Chapter 3: Essential C & C++ Concepts

> C++ is Extended C !!!

Chapter 3.1.: Arrays Basics

Core Array Definition and Purpose

Arrays serve as **collections of similar data elements** that allow you to group multiple values of the same data type under a single name. Instead of declaring separate variables for each piece of data, arrays provide an efficient mechanism to manage related information systematically. This fundamental concept becomes crucial when working with data structures, as arrays form the backbone of many complex data organization methods.[1][2]

> Array Memory Layout of eg: int A[5];

```
Stack Memory
+----------+----------+----------+----------+----------+
|  A[0]    |  A[1]    |  A[2]    |  A[3]    |  A[4]    |
|  0x1000  | 0x1004   | 0x1008   | 0x100C   | 0x1010   |
|   27     |   10     |    0     |    0     |    0     |
+----------+----------+----------+----------+----------+
   ^          ^          ^          ^          ^
   |          |          |          |          |
  Base    Base+4B    Base+8B    Base+12B   Base+16B

- Each box represents 4 bytes (assuming `int` is 4 bytes).
- The addresses increase by 4 for each subsequent element.
- The array name `A` points to the base address (`0x1000` in this example).
- Elements are accessed as `A[0]`, `A[1]`, ..., `A[4]` using zero-based indexing.
- All elements are stored contiguously in stack memory.
```

Array Declaration and Memory Allocation

When you declare an array in C using the syntax `int A[5]`, several important processes occur:[3][1]

Declaration Process:

- The compiler allocates a contiguous block of memory in the **stack section** of main memory[4]
- For `int A[5]`, exactly 20 bytes are reserved (5 integers × 4 bytes each)[1]
- The array name `A` becomes a pointer to the first element's memory address[5]

Memory Layout Characteristics:

- Elements are stored in **adjacent memory locations** for optimal access[6]
- Each element occupies the same amount of memory space based on data type[7]
- The stack allocation means arrays are **automatically managed** - created when entering function scope and destroyed when exiting[6]

```
C and C++ Concepts
------------------
          Arrays

Code:
------
int main() {
    int A[5];
    int B[5] = {2, 4, 6, 8, 10};
    int i;
    for(i = 0; i < 5; i++) {
        printf("%d", B[i]);
    }
}

Main Memory Representation:
---------------------------

            Main Memory
+-----------------------------------+
|             Heap                  |
+-----------------------------------+
|             Stack                 |
+-----------------------------------+
|              main                 |
|   +---------------------------+   |
|   |   A[,,,,]    B[2,4,6,8,10]|   |
|   +---------------------------+   |
+-----------------------------------+
|          Code Section             |
|   +---------------------------+   |
|   |         main()            |   |
|   |   printf(), for loop      |   |
|   +---------------------------+   |
+-----------------------------------+
```

Array Initialization Techniques

C provides multiple approaches for array initialization, each with specific use cases:[8][9]

Complete Initialization:

```
int B[5] = {2, 4, 6, 8, 10};
```

This creates an array and immediately fills all positions with specified values.[4][1]

Partial Initialization:

```
int C[5] = {1, 2};
```

Remaining elements (C, C, C) are automatically set to Garbage Values.[9][10][11][12][13]

> Garbage Values sometimes can be Zero(0) or something random (-2342342345)

Size Inference:

```
int D[] = {10, 20, 30, 40};
```

The compiler determines array size (4 elements) from the number of initializers provided.[13][9]

Variable Initialization:

```
cin>>n;
int A[n]; // int A[n]={1,3,5,929}; NO , Variable-size object cannot be initialized
, Sometimes are warned by compilers, sometimes not and glitches happens !!!
```

Zero-Based Indexing System

Arrays in C use **zero-based indexing**, meaning the first element is accessed as `A[0]`, not `A[1]`. This design choice stems from fundamental computer science principles:[14][5]

Memory Address Calculation: The address of `A[i]` equals `Base_Address + (i × sizeof(datatype))`. With zero-based indexing:[5][14]

- `A[0]` address = Base_Address + (0 × 4) = Base_Address
- `A[1]` address = Base_Address + (1 × 4) = Base_Address + 4
- `A[2]` address = Base_Address + (2 × 4) = Base_Address + 8

```
int A[5];
A[0]=1;
A[1]=2;
A[2]=4;
cout<<sizeof(A); // 20
```

