2. Basic of C Programming (Part-1)

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

January, 2025

Content

1. Basic Structure of C program
2. Process of compiling and running a C Program
3. Constant
4. Variable
5. Data types
6. Type modifier
7. Format specifiers
8. size_t and sizeof()

1. Basic Structure of C program

1. Documentation Section
2. Preprocessor Directives Section a. Macro Definitions Section
3. Global Declarations Section
4. User-defined Functions Prototype Section
5. Main Function Section
6. User-defined Functions Definition Section

1. Basic Structure of C program

1. Documentation Section

```c
/*
   Multi-line comment style
   Good for file headers and block comments
*/

// Single-line comment style
// Good for short explanations
```

```c
/*
 * FILENAME: program_name.c
 * DESCRIPTION: Brief description of what program does
 * AUTHOR: Your Name
 * DATE: Creation/Modification date
 * VERSION: 1.0
 */
```

2. Preprocessor Directives Section

- Start with #
- Processed before compilation
- `#include` - adds library code
- `#define` - creates macros
- `#ifdef / #endif` - conditional compilation

```c
// Macro definitions
#define PI 3.14159265359
#define MAX_SIZE 100
#define MIN(a, b) ((a) < (b) ? (a) : (b))

// Conditional compilation
#ifdef DEBUG
    #define DEBUG_PRINT(msg) printf("DEBUG: %s\n", msg)
#else
    #define DEBUG_PRINT(msg)
#endif
```

```c
#include <stdio.h>    // Include standard I/O library
/*
Common Header Files:
- stdio.h: printf(), scanf(), FILE operations
- stdlib.h: malloc(), free(), exit(), rand()
- math.h: sin(), cos(), sqrt(), pow()
- string.h: strlen(), strcpy(), strcmp()
- ctype.h: isalpha(), isdigit(), toupper()
*/

#include "myheader.h"  // Include user-defined header file
/*
Difference:
- <filename.h>: Searches in system directories
- "filename.h": Searches in current directory first
*/
```

3. Global Declarations Section

- Variables accessible throughout program
- Structure/type definitions
- Use sparingly to avoid side effects

```
// Global variables (accessible throughout program)
int global_var = 10;          // Avoid excessive use
static int file_scope = 20;  // Limited to this file

// Global constants
const double GRAVITY = 9.8;
const char* COMPANY = "ABC Corp";

// Structure definitions
struct Student {
    int id;
    char name[50];
    float marks;
};

// Type definitions
typedef unsigned int uint;
typedef struct Student Student;

// Enumeration
enum Days {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Boolean {FALSE, TRUE};
```

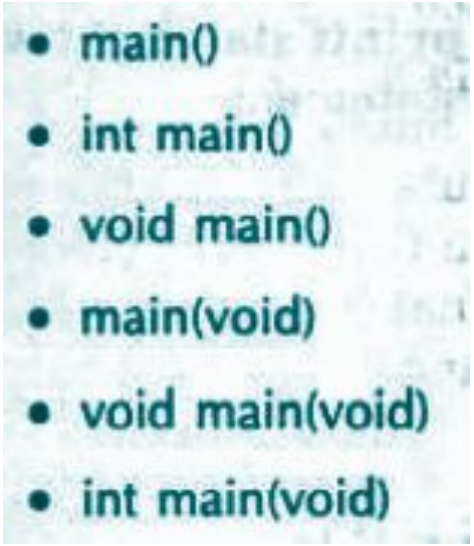
4. User-defined Functions Prototype Section

- Declare functions before defining them
- Enables calling functions in any order
- Required if function defined after main()

```c
// Tell compiler about functions before they're defined
int calculate_sum(int a, int b);
void print_message(char* msg);
double find_average(float arr[], int size);
```

