Unions

Unions are similar to *structs* except that they only have enough memory to hold one member at a time. Often, unions provide flexibility to the programmer for different implementations of the same object, while also aiding memory utilization.

Let's see where unions might prove useful. Let us consider a data structure ***Vec4*** which has four elements ***x, y, z*** and ***w***. ***Vec4*** can either be defined by initializing each of the four elements or using two ***Vec2***s (each ***Vec2*** has two elements). This is also shown in *figure 1*.

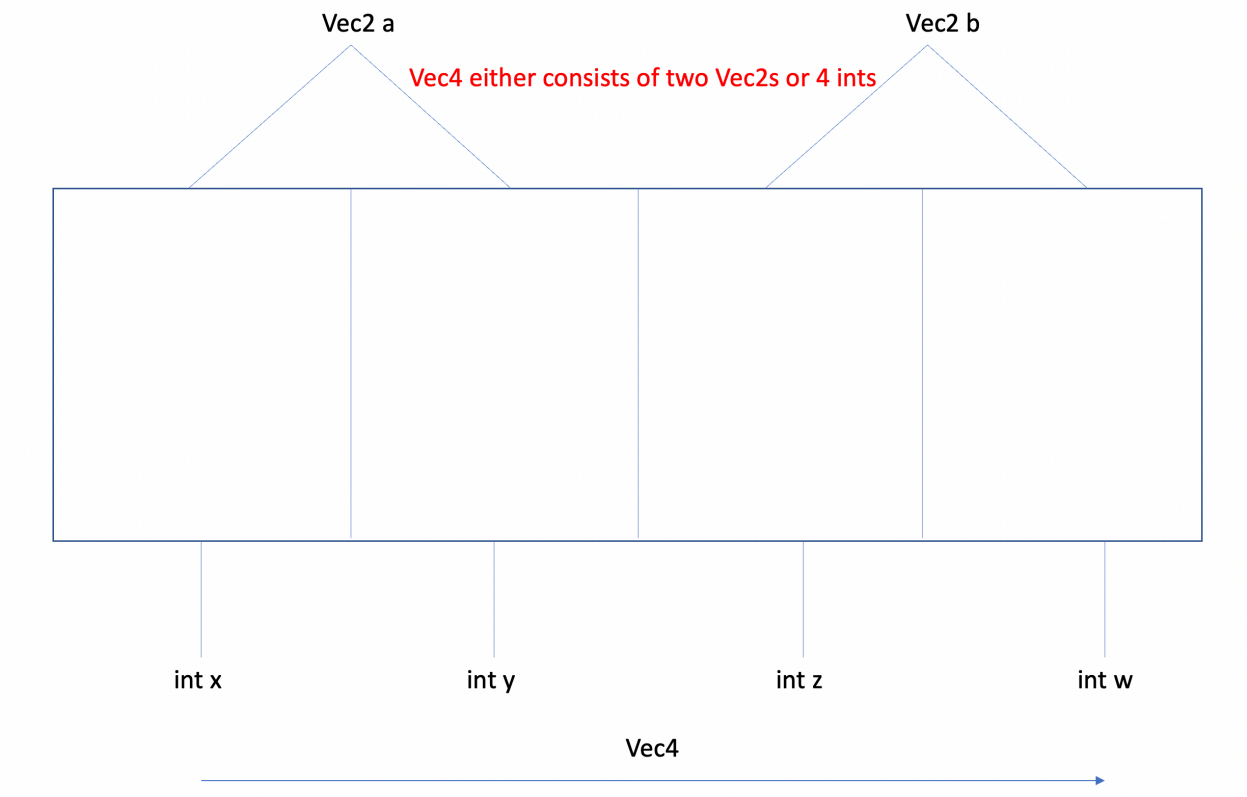


Figure 1

The implementation can be done as follows. This implementation of ***Vec4*** allows us to initialize it using two ***Vec2***s or individual ***Vec4*** elements, while utilizing the same memory.

```
typedef struct Vec2{
    int x,y;
} Vec2;
```

```
typedef struct Vec4{
    union{
        struct
        {
            Vec2 a,b;
        };
        struct{
            int x,y,z,w;
        };
    };
} Vec4;
```