This direct correlation between index and memory offset makes array access computationally efficient.[15][14]

Array Access and Traversal Methods

Individual Element Access:

```
A[0] = 27;  // Assigns value to first element
int value = A[1];  // Reads second element value
```

Loop-Based Traversal: The most common method for accessing all array elements uses a `for` loop:[16][17]

```
for(int i = 0; i < 5; i++) {
    printf("%d ", B[i]);
}
```

Dynamic Size Calculation: For arrays where size may vary, use the `sizeof` formula:[17]

```
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);
for(int i = 0; i < length; i++) {
    printf("%d ", myNumbers[i]);
}
```

Or in C++, use for each loop

```
for (int x:myNumbers){cout<<x<<endl;}
```

This approach makes your code adaptable to arrays of different sizes.[17]

Relationship Between Arrays and Pointers

Arrays and pointers share a fundamental relationship in C:[15][5]

- The array name represents the **base address** of the first element
- `A[i]` is equivalent to `*(A + i)` in pointer arithmetic[5]
- This relationship explains why array indexing starts from zero and enables efficient memory access patterns

Memory Management Considerations

Stack vs. Heap Allocation:

- Arrays declared within functions are **stack-allocated**[7][6]
- Stack memory is **automatically managed** - no manual deallocation required[6]
- Stack arrays have **fixed size** determined at compile time[7][6]
- For **dynamic arrays**, heap allocation using `malloc()` becomes necessary[18][19]

Memory Efficiency:

- Contiguous storage enables **cache-friendly** access patterns[6]
- Sequential memory layout optimizes processor performance[20]
- Zero-based indexing minimizes address calculation overhead[14][20]

Summary Reference Tables

Best Practices for Array Usage

Declaration Guidelines:

- Always specify array size explicitly when possible[3]
- Initialize arrays at declaration time to avoid garbage values[8][1]
- Use meaningful variable names that indicate the array's purpose[3]

Access Patterns:

- Prefer `for` loops for sequential array traversal[16][17]
- Use the `sizeof` formula for flexible array size handling[17]
- Validate array bounds to prevent buffer overflow errors[16]

Memory Considerations:

- Understand stack limitations for large arrays[6]
- Consider dynamic allocation for variable-sized arrays[18][7]
- Remember that local arrays are automatically cleaned up[6]

These foundational array concepts provide the essential knowledge needed for understanding more complex data structures throughout your course. The relationship between arrays, memory management, and pointer arithmetic forms the cornerstone of efficient C programming and data structure implementation.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Code

```
#include<iostream>
int main(){
    int a[5];
    // declared but not initialized

    char b[7]={'1','a','3'};
    // declared but partially initialized

    std::cout<<sizeof(a)<<std::endl;
    // 20
    // 4(int size) 8 5(spaces) =20 (total size of array)
    std::cout<<sizeof(b)<<std::endl;
    // 7
    // 7 = 1*7

    for(int i=0; i<7; i++){
        printf("%c\n",b[i]); // %c
    }
    // 1
    // a
    // 3
    //
    //
    //
```

```
    //
    // last 4 values are empty chars, because those were declared but not
initialised


    for(int i=0; i<7; i++){
        printf("%d\n",b[i]); // %d
    }
    // 49
    // 97
    // 51
    // 0
    // 0
    // 0
    // 0
    // char to int ASCII TypeCasting

    int c[] = {1,2,3,4,5,6,7,8,9,9};
    // here compiler will determine array size from number of initializers
provided
    std::cout<<sizeof(c)<<std::endl;
    // 40

    std::cout<<&c[2]<<std::endl;
    // 0x7ffe2ca0f418
    // suppose
    std::cout<<&c[3]<<std::endl;
    // 0x7ffe2ca0f41c
    // 0x7ffe2ca0f41c - 0x7ffe2ca0f418 = 4 Bytes
    // This direct correlation between index and memory offset makes array access
computationally efficient

    std::cout<<c[4]<<std::endl; // 5
    c[4]=55; // value changed
    std::cout<<c[4]<<std::endl; // 55

    int d = c[4]; //
    // array traversal 1
    std::cout<<c[4]<<std::endl; // 55

    for(int i=0; i <  ( sizeof(c) / sizeof(c[0]) ); i++) { std::cout<<c[i]
<<std::endl; }
    // 1
    // 2
    // 3
    // 4
    // 55
    // 6
    // 7
    // 8
    // 9
    // 9
    // array traversal 2
```

```
    for(int i=0; i <  ( sizeof(c) / sizeof(c[0]) ); i++) { printf("%d\n", c[i]);
//A[i] is equivalent to *(A + i) in pointer arithmetic
        }
    // 1
    // 2
    // 3
    // 4
    // 55
    // 6
    // 7
    // 8
    // 9
    // 9
    // array traversal 3

    for ( int i:c ){std::cout<<i<<std::endl;}
    // 1
    // 2
    // 3
    // 4
    // 55
    // 6
    // 7
    // 8
    // 9
    // 9
    // array traversal 4

    // array has fixed size
    // c[11]=11; // no, as it will not work or works anc corrupts other data and
glitches
    return 0;
}
```

