Chapter 1 User-defined Types 2025/2026

1.1 Introduction

1.1.1 Motivation

In programming, data is used to represent real-world information. The C language provides primitive types such as int, char, float, and double. While these types are sufficient for simple values, they are often insufficient to model complex entities encountered in real applications. For example, consider the notion of a student. A student cannot be represented by a single integer or a single floating-point value. Instead, a student is characterized by several attributes, such as: an identification number, a name, an average grade.

Managing each attribute as a separate variable leads to code that is:

- difficult to read,
- difficult to maintain,
- prone to errors.

To address this problem, the C language allows programmers to define user-defined types, which make it possible to group related data into a single logical unit.

1.1.2 Built-in types vs user-defined types

- **Built-in types :** Built-in types are predefined by the language. Examples include: int for integers, float and double for real numbers, char for characters. These types represent atomic values, meaning they cannot be decomposed into smaller meaningful parts.
- **User-defined types :** User-defined types allow the programmer to create new data types adapted to a specific problem, improve code clarity, better reflect the structure of the data being manipulated.

In this chapter, we study the main user-defined type mechanisms available in C:

- Enumerations (enum)
- Records or Structures (struct)
- Other type definition possibilities (typedef)

1.1.3 Concept of abstraction

One of the main goals of user-defined types is abstraction. Abstraction consists of:

- representing a complex concept using a single type,
- hiding unnecessary details,
- focusing on the essential properties of the data.

For example, instead of manipulating several independent variables, we manipulate a single object that represents a complete concept (such as a student or a date). This approach:

- simplifies program design,
- makes algorithms easier to understand,
- improves code reuse.

Example: limitations of primitive types

Consider the following C code:

```
int id;
chat name[30];
float average;
```

Although these variables describe a student, there is **no explicit link** between them. If we manipulate several students, we must manage multiple arrays and ensure that corresponding elements match correctly. This approach becomes inefficient and error-prone as programs grow in size. User-defined types provide a solution by allowing these related values to be grouped into a **single structured entity**, which we will study in Sections 2 and 3.

By the end of this chapter, the student will be able to:

- understand the need for user-defined types,
- define and use enumerations,
- define and manipulate structures,
- use additional mechanisms for defining and improving types in C.

1.2 Enumerations

1.2.1 Definition and purpose

An enumeration is a user-defined data type that allows a variable to take its values from a finite, predefined set of named constants. In the C programming language, enumerations are defined using the keyword **enum**.

Enumerations are particularly useful when a variable represents a concept that can only assume one value among a limited number of well-defined alternatives. Instead of using arbitrary integers, enumerations associate meaningful names with these values, thereby improving program clarity and reliability.

Typical applications of enumerations include:

- days of the week,
- months of the year,
- menu choices,
- system states (e.g., running, stopped, paused),

- result statuses (success, failure, error).

1.2.2 Syntax and declaration

The general syntax for defining an enumeration in C is:

```
enum EnumName {
    CONSTANT1,
    CONSTANT2,
    CONSTANT3
};
```

- **`EnumeName`** defines a new type.
- `CONSTANT1, CONSTANT2, CONSTANT3` are enumeration constants.

Once defined, a variable of this type can be declared as follows:

```
enum EnumName variableName;
```

Example :

Defining a new enumeration called "**`Color`**":

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

Declaring a variable of type Color :

```
enum Color c;
```

Initializing the variable c :

```
C = GREEN
```

1.2.3 Implicit integer values

By default, the compiler assigns integer values to enumeration constants starting from 0, increasing by 1 for each subsequent constant.

Example : Suppose that we have an enumeration given by :

```
enum Day{
      SUNDAY,
      MONDAY,
      TUESDAY,
      WEDNESDAY,
      THURSDAY,
      FRIDAY,
      SATURDAY,
```

```
};
```

The internal representation is:

Constant	Integer value
SUNDAY	0
MONDAY	1
TUESDAY	2
WEDNESDAY	3
THURSDAY	4
FRIDAY	5
SATURDAY	6

Although enumeration constants are internally integers, they should always be manipulated using their **symbolic names**, not their numeric values.

