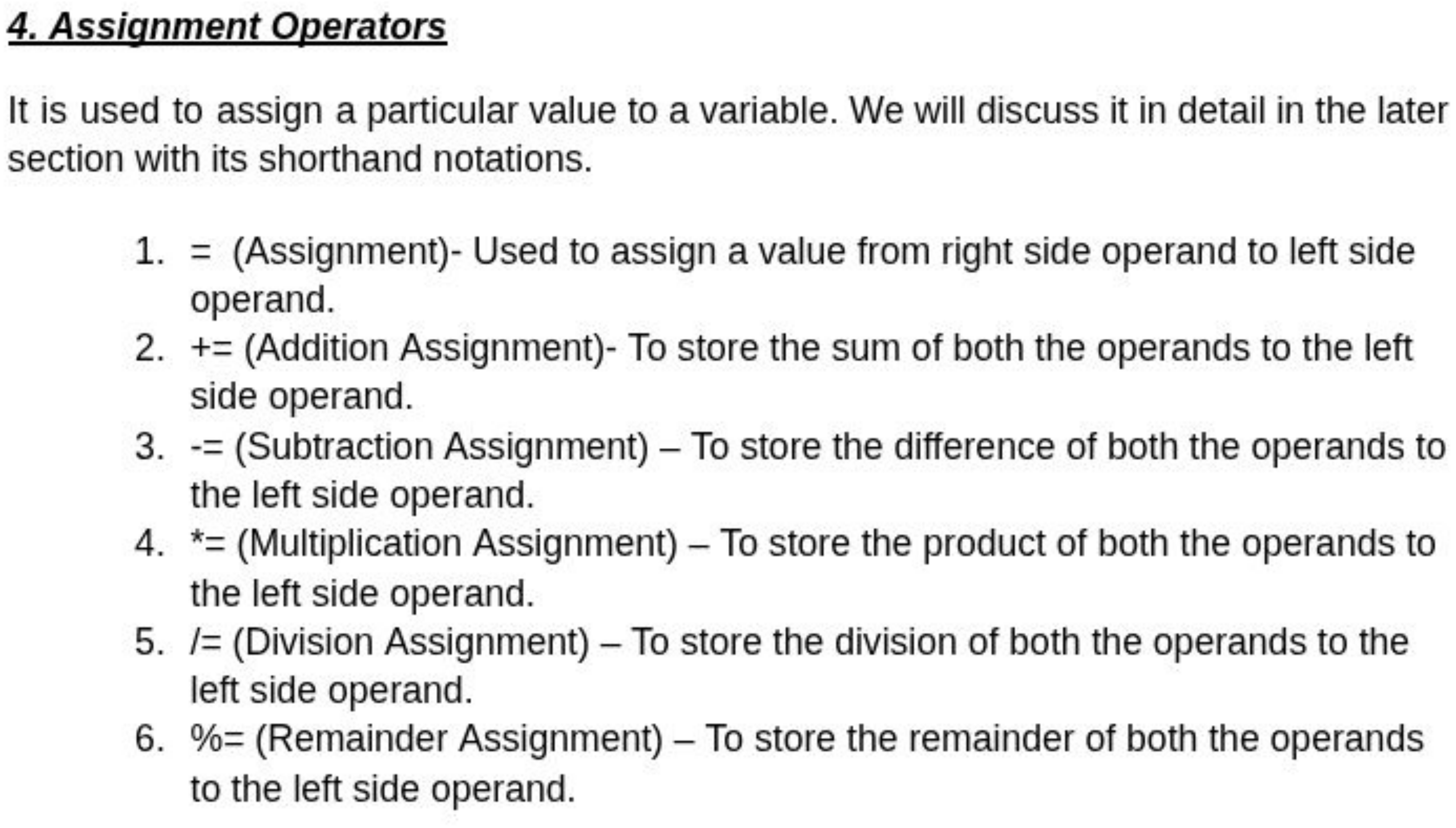
- Binary logical operations, which involves two variables: AND and OR
- Unary logical operation: NOT

Logical Operators in C++ includes –

1. && (AND) – It is used to check if both the operands are true.
2. || (OR) – These operators are used to check if at least one of the operand is true.
3. ! (NOT) – Used to check if the operand is false

If the logical statement is satisfied (it is true), then the program will return the value 1, otherwise, if the relational statement is not satisfied (it is false), the program will return the value 0.

7. Operator



4. Assignment Operators

It is used to assign a particular value to a variable. We will discuss it in detail in the later section with its shorthand notations.