Chapter 3.2.: Structures in C

Table of Contents

Introduction to Structures

A **structure** is a collection of data members (variables) grouped together under one name. These data members can be of:

- **Similar types** (e.g., all integers)
- **Dissimilar types** (e.g., integer, float, character arrays)

Key Characteristics:

- **User-defined data type** created using primitive(integer, float, character arrays) data types
- Groups related data items logically
- Allows creation of complex data representations
- Foundation for creating custom data types in C programming

Why Use Structures?

When dealing with entities that require multiple properties (like a rectangle with length and breadth), structures provide an organized way to group related data rather than using separate variables.

> Structure Definition != Structure Declaration != Structure Initialization

Structure Definition and Syntax

Basic Syntax:

```
struct structure_name {
    data_type member1;
    data_type member2;
    // ... more members
};
```

Rectangle Example:

```
struct rectangle {
    int length;
    int breadth;
};
```

> **Important Notes:**
>
> - `Structure definition` is just a **template** - no memory is allocated
> - Memory allocation happens only when variables are declared
> - Structure names follow C naming conventions

Memory Allocation and Size Calculation

Memory Calculation Rules:

- Size of structure = Sum of all member sizes + padding
- Each member occupies memory based on its data type
- Actual memory allocation occurs during variable declaration

Example Calculations:

Rectangle Structure:

```
struct rectangle {
    int length;    // 2 bytes (assuming 16-bit int)
    int breadth;   // 2 bytes
};
// Total: 4 bytes
```

Student Structure:

```
struct student {
    int rollNo;        // 2 bytes
    char name[25];     // 25 bytes
    char dept[10];     // 10 bytes
    char address[50];  // 50 bytes
};
// Total: 87 bytes
```

Declaration and Initialization

Declaration Methods:

1. Simple Declaration:

```
struct rectangle r;  // Declares variable 'r' of type rectangle
```

2. Declaration with Initialization:

```
struct rectangle r = {10, 5};  // length=10, breadth=5
```

3. Multiple Variables:

```
struct rectangle r1, r2, r3;  // Multiple variables of same type
```

4 . With Defination of Structure:

```
struct rectangle {
    int length;
    int breadth;
} r1, r2= {10, 5};
```

Memory Allocation:

- Variables are created in the **stack frame** of the function
- Each variable occupies memory equal to structure size
- Members are stored consecutively in memory

```
C and C++ Concepts
------------------
           Structure

Code:
------
struct Rectangle {
    int length;   // 2 bytes
    int breadth;  // 2 bytes
};                // Total: 4 bytes

int main() {
    struct Rectangle r;
    struct Rectangle r1 = {10, 5};
}

Main Memory Representation:
---------------------------

+-----------------------------------+
|             Heap                  |
+-----------------------------------+
|             Stack                 |
+-----------------------------------+
|              main                 |
|                    r              |
|                  +----+           |
|         length = | 10 |           |
|                  +----+           |
|        breadth = | 5  |           |
|                  +----+           |
+-----------------------------------+
|          Code Section             |
|   +--------------------------+    |
|   |  main(), struct defn     |    |
|   +--------------------------+    |
```

```
+----------------------------------+
```

Accessing Structure Members

Dot (.) Operator:

The dot operator is used to access structure members through structure variables.

