Structure

# Structures in C

* user defined data type in C/C++
* used to group items of possibly different types into a single type

# Structures definition

struct <strct\_name>

{

data\_type memeber1;

data\_type memeber2;

.......

.......

};

# Declare structure variables

* can either be declared with structure declaration or as a separate declaration like basic types.
* In C++, the struct keyword is optional before in declaration of variable.

In C, it is mandatory.

struct Point

{

int x, y;

} p1; // p1 is declared with structure 'Point'

int main() {

struct Point p1; // The variable p1 is declared like a normal variable

Point p2 // C++ struct is optional

}

# Initialize structure members

* In C, Structure members cannot be initialized with declaration
* Reason, structure is a user defined data type and
* when a datatype is declared, no memory is allocated for it,
* Memory is allocated only when variables are created
* can be initialized using curly braces ‘{}’

struct Point

{

int x = 0; // COMPILER ERROR: cannot initialize members here

// gcc : warning: non-static data member initializers

only available with –std=c++11 or -std=gnu++11

int y = 0; // COMPILER ERROR: cannot initialize members here

};

int main() {

struct Point p1 = {0, 1}; // The order of declaration is followed.

}

## Designated Initialization

* allows structure members to be initialized in any order
* added in C99 standard
* not available in C++ and works only in C.
* In C++ Compilation error : sorry, unimplemented: non-trivial designated initializers not supported

int main()

{

// Examples of initializtion using designated initialization

struct Point p1 = {.y = 0, .z = 1, .x = 2};

struct Point p2 = {.x = 20};

printf ("x = %d, y = %d, z = %d\n", p1.x, p1.y, p1.z);

printf ("x = %d", p2.x);

return 0;

}

Output:

x = 2, y = 0, z = 1

x = 20

# Access structure elements

* using dot (.) operator for variables to structure
* using arrow ( -> ) operator for pointer to structure

struct Point{int x, y;};

int main() {

struct Point p1 = {0, 1};

struct Point \*p2 = &p1; // p2 is a pointer to structure p1

p1.x = 20; p1.y = 15;

printf("x = %d, y = %d\n", p1.x, p1.y); // using .

printf("x = %d, y = %d\n", p2->x, p2->y); // using ->

return 0;

}

Output:

x = 20, y = 15

x = 20, y = 15

# Operations on struct variables in C

* only assignment operation that can be applied to struct variables
* Any other operation (e.g. equality check) is not allowed

#include <stdio.h>

struct Point { int x; int y; };

int main()

{

struct Point p1 = {10, 20};

struct Point p2 = p1; // works: contents of p1 are copied to p1

printf(" p2.x = %d, p2.y = %d", p2.x, p2.y);

/\*

if (p1 == p2) // In C compiler error: invalid operands to binary == (have 'struct Point' and

'struct Point')

// In C++ error: no match for 'operator==' (operand types are 'Point' and 'Point')

printf("p1 and p2 are same ");

\*/

return 0;

}

# Array of structures

we can create an array of structures

#include <stdio.h>

struct Point{int x, y;};

int main()

{

struct Point arr[10]; // array of structures

arr[0].x = 10; arr[0].y = 20; // Access array members

arr[1].x = 17; arr[1].y = 34;

printf("%d %d\n", arr[0].x, arr[0].y);

printf("%d %d", arr[1].x, arr[1].y);

return 0;

}

Output:

10 20

17 34

# Size of structure

1. By observation

Calculate by adding Individual Sizes

1. By Using sizeof Operator

sizeof is not a library function

sizeof is C Operator.

1. Without using sizeof Operator, using array of structure

struct Book

{

int pages;

char name[10];

char author[10];

float price;

}b1;

Size = size of 'Pages' + size of 'Name' +

size of 'Author' + size of 'Price'

= 4 + 10 \* 1 + 10 \* 1 + 4

= 4 + 10 + 10 + 4

= 28

#include<stdio.h>

typedef struct b1 { // typedef can be used with structure

int ssn;

int pages;

char bname[30];

}BOOK;

int main()

{

BOOK b1;

printf("\nSize of Structure : %lu",sizeof(b1));

BOOK a[2];

int start,last;

start = &a[1].ssn;

last = &a[0].ssn;

printf("\nSize of Structure : %d Bytes",start-last);

return(0);

}

# Nested Structure

* Structure written inside another structure is called as nesting of two structures.
* Nested Structures are allowed in C Programming Language.
* 2 way to declare nested structure:

1. Declare two separate structures
2. Declare Embedded structures

We can write one Structure inside another structure as member of another structure.

struct Employee {

char ename[20];

int ssn;

float salary;

struct date {

int date;

int month;

int year;

}doj;

}emp = {"Pritesh",1000,1000.50,{22,6,1990}};

int main(int argc, char \*argv[])

{

printf("\nEmployee Name : %s",emp.ename);

printf("\nEmployee SSN : %d",emp.ssn);

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

printf("\nEmployee DOJ : %d/%d/%d", \

emp.doj.date,emp.doj.month,emp.doj.year);

return 0;

}

Output :

Employee Name : Pritesh

Employee SSN : 1000

Employee Salary : 1000.500000

Employee DOJ : 22/6/1990

# Anonymous Structure in C

also known as unnamed structures

// like anonymous structure, anonymous union is also similar

// Anonymous structure declaration

struct { char alpha; int num; };

* Since there is no names, direct objects(or variables) of them are not created and we use them in nested structure or unions
* Since there is no variable and no name, we can directly access members

#include<stdio.h>

struct Scope {

// Anonymous structure

Struct {

char alpha;

int num;

};

};

int main() {

struct Scope x;

x.num = 65;

x.alpha = 'B';

printf("x.alpha = %c, x.num = %d", x.alpha, x.num);

return 0;

}

Output:

x.alpha = B, x.num = 65

* **This is a C only feature**
* Anonymous Unions and Structures are NOT part of C++ 11 standard, but most of the C++ compilers support them.
* The C++ implementations don’t allow to anonymous struct/union to have private or protected members, static members, and functions.

