SWEG2102 Fall 2026

Fundamentals of Programming II

Lecture 06

User-Defined Data Types
Part II – Advanced

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia

C++ **Standard ISO/IEC 14882**
Programming Language

Outline

Topics Covered

01 Nested Structures

02 Structures and Pointers

03 Structures and Function

04 Unions

05 Enumeration

06 Practical Exercises

07 Common Pitfalls and Best Practices

Learning Objectives

By the end of this lecture, you will be able to:

- Design and use nested structures to model complex real-world data.
- Manipulate (access and modify) structures using pointers.
- Pass structures to and from functions using parameter techniques.
- Differentiate between structures and unions, select appropriate representation
- Define and use enumerations (enum) to represent symbolic constants.

Advanced Structures

1.1 Nested Structure

Composing structures within structures for complex data modeling

Core Concept

- **Structures** can contain other structures **as members**, allowing you to model hierarchical data relationships.
- This is essential for representing complex entities **like students with date of birth, addresses, and contact information**.

Use cases:

(1) A nested structure is created when the member of a structure is itself a structure (variable).

(2) When the structure definition placed within an other structure definition *[not good practice].*

Cont'd

Example 1: use-case / option 1

```
struct Date {
    int y, m, d;
};

struct Student {
    int id;
    string name;
     Date dob;  // Nested structure
};
```

1. Structure within Structure:

→ Defined a **Date** structure, then use it as a member in **Student** structure

2. Chained Access

→ Use **dot operator** to access nested members: stud.dob.y where '**stud**' is an instance of student structure

3. Memory Layout

→ Nested members are also stored contiguously in memory

Cont'd

Example 2: use-case / option 2

```
struct moment  {
    int eventID;
    char eventVenue[20];

   // Nested structure
    struct date {
        int day, month, year;
     };

    struct time {
        int second, minute, hour;
     };
};
```

Structure definition within Structure:

→ Provide **Date** and **Time** structure definition within the **moment**.

→ this is not a good practice

Accessing Members of Nested Structure

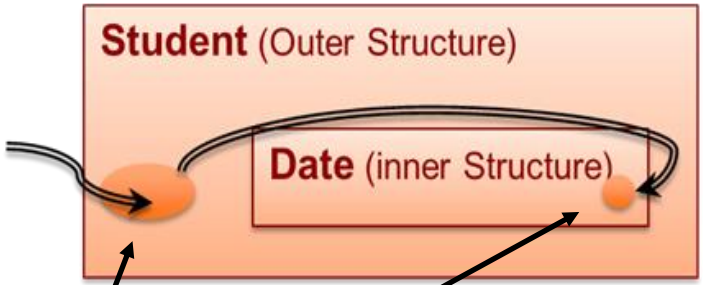
```
struct date {
        int day, month, year;
    };

struct Student {
    char name[20];
    int age;
     float mark;
     Date dateOfBirth;
};
```

- Let's define a **Student** structure and initialize it

  Student stud;

- Then assign values for dateOfBirth

  stud. dateOfBirth.day = 23**;**
  stud. dateOfBirth.month = 5**;**
  stud. dateOfBirth.day = 1940**;**



- **Multiple dot operators** are used.
  - *The **1st dot operator** refers the member variable of outer structure*
  - *The **2nd dot operator** refers the inner structure and so on.*

Cont’d

Complete Program Example:

```cpp
#include <iostream>
#include <string>
using namespace std;

// Date structure for nested usage
struct Date {
    int year, month, day;
};

// Student structure with nested usage
struct Student {
    string name;
    int id;
    Date dob; // Nested usage
    float gpa;
};

void displayStudent(Student s) {
    cout << "Name: " << s.name << endl;
    cout << "DOB: " << s.dob.year << "/"
         << s.dob.month << "/"
         << s.dob.day << endl;
    cout << "ID: " << s.id << endl;
    cout << "GPA: " << s.gpa << endl;
}
```

```cpp
int main() {
    // Method 1: Initialize using member access
    Student student1;
    student1.name = "John Doe";
    student1.id = 12345;
    student1.dob.year = 2000;
    student1.dob.month = 5;
    student1.dob.day = 15;
    student1.gpa = 3.75;

    // Method 2: Initialize using curly braces (C++11 and later)
    Student student2 = {"Jane Smith", 67890, {2001, 8, 22}, 3.92};

    // Display all students
    cout << "Student 1:" << endl;
    displayStudent(student1);

    cout << "\nStudent 2:" << endl;
    displayStudent(student2);

    return 0;
}
```

