Union:

Can a room be shared if each occupant stays in non-clashing time intervals? Can a IT company share its computers between day workers and night workers?

Can the variables share the same memory if they do not logically co-exist?

This concept leads us to the design of union.

Syntactically, union is similar to structure.

Union can have one or more fields of any type – but only one can logically exist at a given point in time. All these fields occupy overlapping memory locations. Each field has the same offset in the union. The offset of each field in the union is 0. The size of the union is the size of the biggest field in the union.

Let us examine the following code. The y variable of union type has fields a and b, both occupy the same location. When we assign y.a a value 111, y.b also has the same value.

```
union Y
{
        int a;
        int b;
};
union Y y;
y.a = 111; printf("union b : %d\n", y.b); // 111
```

'C' supports a concept called free union. Given a union variable, we cannot determine which of the fields is active at that point.

Some languages associate with a union a field which tells us which field of the union is active. This concept is called discriminated union. We can simulate this concept in 'C' by having a discriminator field outside the union. The user should take care of properly maintaining the union with its discriminator. We will discuss an example with discriminator later in this document.

Applications of union:

a) We may look upon 32 bit register as two 16 bit register. This concept is used heavily in Intel computers on Microsoft Windows.

We can refer to a register AX or two registers AH and AL.

b) We may want to pass different types of arguments to a function and the function may perform various operations based on an integer as well as the argument passed as union. This integer value acts like a discriminator.

Example: semun : union; used in a function (system call) semctl.

Check the man page.

c) We may use union to provide a uniform interface for the user – thus providing flexibility. We will show that in an example that follows.

```
#include <stdio.h>
#include <stddef.h>
// union:
//       have # of fields - similar to a structure
//  at a given point in time, only one can exist
//  all the fields overlap
//  they have the same offset : 0
// size of a union : size of the biggest component
//  - memory management technique
//              allow variables to occupy the same memory
//              fine as long as all these variables do not have to exist
simultaneously
//  - flexibility

struct X
{
        int a;
        int b;
};

union Y
{
        int a;
        int b;
};

union Z
{
        double a;
        int b;
};
// examples of union
// register : 32 bit
//          or two 16 bit
union regs
{
        int AX;
        struct
        {
                short AL;
                short AH;
        } A;
};
// interprocess communication
```

```
// man semctl
// union semun { }
// flexibility : pass the same argument
// based on an argument called command, we make out what the union
contains

int main()
{
	printf("posn of b in struct : %lu\n", offsetof(struct X, b));
	// this may be non zero unless it is the first field of struct X
	printf("posn of b in union : %lu\n", offsetof(union Y, b));
	// this is always zero

	struct X x;
	x.a = 111; printf("struct b : %d\n", x.b); // undefined
	union Y y;
	y.a = 111; printf("union b : %d\n", y.b); // 111

	// cannot make out which field of the union is in use currently
	// concept : free union
	// may require a variable outside the union to make out which field of the
union is active
	//	discriminator
	union Z z;
	z.a = 3.14; printf("a : %lf\n", z.a);
	z.b = 123; printf("b : %d\n", z.b);

	union regs r;
	r.AX = 0x12345678;
	printf("%x %x %x\n", r.AX, r.A.AH, r.A.AL);
}
```

Let us try to provide a uniform interface for the user to play with different shapes like rectangle and circle.

We have the rectangle interface with the type RECT_H and functions read, disp and find_area.

```
#ifndef RECT_H
#define RECT_H
struct Rect
{
	double l;
	double b;
};
typedef struct Rect rect_t;
void read_rect(rect_t*);
```

```
void disp_rect(const rect_t*);
double find_area_rect(const rect_t*);
#endif
```

This is the rectangle implementation.

```
#include <stdio.h>
#include "1_rect.h"

void read_rect(rect_t* p)
{
	scanf("%lf %lf", &p->l, &p->b);
}
void disp_rect(const rect_t* p)
{
	printf("%lf %lf\n", p->l, p->b);
}
double find_area_rect(const rect_t* p)
{
	return p->l * p->b;
}
```

Similarly we have the circle interface.

```
#ifndef CIRCLE_H
#define CIRCLE_H
struct Circle
{
	double r;

};
typedef struct Circle circle_t;
void read_circle(circle_t*);
void disp_circle(const circle_t*);
double find_area_circle(const circle_t*);
#endif
```

We have the circle implementation.

```
#include <stdio.h>
#include "1_circle.h"

void read_circle(circle_t* p)
{
	scanf("%lf", &p->r);
}
void disp_circle(const circle_t* p)
{
	printf("%lf\n", p->r);
}
double find_area_circle(const circle_t* p)
{
```

```
	return 3.14 * p->r * p->r;
}
```

This is the user interface for shape.