1.2.4 Explicit Value Assignment and Its Use Cases

In some situations, default values are not sufficient or desirable. Explicit assignment is useful when:

- Interfacing with hardware or external systems;
- Representing error codes;
- Maintaining compatibility with predefined numeric conventions.

Example : Enumeration representing error codes

```
enum ErrorCode {
    OK = 0,
    FILE_NOT_FOUND = 404,
    PERMISSION_DENIED = 403,
    UNKNOWN_ERROR = -1
};
```

Rules governing explicit assignment:

- Any constant may be assigned an explicit integer value;
- Following constants continue counting from the last assigned value unless specified otherwise.

Example :

```
enum Test {
      X = 10 ,
      Y ,
      Z = 20 ,
      W
};
```

Resulting values:

X → 10

Y → 11

Z → 20

W → 21

1.2.5 Enumerations versus integers

Consider the following two approaches:

Using integers:

```
int status = 1;
```

Using an enumeration:

```
enum Status { INACTIVE,  ACTIVE };

enum Status status = ACTIVE;
```

The second approach is superior because:

- The meaning of ACTIVE is explicit,
- The code is easier to maintain,
- The risk of using an invalid value is reduced.

1.2.6 Using typedef with enumeration

A type definition associates a new identifier with an existing type:

```
typedef existing_type NewTypeName;
```

Example:

```
typedef enum {
    LOW,
    MEDIUM,
    HIGH
} Priority;
```

A variable of type Priority can be declared as:

```
Priority p;
```

A value can then be assigned to this variable as:

```
p = MEDIUM;
```

1.2.7 Use of enumerations in control structures

Enumerations are commonly used with conditional and selection structures such as if and switch.

Example using if :

Enumeration representing simple states :

```
typedef enum {
          OFF,
          ON
      }State;

      State deviceState = ON;
      if (deviceState == OFF) {
              printf("The device is turned off.\n");
      } else if (deviceState == ON) {
              printf("The device is turned on.\n");
      }
```

Example using switch:

Enumeration representing the result of comparing two numbers :

```
    typedef enum {
          LESS,
          EQUAL,
          GREATER
      }Comparison;

    int a, b;
    Comparison result;

    printf("Enter two integers: ");
    scanf("%d %d", &a, &b);

    if (a < b) {
        result = LESS;
    } else if (a == b) {
        result = EQUAL;
    } else {
        result = GREATER;
```

```
    }
```

Structured control using switch ... case :

```
switch (result) {
      case LESS:
          printf("The first number is less than the second.\n");
          break;
      case EQUAL:
          printf("The two numbers are equal.\n");
          break;
      case GREATER:
          printf("The first number is greater than the second.\n");
          break;
    }
```

1.2.8 Complete example: Classification of an Integer as Even or Odd

In many mathematical algorithms, it is necessary to distinguish between integers according to simple arithmetic properties. One of the most fundamental classifications partitions the set of integers into **even** and **odd** numbers. Since this classification consists of a **finite and well-defined set of cases**, it can be naturally represented using an enumeration.

The following complete example implements the mathematical rule:

- an integer is **even** if $nmod2 = 0$,
- otherwise, it is **odd**.

The complete example is given by :

```
#include <stdio.h>

/* Enumeration representing the parity of an integer */
typedef enum{
    EVEN,
    ODD
}Parity;

/* Determines the parity of an integer */
Parity parity_of(int n) {
    if (n % 2 == 0) {
        return EVEN;
    } else {
```

```
        return ODD;
    }
}

int main(void) {
    int n;
    Parity p;

    printf("Enter an integer: ");
    scanf("%d", &n);

    p = parity_of(n);

    /* Structured control using switch ... case */
    switch (p) {
        case EVEN:
            printf("The integer is even.\n");
            break;

        case ODD:
            printf("The integer is odd.\n");
            break;
    }

    return 0;
}
```