1.2 Pointers and Structures

Key Idea

- A pointer can store the address of a structure variable
- This is called **pointer to structure**

Declaration:

```
Struct_tag *ptr;
```

Example:

```
Struct_tag *ptr;
```

👉 ***ptr*** *is a pointer that can point to a Student structure.*

Accessing Members Using Pointer:

👉 Method 1: Dereferencing + Dot Operator

- ***(*ptr).name***
- ****ptr*** *→ access structure*
- ***.name*** *→ access structure member*
- Access member via dereference

👉 Method 2: Arrow Operator (->) (Preferred)

- ***ptr->name***
- *Direct access (shortcut for method 1)*

Cont’d

Example:

Demonstrates **how to access structure members** using a pointer via

- ✓ **dereferencing (*)** and
- ✓ **arrow operator (->)**.

```
struct Student {
    int id;
    string name;
    float grade;
};

int main() {

    Student s1 = {1, "Sara", 90};
    Student *ptr = &s1;

    // ----------------------------
    // Method 1: Dereferencing + Dot Operator
    // ----------------------------
    cout << "Method 1 ( *ptr ).name: " << (*ptr).name << endl;
    cout << "Method 1 ( *ptr ).grade: " << (*ptr).grade << endl;
    cout << "----------------------------" << endl;

    // ----------------------------
    // Method 2: Arrow Operator (->)
    // ----------------------------
    cout << "Method 2 ptr->name: " << ptr->name << endl;
    cout << "Method 2 ptr->grade: " << ptr->grade << endl;

    return 0;
}
```

1.3 Structures and Functions

Passing structures to functions and returning structures from functions

Both structure variables and structure elements can be **passed** to a function and **returned** in a similar way as normal arguments.

(I) Passing Structures to Functions:

- 👉 **Pass by value:** *A copy of the structure is created and passed to the function and the original structure **remains unchanged**.*
- 👉 **Pass by reference:** *The function receives a **reference** to the original structure, allowing **direct modification** without copying.*
- 👉 **Pass by pointer:** *The function receives the **memory address** of the structure, enabling **indirect access** and modification.*

(II) Return Values: *Entire structures can be returned from functions to **assign to new variables** or **update existing ones**.*

Cont'd

Example Code: Passing structure to function – by value, reference & pointer

```cpp
#include <iostream>
using namespace std;

struct Student {
    string name;
    int id;
    float gpa;
};

// Pass by Value (copy is created)
void passByValue(Student s) {

// Pass by Reference (original is modified)
void passByReference(Student &s) {

// Pass by Pointer (original is modified)
void passByPointer(Student *s) {
```

```cpp
int main() {
    Student s = {"Alice", 101, 3.5};

    cout << "[Before any call] " << s.name << ", "
         << s.id << ", " << s.gpa << endl << endl;

    passByValue(s);
    cout << "[After passByValue] " << s.name << ", "
         << s.id << ", " << s.gpa << endl << endl;

    passByReference(s);
    cout << "[After passByReference] " << s.name << ", "
         << s.id << ", " << s.gpa << endl << endl;

    passByPointer(&s);
    cout << "[After passByPointer] " << s.name << ", "
         << s.id << ", " << s.gpa << endl;

    return 0;
}
```

```cpp
// Pass by Value (copy is created)
void passByValue(Student s) {
    s.id = 999;
    s.gpa = 4.0;
    cout << "[Inside passByValue] " << s.name << ", "
         << s.id << ", " << s.gpa << endl;
}
// Pass by Reference (original is modified)
void passByReference(Student &s) {
    s.id = 888;
    s.gpa = 3.8;
    cout << "[Inside passByReference] " << s.name << ", "
         << s.id << ", " << s.gpa << endl;
}
// Pass by Pointer (original is modified)
void passByPointer(Student *s) {
    s->id = 777;
    s->gpa = 3.6;
    cout << "[Inside passByPointer] " << s->name << ", "
         << s->id << ", " << s->gpa << endl;
}
```

OUTPUT

```
[Before any call]       Alice, 101, 3.5

[Inside passByValue]  Alice, 999, 4
[After passByValue]   Alice, 101, 3.5

[Inside passByReference] Alice, 888, 3.8
[After passByReference]  Alice, 888, 3.8

[Inside passByPointer]  Alice, 777, 3.6
[After passByPointer]   Alice, 777, 3.6
```

