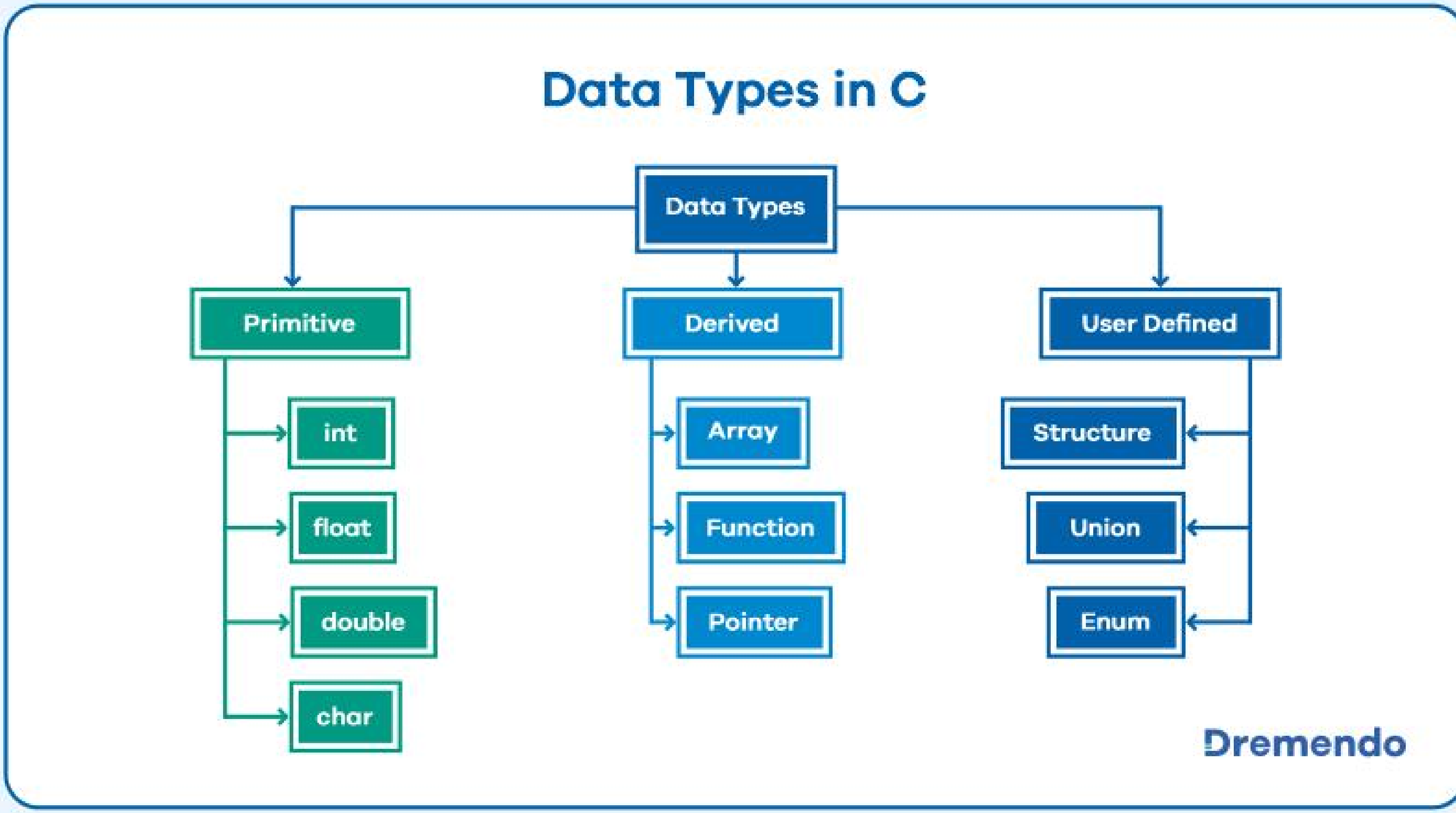
Unit 6:

Data Structures

- Struct
- Union
- Bit Field
- DMA
- Linked List


Data Types in C
Data Types
Primitive
Derived
User Defined
int
float
double
char
Array
Function
Pointer
Structure
Union
Enum
Dremendo