5. Main Function: *main()*

- Program entry point
- Must return int value
- return statement:
  - 0 indicates successful execution
  - Non-zero indicates error
- When main() ends, program terminates
- Inside main() we use:
  - local variable declaration
  - executable statement
  - output statement



```c
// Common main() structure:
int main() {
    // 1. Variable declarations
    int x, y, result;

    // 2. Input operations
    printf("Enter values: ");
    scanf("%d %d", &x, &y);

    // 3. Processing
    result = x + y;

    // 4. Output
    printf("Result: %d\n", result);

    // 5. Return value
    return 0;  // Success
}
```

```c
// Break into small, focused functions
// Each function should do ONE thing well

// Instead of one big main():
int main() {
    // 100 lines of code doing everything
}

// Use multiple functions:
void get_input();
void process_data();
void display_results();

int main() {
    get_input();
    process_data();
    display_results();
    return 0;
}
```

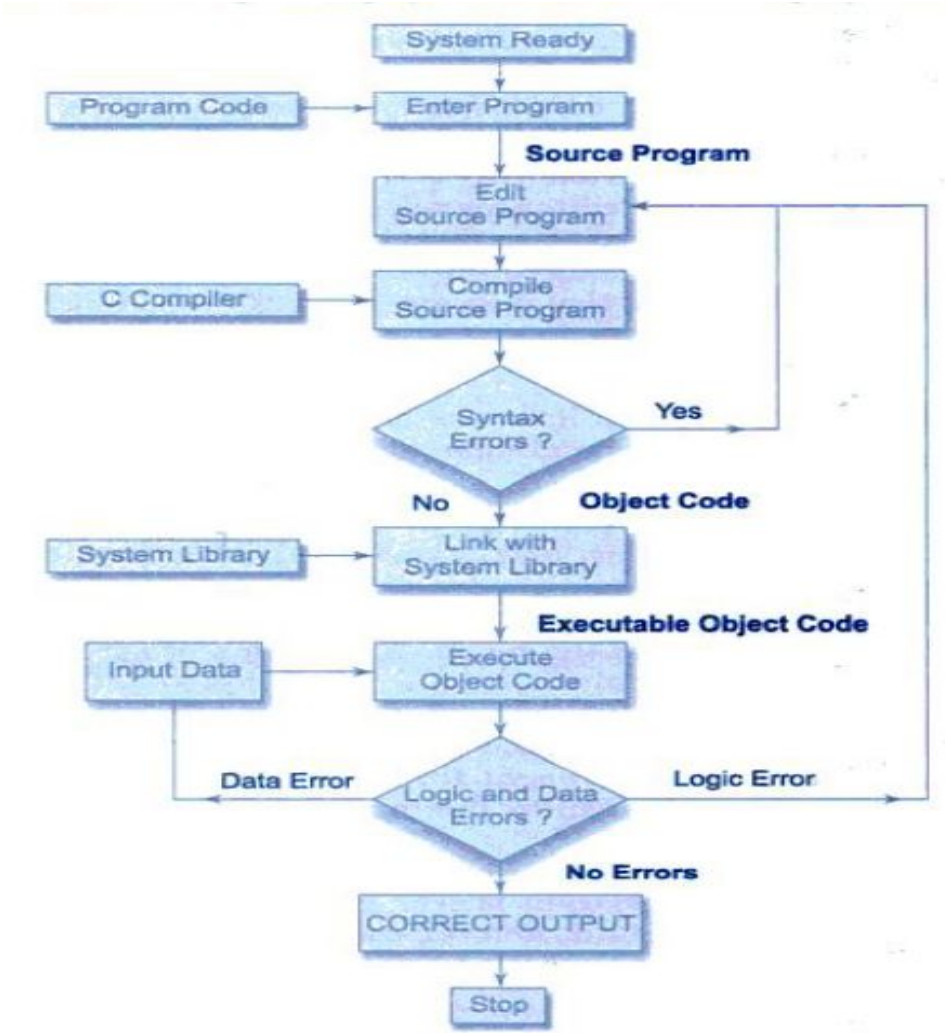
6. User-defined Functions Definition Section

- Implement declared functions
- Should have single responsibility

```c
// Function prototype
float calculate(float a, float b, char operator);
```

```c
// Function definition
float calculate(float a, float b, char operator) {
    switch(operator) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/':
            if(b != 0) return a / b;
            else {
                printf("Error: Division by zero!\n");
                return 0;
            }
        default:
            printf("Error: Invalid operator!\n");
            return 0;
    }
}
```

2. Process of compiling and running a C Program



3. Constant in C Language

A **constant** in C is a value that **does not change during program execution**. Once defined, its value **cannot be modified**.

