**23CS101 Programming in C – UNIT IV**

**UNIT IV STRUCTURES AND UNION**

Structure - Nested structures – Pointer and Structures – Array of structures – Self referential structures – Dynamic memory allocation - Singly linked list - typedef – Union –Storage classes and visibility

**4.1 Introduction**

Using C language we can create new data types. These data types are known as User Defined data types & can be created by using Structures, Unions & Enumerations.

**Need for Structure**

Arrays can store data of same data type. They can’t be used to store data of different data types. For this Structures are used.

**Structures**

A structure is a collection of variables of different types under a single name. It is used for storing different types of data.

3 aspects:

1. Defining a structure type

2. Declaring variables

3. Using & performing operations.

**4.1.1 Structure Definition**

The structure can be defined with the keyword **struct** followed by the name of structure and opening brace with data elements of different type then closing brace with semicolon.

**General Form**

struct [structure tag name]

{

type membername1;

type membername2;

……

}[variable name];

**E.g.**

struct book

{

char title[25];

int pages;

float price;

};

• Structure definition does not reserve any space in the memory.

• It is not possible to initialize the structure members during the structure definition.

• A structure definition must always be terminated with a semicolon.

**Rules for Structure members**

1. A structure can have data of different types

2. A structure can’t contain an instance of itself.

E.g.

struct box

{

struct box a; // not possible

};

3. A structure can contain members of other complete types. E.g.

struct name

{

char firstname[20];

char lastname[20];

};

struct person

{

struct name personname;

float salary;

}

4. A structure can contain a pointer to itself;

**4.1.2 Declaration**

Variables of structure type can be declared either at the time of structure definition or after the structure definition.

**General Form**

struct structurename variablename[=initialization list];

E.g.

struct book b1,b2;

struct book b3={“CP”,500,385.00};

**Accessing Members of Structure**

There are two types of operators used for accessing members of a structure. 1. Direct member access operator (dot operator) (.)

2. Indirect member access operator (arrow operator) (->)

**Using Dot operator**

**General form:**

structure variable name.member variable name

E.g.

Suppose, we want to access title of structure variable b1, then, it can be accessed as:

b1.title

We can also directly assign values to members.

b1.title= “CP”;

**4.2 Structures within a Structure (Nested Structures)**

A structure can be nested within another structure. Structure within structure is known as nested structure i.e.) one structure can be declared inside other.

**Example program:**

#include<stdio.h>

struct name

{

char fname[20],lastname[20];

};

struct student

{

int sno,m1,m2,m3;

int tot;

float avg;

struct name sname;

};

void main()

{

struct student s[10];

float,avg;

int n,i;

printf(“Enter the number of students \n”);

scanf(“%d”,&n);

for(i=0;i<n;i++)

{

printf(“Enter student details \n”);

scanf(“%d”,&s[i].sno);

scanf((“%d%d%d”,&s[i].m1, &s[i].m2, &s[i].m3);

scanf(“%s”, s[i].sname.fname);

scanf((“%s”,s[i].sname.lastname);

s[i].tot=s[i].m1+s[i].m2+s[i].m3;

s[i].avg=s[i].tot/6.0;

}

printf(“Student Mark lists\n”);

for(i=0;i<n;i++)

printf(“%s\t%s\t%f”,s[i].sname.fname, s[i].sname.lastname,s[i].avg); }

Output:

Enter the number of students

2

Enter student details

1 70 58 68 AjithKesav

2 90 86 95 RishikKesav

Student Mark lists

AjithKesav 70.000

RishikKesav 90.000

**Accessing members in nested structure**

outerstructure variable.innerstructure variable.member name

E.g) s[i].sname.lastname





**4.3 Pointer and Structures**





It is possible to create a pointer to a structure. A Structure containing a member that is a pointer to the same structure type is called self referential structure. A pointer variable for the structure can be declared by placing an asterisk(\*) in front of the structure pointer variable.