To validate, let's initialize a ***Vec4*** and make changes to the elements and logging the intermediate output. It should show the operations shown in *Figure 2* and *Figure 3*.

```
    //Initialize two vec2s
    Vec2 p={1,2};
    Vec2 q={3,4};

    //Assign to vec4
    Vec4 r;
    r.a=p, r.b=q;

    //Check values for int x,y,z,w
    printf("x: %d, y: %d, z: %d, w: %d\n",r.x,r.y,r.z,r.w);
    printf("Address of x: %p, Address if a.x: %p\n",&r.x, &r.a.x
);

    //Change value for z
    r.z=300;
    printf("a: { %d, %d }, b: { %d, %d }\n", r.a.x, r.a.y,
r.b.x, r.b.y);
```

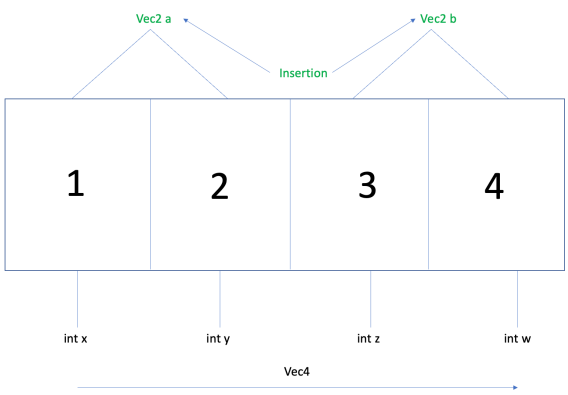


Figure 2

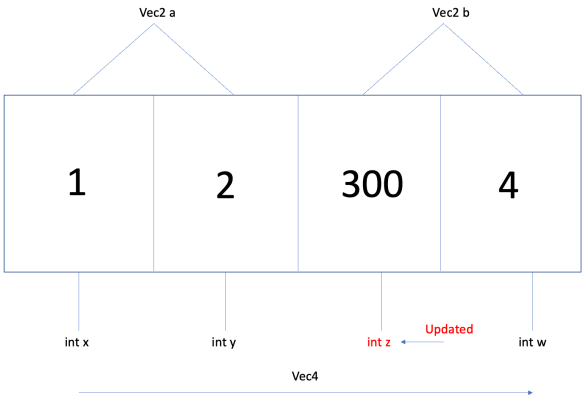


Figure 3

Tagged Unions

Before we talk about tagged unions, let us consider a scenario where we have unions of different data types.

```
typedef struct MyData{
    union{
        int a;
        float b;
    };
} MyData;
```

Although, both data types consume 4 bytes (on 64-bit GCC), the way in which integers and float data types are represented in memory is completely different.

```
MyData data;
data.a=4000;
data.b =3453.25f;
printf("%f, %d\n",data.b, data.a);
```

In this program, we are overwriting the memory consisting of an integer data type with a floating-point number, and accessing the integer data type. While accessing one data type after setting another member in a union is simply foolish, it is general convention to utilize same struct operations for an object, regardless of which union member is set. So, say, ***MyData*** struct needs to perform addition operation between two objects. It is necessary to know whether the integer or the float data type is set to perform correct operation. This is where tagged unions fit in.

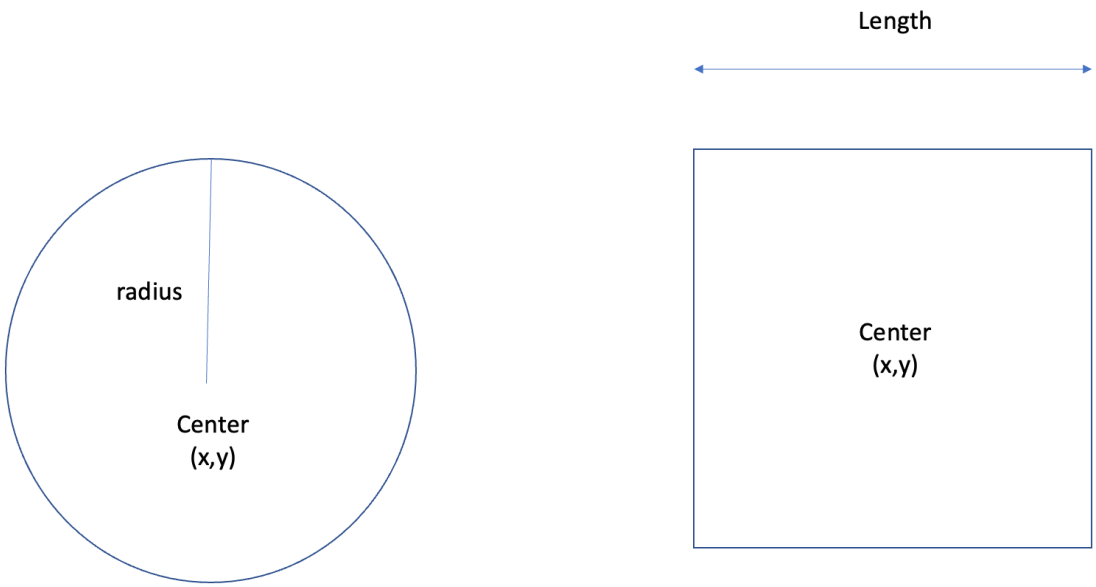
Tagged unions (also known as discriminated unions), consist of data types (usually occupying small space, with all members requiring nearabout same memory size) along with a special data type which identifies which member in the union is currently set.

```
typedef struct MyData{
    union{
        int a;
        float b;
        int member_set; //0 for int, 1 for float
    };
} MyData;
```