1. Using `#define` (Preprocessor Constant)

- No memory allocated
- Value replaced before compilation

```
#define PI 3.14159
#define MAX_SIZE 100
#define GREETING "Hello, World!"
```

2. Using `const` Keyword

- Memory allocated
- Type checking supported

```
const float PI = 3.14159;
const int MAX_SIZE = 100;
const char GREETING[] = "Hello, World!";
```

Key Differences

`#define`	`const`
Preprocessor directive (text replacement)	Compiler-managed variable
No memory allocation	Allocates memory
No type checking	Type checking occurs
Global scope	Respects scope rules
Can't be used with pointers	Can have pointers to it

4. Variable in C language

A variable is a named **memory location used** to store data that **can be changed during program execution**.

❖ Variable Declaration Syntax:

```
data_type variable_name;
// or
data_type variable_name = initial_value;
```

❖ Variable Naming Rules:

1. Must begin with letter or underscore
2. Can contain letters, digits, underscores
3. Case-sensitive
4. Cannot use C keywords
5. No spaces allowed

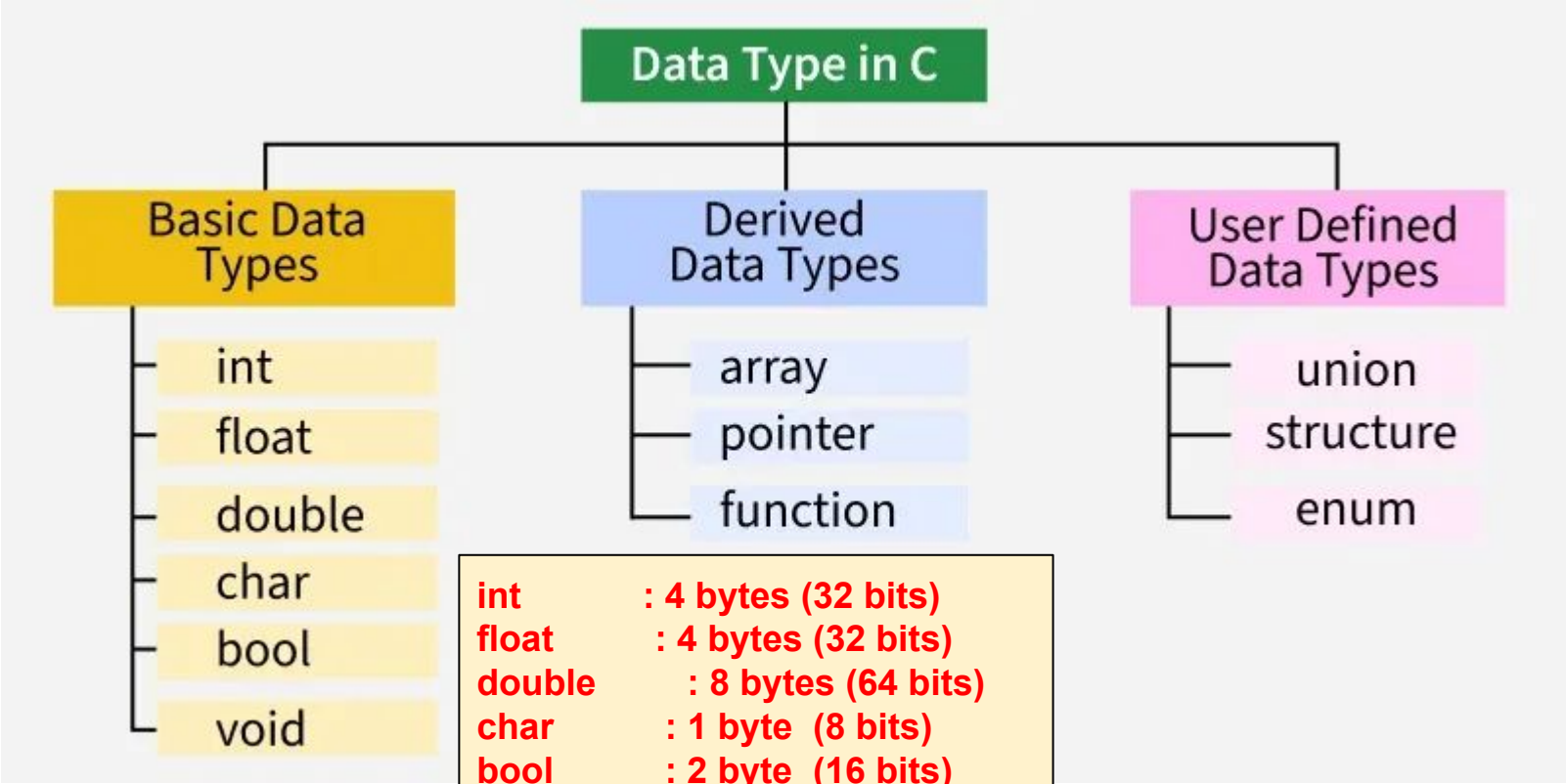
✔ Valid: `sum`, `_total`, `num1`

✘ Invalid: `1sum`, `float`, `total-amount`

```c
// Declaration and initialization
int score = 100;
float temperature = 98.6f;
char initial = 'J';

// Declaration without initialization
int count;
float average;

// Assignment
count = 10;
average = 85.5f;

// Modifying values
score = score + 20;     // score becomes 120
temperature += 1.5;     // temperature becomes 100.1
```

5. Data types in C language

In C, **data types** define **what kind of data a variable can store**.



int : 4 bytes (32 bits)
float : 4 bytes (32 bits)
double : 8 bytes (64 bits)
char : 1 byte (8 bits)
bool : 2 byte (16 bits)
void : 0 byte (0 bits)

6. Type modifier in C language

Type modifiers in C are keywords that **modify the size (memory)** and/or **range of values** of basic data types (mainly int and char). They help to choose:

- **How much memory** to use *(short, long, long long)*