**Syntax:**

struct namedstructuretype

\*identifiername;

Example:

struct struct\_name

{

data\_type member\_name1;

data\_type member\_name2;

.....................................

}\*ptr;

OR

struct struct\_name \*ptr;

Dot(.) operator is used to access the data using normal structure variable and arrow (->) is used to access the data using pointer variable.

**1. Illustration of Structures using pointers**

#include <stdio.h>

#include <string.h>

struct student

{

int id;

char name[30];

float percentage;

};

int main()

{

int i;

struct student record1 = {1, "Raju", 90.5};

struct student \*ptr;

ptr = &record1;

printf("Records of Student: \n");

printf(" Id is: %d \n", ptr->id);

printf(" Name is: %s \n", ptr->name);

printf(" Percentage is: %f \n\n", ptr->percentage);

return 0;

}

Output:

**Records of Student:**

Id is: 1

Name is: Sankar

Percentage is: 90.500000

**4.4 Array of Structures**

Array of Structures is nothing but a collection of structures. It is an array whose elements are of structure type. This is also called as Structure Array in C.

Consider the structure type struct student. This structure contains student information like student name, s.no etc.

struct student

{

char name[20];

int sno, m1, m2, m3;

float average;

};

Using a single structure variable we can store single student details. To store information about several students, we have to create a separate variable for each student. It is not feasible. So array of structures are used.

**General form**

struct structurename arrayname[size];

**E.g. Program:**

#include<stdio.h>

#include<conio.h>

struct employee

{

char name[15];

int empid,bsal;

float net,gross;

};

void main()

{

struct employee emp[10];

float hra,da,tax;

int n,i,j;

clrscr();

printf("Enter the number of employees\n");

scanf("%d",&n);

for(i=1;i<=n;i++)

{

printf("\nEnter the employee name");

scanf("%s",emp[i].name);

printf("\nEnter the employee id");

scanf("%d",&emp[i].empid);

printf("\nEnter the basic salary");

scanf("%d",&emp[i].bsal);

hra=((10\*emp[i].bsal)/100);

da=((35\*emp[i].bsal)/100);

tax=((15\*emp[i].bsal)/100);

emp[i].gross=emp[i].bsal+hra+da;

emp[i].net=emp[i].gross-tax;

}

printf("Employee Name Employee ID Employee Net Salary \n"); for(i=1;i<=n;i++)

printf("%s\t\t%d\t\t%f\n",emp[i].name,emp[i].empid,emp[i].net); getch();

}

**Output:**

Enter the number of Employees

2

Enter the employee name

Anu

Enter the employee id

01

Enter the basic salary

1000

Enter the employee name

Meena

Enter the employee id

02

Enter the basic salary

2000

Employee Name Employee ID Net Salary

Anu 01 1300.000

Meena 02 2600.000

**4.5 Example Program using structures and pointers**

**1. C program to read and print employee's record using structure** #include <stdio.h>

/\*structure declaration\*/

struct employee{

char name[30];

int empId;

float salary;

};

int main()

{

/\*declare structure variable\*/

struct employee emp;

/\*read employee details\*/

printf("\nEnter details :\n");

printf("Name ?:"); gets(emp.name);

printf("ID ?:"); scanf("%d",&emp.empId);

printf("Salary ?:"); scanf("%f",&emp.salary);

/\*print employee details\*/

printf("\nEntered detail is:");

printf("Name: %s" ,emp.name);

printf("Id: %d" ,emp.empId);

printf("Salary: %f\n",emp.salary);

return 0;

}

Output:

Enter details :

Name ?:Raju

ID ?:007

Salary ?:76543

Entered detail is:

Name: Raju

Id: 007

Salary: 76543.000000

**4.6 Self referential structures**

Self referential Structures are those structures that contain a reference to data of its same type. i.e in addition to other data a self referential structure contains a pointer to a data that it of the same type as that of the structure. For example: consider the structure node given as follows:

struct node

{

int val;

struct node \*next;

};

Here the structure node will contain two types of data an integer val and next which is a pointer a node. Self referential structure is the foundation of other data structures.