The union shape_t at a given point in time can have a rectangle or a circle.

But, we should be able to make out which is being used currently. So, we put the union within a structure shapeinfo_t which has an additional field called type. The value of type indicates which of the shapes is currently active. This field acts like a discriminator.

The user interface is exactly same whether the user wants to use a rectangle or a circle. Observe the shape interface.

```
#ifndef SHAPE_H
#define SHAPE_H
#include "1_rect.h"
#include "1_circle.h"

union Shape
{
	rect_t r;
	circle_t c;
};
typedef union Shape shape_t;
struct ShapeInfo
{
	shape_t s;
	char type; // r for rect and c for circle
};
typedef struct ShapeInfo shapeinfo_t;
void read_shapeinfo(shapeinfo_t*);
void disp_shapeinfo(const shapeinfo_t*);
double find_area_shapeinfo(const shapeinfo_t*);
#endif
```

This is the implementation of shape. Based on the discriminator, the functions of the corresponding shape are invoked.

```
#include <stdio.h>
#include "1_shape.h"

void read_shapeinfo(shapeinfo_t* p)
{
	if(p->type == 'r')
	{
		read_rect(&p->s.r);
	}
```

```
	else if(p->type == 'c')
	{
		read_circle(&p->s.c);
	}
	else
	{
		printf("type error\n");
	}
}
void disp_shapeinfo(const shapeinfo_t* p)
{
	if(p->type == 'r')
	{
		disp_rect(&p->s.r);
	}
	else if(p->type == 'c')
	{
		disp_circle(&p->s.c);
	}
	else
	{
		printf("type error\n");
	}
}
double find_area_shapeinfo(const shapeinfo_t* p)
{
	if(p->type == 'r')
	{
		return find_area_rect(&p->s.r);
	}
	else if(p->type == 'c')
	{
		return find_area_circle(&p->s.c);
	}
	else
	{
		return 0.0;
	}

}
```

Now, at last, the client code.

```
#include <stdio.h>
#include "1_shape.h"

int main()
{
	shapeinfo_t s;
	s.type = 'r';
	read_shapeinfo(&s);
```

```
	disp_shapeinfo(&s);
	printf("area : %lf\n", find_area_shapeinfo(&s));

	s.type = 'c';
	read_shapeinfo(&s);
	disp_shapeinfo(&s);
	printf("area : %lf\n", find_area_shapeinfo(&s));

}
```

---

bit fields:

There are cases where we may require very very small integers. Instead of creating lots of integer variables – each of which could occupy a word, we put these together with a word itself.

We specify the size of each integer in terms of bits. Otherwise the syntax of bit fields is similar to that of a structure.

A few points to note about bit fields.

- Bit fields are normally unsigned int.
- Each bit field occupies a few bits.
- As these fields occupy part of a byte, they do not have addresses. Address operator on a bit field results in a syntax error.
- Overflow of a bit field may result in lower bits being stored in the bit field.
- A variable of bit field may occupy one or more words.
- Bit fields cannot be arrays.
-

```
#include <stdio.h>
// bit fields
//	allow for storing small integer values in named locations
//  allow # of small integers to be put in a word
//  optimize on space for small integers
//  flexibility with names

// syntactically , similar to structure
struct myfields
{
	unsigned int a : 4;
	unsigned int b : 5;
	unsigned int c : 25;
};
int main()
{
	struct myfields f;
	f.a = 5; // 0101
	f.b = 9; // 01001
	printf("%d %d\n", f.a, f.b);
```

```
        //printf("addr a : %p\n", &f.a);

        printf("size : %lu\n", sizeof(f));
}
// each of the fields will occupy part of an int
// cannot apply address operator on these fields
// overflow of a field : results in truncation
```