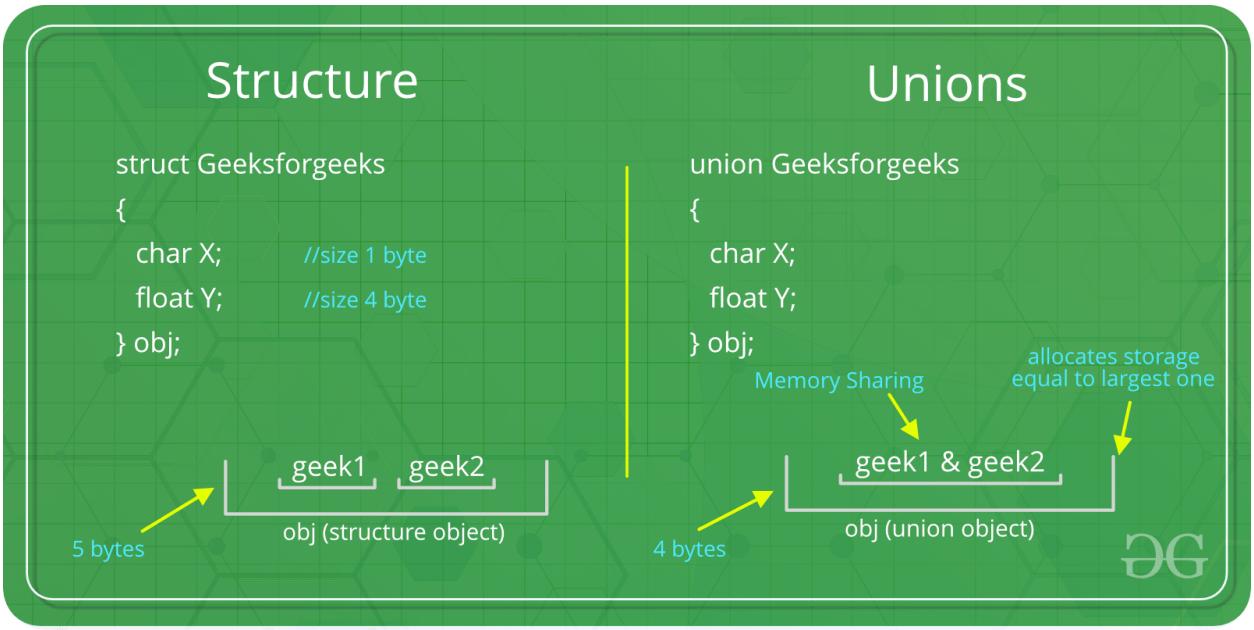
Union in C

Like Structures, union is a user defined data type. In union, all members share the same memory location. At once, only one member of the union can occupy the memory. In other words, we can say that the size of the union in any instance is equal to the size of its largest element.



Example:

```
#include <stdio.h>

// Declaration of union is same as structures
union test {
    int x;
    char y;
};

int main()
{
    // A union variable t
    union test t;

    t.x = 55; // t.y also gets value 2
    printf("After making x = 55:\n x = %d, y = %c\n\n",
           t.x, t.y);

    t.y = 'C'; // t.x is also updated to 10
    printf("After making y = 'C':\n x = %d, y = %c\n\n",
           t.x, t.y);
    return 0;
}
```

OUTPUT

```
After making x = 55:
 x = 55, y = 7

After making y = 'C':
 x = 67, y = C
```

How is the size of union decided by compiler?

```
#include <stdio.h>

union test1 {
    int x;
    int y;
} Test1;

union test2 {
    int x;
    char y;
} Test2;

union test3 {
    int arr[10];
    char y;
} Test3;

int main()
{
    printf("sizeof(test1) = %lu, sizeof(test2) = %lu, "
           "sizeof(test3) = %lu",
           sizeof(Test1),
           sizeof(Test2), sizeof(Test3));
    return 0;
}
```

OUTPUT

```
sizeof(test1) = 4, sizeof(test2) = 4, sizeof(test3) = 40
```

Advantage of union over structure

It **occupies less memory** because it occupies the size of the largest member only.