**4.7 Dynamic memory allocation**

Dynamic memory allocation refers to the process of manual memory management (allocation and deallocation).

The functions supports for dynamic memory allocation are,

1. malloc()

2. calloc()

3. realloc()

4. free()

**1. malloc() function**

malloc() allocates N bytes in memory and return pointer to allocated memory. The returned pointer contains link/handle to the allocated memory.

void \* malloc(number\_of\_bytes);

• It returns void pointer (generic pointer). Which means we can easily typecast it to any other pointer types.

• It accepts an integer number\_of\_bytes, i.e. total bytes to allocate in memory. **Note:** malloc() returns NULL pointer on failure.

**Example**

int N = 10; // Number of bytes to allocate

int \*ptr; // Pointer variable to store address

ptr = (int \*) malloc(N \* sizeof(int)); // Allocate 10 \* 4 bytes in memory

Here,

• ptr is a pointer to integer to store address of the allocated memory. • (int \*) is typecast required. As, I mentioned above that malloc() return void \*. Hence, to work with void pointer we must typecast it to suitable type. • N \* sizeof(int) - Since size of int is not fixed on all compilers. Hence, to get size of integer on current compiler I have used sizeof() operator.

2. **calloc() function**

calloc() function allocates memory contiguously. It allocates multiple memory blocks and initializes all blocks with 0 (NULL).

**Note:** malloc() allocates uninitialized memory blocks.

**Syntax**

void\* calloc(number\_of\_blocks, number\_of\_bytes);

Here,

• Similar to malloc() it returns void pointer.

• It accepts two parameters number\_of\_blocks i.e. total blocks to allocate and number\_of\_bytes i.e. bytes to allocate per block.

Therefore, you can say that calloc() will allocate

total (number\_of\_blocks \* number\_of\_bytes) bytes. Each block initialized with 0 (NULL).

**Example:**

int \*ptr;

ptr = (int \*) calloc(N, sizeof(int));

Here, all memory blocks are initialized with 0.

**3. realloc() function**

When working with huge data and if the allocated memory is not sufficient to store data. In that case, we need to alter/update the size of an existing allocated memory blocks (which has been created by either malloc() or calloc()).

We use realloc() function to alter/update the size of exiting allocated memory blocks. The function may resize or move the allocated memory blocks to a new location. **Syntax**

void\* realloc(ptr, updated\_memory\_size);

• Similar to all other functions for Dynamic Memory Allocation in C, it returns void pointer. Which points to the address of existing or newly allocated memory. • ptr is a pointer to memory block of previously allocated memory. • updated\_memory\_size is new (existing + new) size of the memory block. **Example**

// Original memory blocks allocation

int N = 10;

int \*ptr;

ptr = (int \*) malloc(N \* sizeof(int));

// Increase the value of N

N = 50;

// Reallocate memory blocks

ptr = (int \*) realloc(ptr, N \* sizeof(int));

4. **free() function**

C programming has a built-in library function free() to clear or release the unused memory.

The free() function clears the pointer (assigns NULL to the pointer) to clear the dynamically allocated memory. If pointer contains NULL, then free() does nothing (because pointer will not be pointing at any memory addresses). If it contains any address of dynamically allocated memory, free() will clear pointer by assigning NULL.

**Syntax**

free(ptr);

The function accepts a void pointer ptr. It points to previously allocated memory using any of Dynamic Memory Allocation functions in C.

**Example:**

int N=10;

int \*ptr;

// Allocate memory using malloc

ptr=(int \*) malloc (N\* size of (int));

//Free allocated memory

free(ptr);

**4.8 Singly Linked List**

• A linked list in simple terms is a linear collection of data elements. These data elements are called nodes.

• Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as building block to implement data structures like stacks, queues and their variations.

• A linked list can be perceived as a train or a sequence of nodes in which each node contain one or more data fields and a pointer to the next node.