Cont'd

Comparison Summary: Passing Structure to Functions

Method	Syntax	Original Data	Memory Overhead	Best For
1 Pass by Value	`void func( Student s)`	Unchanged (copy used inside function)	**High** (copy of structure is created)	Small structures, read-only operations
2 Pass by Reference	`void func( Student &s)`	**Modified** (if changed in function)	**Low** (no copy)	Large structures, when modification is needed
3 Pass by Pointer	`void func( Student *s)`	**Modified** (if changed in function)	**Low** (no copy)	Dynamic data, optional/NULL handling, flexibility

Key Insight:

- **Value** → Safe but expensive (creates copy)
- **Reference** → Efficient and preferred for modification
- **Pointer** → Most flexible but requires careful handling (NULL, validity)

Cont’d

Example Code: Returning Structures from Functions

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Student {
    string name;
    int id;
    float gpa;
};

// Returns a normalized COPY of the structure
Student normalize_copy(Student s) {
    if (s.gpa > 4.0f) s.gpa = 4.0f;
    if (s.gpa < 0.0f) s.gpa = 0.0f;
    return s;
}

// In-place modification using reference
void set_honors(Student& s) {
    if (s.gpa >= 3.7f)
        s.name += " [Honors]";
}
```

```cpp
int main() {

    // Structure initialized directly in main
    Student a = {"Dawit", 1102, 3.85f};

    cout << a.name << ", id=" << a.id << ", gpa=" << a.gpa << endl;

    Student b = normalize_copy(a);

    cout << "After normalize_copy: "
         << "a.gpa=" << a.gpa
         << ", b.gpa=" << b.gpa << endl;

    set_honors(a);

    cout << "After set_honors: "
         << a.name << ", gpa=" << a.gpa << endl;