Syntax:

```
structure_variable.member_name
```

Examples:

```c
struct rectangle r;
r.length = 15;        // Assign value to length
r.breadth = 10;       // Assign value to breadth
int area = r.length * r.breadth;  // Calculate using members
```

Key Points:

- Dot operator has **highest precedence** in C
- Used for both reading and writing member values
- Essential for manipulating structure data

Practical Examples

1. Complex Number Structure

```c
// Complex Number
// a+ib
// i = (-1)**(1/2), i.e. sq.root of -1
struct complex {
    float real;      // Real part (A in A+iB)
    float imaginary; // Imaginary part (B in A+iB)
};

// Usage
struct complex c1 = {3.5, 2.8};
printf("Complex number: %.2f + %.2fi\n", c1.real, c1.imaginary);
```

2. Student Information System

```
struct student {
    int rollNo;
    char name[25];
    char dept[10];
    char address[50];
};

// Usage
struct student s1;
s1.rollNo = 101;
strcpy(s1.name, "John Doe");
strcpy(s1.dept, "CSE");
strcpy(s1.address, "123 Main St");
```

3. Playing Card Structure

```
struct card {
    int face;    // 1-13 (Ace=1, Jack=11, Queen=12, King=13)
    int shape;   // 0=Club, 1=Spade, 2=Diamond, 3=Heart
    int color;   // 0=Black, 1=Red
};

// Usage
struct card aceOfSpades = {1, 1, 0};  // Ace of Spades (Black)
```

Array of Structures

Declaration Syntax:

```
struct structure_name array_name[size];
```

Deck of Cards Example:

```
struct card deck[52];  // Array of 52 card structures
```

Initialization:

```
struct card deck[52] = {
    {1, 0, 0},    // Ace of Clubs (Black)
    {2, 0, 0},    // 2 of Clubs (Black)
```

```
    {1, 1, 0},    // Ace of Spades (Black)
    // ... continue for all 52 cards
};
```

Accessing Array Elements:

```
// Access first card's face value
printf("First card face: %d\n", deck[0].face);

// Access second card's shape
printf("Second card shape: %d\n", deck[1].shape);

// Loop through all cards
for(int i = 0; i < 52; i++) {
    printf("Card %d: Face=%d, Shape=%d, Color=%d\n",
           i+1, deck[i].face, deck[i].shape, deck[i].color);
}
```

Memory Calculation for Arrays:

- Single card structure: 6 bytes (3 integers × 2 bytes each)
- Array of 52 cards: 52 × 6 = 312 bytes total

Memory Layout and Stack Allocation

Stack Frame Allocation:

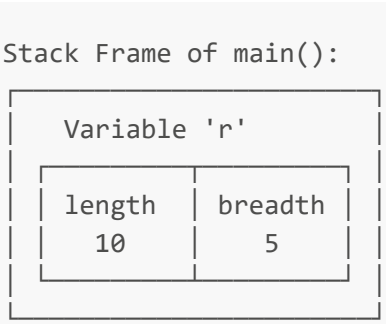
When structures are declared in functions:

- Memory allocated in function's **stack frame**
- Automatic memory management
- Memory deallocated when function ends

Memory Layout Example:

```
int main() {
    struct rectangle r = {10, 5};
    // Memory layout in stack:
    // [length: 10][breadth: 5]
    // |    2B   |    2B     | = 4 bytes total
}
```

Visualization:



Structure Padding Concepts

What is Structure Padding?

Structure padding is the insertion of empty bytes between structure members to align data according to processor requirements.

Why Padding Occurs: to make Accessibility Easy !!!

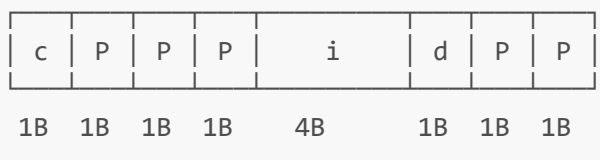
- **Processor Architecture**: 32-bit processors read 4 bytes at a time
- **Performance Optimization**: Aligned data access is faster
- **Hardware Requirements**: Some processors require aligned memory access

> it's easy for machine to read 4-4 bytes at a time. like it's easy for pharmacist to sell medicines as strips, not custom size.

