SWEG2102 Fall 2026

Fundamentals of Programming II

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

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

Lecture 05

User-Defined Data Types
Part I - Structures

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

Outline

Topics Covered

Learning Objectives

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

- Define user-defined data types and explain the need for it.
- Define and declare structures, and create variables for different data types.
- Initialize, and access structures to model complex data effectively.
- Apply best practices and identify common pitfalls in the use of structures.
- Develop small programs that integrate structures with function concepts.

Introduction

1.1 What are User-Defined Data Types?

Definition

- **UDTs** are Data types created by programmers
- Allow **grouping** of different data types and representing in a meaningful way
- Enable you to **combine multiple variables** into a single logical unit instead of dealing with them separately

Key Points:

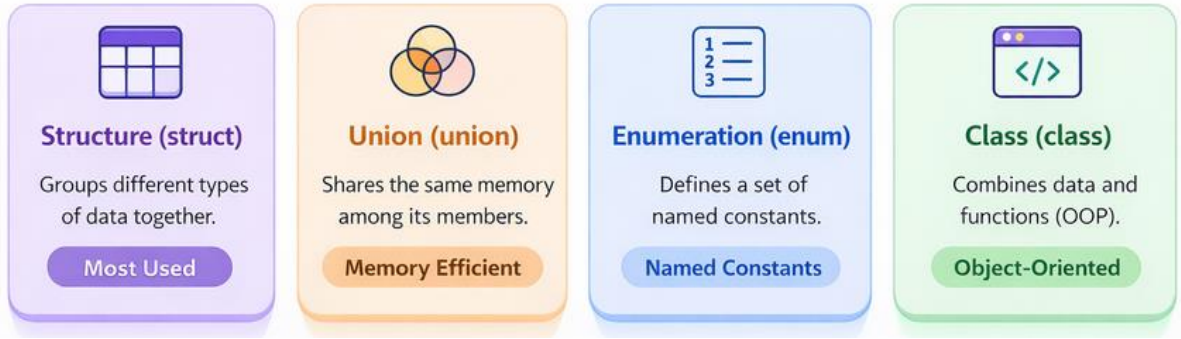
👉 **Built-in Types**

→ Basic building blocks

👉 **User-Defined Types (UDTs)**

→ Custom structures built from the basic blocks

Common Types of UDTS



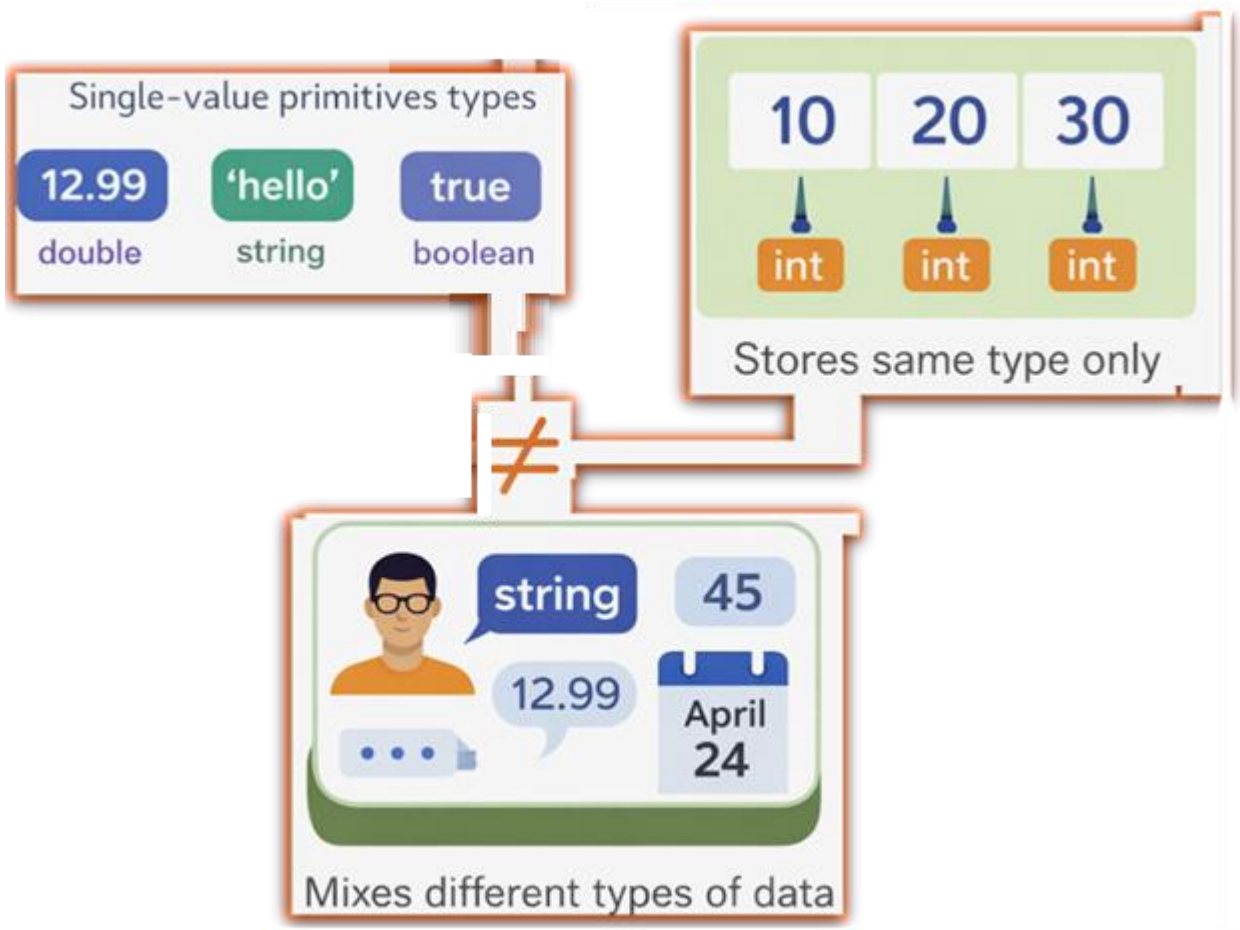
1.2 The Need of User Defined Data Types

Why User-Defined Data Types?

💡 Limitations of primitive types and arrays

- **Primitive types:**
  - → store **single value**
- **Arrays:**
  - → Store same type only
  - → Cannot mix different data types

👉 Real-world data is **heterogeneous**



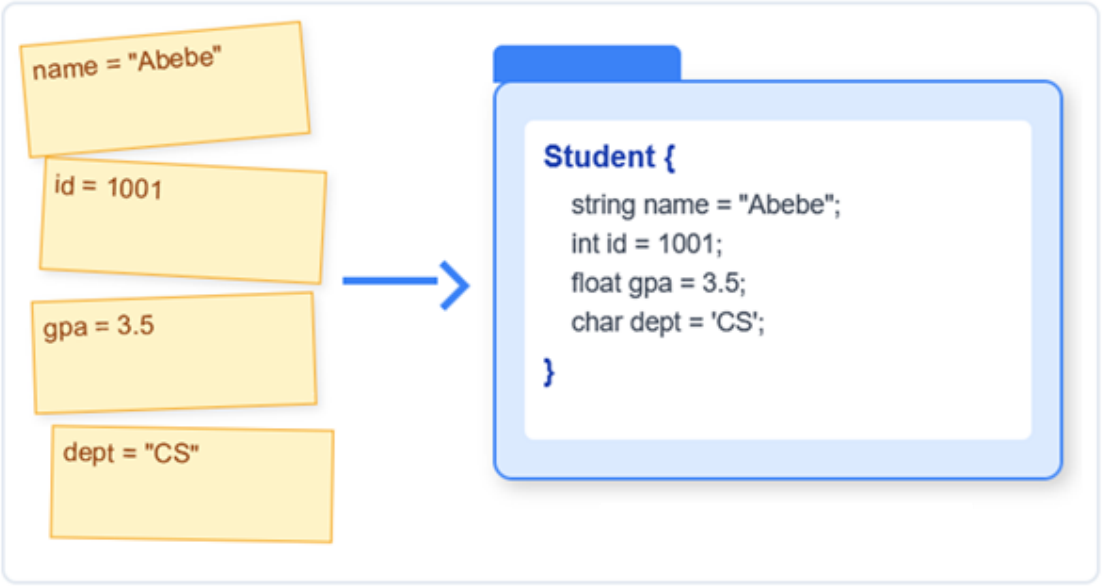
Cont’d

Scenario: Student Information Management

Imagine you need to manage information for a student: *name, ID number, GPA, and department*.