User Defined Data types- **typedef, enum**

- "type definition" allows user to define variable of existing data type.
- Example.
  - typedef float marks ;
  - marks m1,m2,m3;

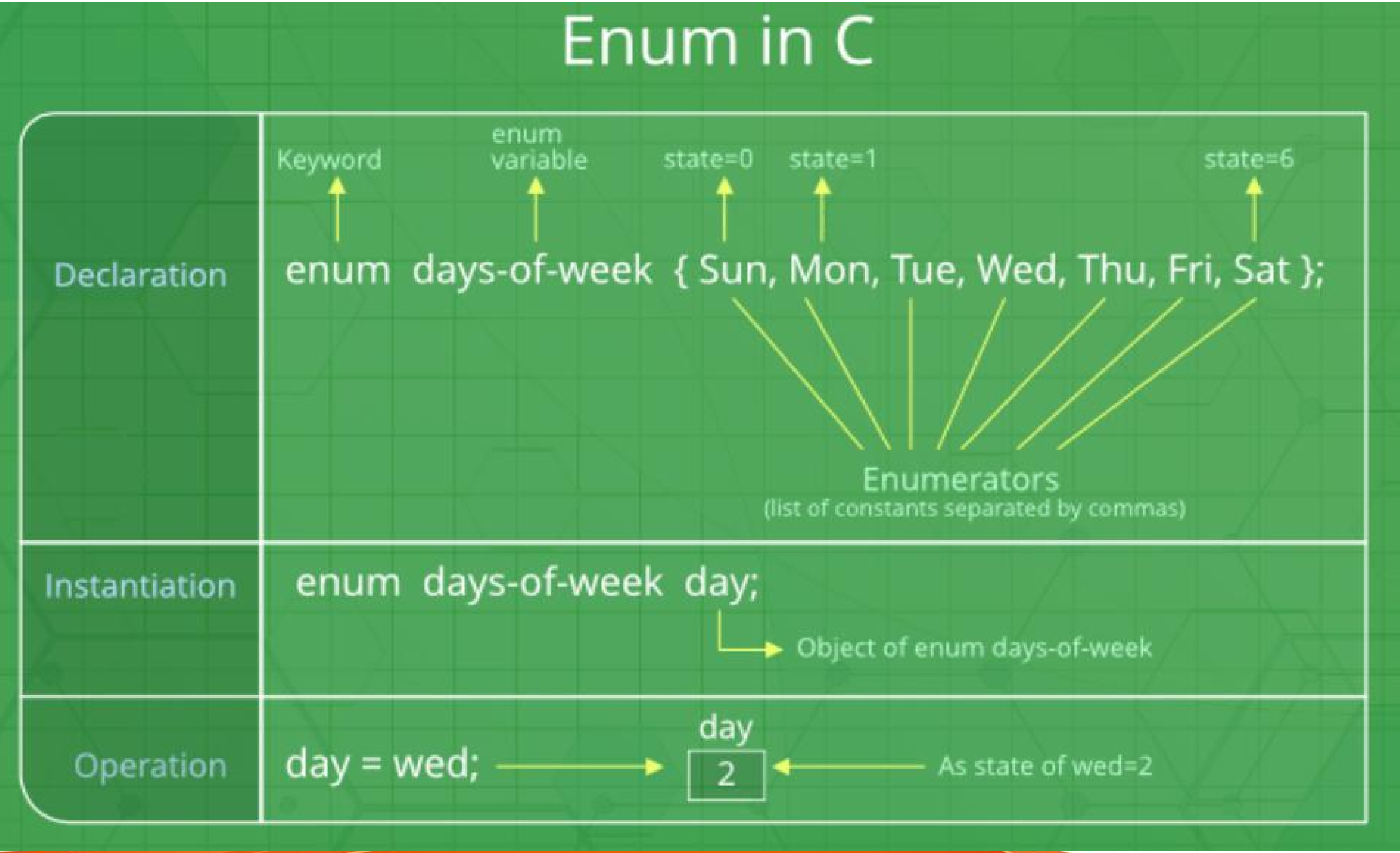
  (in above example marks is used as float data type)