Example with Padding:

```
struct example {
    char c;      // 1 byte
    // 3 bytes padding here
    int i;       // 4 bytes
    char d;      // 1 byte
    // 3 bytes padding here
};
// Total: 12 bytes (not 6 bytes)
```

Memory Layout with Padding:



(P = Padding bytes)

Best Practices

1. Meaningful Names:

```
// Good
struct employee {
    int empId;
    char name[50];
    float salary;
};

// Avoid generic names like 'data', 'info', etc.
```

2. Logical Grouping:

Group related data that naturally belongs together:

```
// Good - Related geometric properties
struct rectangle {
    int length;
    int breadth;
};

// Avoid - Unrelated data
struct mixed {
    int age;
    float temperature;
    char color[10];
};
```

3. Memory Efficiency:

Order members at descending order of size to minimize padding:

```
// Less efficient (more padding)
struct inefficient {
    char c1;     // 1 byte + 3 padding
    int i;       // 4 bytes
    char c2;     // 1 byte + 3 padding
};  // Total: 12 bytes

// More efficient (less padding)
struct efficient {
    int i;       // 4 bytes
    char c1;     // 1 byte
    char c2;     // 1 byte + 2 padding
};  // Total: 8 bytes
```

To minimize padding, **declare the struct members in descending order of their size**. Put the largest types (like `double`, `int`) first, followed by smaller types (like `char`).

Why This Works 🧠

Computers can read data from memory much faster when it starts at a memory address that is a multiple of its own size. This is called **alignment**.

- An `int` (4 bytes) `wants to start at an address divisible by 4` (like address 0, 4, 8, ...).
- A `char` (1 byte) can start at any address.

To enforce this, the compiler automatically inserts empty bytes called **padding** to push the next member to a "friendly" starting address. By putting the biggest members first, you use the available space more naturally, and the compiler needs to add less padding.

Inefficient Layout (More Padding)

In this case, a 4-byte `int` is sandwiched between two 1-byte `char`s. The compiler has to add padding before *and* after to keep everything aligned.

```
struct inefficient {
    char c1; // 1 byte
    int  i;  // 4 bytes
    char c2; // 1 byte
};
```

Memory Layout (12 bytes total):

The compiler adds 3 bytes of padding after `c1` so that `i` can start on an address divisible by 4. It then adds 3 more bytes at the end so the total size of the struct (12) is a multiple of the largest member's alignment (4).

```
Byte: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
      +---+---+---+---+---+---+---+---+---+---+---+----+
Data: |c1 | P | P | P |     i     |c2 | P | P | P  |
      +---+---+---+---+---+---+---+---+---+---+---+----+
                    (P = Padding)
```

- `c1`: 1 byte
- **Padding**: 3 bytes
- `i`: 4 bytes
- `c2`: 1 byte
- **Padding**: 3 bytes

Efficient Layout (Less Padding)

Here, we declare the largest member (`int`) first. The smaller `char`s can be packed together right after it without needing any padding in between.

```
struct efficient {
    int  i;  // 4 bytes
    char c1; // 1 byte
    char c2; // 1 byte
};
```

Memory Layout (8 bytes total):

The only padding needed is at the very end to make the total struct size a multiple of 4.

```
Byte: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
      +---+---+---+---+---+---+---+---+
Data: |     i     |c1 |c2 | P | P |
      +---+---+---+---+---+---+---+---+
```

- `i`: 4 bytes
- `c1`: 1 byte
- `c2`: 1 byte
- **Padding**: 2 bytes

By simply reordering the members, you save 4 bytes of memory for every instance of this struct!