Disadvantage of union over structure

Only the last entered data can be stored in the union. It overwrites the data previously stored in the union.

Defining union

The **union** keyword is used to define the union. Let's see the syntax to define union in c.

```
union union_name
{
   data_type member1;
   data_type member2;
```

```
   .
   .
   data_type memeberN;
};
```

Let's see the example to define union for an employee in c.

```
union employee
{   int id;
   char name[50];
   float salary;
};
```

Example:

```
#include <stdio.h>
#include <string.h>
union employee
{   int id;
   char name[50];
}e1;  //declaring e1 variable for union
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  return 0;
}
```

OUTPUT

```
employee 1 id : 1869508435
employee 1 name : Sonoo Jaiswal
```

Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

```
union car
{
  char name[50];
  int price;
```

```
};

int main()
{
  union car car1, car2, *car3;
  return 0;
}
```

Another way of creating union variables is:

```
union car
{
  char name[50];
  int price;
} car1, car2, *car3;
```

In both cases, union variables *car1*, *car2*, and a union pointer *car3* of `union car` type are created.

Initializing the members of union

like we initialize the structure elements, we can initialize the union elements too, but any one member of union.

union category {
int intClass;
char chrDeptId [10];
} student_category = {10}; // this will initialize intClass

Here student _category is the union variable and is initialized its element intClass to 10. Since we have not specified any element name while initializing, it initializes first element.
union category {
int intClass;
char chrDeptId [10];
};
union category student_category = {10}; // this will initialize intClass

By the above method we can only initialize the first element of the union.

By specifying element names

This is another method of initializing the elements of the union. Here we can explicitly specify the member names to which we need to assign values.

union category {
int intClass;
char chrDeptId [10];
};

```
union category student_category = {.intClass= 10}; // this will initialize intClass
union category emp_categeory = {.chrDeptId= "DEPT_100”}; // this will initialize chrDeptId
```

OR

```
union category {
int intClass;
char chrDeptId [10];
};
union category student_category;
student_category.intClass = 10; // this will initialize intClass
strcpy (student_category.chrDeptId, “DEPT_100”); // this will initialize chrDeptId on union variable student_category, but it will overwrite the value of intClass which will have garbage value now
```

Access members of a union

We use the . operator to access members of a union. To access pointer variables, we use also use the -> operator.

In the above example,

- To access *price* for car1, car1.price is used.
- To access *price* using car3, either (*car3).price or car3->price can be used.

Example:

```
#include <stdio.h>

union test {
   int x;
   char y;
};

int main()
{
   union test p1;
   p1.x = 65;
   printf("%d %c\n", p1.x, p1.y);
   // p2 is a pointer to union p1
   union test* p2 = &p1;

   // Accessing union members using pointer
   printf("%d %c", p2->x, p2->y);
   return 0;
}
```

OUTPUT

```
65 A
```

65 A

Array of Union

Like array of structures we can also create array of unions and access them in similar way. When array of unions are created it will create each element of the array as individual unions with all the features of union. That means, each array element will be allocated a memory which is equivalent to the maximum size of the union member and any one of the union member will be accessed by array element.

Example:
union category {
int intClass;
char chrDeptId[10];
};
union category catg [10]; // creates an array of unions with 10 elements of union type

Anonymous Union and Structure in C

Anonymous unions/structures are also known as unnamed unions/structures as they don't have names. Since there is no names, direct objects(or variables) of them are not created and we use them in nested structure or unions.

Definition is just like that of a normal union just without a name or tag. For example,

```
// Anonymous union example
union
{
   char alpha;
   int num;
};
// Anonymous structure example
struct
{
   char alpha;
   int num;
};
```

Since there is no variable and no name, we can directly access members. This accessibility works only inside the scope where the anonymous union is defined.