Enumerated data type:

It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

- enum day { Mon , Tue ,wed ,Thu ,Fri ,Sat , Sun }
- enum day today;
- today=Thu

find mistake.



```c
// An example program to demonstrate working
// of enum in C
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};

int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

```c
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
    printf("%d, %d, %d", Working, Failed, Freezed);
    return 0;
}
```

- Two enum names can have same value.
- If we do not explicitly assign values to enum names, the compiler by default assigns values starting from 0.
- Values can be assigned in any order.

Derived data types- **Array**

- Array:
  - It is collection of homogeneous data type elements stored in contiguous memory locations.
  - E.g.
    - int a[5];
    - Element of array is accessed with a[i] where "**a**" is the array and "i" is the index .

a[0]	a[1]	a[2]	a[3]	a[4]

Structure (Struct) in C

- created to store different types of values under a single variable.

Defining a Structure

```
struct record
{
    int roll;
    char name[20];
    float per;
};
```

Syntax:

```
struct structure_name
{
    data_type member_name1;
    data_type member_name1;
    ....
    ....
};
```

Creating Structure Variable

- After structure definition, we have to create variable of that structure to use it. It is similar to the any other type of variable declaration:

```
struct strcuture_name var;
```

- We can also declare structure variables with structure definition.

```
struct structure_name {
  …
}var1, var2….;
```

- Structures—sometimes referred to as aggregates—are collections of related variables under one name.

- Structures may contain variables of ***many different data types***—
  - in contrast to arrays, which contain *only* elements of the same data type.

- To access or modify members of a structure, we use the (.) dot operator.

  structure_name . member1;

strcuture_name . member2;

E.g.

```
struct stud
    {
        int char name[35];
        int roll_no;
        char addr[50]
    }s;
```

s.Name //access the name field

- In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

```
structure_ptr -> member1
structure_ptr -> member2
```

```c
struct record
{
    int roll;
    char name[20];
    float per;
};

int main()
{
    struct record x;
    x.roll=1;
    strcpy(x.name,"Allen");
    x.per=92.36;

    printf("Roll: %d\n", x.roll);
    printf("Name: %s\n", x.name);
    printf("Percentage: %f", x.per);
    return 0;
}
```

Example:

```c
// Create a structure
struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Print values
  printf("%d %c %s", s1.myNum, s1.myLetter, s1.myString);

  return 0;
}
```

Copy structure

```
#include <stdio.h>

struct myStructure {
  int myNum;
  char myLetter;
  char myString[30];
};

int main() {
  // Create a structure variable and assign values to it
  struct myStructure s1 = {13, 'B', "Some text"};

  // Create another structure variable
  struct myStructure s2;

  // Copy s1 values to s2
  s2 = s1;

  // Print values
  printf("%d %c %s", s2.myNum, s2.myLetter, s2.myString);

  return 0;
```

Real-Life Example

```c
struct Car {
  char brand[50];
  char model[50];
  int year;
};

int main() {
  struct Car car1 = {"BMW", "X5", 1999};
  struct Car car2 = {"Ford", "Mustang", 1969};
  struct Car car3 = {"Toyota", "Corolla", 2011};

  printf("%s %s %d\n", car1.brand, car1.model, car1.year);
  printf("%s %s %d\n", car2.brand, car2.model, car2.year);
  printf("%s %s %d\n", car3.brand, car3.model, car3.year);

  return 0;
}
```

Structure Pointer

```c
#include <stdio.h>

// Structure declaration
struct Point {
    int x, y;
};

int main() {
    struct Point p = { 1, 2 };

    // ptr is a pointer to structure p
    struct Point* ptr = &p;

    // Accessing structure members using structure pointer
    printf("%d %d", ptr->x, ptr->y);

    return 0;
}
```

Syntax of `typedef` with `struct`

```c
typedef struct [struct_name] {
    // Members (fields) of the struct
    data_type member1;
    data_type member2;
    // ...
} new_type_name;
```

- `struct_name`: The name of the struct (optional; can be omitted).
- `new_type_name`: The new name (alias) for the struct type.