4. Initialization Best Practices:

```
// Clear initialization
struct point p1 = {10, 20};

// Partial initialization (remaining members set to 0)
struct student s1 = {101, "John"};  // Other members become 0 or empty

// Zero initialization
struct rectangle r = {0};  // All members set to 0
```

5. Array of Structures Usage:

```
// Declare and initialize efficiently
struct student class[30] = {
    {101, "Alice", "CSE", "Address1"},
    {102, "Bob", "ECE", "Address2"},
    // ... more students
};
```

```
// Process using loops
for(int i = 0; i < 30; i++) {
    if(class[i].rollNo != 0) {  // Check if student exists
        printf("Roll: %d, Name: %s\n", class[i].rollNo, class[i].name);
    }
}
```

6. Function Parameter Considerations:

While not covered in this video, consider:

- Passing structures by reference (pointers) for efficiency
- Returning structures from functions
- Dynamic memory allocation for large structures

Key Takeaways

1. **Structures group related data** under one name for better organization
2. **Memory allocation** happens only during variable declaration, not definition
3. **Dot operator (.)** is essential for accessing structure members
4. **Array of structures** enables handling multiple instances efficiently
5. **Structure padding** affects memory usage and should be considered for optimization
6. **Stack allocation** occurs for local structure variables in functions
7. **Proper design** and naming conventions improve code maintainability

Advanced Topics to Explore

1. **Pointers to Structures** - Using arrow operator (->)
2. **Structures as Function Parameters** - Pass by value vs. reference
3. **Dynamic Memory Allocation** - Using malloc() for structures
4. **Nested Structures** - Structures within structures
5. **Structure Bit Fields** - Optimizing memory for flag variables
6. **Union vs. Structures** - Understanding the differences
7. **Structure Packing Directives** - Compiler-specific optimizations