→ Using **primitive types** and **arrays**, you would need to declare separate variables for each piece of data

```
string studentName = "Amanuel";
int studentID = 1023;
float studentGPA = 3.45;
string department = "Computer Science";
// Problem: 4 separate variables for 1 student!
```



Loose Variables
→ Separate, scattered

Organized Structure
→ Grouped, organized

Cont’d

More Limitation of Primitive DT & Array

- Scattered State
  - ✓ *Related data is **spread/scattered** across multiple variables*
  - ✓ No logical grouping of related information
- Poor readability
  - ✓ Variable names don't indicate relationships
  - ✓ And hard to understand relationships

- Scaling issue
  - ✓ *Managing 100+ students becomes a nightmare with 400+ separate variables to track and maintain*
- Parameter passing issue
  - ✓ Multiple parameters to function
  - ✓ **E.g.** Functions might need 4 separate arguments just to **pass 1 student's** complete information

Cont'd

The Solution: UDTs

Key Idea: Group related data into a single unit

- **E.g.** You can define a **Student** structure that **groups related fields** into a single, cohesive unit
- So, instead of managing 4 separate variables **(name, id, gpa, dept),** you create **1 structure** that contains all related data

Using UDTs results - in better code quality

- **Cohesion**:
  - ✓ *All related data grouped together and no more scattered variables*
- **Clarity**:
  - ➢ *Clear data relationships that is easy to understand structure*
- **Scalability**:
  - ✓ Handle 100+ records efficiently using Arrays of structures for bulk data
- **Maintainability**:
  - ✓ *Easier to modify*

1.3 Benefits of User-Defined Data Types

Key benefits of UDTs

Model real-world entities

*Represent real-world objects with their **attributes** and **behaviors** in a natural way.*

Reduce complexity

Bundle multiple variables into one structure, avoiding scattered and repetitive code.

Flexibility

Allow creation of custom data structures tailored to specific problem requirements.

Improve organization & readability

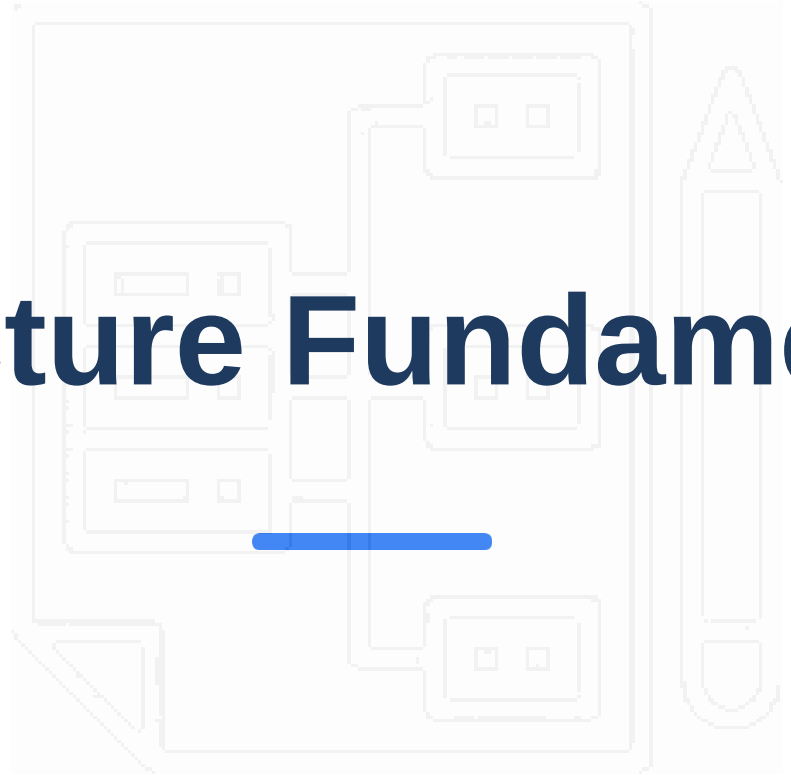
Group related data into a single unit, making code easier to understand and maintain.