    return 0;
}
```

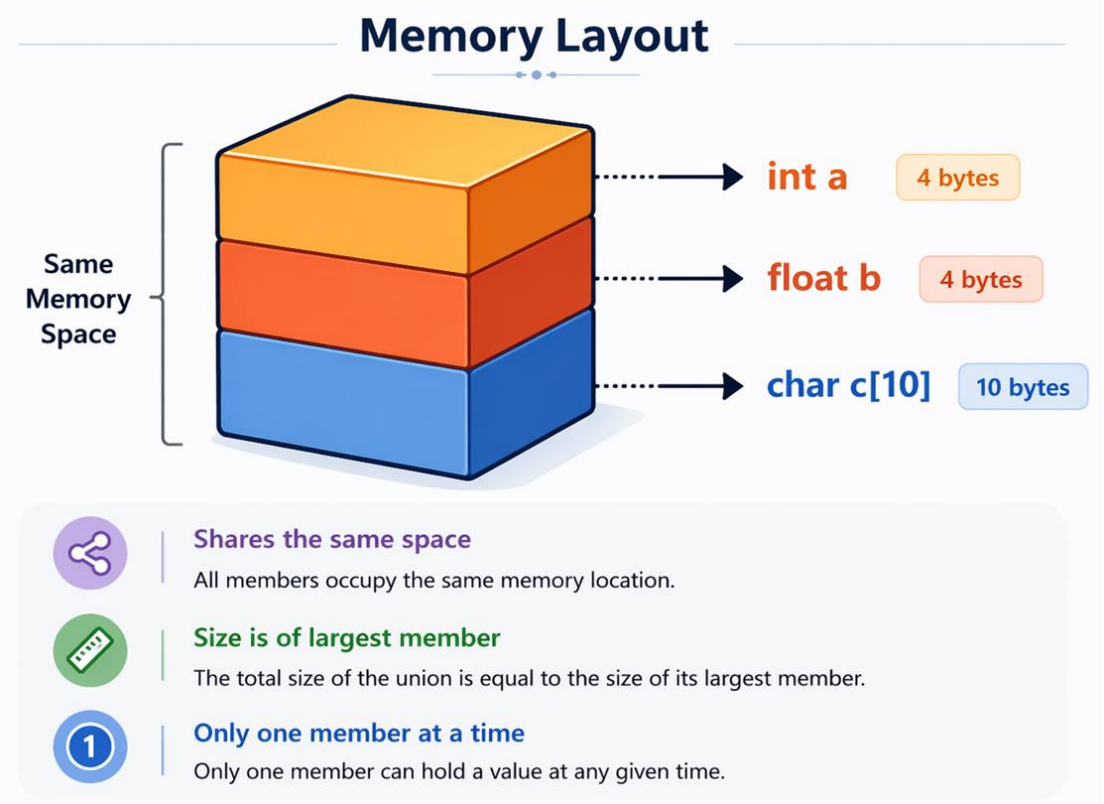
```
Output
Dawit, id=1102, gpa=3.85
After normalize_copy: a.gpa=3.85, b.gpa=3.85
After set_honors: Dawit [Honors], gpa=3.85
```

Unions

2.1 Definition and Syntax

Core Concept

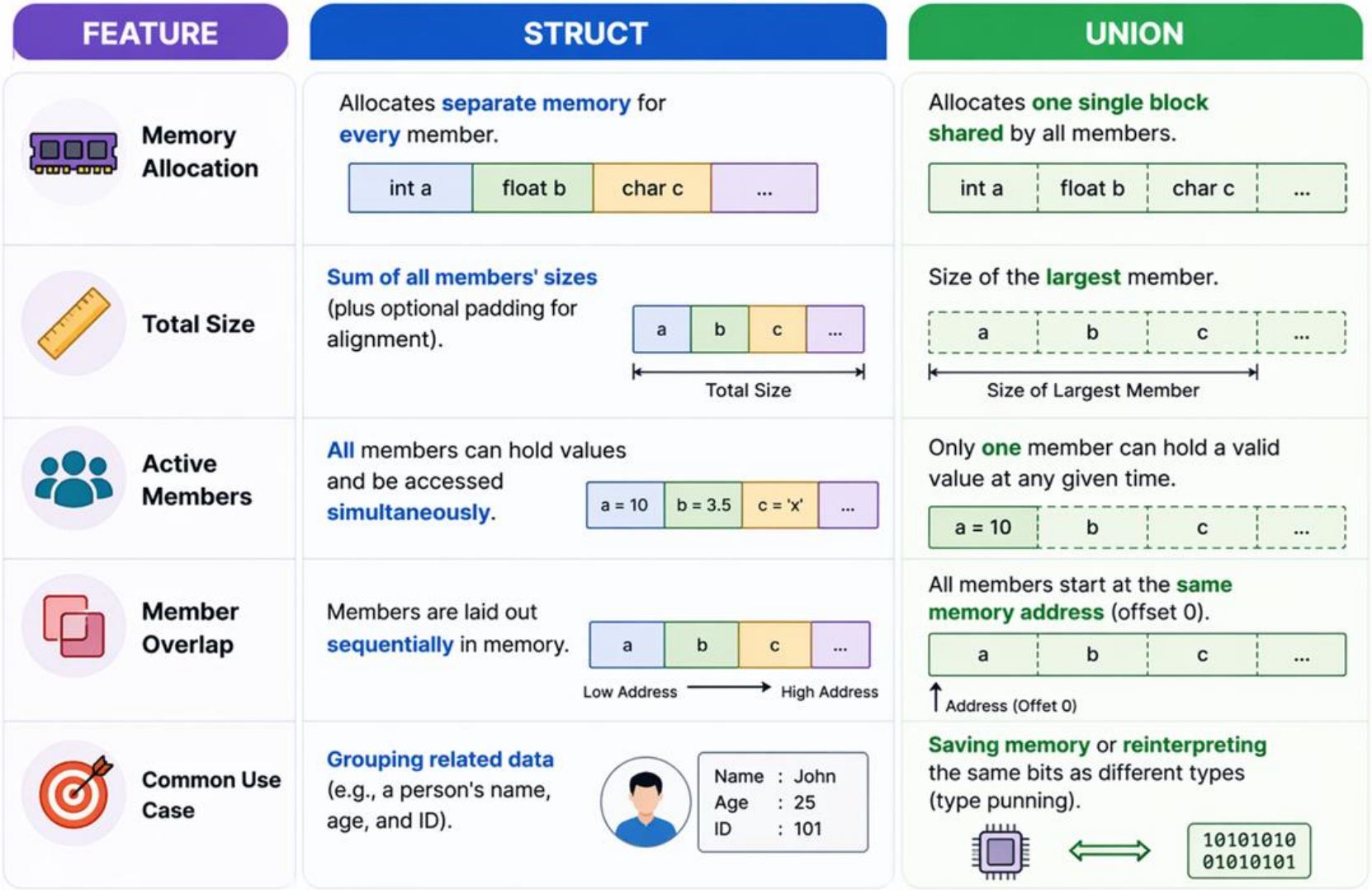
- A **union** is a **user-defined data type** in which we can define members of different types of data types just like **structures**.
  - 👉 *All members share **the same memory**, which means you can only use one of the values at a time.*
- Declared and used **in the same way** as structures, except here we use the keyword '**enum**'
- *Similarly, the members of union can be accessed using **dot operator (.)***



Source: *AI generated image (ChatGPT 5.3, 2026)*

Cont'd

Structure Vs. Union



Source: *AI generated image (ChatGPT 5.3, 2026)*

Cont'd

Union definition and variable declaration

- Once a union is declared/defined, you can **create variables** of the new type as follow.

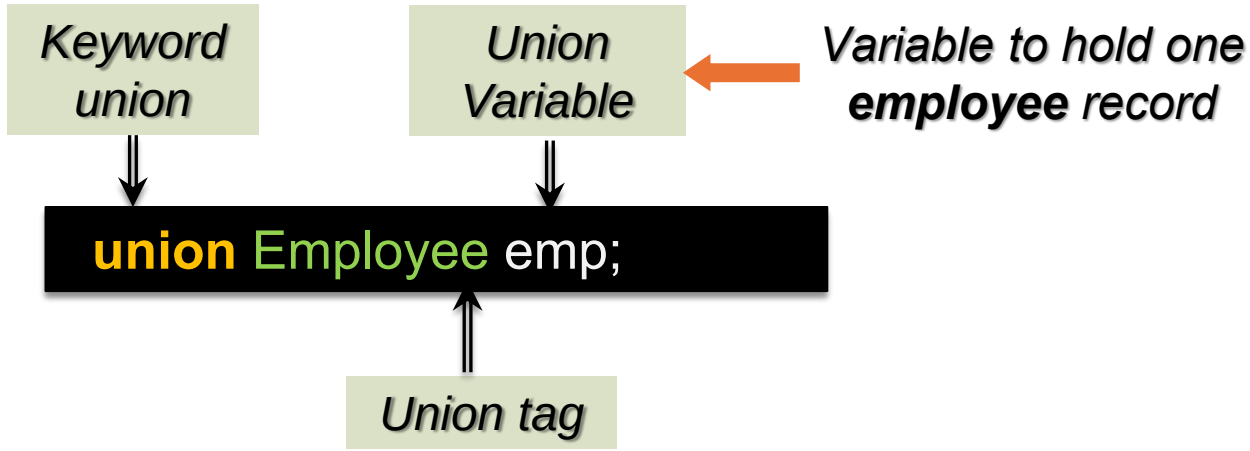
Syntax:

<union> <union_tag> <identifier>; or

< union_tag > <identifier>;

Example:

```
union Employee {
        int EmpID;
        char Name[25],
        char Gender;
        char Position;
        float Salary
 };
```

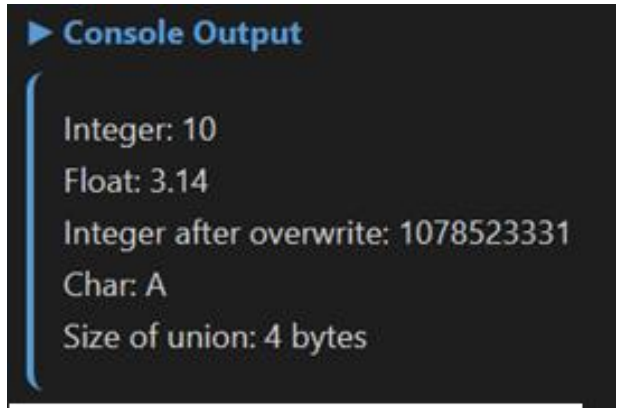


Cont'd

Example:

Demonstrate Union memory behavior

```cpp
#include <iostream>
using namespace std;

// Define union
union Data {
    int i;
    float f;
    char c;
};

int main() {
    Data d;

    d.i = 10;
    cout << "Integer: " << d.i << endl;

    d.f = 3.14;
    cout << "Float: " << d.f << endl;