Code

```
#include<bits/stdc++.h>
using namespace std;

// Structure Defination eg: Polygon ( just Chull )
struct polygon {
    int length; // also it's redundant as triangle.edges.size(); gives the same
stuff
    vector<int> edges; //int edges[]; is wrong, int edges[length]; is wrong
    vector<int> angles;
};

// struct eg: Rectangle
```

```c
struct rect
{
    /* data */
    int l; // -> 2
    int b; // -> 2
    // total = -> 4
    // also Structure definition is just a template - no memory is allocated
} r1, r2={1,2};


// struct eg: complex no.
struct cp{
    float r;
    float i;
} c1={1.5,0.3};

// struct eg: student info system
struct student{
    int rollNo;
    char name[25];
    // this is not string, but character array,
        // basically You can't use direct assignment like s1.dept = "CSD"; because in C and C++, arrays are not assignable.
        // strcpy(s1.name, "Siddhant Bali"); // Correct
    char dept[50];
    char address[50];
};


// strcut eg: playing cards
struct card{
    int face; // 1-13 { 1: Ace, 2: 2, 3: 3, ... , 10: 10, 11: Jack, 12: Queen, 13: King}
    int shape; // 0-3 {0: Club, 1: Spade, 2: Diamond, 3:Heart }
    int colour; // 0: Black, 1: Red
    // & also Diamond/Heart are red, Club/Spade are Black
    // Construction of Deck of cards
        // Thus constructing it NOT be loop of face*shape*colour
        // then even NOT do face*shape & inside it colour 0, 1 2 times add
        // logic of making it for colour is: if shape==0 or shape==1 => colour is 0  &  if shape==2 or shape==3 => colour is 1
};

// structure padding
// inefficient
struct eg{
    char a; // 1(should be) => 1(real life)
    // (+3 offset for int to attach stable)
    int b; // 4(should be) => 4(real life)
    char c; // 1(should be) => 1(real life)
    // (+3 offset to end array stable)
    // Total = 6 Bytes(should be) => 12 Bytes(real life)
} eg1={1,2,3};
```

```
// efficient
struct eg2{
    int b; // 4(should be) => 4(real life)
    char a; // 1(should be) => 1(real life)
    char c; // 1(should be) => 1(real life)
    // (+2 offset to end array stable)
    // Total = 6 Bytes(should be) => 8 Bytes(real life)
    // * An `int` (4 bytes) `wants to start at an address divisible by 4` (like
address 0, 4, 8, ...).
    // * A `char` (1 byte) can start at any address.
} eg21={1,2,3};



int main(){
    struct polygon triangle; // also // The 'struct' keyword is not needed here in
C+
    // polygon triangle;
    triangle.length = 3;
    triangle.edges = {5,5,5};
    triangle.angles = {60,60,60};

    struct rect r;
    rect r3,r4;
    rect r5={1,22};
    r1 = {11,11};
    r1.l = 12;
    cout<<r1.l<<"\t"<<r1.b<<endl; // no direct cout<<r1;
    // 12      11
    cout<<r2.l<<"\t"<<r2.b<<endl;
    // 1       2
    cout<<"area of r2 = "<<(r2.l) * (r2.b)<< endl;
    // 2
    // because 1*2 = 2

    // Dot operator has highest precedence in C,  Used for both reading and
writing member values

    cout<<c1.r<<"\t"<<c1.i<<endl;
    // 1.5     0.3
    printf(" Complex Number :%.2f\t%.2f\n",c1.r,c1.i);
    // 1.50    0.30
    printf("%.3f\t%.2f\n",c1.r,c1.i);
    // 1.500   0.30

    // student struct
    student s1;
    s1.rollNo = 2022496;
    // NO this // s1.dept = "CSD"; // expression must be a modifiable value C/C++
(137)
    strcpy(s1.name, "Siddhant Bali"); // Correct
    strcpy(s1.address, "India");
    strcpy(s1.dept, "Computer Science and Design");
```

```
    // Array of Structures
    card cards[52];
    int idx= 0;

    for (int shape=0; shape< 4; shape++){
        for (int face=1; face< 14; face++){
            if (shape<2) { cards[idx] = {face,shape,0}; }
            if (shape>=2) { cards[idx] = {face,shape,1}; }
            idx++;
        }
    }

    idx=0;

    for (int shape=0; shape< 4; shape++){
        for (int face=1; face< 14; face++){
            cout<<idx+1<<":\t"<<cards[idx].face<<"\t"<<cards[idx].shape<<"\t"
<<cards[idx].colour<<endl;
            idx++;
        }
    }
    idx=0;
    // 1:      1       0       0
    // 2:      2       0       0
    // 3:      3       0       0
    // 4:      4       0       0
    // 5:      5       0       0
    // 6:      6       0       0
    // 7:      7       0       0
    // 8:      8       0       0
    // 9:      9       0       0
    // 10:     10      0       0
    // 11:     11      0       0
    // 12:     12      0       0
    // 13:     13      0       0
    // 14:     1       1       0
    // 15:     2       1       0
    // 16:     3       1       0
    // 17:     4       1       0
    // 18:     5       1       0
    // 19:     6       1       0
    // 20:     7       1       0
    // 21:     8       1       0
    // 22:     9       1       0
    // 23:     10      1       0
    // 24:     11      1       0
    // 25:     12      1       0
    // 26:     13      1       0
    // 27:     1       2       1
    // 28:     2       2       1
    // 29:     3       2       1
    // 30:     4       2       1
    // 31:     5       2       1
    // 32:     6       2       1
```

```
    // 33:     7       2       1
    // 34:     8       2       1
    // 35:     9       2       1
    // 36:     10      2       1
    // 37:     11      2       1
    // 38:     12      2       1
    // 39:     13      2       1
    // 40:     1       3       1
    // 41:     2       3       1
    // 42:     3       3       1
    // 43:     4       3       1
    // 44:     5       3       1
    // 45:     6       3       1
    // 46:     7       3       1
    // 47:     8       3       1
    // 48:     9       3       1
    // 49:     10      3       1
    // 50:     11      3       1
    // 51:     12      3       1
    // 52:     13      3       1

    // structure padding
    cout<<"Sum "<<sizeof(eg1.a) + sizeof(eg1.b) + sizeof(eg1.c)<<endl;
    // Sum 6
    // 1 + 4 + 1
    // sizeof() operator returns the size of a variable or data type in bytes.
    // sizeof(eg1.a) = sizeof(char) = 1
    cout<<"Sum "<<sizeof(eg1)<<endl;
    // Sum 12
    // 4 + 4 + 4
    // (1+3) + 4 + (1+3)
    // sizeof(eg1) = size of eg1 custom datatype/datastructure, this struct is
data type too
    cout<<"Sum "<<sizeof(eg21)<<endl;
    // Sum 8
    // 4 + 2 + 2

    printf("%d\n",sizeof(eg21) ); // working, but NOT recommended
    // 2.cpp:202:14: warning: format ‘%d’ expects argument of type ‘int’, but
argument 2 has type ‘long unsigned int’ [-Wformat=]
    // 202 |     printf("%d\n",sizeof(eg21) );
    //     |             ~^    ~~~~~~~~~~~~
    //     |              |    |
    //     |              int  long unsigned int
    //     |             %ld
    // 8
    printf("%lu\n",sizeof(eg21) ); // %lu is long unsigned int
    // 8
    return 0;
}
```

Chapter 3.3.: Introduction to Pointers

Key Takeaway: Pointers allow a program to store and manipulate addresses of data rather than the data itself. They are essential for dynamic memory management, resource access, and efficient parameter passing.