Support modular programming

Enable breaking programs into smaller, manageable, and reusable components.

Reusability

Once defined, UDTs can be reused across programs or modules, saving development time



Structure Fundamentals

2.1 Structure Definition & Syntax

Core Concept

- A **structure** is a **user-defined data type** that groups **related variables** of different data types under a single name.
  - 👉It allows you to create **composite data types** that represent real-world entities.

Key characteristics:

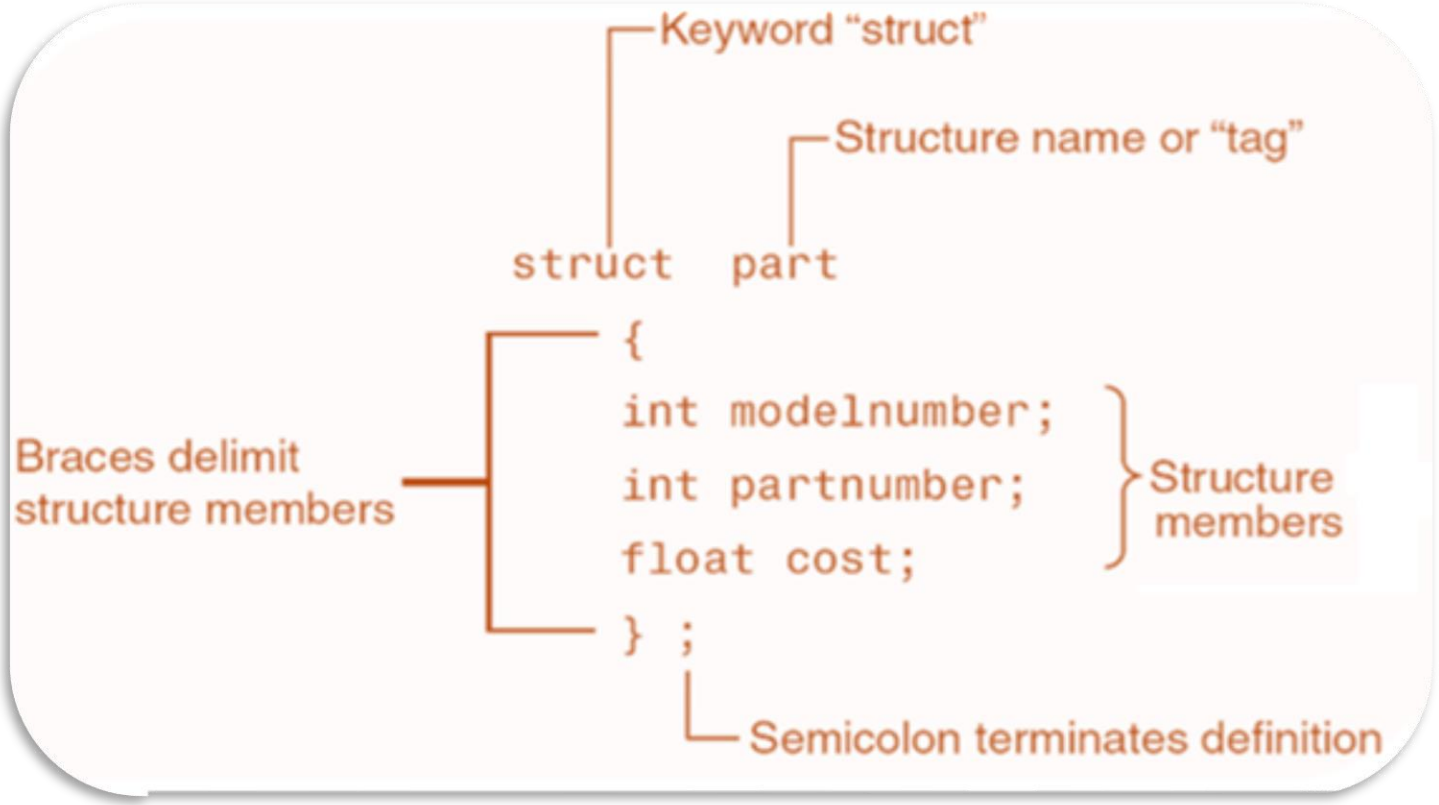
- Groups related data together
- Members can be of different types
- Creates a new data type
- Memory is allocated for all members