```c
// Defining structure
typedef struct {
    int a;
} str1;

// Another way of using typedef with structures
struct st_name {
        int x;
};
typedef st_name str2;
int main() {

    // Creating structure variables using new names
    str1 var1 = { 20 };
    str2 var2 = { 314 };

    printf("var1.a = %d\n", var1.a);
    printf("var2.x = %d\n", var2.x);
    return 0;
}
```

Passing Structure to Functions

```c
#include <stdio.h>

// Structure definition
struct A {
    int x;
};

// Function to increment values
void increment(struct A a, struct A* b) {
    a.x++;
      b->x++;
}

int main() {
    struct A a  = { 10 };
      struct A b  = { 10 };

      // Passing a by value and b by pointer
      increment(a, &b);

      printf("a.x: %d \tb.x: %d", a.x, b.x);
    return 0;
}
```

Nested Structures

- Nested structure refers to a structure that contains another structure as one of its members.
- Two ways:
  - **Embedded Structure Nesting:** The structure being nested is also declared inside the parent structure.
  - **Separate Structure Nesting:** Two structures are declared separately and then the member structure is nested inside the parent structure.

```c
#include <stdio.h>
#include <string.h>

// Declaration of the main
// structure
struct Organisation
{
  char organisation_name[20];
  char org_number[20];

  // Declaration of the dependent
  // structure
  struct Employee
  {
    int employee_id;
    char name[20];
    int salary;

  // variable is created which acts
  // as member to Organisation structu
  } emp;
};
```

```c
int main()
{
  struct Organisation org;

  // Print the size of organisation
  // structure
  printf("The size of structure organisation : %ld\n",
         sizeof(org));

  org.emp.employee_id = 101;
  strcpy(org.emp.name, "Robert");
  org.emp.salary = 400000;
  strcpy(org.organisation_name,
         "GeeksforGeeks");
  strcpy(org.org_number, "GFG123768");


  // Printing the details
  printf("Organisation Name : %s\n",
         org.organisation_name);
  printf("Organisation Number : %s\n",
         org.org_number);
  printf("Employee id : %d\n",
         org.emp.employee_id);
  printf("Employee name : %s\n",
         org.emp.name);
  printf("Employee Salary : %d\n",
         org.emp.salary);
}
```

```c
#include <stdio.h>
#include <string.h>

// Declaration of the
// dependent structure
struct Employee
{
  int employee_id;
  char name[20];
  int salary;
};

// Declaration of the
// Outer structure
struct Organisation
{
  char organisation_name[20];
  char org_number[20];

  // Dependent structure is used
  // as a member inside the main
  // structure for implementing
  // nested structure
  struct Employee emp;
};
```

```c
int main()
{
  // Structure variable
  struct Organisation org;

  // Print the size of organisation
  // structure
  printf("The size of structure organisation : %ld\n",
         sizeof(org));

  org.emp.employee_id = 101;
  strcpy(org.emp.name, "Robert");
  org.emp.salary = 400000;
  strcpy(org.organisation_name,
        "GeeksforGeeks");
  strcpy(org.org_number, "GFG123768");
  // Printing the details
  printf("Organisation Name : %s\n",
         org.organisation_name);
  printf("Organisation Number : %s\n",
         org.org_number);
  printf("Employee id : %d\n",
         org.emp.employee_id);
  printf("Employee name : %s\n",
         org.emp.name);
  printf("Employee Salary : %d\n",
         org.emp.salary);
}
```

Union

- It is similar to structure data type .
- Difference between structure and union is that in structure every data member has its own storage where members of union shares same memory locations.
- Elements of union can be accessed using dot(.) operator
- Total memory allocated is equivalent to the size of the largest member.
- E.g

```
union item
{
    int m[5];    //20 bytes
    float z;  //4 bytes
    char c;   //1 byte
}u;

u.m  //access m field
```

BIT Field

- Can specify the size (in bits) of the structure and union members.
- to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is within a small range.
- C Bit fields are used when the storage of our program is limited.
- Syntax

```
struct {
        type [member_name] : width;
};
```

Size of structure without Bit Field