**START**

****1 2 3 4 5 6 7 **X**

In the above linked list, every node contains two parts- one integer and the other a pointer to the next node. The left part of the node which contains data may include a simple data type, an array or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL.

A **singly linked list** is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node we mean that the node stores the address of the next node in sequence.

**4.8.1 Traversing a singly linked list**

Traversing a linked list means accessing the nodes of the list in order to perform some operations on them. A Linked list always contains a pointer variable START which stores the address of the first node of the list. The end of the list is marked by string NULL or -1 in the NEXT field of the last node. For traversing the singly linked list, we make use of another pointer variable PTR which points to the node that is correctly being accessed. The algorithm to traverse a linked list is shown below:

**Algorithm for traversing a linked list**

**Step 1: [INITIALIZE] SET PTR = START**

**Step 2: Repeat Steps 3 and 4 while PTR != NULL**

**Step 3: Apply Process to PTR->DATA**

**Step 4: SET PTR = PTR->NEXT**

**[END OF LOOP]**

**Step 5: EXIT**

In this algorithm, we first initialize PTR with the address of start. So now PTR points to the first node of the linked list.

Then in step 2 while loop is executed which is repeated till PTR processes the last node, that is, until it encounters NULL.

In step 3, we apply the process to the current node.

In step 4, we move to the next node by making PTR point to the node whose address is stored in the NEXT field.

The algorithm print the information stored in each node of the linked list is shown below:

**Algorithm to print the information stored in**

**each node of the linked list**

**Step 1: [INITIALIZE] SET PTR = START**

**Step 2: Repeat Steps 3 and 4 while PTR != NULL**

**Step 3: Write PTR->DATA**

**Step 4: SET PTR = PTR->NEXT**

**[END OF LOOP]**

**Step 5: EXIT**

We will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Figure below shows the algorithm to print the number of nodes in a linked list.

**Algorithm to print the number of nodes in the linked list Step 1: [INITIALIZE] SET Count = 0**

**Step 2: [INITIALIZE] SET PTR = START**

**Step 3: Repeat Steps 4 and 5 while PTR != NULL**

**~~Step 4: SET Count = Count + 1~~**

**Step 5: SET PTR = PTR->NEXT**

**[END OF LOOP]**

**Step 6: EXIT**

**4.8.2 Searching for a value in a Linked list**

Searching a linked list means to find a particular element in the linked list. A linked list consists of two parts – the DATA part and NEXT part, where DATA stores the relevant information and NEXT stores the address of the next node in the sequence. Figure below shows the algorithm to search a linked list.

**Algorithm to search an unsorted linked list**

**Step 1: [INITIALIZE] SET PTR = START**

**Step 2: Repeat Steps 3 while PTR != NULL**

**Step 3: IF VAL = PTR->DATA**

**SET POS = PTR**

**Go To Step 5**

**ELSE**

**SET PTR = PTR->NEXT**

**[END OF IF]**

**[END OF LOOP]**

**Step 4: SET POS = NULL**

**Step 5: EXIT**

Consider the linked list shown in figure we have val=4, then the flow of the algorithm can be explained as shown in figure

1 7 3 4 2 6 5 **X **1 7 3 4 2 6 5 **X**

1 7 3 4 2 6 5 **X 4.8.3 Insertion in a Singly Linked List** 

****1 7 3 4 2 6 5 **X**

Insert a new node at the head of the list is straightforward. The main idea is that we create a new node, set its next link to refer to the current head, and then set head to point to the new node.

***Algorithm addFirst(String newData):***

***create a new node v containing newData***

***v.setNext(head)***

***head = v***

***size = size + 1***

**4.8.3 Insertion at the tail**

If we keep a reference to the tail node, then it would be easy to insert an element at the tail of the list. Assume we keep a tail node in the class of SLinkedList, the idea is to create a new node, assign its next reference to point to a null object, set the next reference of the tail to point to this new object, and then assign the tail reference itself to this new node. Initially both head and tail point to null object.