Syntax

```
struct StructName {
    data_type member1;
    data_type member2;
    data_type member3;
    // ... more members
}; // Don't forget the semicolon!
  // The semicolon after the closing
    brace is mandatory in C++!
```

Cont'd

Example and Visualization of Declaration/Definition



Descriptions

👉 **Tag**

- *Name of newly created user defined **Data Type***

👉 **Structure members (or fields)**

- *Individual components of structure*
- *Defined with their data-type*
  (simple, array or object type)
- *Must have unique names/identifier*

Cont'd

More Example:

```
struct BankAccount {
        char Name[15];
        int AcountNo[10];
        double balance;
        Date Birthday;
};
```

*The "**BankAccount**" structure has **regular variable**, **array** and **object** types as members.*

```
struct StudentRecord {
        char Name[15];
        int Id;
        char Dept[5], Gender;
};
```

*The "**StudentRecord**" structure has 4 members.*

2.2 Creating Structure Variables

How Structures Become User-Defined Types?

- Once a **structure** is declared, **its name acts** as a **user-defined data type.**
- You can use the **new type** and create/declare **variables, arrays, function parameters, and return types**
- The **struct-tag** is just used like primitive types (int, float, etc.).

Syntax:

<struct> <struct tag> <identifier>;

or

<struct tag> <identifier>;

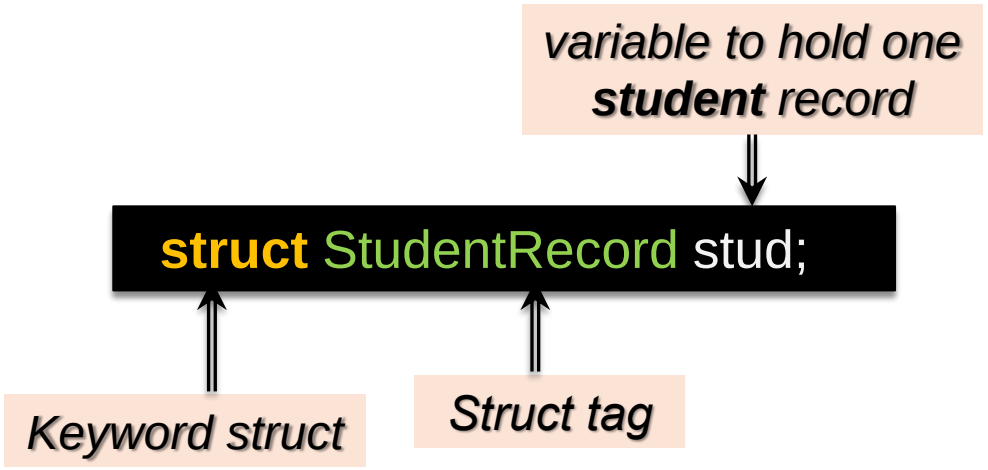
Note:

- ***Structure variable creation*** *tells the compiler to* ***allocate memory space.***
- *And the compiler* ***allocates adequate memory*** *according to the elements of the structure.*

Cont'd

Examples:

```
struct StudentRecord {
        char Name[15];
         char Dept[5];
         char Gender;
        int Id;
        float mark;
};
```




```
struct Book {
    string title;
    string author;
    int yearPublished
    float price;
};
```

```
Book b1  // Variable

Book library [50];     // Array of structures
```

Cont'd

Variable creation with structure definition

- A **structure variable** can also be declared/created alongside the definition of the structure

Example:

```
struct Point {
        int x;
        int y;
} p1, p2;

// p1 & p2 are structure variables
```

Structure without tag but with variable declaration

```
struct {
        char name[20];
        int id;
        float gpa;
} stud1, stud2;
```

*This kind of structure declaration is **not recommendable** because then you **cannot define a structure of this type** somewhere else.*

2.3 Declaration vs Variable Creation

📘 Declaration (Type Definition)

- **Declaration** defines the structure type, specifying the members of the structure
- Tell the complier how the **structure is organized**, but memory is not allocated
- Simply it creates a **blueprint** for the data type

📦 Variable Creation

- **Variable creation** actually creates **instances** of the structure type, allocating memory for the data.
- Each variable gets its own memory space for all members.