1. = (Assignment)- Used to assign a value from right side operand to left side operand.
2. += (Addition Assignment)- To store the sum of both the operands to the left side operand.
3. -= (Subtraction Assignment) – To store the difference of both the operands to the left side operand.
4. *= (Multiplication Assignment) – To store the product of both the operands to the left side operand.
5. /= (Division Assignment) – To store the division of both the operands to the left side operand.
6. %= (Remainder Assignment) – To store the remainder of both the operands to the left side operand.

7. Operator

5. Bitwise Operators

It is based on the principle of performing operations bit by bit which is based on boolean algebra. It increases the processing speed and hence the efficiency of the program.Bitwise operators are not applicable in the case of float and double.The Bitwise Operators in C Includes –

1. & (Bitwise AND) – Converts the value of both the operands into binary form and performs AND operation bit by bit.
2. | (Bitwise OR) – Converts the value of both the operands into binary form and performs OR operation bit by bit.
3. ^ (Bitwise exclusive OR) – Converts the value of both the operands into binary form and performs EXCLUSIVE OR operation bit by bit.
4. ~ (One's complement operator): Converts the operand into its complementary form.
5. << – Left shift
6. >> – Right shift

7. Operator

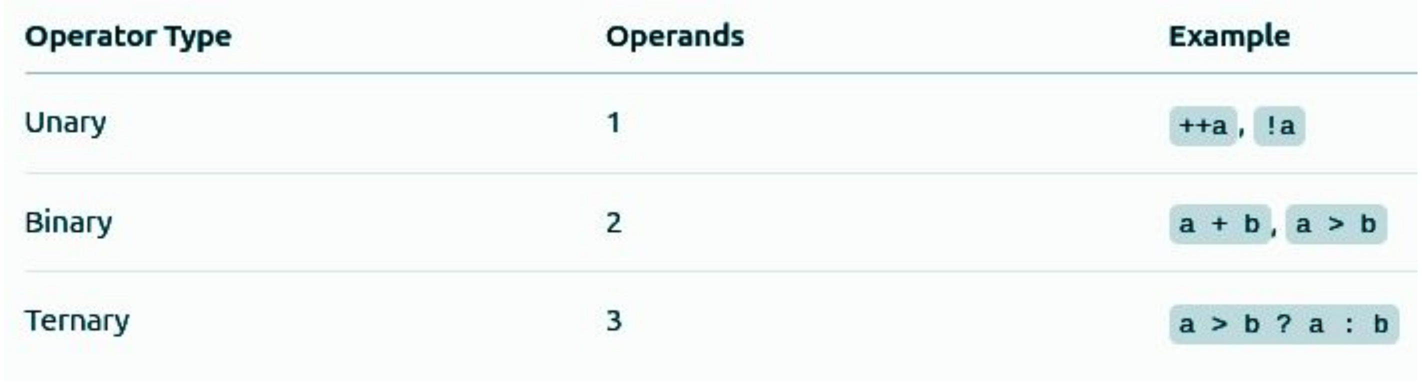
6. Miscellaneous Operators Apart from the above-discussed operators, there are certain operators which fall under this category which include sizeof and ternary (conditional) operators.Here is a list which illustrates the use of these operators:

1. sizeof – It returns the memory occupied by the particular data type of the operand
2. & (reference or address) – It refers to the address (memory location) in which the operand is stored.
3. * (Pointer) – It is a pointer operator
4. ? (Condition) – It is an alternative for if-else condition

7. Operator

In another way, operators are classified by **how many operands they work on**. That's where **unary**, **binary**, and **ternary** operators come in.

- A **unary operator** operates on **only one variable/value**.
- A **binary operator** operates on **two operands**.
- The **ternary operator** is the **only operator** in C that works on **three operands**. It is used as a shorthand for if–else.
  - Syntax: ***condition ? expression1 : expression2;***
    - If `condition` is **true** → `expression1` executes
    - If `condition` is **false** → `expression2` executes



Operator Type	Operands	Example
Unary	1	`++a`, `!a`
Binary	2	`a + b`, `a > b`
Ternary	3	`a > b ? a : b`