    // Previous value overwritten
    cout << "Integer after overwrite: " << d.i << endl;

    d.c = 'A';
    cout << "Char: " << d.c << endl;

    cout << "Size of union: "
         << sizeof(Data) << " bytes" << endl;

    return 0;
}
```



Note:

- ✓ The value **1078523331** is garbage/undefined
  - 👉 *This is because the memory was **overwritten** when assigning 3.14 to **f**, and then interpreted as an **int**.*

2.2 Application of Union

👉 Use cases of **Union:**

✅ Memory Optimization in Embedded Systems

- Used in **microcontrollers** and IoT devices reduce RAM usage by storing mutually exclusive values in the same memory region.

✅ Graphics Programming / Color Representation

- Used to **represent pixel data** as either a 32-bit integer (fast operations), or individual RGBA components (fine control).

✅ Network Protocol and Packet Parsing

- Used in low-level systems to **interpret the same memory buffer** in multiple structured formats depending on protocol

✅ Hardware Register Mapping:

- Used in embedded systems to map hardware registers as both raw values and bit-fields for control/status access..

Enumerations (enum)

3.1 Enum Definition & Syntax

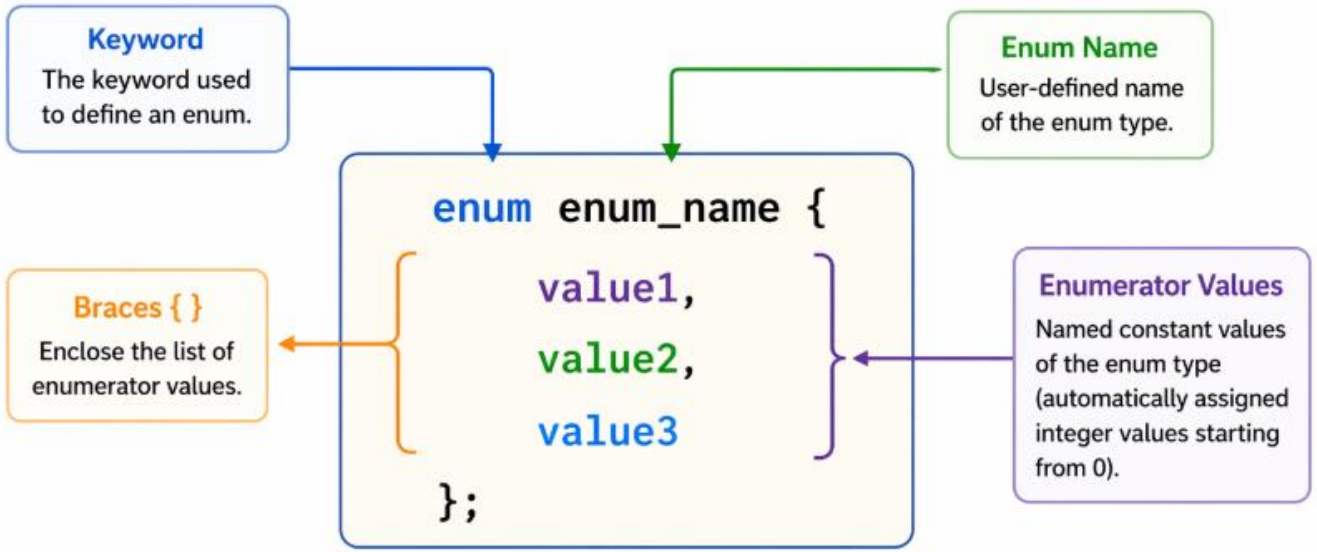
Core Concept

➢ A **enumerations (or enum)** is a **user-defined data type** used to create new data types of **multiple integer constants**.

👉 Used to set up collections of **named integer constants**.

👉 In other term, used to **assign meaningful names** to integer values.

Syntax



Note: *By default, the first enumerator is assigned value 0, the next value 1, and so on. However, you can also assign specific values to enumerators.*

Cont'd

Example

```
enum Level {
    LOW,
    MEDIUM,
    HIGH
};
```

Description

- **enum** - keyword indicate declaration of enumeration
- **Level** - name of the enumeration
- The **enum items** are enclosed within curly brace & separated by a comma
- **Uppercase** is preferred for naming the enum items
- Here, the name **LOW** is automatically assigned 0, **MEDIUM** is assigned 1 and **HIGH** is assigned 2n.

Note: ***enum*** *declaration is only a* ***blueprint*** *for the variable is created and does not contain any fundamental data type.*

3.2 Create enum Variables

- When we define an **enum type**, the blueprint for the variable is created but memory is not allocated.
- To access the **enum items**, we must create a variable of the defined enum type.
  - 👉 *We can **create variables** of enum by using its specified name.*
- An **enum variable** can be initialized/assigned value either with a item-name provided in the enum definition or directly with its integer value.

Note: *An enum variable can take only one value.*

Example code