Cont’d

Example

Declaration

```
struct Student {
    string name;
    int id;
    float gpa;
    char section;
}; // Type definition complete
```

What it does?

- Creates a new data type called "**Student**“
- Defines the **structure's members**
- **No memory** is allocated at this point

Variable Creation

```
// Creates first student variable
Student s1;

// Creates second student variable
Student s2;

Student s3 =
    {"Liya", 1023, 3.45f, 'A'};
```

What it does?

- **Allocates memory** for the structure
- Creates actual **instances**
- Can be **initialized** with values

2.4 Memory Layout Visualization

Key Points

- *Members stored contiguously in memory*
- *Padding added for alignment requirements*
- *Structure size*
  - ✓ *sum of member sizes + padding*
  - ✓ *E.g. for the **Student structure** provide here on the code example*

    Total size = 32 bytes (29 + 3 padding)

Code Example

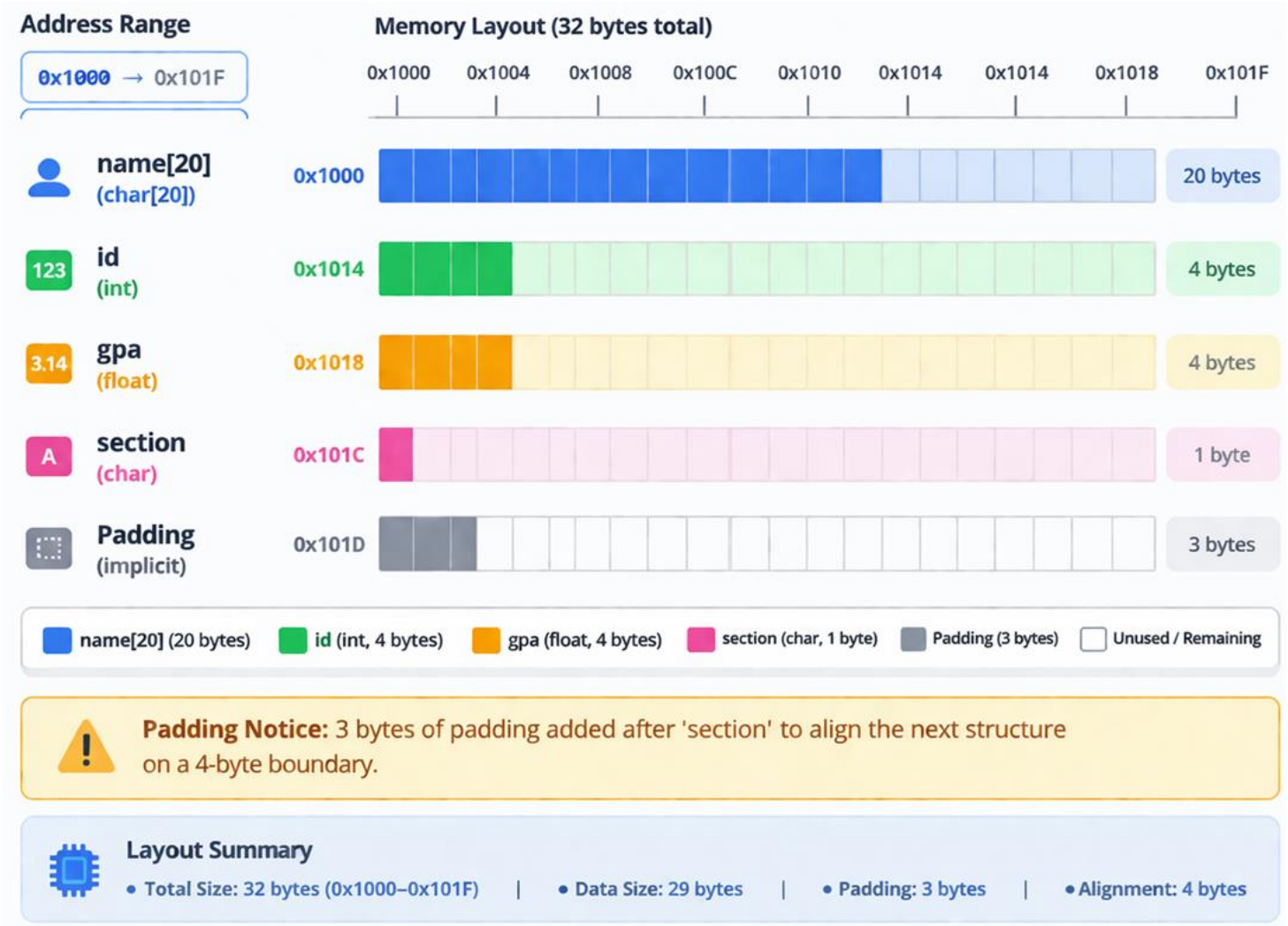
```
struct Student {
        char name[20];          // 20 bytes
        int id;                 // 4 bytes
        float gpa;              // 4 bytes
        char section;           // 1 byte
}; // Total: 29 bytes + padding

int main() {
        Student stud1;
        cout << sizeof(stud1);      // Output: 32
        return 0;
}
```