8. Type Casting

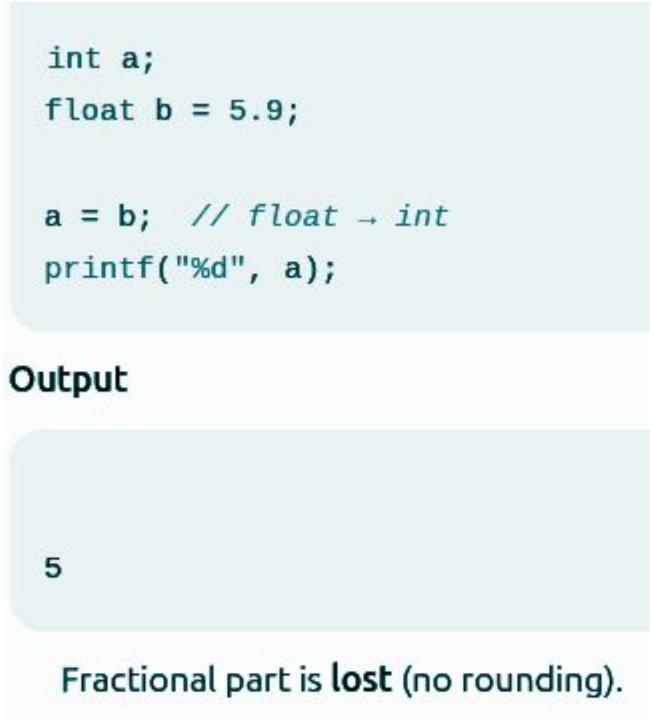
Type casting (also called type conversion) is the process of converting a value from one data type to another in programming. In C, this can happen in two ways:

- **Implicit Type Conversion (Automatic):**The compiler automatically converts types without programmer intervention.
- **Explicit Type Conversion (Manual):** Programmer manually specifies the conversion using cast operator.

8. Type Casting

Implicit Type Conversion (Automatic):

The compiler automatically converts types without programmer intervention.



```c
int a;
float b = 5.9;

a = b;  // float → int
printf("%d", a);
```

Output

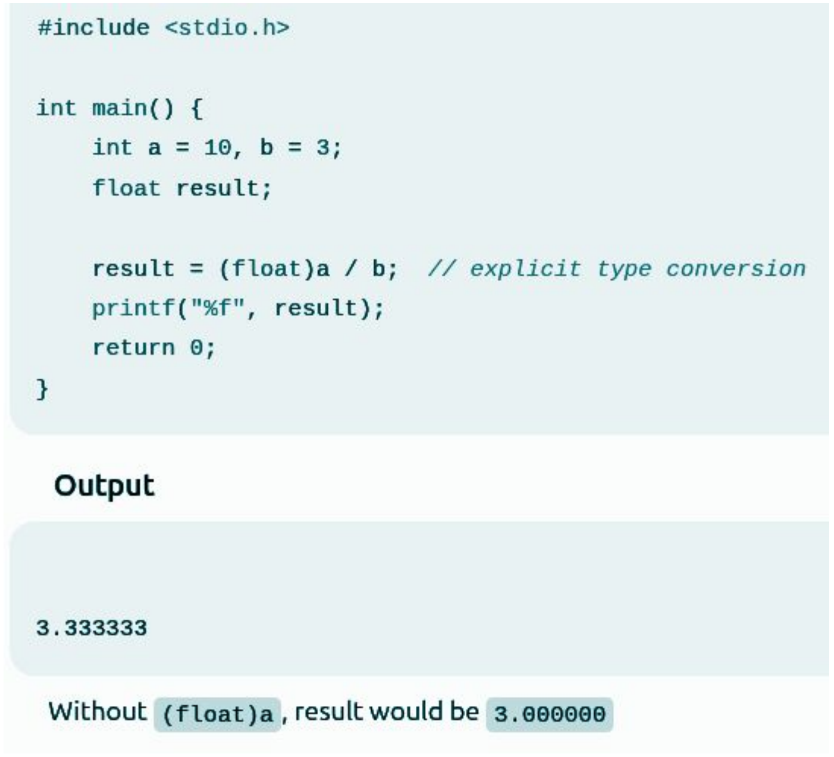
```
5
```

Fractional part is **lost** (no rounding).

8. Type Casting

Explicit Type Conversion (Manual):

Programmer manually specifies the conversion using cast operator.



```c
#include <stdio.h>

int main() {
    int a = 10, b = 3;
    float result;

    result = (float)a / b;  // explicit type conversion
    printf("%f", result);
    return 0;
}
```

Output

```
3.333333
```

Without `(float)a`, result would be `3.000000`

END
of
Chapter 2

***Reference**: E. Balaguruswamy; Programming in ANSI C; chapter-1, chapter-2, chapter-3, chapter-4, chapter-14;*