```
// C Program to illustrate the structure without bit field
#include <stdio.h>

// A simple representation of the date
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};

int main()
{
    // printing size of structure
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
}
```

Size of structure with Bit Field

```c
// C program to demonstrate use of Bit-fields
#include <stdio.h>

// Space optimized representation of the date
struct date {
    // d has value between 0 and 31, so 5 bits
    // are sufficient
    int d : 5;

    // m has value between 0 and 15, so 4 bits
    // are sufficient
    int m : 4;

    int y;
};

int main()
{
    printf("Size of date is %lu bytes\n",
           sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}
```

Dynamic Memory Allocation

- Process of allocating memory at runtime (during program execution) rather than at compile time.
- Allows programs to request memory from the operating system as needed and release it when it is no longer required.
- Used when size is not known in advance, such as arrays, linked lists, trees, and more.

key functions in stdlib.h

- malloc: Allocates a block of memory of a specified size.
- calloc: Allocates a block of memory for an array of elements and initializes all bytes to zero.
- realloc: Resizes a previously allocated block of memory.
- free: Deallocates a block of memory, making it available for future use.

1. malloc (Memory Allocation)

- Allocates a block of memory of a specified size in bytes.
- The memory is not initialized (contains garbage values).
- Returns a pointer to the first byte of the allocated memory.
- If the allocation fails, it returns NULL.

- Syntax

```
void* malloc(size_t size);
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5;

    // Allocate memory for an array of 5 integers
    arr = (int*) malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Use the allocated memory
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    // Print the array
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

2. calloc (Contiguous Allocation)

- Allocates memory for an array of elements and initializes all bytes to zero.
- Takes two arguments: the number of elements and the size of each element.
- Returns a pointer to the first byte of the allocated memory.
- If the allocation fails, it returns NULL.

- Syntax
- void* calloc(size_t num, size_t size);

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5;

    // Allocate memory for an array of 5 integers and initialize to zero
    arr = (int*) calloc(n, sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Print the array (all elements are initialized to 0)
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

3. realloc (Reallocation)

- Resizes a previously allocated block of memory.
- Takes two arguments: a pointer to the previously allocated memory and the new size.
- If the new size is larger, the additional memory is uninitialized.
- If the new size is smaller, the excess memory is deallocated.
- Returns a pointer to the resized memory block.
- If the allocation fails, it returns NULL, and the original block remains unchanged.
- Syntax

```
void* realloc(void* ptr, size_t size);
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr;
    int n = 5;

    // Allocate memory for an array of 5 integers
    arr = (int*) malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Initialize the array
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }
```

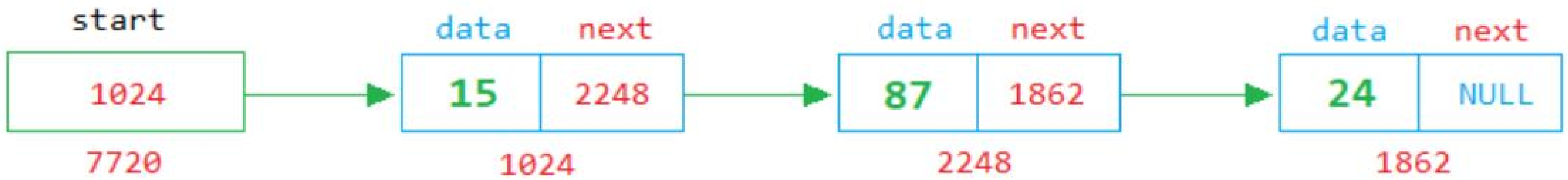
```c
    // Resize the array to 10 integers
    arr = (int*) realloc(arr, 10 * sizeof(int));
    if (arr == NULL) {
        printf("Memory reallocation failed!\n");
        return 1;
    }

    // Initialize the additional elements
    for (int i = n; i < 10; i++) {
        arr[i] = i + 1;
    }

    // Print the resized array
    for (int i = 0; i < 10; i++) {
        printf("%d ", arr[i]);
    }

    // Free the allocated memory
    free(arr);

    return 0;
}
```