Cont'd

Memory Layout Visualization

- See how data is stored in memory with **addresses, sizes, and padding**.
- This diagram helps you understand how each **structure field** is arranged **in memory** and how alignment affects storage.



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

Initializing and Accessing Members

3.1 Initializing Structure Variable

- This refers to the **process of assigning values** to *structure elements* during declaration or creation of structure variable

General Syntax: <StructureTag> <identifier> = { *// list of initial values separated by comma* };

*** also known as Brace Initialization*

Example:

```
struct part {
    int modelnumber;
    int partnumber;
    float cost;
};
```

part tyre = { 2011, 34, 13.50};

Key Points:

- Order of values must match member order
- Type-safe initialization with compile-time checking
- No narrowing conversions allowed

Cont’d

Partial Initialization

- You can **initialize only some members** of a structure at declaration time.
- The remaining members will be **value-initialized** (set to zero for numeric types, empty for strings).

Example:

```
struct Student {
    string name;
    int id;
    float gpa;
    char section;
};

// Partial initialization
Student s1 = {"Alice", 1001};
// gpa = 0.0f, section = '\0'

Student s2 = {"Bob", 1002, 3.5f};
// section = '\0'

Student s3 = {};
// All members zero-initialized
```

3.2 Accessing Members (Dot Operator)

- Once a structure is declared, use the **dot operator (.)** to access members of a structure variable.
- The *dot operator* is also called *member access operator.*

Syntax: <StructureVariable>.<MemberVariable>;

Example:

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

Student s1; // Create structure variable
s1.name = "Amanuel"; // Assign value to member
s1.gpa += 0.1f; // Modify member value
cout << s1.id; // Read member value
```

3.3 Array of Structure

Managing Multiple Records with Arrays

- **Arrays of structures** allow you to manage multiple records of the same type efficiently.
- It is essential for real-world applications like student and employee records, or inventory systems.

Example:

Consider this array of ***Student structures*** *to manage multiple student records efficiently.*

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

// Declare array of 5 students
Student roster[5];
```

Access Pattern:

- *Use loops to iterate through the array and process each student record*

```
// Input loop
for (int i = 0; i < 5; i++) {
    cout << "Enter name " << i+1 << ": ";
    cin >> roster[i].name;
    // Access using dot operator
}
```

Quick Check and Practice Exercise

Quick Check: Pause & Predict

Consider the below code segment, answer the following questions

```
struct Book {
    int id;
    char title [50];
    char author [20];
    float price;
};
```

1. What does this **code segment** represent**?**
   - A. It creates a variable of type Book
   - B. It defines a new data type named Book
   - C. It allocates memory for a book record

Answer: B → *the code segment refers to a structure definition/declaration*

2. What is happening with the statement *Book b1;* **?**
   - A. A new structure is being defined
   - B. Members are being initialized
   - C. A variable of type **Book** is created