Following is a complete working example of anonymous union.

```
// C Program to demonstrate working of anonymous union
#include<stdio.h>
struct Scope
{
    // Anonymous union
    union
    {
        char alpha;
        int num;
    };
```

```
};

int main()
{
    struct Scope x;
    x.num = 65;

    // Note that members of union are accessed directly
    printf("x.alpha = %c, x.num = %d", x.alpha, x.num);

    return 0;
}
```

OUTPUT

```
x.alpha = A, x.num = 65
```

```
// C Program to demonstrate working of anonymous struct
#include<stdio.h>
struct Scope
{
    // Anonymous structure
    struct
    {
        char alpha;
        int num;
    };
};

int main()
{
    struct Scope x;
    x.num = 65;
    x.alpha = 'B';

    // Note that members of structure are accessed directly
    printf("x.alpha = %c, x.num = %d", x.alpha, x.num);

    return 0;
}
```

OUTPUT

```
x.alpha = B, x.num = 65
```

```
#include<stdio.h>
#include<conio.h>

void main() {
  struct student {
    char name[30];
    char sex;
    int rollno;
    float percentage;
  };
```

```
  union details {
    struct student st;
  };
  union details set;

  printf("Enter details:");

  printf("\nEnter name : ");
  scanf("%s", set.st.name);
  printf("\nEnter roll no : ");
  scanf("%d", &set.st.rollno);
  printf("\nEnter sex : ");
  scanf(" %c", &set.st.sex);
  printf("\nEnter percentage :");
  scanf("%f", &set.st.percentage);

  printf("\nThe student details are : \n");
  printf("\name : %s", set.st.name);
  printf("\nRollno : %d", set.st.rollno);
  printf("\nSex : %c", set.st.sex);
  printf("\nPercentage : %f", set.st.percentage);

  getch();
}
```

OUTPUT

```
Enter details:
Enter name : Pritesh
Enter rollno: 10
Enter sex: M
Enter percentage: 89

The student details are:
Name : Pritesh
Rollno : 10
Sex : M
Percentage : 89.000000
```

typedef in C

The **typedef** is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

```
typedef <existing_name> <alias_name>
```

In the above syntax, '**existing_name'** is the name of an already existing variable while '**alias name'** is another name given to the existing variable.

For example, suppose we want to create a variable of type **unsigned int**, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use **a typedef** keyword.

```
typedef unsigned int unit;
```

In the above statements, we have declared the **unit** variable of type unsigned int by using **a typedef** keyword.

Now, we can create the variables of type **unsigned int** by writing the following statement:

```
unit a, b;
```

instead of writing:

```
unsigned int a, b;
```

Till now, we have observed that the **typedef** keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

Let's understand through a simple example.

```
#include <stdio.h>
int main()
{
typedef unsigned int unit;
unit i,j;
i=10;
j=20;
printf("Value of i is :%d",i);
printf("\nValue of j is :%d",j);
return 0;
}
```

OUTPUT

```
Value of i is :10
Value of j is :20
```

Using typedef with structures

Consider the below structure declaration:

```
struct student
{
```

```
char name[20];
int age;
};
struct student s1;
```

In the above structure declaration, we have created the variable of **student** type by writing the following statement:

```
struct student s1;
```

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the **typedef** keyword to create the variable of type **student**.

```
struct student
{
char name[20];
int age;
};
typedef struct student stud;
stud s1, s2;
```

In the above statement, we have declared the variable **stud** of type struct student. Now, we can use the **stud** variable in a program to create the variables of type **struct student**.