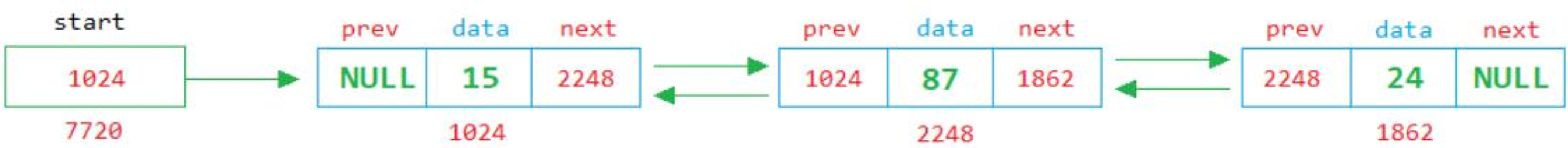
4. free (Deallocation)

- Releases a block of memory previously allocated by malloc, calloc, or realloc.
- The pointer passed to free must be the same pointer returned by the allocation function.
- After calling free, the pointer should not be used (it becomes a dangling pointer).

- Syntax
- void free(void* ptr);

Linked List(Singly, Doubly, Circular)





Linked List

- LL is a linear data structure in which elements are stored in nodes, and each node points to the next node in the sequence.
- LL do not require contiguous memory allocation.
- Each node in a linked list contains two parts:
  - Data: The value or information stored in the node.
  - Pointer: A reference (or link) to the next node in the list.
- The last node in the list typically points to NULL, indicating the end of the list.

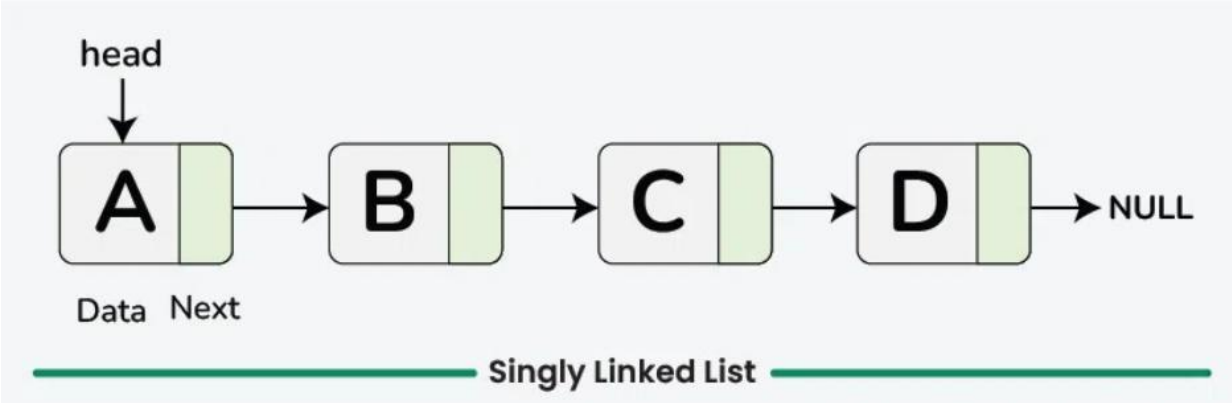
Array VS linked List

	Array	Linked List
Data Structure	Contigious	Non-Contigious
Memory Allocation	allocated to whole arrary	allocated to individual element
Insertion/ Deletion	Inefficient	Efficient
Access	Random	Sequantial

Operations on Singly Linked List

- Traversal
- Searching
- Length
- Insertion:
    - Insert at the beginning
    - Insert at the end
    - Insert at a specific position
- Deletion:
    - Delete from the beginning
    - Delete from the end
    - Delete a specific node

Singly Linked List




```
// Definition of a Node in a singly linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new Node
struct Node* newNode(int data) {
    struct Node* temp =
      (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->next = NULL;
    return temp;
}
```

Insertion at Beginning

Algorithm

```
1. START
2. Create a node to store the data
3. Check if the list is empty
4. If the list is empty, add the data to the node and
   assign the head pointer to it.
5. If the list is not empty, add the data to a node and link to the
   current head. Assign the head to the newly added node.
6. END
```