- **Whether negative values** are allowed *(signed, unsigned)*
- **How large numbers** can be stored

Type Modifier	What It Does	Affects	Example
`signed`	Allows positive and negative values	Range	`signed int x;`
`unsigned`	Allows only non-negative values	Range	`unsigned int x;`
`short`	Reduces memory size	Size	`short int x;`
`long`	Increases memory size	Size	`long int x;`
`long long`	Greatly increases memory size	Size	`long long int x;`

char : 1 byte (8 bits)
int : 4 bytes (32 bits)
short : 2 bytes (16 bits)
long : 8 bytes (64 bits)
long long : 8 bytes (64 bits)
float : 4 bytes (32 bits)
double : 8 bytes (64 bits)
long double : 16 bytes (128 bits)

7. Format Specifiers in C language

A format specifier tells printf() or scanf() **what type of data to display or read**. It starts with %.

Data Type	Format Specifier
int	%d
unsigned int	%u
short int	%hd
long int	%ld
float	%f
double	%lf
char	%c
string	%s
octal	%o
hexadecimal	%x

8. size_t and sizeof()

❖ **What is size_t?**

size_t is an **unsigned integer type** defined by the C standard to represent **sizes and counts** (especially memory sizes).

❖ **What is %zu?**

%zu is a **format specifier** used with printf() to print values of type **size_t**.

- **z** → length modifier meaning "the type is size_t"
- **u** → unsigned integer output

❖ **Why %zu is Important**

The return type of sizeof() is **size_t**, not int. Using %d or %ld for sizeof can cause:

- Wrong output
- Compiler warnings
- Undefined behavior (on some systems)

8. size_t and sizeof()

Correct:

```
printf("%zu\n", sizeof(int));
```

Wrong:

```
printf("%d\n", sizeof(int));
```

```
#include <stdio.h>

int main() {
    size_t a = sizeof(char);
    size_t b = sizeof(int);
    size_t c = sizeof(double);

    printf("char: %zu bytes\n", a);
    printf("int: %zu bytes\n", b);
    printf("double: %zu bytes\n", c);

    return 0;
}
```

END
of
Chapter 2
(Part-1)

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