Answer: C → *here **b1** is a structure variable and memory is now allocated to store book's: id, name, author & price*

Cont'd

3. What is the role of the **dot (.)** operator in structure?

A. Creates a structure variable
B. Allocates memory dynamically
C. Used to access structure members

Answer: C → *The **dot operator (.)** is used to access or modify members of a structure.*

4. What does this declaration represent? `Book b[3];`

A. A single **Book** variable
B. An array of 3 Student structures
C. Three variables of different data types

Answer: B → *This creates 3 structure variables: b[0], b[1], b[2]. Each element is a complete Student object.*

5. What will be printed?

```
Book b = {123, "Programming" };
cout << b.id << " " << b.price;
```

Answer: 123 0.0

→ print id & price separated by comma

→ since this is partial initialization, the price is roll back to default value 0.0

Practice Exercise

Exercise — Student Record System

Objective: To build a practical student record management system using structures in C++

Requirements

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

- **Expected fields**
  - ✓ Student ID (int),
  - ✓ Name (string),
  - ✓ GPA (float),
  - ✓ Department (string)

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

Cont’d

Solution : *Student Record System Implementation*

```cpp
#include <iostream>
#include <string>
#include <iomanip>

using namespace std;

const int MAX_STUDENTS = 100;

// structure declaration/definition
struct Student {
    int id;
    string name;
    float gpa;
    string department;
};

// function prototype declaration
void addStudent(Student s[], int &count);
void listStudents(Student s[], int count);
int findById(Student s[], int count, int id);
void searchStudent(Student s[], int count);
```

```cpp
int main() {
    Student students[MAX_STUDENTS];
    int count = 0, choice;

    while (true) {
        cout << "\n1. Add 2. List 3. Search 4. Exit: ";
        cin >> choice;

        switch (choice) {
            case 1:
                addStudent(students, count);
                break;
            case 2:
                listStudents(students, count);
                break;
            case 3:
                searchStudent(students, count);
                break;
            case 4:
                return 0;
            default:
                cout << "Invalid choice!\n";
        }
    }

    return 0;
}
```

Cont’d

Solution : *Student Record System Implementation*

```
void addStudent(Student s[], int &count) {
    if (count >= MAX_STUDENTS) {
        cout << "Student limit reached!" << endl;
        return;
    }

    cout << "Enter ID: ";
    cin >> s[count].id;

    cout << "Enter Name: ";
    cin >> s[count].name;

    cout << "Enter GPA: ";
    cin >> s[count].gpa;

    cout << "Enter Department: ";
    cin >> s[count].department;

    count++;
    cout << "Student added successfully!\n";
}
```

The complete Solution Found Here

Common Pitfalls and Best Practices

Common Pitfalls

✕ Structure Declaration & Design

- ✓ Missing semicolon after structure definition
- ✓ Confusing structure definition with variable declaration/creation
- ✓ Poor or inconsistent naming conventions

✕ Structure Variable Creation

- ✓ Forgetting to create variables after defining a structure
- ✓ Incorrect variable declaration syntax
- ✓ Declaring unnecessary or unused variables (memory waste)

✕ Structure Initialization Issues

- ✓ Using uninitialized members (garbage values)
- ✓ Incorrect initialization order (does not match structure definition)
- ✓ Misunderstanding partial initialization

✕ Accessing Structure Members

- ✓ Using wrong operator (. vs ->)
- ✓ Typographical errors in member names (case-sensitive issues)
- ✓ Accessing members before initialization

Best Practices

✓ Structure Design

- Always terminate structure definition with a **semicolon**
- Use **clear, meaningful**, and consistent naming conventions
- Define dependent structures before use
- Keep structures **cohesive and logically** grouped

✓ Structure variable Creation

- Declare variables only when needed
- Use grouped declarations where appropriate
- Prefer local variables over global variables

✓ Structure Initialization

- Always initialize structure variables or assign value before use
- Properly initialize all elements
- Maintain consistent member order during initialization

✓ Structure member access

- Use correct operators (**dot**) for accessing structure members
- Ensure structure members are initialized before access/used
- Follow consistent naming to avoid errors

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