The above typedef can be written as:

```
typedef struct student
{
char name[20];
int age;
} stud;
stud s1,s2;
```

From the above declarations, we conclude that **typedef** keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

Let's see another example where we typedef the structure declaration.

```
#include <stdio.h>
typedef struct student
{
char name[20];
int age;
}stud;
int main()
{
```

```
stud s1;
printf("Enter the details of student s1: ");
printf("\nEnter the name of the student:");
scanf("%s",&s1.name);
printf("\nEnter the age of student:");
scanf("%d",&s1.age);
printf("\n Name of the student is : %s", s1.name);
printf("\n Age of the student is : %d", s1.age);
return 0;
}
```

OUTPUT

```
Enter the details of student s1:
Enter the name of the student: Peter
Enter the age of student: 28
Name of the student is : Peter
Age of the student is : 28
```

Using typedef with pointers

We can also provide another name or alias name to the pointer variables with the help of **the typedef**.

For example, we normally declare a pointer, as shown below:

```
int* ptr;
```

We can rename the above pointer variable as given below:

```
typedef int* ptr;
```

In the above statement, we have declared the variable of type **int***. Now, we can create the variable of type **int*** by simply using the '**ptr**' variable as shown in the below statement:

```
ptr p1, p2 ;
```

In the above statement, **p1** and **p2** are the variables of type '**ptr**'.

Bit Field in C

Bit field can be used to reduce memory consumption when it is known that only some bits would be used for a variable. Bit fields allow efficient packaging of data in the memory.

As we know, integer takes two bytes(16-bits) in memory. Some times we need to store value that takes less then 2-bytes. In such cases, there is wastages of memory. For example, if we use a variable ***temp*** to store value either 0 or 1. In this case only one bit of memory will be used rather

then 16-bits. By using bit field, we can save lot of memory. In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is with in a small range.

Syntax for bit field

```
struct struct-name
{
       datatype var1 : size of bits;
       datatype var2 : size of bits;
       - - - - - - - - - -
       - - - - - - - - - -
       datatype varN : size of bits;
};
```

Example for bit field

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

/* define simple structure */
struct {
   unsigned int widthValidated;
   unsigned int heightValidated;
} status1;

/* define a structure with bit fields */
struct {
   unsigned int widthValidated : 1;
   unsigned int heightValidated : 1;
} status2;

int main( ) {
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
   return 0;
}
```

OUTPUT

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Following are some interesting facts about bit fields in C.

1) A special unnamed bit field of size 0 is used to force alignment on next boundary. For example consider the following program.

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

```
// A structure without forced alignment
struct test1 {
    unsigned int x : 5;
    unsigned int y : 8;
};

// A structure with forced alignment
struct test2 {
    unsigned int x : 5;
    unsigned int : 0;
    unsigned int y : 8;
};

int main()
{
    printf("Size of test1 is %lu bytes\n",
           sizeof(struct test1));
    printf("Size of test2 is %lu bytes\n",
           sizeof(struct test2));
    return 0;
}
```

OUTPUT

```
Size of test1 is 4 bytes
Size of test2 is 8 bytes
```

2) We cannot have pointers to bit field members as they may not start at a byte boundary.

```
#include <stdio.h>
struct test {
    unsigned int x : 5;
    unsigned int y : 5;
    unsigned int z;
};
int main()
{
    struct test t;

    // Uncommenting the following line will make
    // the program compile and run
    printf("Address of t.x is %p", &t.x);

    // The below line works fine as z is not a
    // bit field member
    printf("Address of t.z is %p", &t.z);
    return 0;
}
```

Output:

```
prog.c: In function 'main':
prog.c:14:1: error: cannot take address of bit-field 'x'
 printf("Address of t.x is %p", &t.x);
```

```
 ^
```

3) It is implementation defined to assign an out-of-range value to a bit field member.

```
#include <stdio.h>
struct test {
    unsigned int x : 2;
    unsigned int y : 2;
    unsigned int z : 2;
};
int main()
{
    struct test t;
    t.x = 5;
    printf("%d", t.x);
    return 0;
}
```

Output:

```
Implementation-Dependent
```

4) Array of bit fields is not allowed. For example, the below program fails in the compilation.

```
struct test {
    unsigned int x[10] : 5;
};

int main()
{
}
```

Output:

```
prog.c:3:1: error: bit-field 'x' has invalid type
 unsigned int x[10]: 5;
 ^
```