```cpp
#include <iostream>
using namespace std;

// Enum declaration (defines a new data type)
enum Week {
    Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday
};

int main() {
    // Variable creation (memory is allocated here)
    Week today;

    // Assigning a value
    today = Wednesday;

    cout << "Today (numeric value): " << today << endl;

    return 0;
}
```

Cont’d

Complete Example

```cpp
#include <iostream>
using namespace std;

// Enum 1: Color (default values start from 0)
enum Color {
    Red,        // 0
    Green,      // 1
    Blue,       // 2
    Yellow,     // 3
    Black = 10, // custom value
    White       // 11 (auto-increment from Black)
};

// Enum 2: FontStyle (mixed assignment)
enum FontStyle {
    Regular = 1,
    Bold = 2,
    Italic = 5,
    Underline,   // 6 (auto-increment)
    StrikeThrough // 7
};
```

```cpp
int main() {
    // Creating enum variables
    Color bgColor;
    FontStyle textStyle;

    // Assign using enum names
    bgColor = Green;
    textStyle = Bold;

    cout << "Background Color (Green): " << bgColor << endl;
    cout << "Text Style (Bold): " << textStyle << endl;

    // Assign using custom values
    bgColor = Black;
    textStyle = Italic;

    cout << "\n=== After Reassignment (Custom Values) ===\n";
    cout << "Background Color (Black): " << bgColor << endl;
    cout << "Text Style (Italic): " << textStyle << endl;

    // Assign using integer values
    bgColor = Color(11);
    textStyle = FontStyle(6);

    cout << "Background Color (White = 11): " << bgColor << endl;
    cout << "Text Style (Underline = 6): " << textStyle << endl;

    // Size of enums
    cout << "Size of Color enum: " << sizeof(Color) << " bytes" << endl;
    cout << "Size of FontStyle enum: " << sizeof(FontStyle) << " bytes" << endl;

    return 0;
}
```

3.3 Application of Enum

👉 **Enums** are extensively used in various real-world applications in programming

✓ State Representation:

- Enums are commonly used to **represent different states or modes** in state machines, like in game development or process control systems.

✓ Error Codes:

- Enums are useful for **defining a set of related error codes** with meaningful names, making the code easier to debug and maintain.

✓ Menu Choices or User Inputs:

- Enums are useful for handling **menu options or user input** and can represent variety of user input such as days of the week.

✓ File Permissions:

- Enums are useful for handling file permissions in an operating system or file management system, with different permission levels such as READ, WRITE, and EXECUTE.

Quick Check and Practice Exercise

Quick Check: Pause & Predict

1. What will happen in the following code?

```
Student s = {101, "Abel", 3.5};
Student *ptr = &s;

ptr.id = 200;
```

Answer: Error --> *the code will not compile because of wrong operator use.*

Correct use: ***ptr->id = 200;***

2. Predict the output **?**

```
enum Department { CS, IT, SE };

cout << IT;
```

Answer: 1 → *Enum values are integers and considering the default indexing*

3. Which is more appropriate **(struct or union)** and why?

👉 **Scenario:** A student has GPA & Total Marks

Answer: Structure → *Both values needed simultaneously*

Cont'd

4. What value is printed with this code segment?

```
union Result {
    float gpa;
    int marks;
};