***Algorithm addLast(String newData):***

***create a new node v containing newData***

***v.setNext(null)***

***if (head == null) { // list is empty***

***head = v***

***} else { // list is not empty***

***tail.setNext(v)***

***}***

***tail = v***

***size = size + 1***

**4.8.4 Deletion in a Singly Linked List**

***Deletion at the head***

Removal of an element at the head of a singly linked list is relatively easy. However removing a tail node is not easy.

***Algorithm removeFirst()***

***if (head = = null) then***

***Indicate an error: the list is empty***

***tmp = head***

***head = head.getNext()***

***tmp.setNext(null)***

***size = size - 1***

**Example:**

include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

struct test\_struct

{

int val;

struct test\_struct \*next;

};

struct test\_struct \*head = NULL;

struct test\_struct \*curr = NULL;

struct test\_struct\* create\_list(int val)

{

printf("\n creating list with headnode as [%d]\n",val);

struct test\_struct \*ptr = (struct test\_struct\*)malloc(sizeof(struct test\_struct)); if(NULL == ptr)

{

printf("\n Node creation failed \n"); return NULL;

}

ptr->val = val;

ptr->next = NULL;

head = curr = ptr;

return ptr;

}

struct test\_struct\* add\_to\_list(int val, bool add\_to\_end)

{

if(NULL == head)

{

return (create\_list(val));

}

if(add\_to\_end)

printf("\n Adding node to end of list with value [%d]\n",val); else printf("\n Adding node to beginning of list with value [%d]\n",val); struct test\_struct \*ptr = (struct test\_struct\*)malloc(sizeof(struct test\_struct)); if(NULL == ptr)

{ printf("\n Node creation failed \n"); return NULL;

}

ptr->val = val;

ptr->next = NULL;

if(add\_to\_end)

{ curr->next = ptr; curr = ptr;

}

else { ptr->next = head; head = ptr;

}

return ptr;

}

struct test\_struct\* search\_in\_list(int val, struct test\_struct \*\*prev) {

struct test\_struct \*ptr = head;

struct test\_struct \*tmp = NULL;

bool found = false;

printf("\n Searching the list for value [%d] \n",val);

while(ptr != NULL)

{

if(ptr->val == val)

{ found = true; break;

}

else

{

tmp = ptr; ptr = ptr->next;

}

}

if(true == found)

{

if(prev) \*prev = tmp;

return ptr;

}

else

{

return NULL; } }

int delete\_from\_list(int val)

{

struct test\_struct \*prev = NULL;

struct test\_struct \*del = NULL;

printf("\n Deleting value [%d] from list\n",val); del = search\_in\_list(val,&prev);

if(del == NULL) { return -1; }

else

{

if(prev != NULL)

prev->next = del->next;

if(del == curr)

{ curr = prev;

}

else if(del == head)

{

head = del->next;

} }

free(del);

del = NULL; return 0;

}

void print\_list(void) {

struct test\_struct \*ptr = head;

printf("\n -------Printing list Start------- \n");

while(ptr != NULL)

{

printf("\n [%d] \n",ptr->val);

ptr = ptr->next;

}

printf("\n -------Printing list End------- \n"); return; }

int main(void)

{

int i = 0, ret = 0;

struct test\_struct \*ptr = NULL;

print\_list();

for(i = 5; i<10; i++)

add\_to\_list(i,true); print\_list();

for(i = 4; i>0; i--)

add\_to\_list(i,false);

print\_list();

for(i = 1; i<10; i += 4)

{ ptr = search\_in\_list(i, NULL);

if(NULL == ptr)

{

printf("\n Search [val = %d] failed, no such element found\n",i);

}

else

{

printf("\n Search passed [val = %d]\n",ptr->val); }

print\_list(); ret = delete\_from\_list(i);

if(ret != 0) { printf("\n delete [val = %d] failed, no such element found\n",i); }

else

{

printf("\n delete [val = %d] passed \n",i); } print\_list(); } return 0; }

**4.9 Typedef**

The typedef keyword enables the programmer to create a new data type name by using an existing data type.