The ***member_set*** now allows the addition operation of any two ***MyData*** objects.

Example 1

Let us consider an object which can either be a circle or a square. Implementing them as a union, how can we know which shape is set to perform object operations?



Here, the ***enum Object_shape***, is used to determine which shape is set in the ***Object*** struct.

```
//Position data structure
typedef struct Position{
    int x;
    int y;
} Position;

//Circle and square shapes
typedef struct Circle{
    int radius;
    Position center;
} Circle;

typedef struct Square{
    int length;
    Position center;
} Square;

//Object shape enum
typedef enum Object_shape {
```

```
    Object_shape_circle,
    Object_shape_square
} Object_shape;

//Object structure
typedef struct Object
{
    Object_shape shape;
    union {

        Circle c;
        Square s;

    };

} Object;
```

Switch vs function dispatch

Before we write the operations for Example 1, let us revisit the switch case statement. For this, let's consider the ***MyData*** struct in Tagged Unions. To implement the addition operation of a ***MyData*** object and a floating-point number, the following switch case can be used.

```
    MyData sample_data;
    sample_data.b=3.3f;
    float result;
    int num=7;

    switch (sample_data.member_set)
    {
    case 0:
        result= (float)(sample_data.a + num);
        break;
    case 1:
        result= sample_data.b + (float)num;
        break;
    }
```

Assuming previous knowledge of function pointers, we can always shorten the code above using an array of function pointers, where the correct function would dispatch given the ***member_set***.

```
typedef float (*op)(MyData,int);

float add_int(MyData x, int num){ return (float)x.a +num); };
float add_float(MyData x, int num){ return x.b + (float)num); };

op add[]={add_int, add_float};
MyData sample_data;
sample_data.b=3.3f;
float result;
int num=7;
result = add[sample_data.member_set](sample_data,num);
```

Calculating area in Example 1

We have iterated the use of a single operation which returns the outcome according to the object characteristic. This behavior is known as polymorphic behavior or polymorphism. We have an ***Object*** struct in Example 1 which can be a Circle or a Square. Although the ergonomics remain different for the shapes, the area of the ***Object*** resonates the same meaning, the space covered by the shape. So, we perform the area operation using switch and function dispatch as follows in Example 1 to wrap it up.

```
#define PI 3.14
typedef int (*op)(Object);

//Area function implementations
int areaCircle(Object obj){ return PI * obj.c.radius *
obj.c.radius; };
int areaSquare(Object obj){ return obj.s.length * obj.s.length;
};

 Position p={1,2};
 Circle c;
 c.radius= 5;
 c.center = p;
 Object myObj;
 myObj.shape=Object_shape_circle;
 myObj.c=c;

 //object area calculation using switch case

    switch (myObj.shape){

        case Object_shape_circle:
```

```
            printf("Circle area: %d\n",areaCircle(myObj));
            break;
        case Object_shape_square:
            printf("Square area: %d\n", areaSquare(myObj));
            break;

    }

    //object area calculation using function dispatch
    op area[]={areaCircle,areaSquare};
    printf("Area: %d\n",area[myObj.shape](myObj));
```

Further exploration

- Polymorphism with void pointer vs discriminated unions: Heap vs stack allocation
- Switch case vs function dispatch performance comparison: Jump tables perspective

References

1. Chernikov, Y. (2018, May 13). Unions in C++. YouTube. Retrieved December 22, 2021, from https://www.youtube.com/watch?v=6uqU9Y578n4
2. Maclaine, A. (2019, September 10). C/C++ tagged/discriminated union. Medium. Retrieved December 22, 2021, from https://medium.com/@almtechhub/c-c-tagged-discriminated-union-ecd5907610bf