Result r;
r.gpa = 3.5;
r.marks = 80;
cout << r.gpa;
```

Answer: Undefined/garbage output

→ *Because union shares memory*

5. Which combination is most appropriate for designing a university system?

A. Flat structure with all fields

B. Nested structures + enum + functions

C. Union-based design

Answer: B → *Real-world modeling, Maintainability & Type safety.*

Practice Exercise

Exercise #1 — Student Record Management System

Objective:

To design and implement an **Student Record Management System** in C++ using **user-defined data types (structures)** and related programming concepts.

👉 This exercise will allow you progressively apply:

- ✓ Structures and nested structures
- ✓ Arrays of structures
- ✓ Pointers for structure manipulation
- ✓ Functions with different parameter passing techniques
- ✓ Enumerations for symbolic constants

Cont’d

Exercise #1 — Student Record Management System

Requirements

- The structure should store up to **100 student records** with the following fields:

1. Basic Information

- ✓ Student ID (int),
- ✓ Name (string),
- ✓ GPA (float),

2. Additional – nested structure

- ✓ Date of Birth
  -> {day (int), month(int), year(int)}
- ✓ Address
  -> {city (string) & country (string)},

3. Enumeration

- ✓ Department
  e.g. CS, IT, SE, etc.

- **Features**
  - ✓ Add student,
  - ✓ List all students,
  - ✓ Search by Id
  - ✓ Update Student GPA

Cont’d

Solution : *Student Record System Management Implementation*

```cpp
#include <iostream>
#include <string>
using namespace std;

// constant defined for specifying array size
#define MAX 100

// ENUMERATION (Symbolic constants for Department)
enum Department { CS, IT, SE, IS };

// STRUCTURES
struct Date {
    int day, month, year;
};

struct Address {
    string city;
    string country;
};

struct Student {
    int id;
    string name;
    float gpa;
    Department dept;
    Date dob;
    Address address;
};
```

```cpp
// FUNCTION PROTOTYPES
void addStudent(Student students[], int &count);
void listStudents(Student students[], int count);
void searchStudent(Student students[], int count);
void updateGPA(Student &s);
void displayStudent(Student s);
string getDeptName(Department d);

int main() {
    Student students[MAX];
    Student *ptr = students;
    int count = 0;
    int choice;

    do {
        cout << "\n===== Student Record System =====\n";
        cout << "1. Add Student\n";
        cout << "2. List Students\n";
        cout << "3. Search Student\n";
        cout << "4. Update GPA\n";
        cout << "0. Exit\n";
        cout << "Enter choice: ";
        cin >> choice;
```

The complete Solution Found Here

[Fundamentals Programming II] Week 6 - Practical Exercise Solutions.pdf

Common Pitfalls and Best Practices

Common Pitfalls

✕ Nested Structures

- ✓ Deeply nested structures
- ✓ Poor naming of nested members

✕ Passing Struct to and from Functions

- ✓ Passing large structures by value
- ✓ Modifying data unintentionally
- ✓ Returning pointers/references to local variables

✕ Manipulating Structs through pointers

- ✓ Dereferencing null or uninitialized pointers
- ✓ Overuse of pointers where not needed

✕ Union Data Types

- ✓ Accessing a member that is not currently active
- ✓ Misusing unions where safer alternatives exist

✕ Enumerations (enum)

- ✓ Using raw integers instead of enum values
- ✓ Mixing unrelated enum types
- ✓ Not handling all enum cases

Best Practices

✓ Nested Structures

- Keep structures simple and shallow
- Use clear, domain-based naming for struct members
- Design structures for **single responsibility**

✓ Passing Struct to and from Functions

- Use const reference **(const &)** for read-only access
- Use reference (&) or pointer when modification is required
- Return by value when safe

✓ Manipulating Structs through pointers

- Always initialize pointers before use
- Validate pointer before dereferencing

✓ Union Data Types

- Use unions only for specific memory optimization
- Avoid unions when type safety is critical

✓ Enumerations (enum)

- Use **enums** to represent fixed sets of integer constants only
- Handle all enum values explicitly

References

Textbooks

- **C++ How to Program** [10th edition], Deitel, P. & Deitel, H., Global Edition, Global Edition (2017).
- **Problem Solving With C++** [10th edition], Walter Savitch, University of California, San Diego, 2018.

Reference Books

- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup, Addison-Wesley, 2014.
- **An Introduction to Programming with C++** (8th Edition), Diane Zak, Cengage Learning, 2016

Online Resources

- https://www.geeksforgeeks.org/cpp/c-plus-plus/
- https://www.w3schools.com/cpp/default.asp
- https://programiz.pro/resources/cpp
- https://www.hackerrank.com/domains/cpp
- https://cplusplus.com/doc/tutorial/

💡 ***Study Tip:*** *Don't just read the code! Retype the examples from these slides and resources into your IDE, compile them, and modify them to see what happens.*

Thank You!

Chere Lemma (M.Tech)
Lecturer, Dept. of Software Engineering

Addis Ababa Science and Technology University (AASTU)
Addis Ababa, Ethiopia