By using typedef, no new data is created, rather an alternate name is given to a known data type.

***Syntax: typedef existing\_data\_type new\_data\_type;***

It is used to create a new data using the existing type.

**Syntax**: typedef data type name;

**Example**: typedef int hours: hours hrs;/

\* Now, hours can be used as new datatype \*/

**UNION**

* **A union is a collection of dissimilar elements that are stored under a common / single name and each member occupy same memory space.**
* **It’s a collection of heterogeneous data.**
* **Keyword : union is used.**
* **Union may contain int, float, char, etc.**
* **Each element in structure is called union members.**
* **It provides a convenient way of grouping several pieces of related information together.**

union union\_name union union\_name

{ {

datatype element1; datatype element1;

datatype element2; (OR) datatype element2;

……………………. …………………….

}; } variable;

void main()

{

union union\_name variable;

}

Example:

#include<stdio.h>

union student

{

int rollnum;

float cgpa;

}s;

void main()

{

scanf(“%d”,&s.rollnum);

printf(“%d”,s.rollnum);

scanf(“%f”,&s.cgpa);

printf(“%d”, s.cgpa);

}

**STORAGE CLASSES**

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

Automatic

External

Static

Register

* Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C
* Automatic
* External
* Static
* Register

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Storage Classes** | **Storage Place** | **Default Value** | **Scope** | **Lifetime** |
| auto | RAM | Garbage Value | Local | Within function |
| extern | RAM | Zero | Global | Till the end of the main program Maybe declared anywhere in the program |
| static | RAM | Zero | Local | Till the end of the main program, Retains value between multiple functions call |
| register | Register | Garbage Value | Local | Within the function |

AUTOMATIC:

Automatic variables are allocated memory automatically at runtime.

The visibility of the automatic variables is limited to the block in which they are defined.

The scope of the automatic variables is limited to the block in which they are defined.

The automatic variables are initialized to garbage by default.

The memory assigned to automatic variables gets freed upon exiting from the block.

The keyword used for defining automatic variables is auto.

Every local variable is automatic in C by default.

EXAMPLE:

#include <stdio.h>

int main()

{

int a; //auto

char b;

float c;

printf("%d %c %f“,a,b,c); // printing initial default value of automatic variables

a, b, and c.

return 0;

}

Output:

garbage garbage garbage

STATIC:

The variables defined as static specifier can hold their value between the multiple function calls.

Static local variables are visible only to the function or the block in which they are defined.

A same static variable can be declared many times but can be assigned at only one time.

Default initial value of the static integral variable is 0 otherwise null.

The visibility of the static global variable is limited to the file in which it has declared.

The keyword used to define static variable is static.

EXAMPLE:

#include<stdio.h>

void sum()

{

static int a = 10;

static int b = 24;

printf("%d %d \n",a,b);

a++;

b++;

}

void main()

{

int i;

for(i = 0; i< 3; i++)

{

sum(); // The static variables holds their value between multiple function calls.

}

}

Output:

10 24

11 25

12 26

REGISTER:

The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.

We can not dereference the register variables, i.e., we can not use &operator for the register variable.

The access time of the register variables is faster than the automatic variables.

The initial default value of the register local variables is 0.

The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler?s choice whether or not; the variables can be stored in the register.

We can store pointers into the register, i.e., a register can store the address of a variable.

Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

EXAMPLE:

#include <stdio.h>

int main()

{

register int a = 0;

printf("%u",&a); // This will give a compile time error since we can not access  the address of a register variable.

}

Output:

main.c:5:5: error: address of register variable ?a? requested

printf("%u",&a);

^~~~~~

EXAMPLE:

The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.

The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.

The default initial value of external integral type is 0 otherwise null.

We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.

An external variable can be declared many times but can be initialized at only once.

If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

EXAMPLE:

#include <stdio.h>

int main()

{

extern int a;

printf("%d",a);

}

Output

main.c:(.text+0x6): undefined reference to `a'

collect2: error: ld returned 1 exit status