# Structure is used in C programming for following purposes –

* Clearing screen
* Adjusting Cursor Position
* Drawing any graphics shape on the screen
* Receiving a key from the keyboard
* Finding out the list of equipment attached to the computer
* Changing the size of the cursor
* Formatting a floppy
* Hiding a file from the directory
* Displaying the directory of a disk
* Checking the memory size
* Sending the output to printer
* Interacting with the mouse

# Similarities and difference between Structure and Union

**Similarities**:

1. Both are user-defined data types used to store data of different types as a single unit.
2. Their members can be objects of any type, including other structures and unions or arrays.

A member can also consist of a bit field.

1. Both support only assignment = and sizeof operators.

The two structures or unions in the assignment must have the same members and member types.

1. A structure or a union can be passed by value to functions and returned by value by functions.

The argument must have the same type as the function parameter.

A structure or union is passed by value just like a scalar variable as a corresponding parameter.

**Differnece**:

|  |  |  |
| --- | --- | --- |
|  | **Structure** | **Union** |
| Keyword | struct is used to define structure | union is used to define a union |
| Size | greater than or equal to the sum of size of its members | equal to the size of largest member |
| Memory | each member is assigned unique storage area of location | memory is shared by individual members |
| Value altering | altering the value of a member will not  affect other members | altering the value of any of the member will alter other member values |
| accessing members | individual members can be accessed at a time | only one member can be accessed at a time |
| Initialization of members | several members of a structure can be initialized at once | only the first member of a union can be initialized |

# Difference between C structures and C++ structures

|  |  |  |
| --- | --- | --- |
|  | C | C++ |
| Member functions  inside structure | cannot have member functions | can have member functions along with data members |
| Direct Initialization | not permitted | permitted since C++ 11 |
| variable declaration | struct keyword is necessary | struct keyword is optional |
| static member | not permitted  // C error: expected specifier-qualifier-list before 'static' | Permitted |
| sizeof operator | 0 for an empty structure in C | 1 for an empty structure in C++ |
| Data Hiding | does not allow concept of Data hiding | Data hiding is permitted |
| Access Modifiers | does not have access modifiers | access modifiers is inbuilt in |

# Flexible Array Members (FAM) in a structure in C

* Flexible Array Member(FAM) is a feature introduced in the C99 standard of the C programming language.
* For the structures in C programming language from C99 standard onwards, we can declare an array without a dimension and whose size is flexible in nature.
* Such an array inside the structure should preferably be declared as the last member of structure and its size is variable(can be changed be at runtime).
* The structure must contain at least one more named member in addition to the flexible array member.

Important Points:

* Adjacent memory locations are used to store structure members in memory.
* In previous standards of the C programming language, we were able to to declare a zero size array member in place of a flexible array member.
* The GCC compiler with C89 standard considers it as zero size array.

# Struct Hack (FAM)

struct employee {

int emp\_id;

int name\_len;

char name[0];

};

4 + 4 + 0 = 8 bytes.

* In gcc, when we create an array of zero length, it is considered as array of incomplete type
* This technique is known as “Struct Hack”
* array of zero length inside structure, must be (and only) last member of structure
* “Struct Hack” technique is used to create variable length member in a structure
* string length of “name” is not fixed, so we can use “name” as variable length array

struct employee \*e = malloc(sizeof(\*e) + sizeof(char) \* 128);

is equivalent to

struct employee

{

int emp\_id;

int name\_len;

char name[128]; /\* character array of size 128 \*/

};

When we allocate memory as given above, compiler will allocate memory to store “emp\_id” and “name\_len” plus

contiguous memory to store “name” (gcc guaranties that, “name” will get contiguous memory).

Other advantage of this is, we **can write whole data by using single “write()” call**. e.g.

write(fd, e, sizeof(\*e) + name\_len); /\* write emp\_id + name\_len + name \*/

If we use character pointer, then we need 2 write calls to write data. e.g.

write(fd, e, sizeof(\*e)); /\* write emp\_id + name\_len \*/

write(fd, e->name, e->name\_len); /\* write name \*/

if we use character pointer, there is no guarantee that character pointer will get contiguous memory

**In C99, there is feature called “flexible array members”, which works same as “Struct Hack”**

# Compound Literals in C

#include <stdio.h>

int main() {

int \*p = (int []){2, 4, 6};

printf("%d %d %d", p[0], p[1], p[2]);

return 0;

}

Output:

2 4 6

The above example is an example of compound literals.

* Compound literals were introduced in C99 standard of C.
* Compound literals feature allows us to create unnamed objects with given list of initialized values.

In the above example, an array is created without any name. Address of first element of array is assigned to pointer p.

**What is the use of it?**

* Compound literals are mainly used with structures and are particularly useful when passing structures variables to functions.
* We can pass a structure object without defining it

#include <stdio.h>

struct Point { int x, y; };

void printPoint(struct Point p) {

printf("%d, %d", p.x, p.y);

}

int main() {

printPoint((struct Point){2, 3});

/\*

Without compound literal, above statement would have been written as

struct Point temp = {2, 3};

printPoint(temp);

\*/

return 0;

}