Definition and Motivation

A **pointer** is a special variable whose value is the **address** of another variable or resource, rather than the data itself. While normal variables directly hold data (e.g., integers, characters), pointers hold the location of where that data resides in memory. By using pointers, a program can:

- **Indirectly access** data stored in different regions of memory.
- **Dynamically allocate** and free memory on the **heap**, enabling flexible data structures.
- **Access external resources** (files, network sockets, devices) through handles or descriptors.
- **Efficiently pass parameters** to functions by reference, avoiding large data copies.

Memory Layout Overview

Modern programs divide **main memory** into three segments:

1. **Code (Text) Segment**
   Contains the compiled program instructions.
2. **Stack Segment**
   Stores local variables and function call frames.
3. **Heap Segment**
   Provides dynamically allocated memory at runtime.

By default, a running program can directly access its Code and Stack segments. To use or manage memory in the Heap (or interact with external resources like files or devices), **pointers** are required.

Pointer Declaration, Initialization, and Dereferencing

1. Declaring a Pointer

```
int* p;
```

- The asterisk (`*`) in the declaration specifies that `p` is a pointer to an `int`.
- This pointer variable itself is stored on the **stack** and consumes memory (8 bytes on modern 64-bit systems, regardless of the pointed-to type).

2. Initializing a Pointer

To make `p` hold the address of an existing integer variable:

```
int a = 10;
int* p;
p = &a;   // &a yields the address of 'a'
```

- The address operator (`&`) retrieves the memory location of `a`.
- **Do not** use `*` when assigning an address—only at declaration and when dereferencing.

3. Dereferencing a Pointer

To access the data stored at the address held by `p`:

```
int value = *p;
std::cout << *p;  // prints 10
```

- The dereference operator (`*`) tells the compiler to access the data at the pointer's address.
- Without dereferencing, printing `p` yields the address itself.

Dynamic Memory Allocation

Pointers shine when managing heap memory:

In C (using `malloc` and `free`)

```
#include <stdlib.h>

int* p = (int*)malloc(5 * sizeof(int));  // allocate array of 5 ints
// ... use p[0] through p[4]
free(p);                                 // release heap memory
```

- `malloc` returns a `void*` which must be cast to the appropriate pointer type.
- Always call `free` when done to prevent memory leaks (critical in large programs).

In C++ (using `new` and `delete[]`)

```
int* p = new int[5];  // allocate array of 5 ints
// ... use p[0] through p[4]
delete[] p;           // release heap memory
```

- `new` automatically returns the correctly typed pointer.
- Use `delete[]` for arrays; use plain `delete` for single objects.
- In short-lived student programs, freeing heap memory may be optional, but it's **essential** in larger applications.

Pointers and Arrays

- The **name of an array** acts as a constant pointer to its first element.
- You can assign an array to a pointer without `&`:

```
int A[5] = {2,4,6,8,10};
int* p = A;          // same as &A[0]
```

- Both `A[i]` and `p[i]` access the ith element.

Pointer Size Is Uniform

On modern 64-bit systems, **all** pointers—regardless of the data type they point to—occupy **8 bytes**. This uniformity holds whether they point to `int`, `char`, `struct`, or function types.

Best Practices

- Always **initialize** pointers before use.
- **Dereference** only valid, non-null pointers.
- **Match** each `malloc`/`new` with a corresponding `free`/`delete[]`.
- Consider using **smart pointers** (e.g., `std::unique_ptr`, `std::shared_ptr`) in C++ to automate memory management.
- Practice pointer operations through small code experiments to solidify understanding.

By mastering pointers, you gain precise control over memory, enabling advanced data structures, resource management, and performance optimizations in both C and C++.