Struct in C

- C allow us to combine related data into a structure using the keyword `struct`.
- All data in a `struct` variable can be accessed by any code.

Struct definition

- **The general form of a structure definition is**

```
struct tag
{
    member1_declaration;
    member2_declaration;
    member3_declaration;
      . . .
    memberN_declaration;
};
```

Note the semi-colon

where `struct` is the keyword, `tag` names this kind of `struct`, and `member_declarations` are variable declarations which define the members.

C `struct` Example

- **Defining a `struct` to represent a point in a coordinate plane**

```
struct point
{
   int x;     /* x-coordinate */
   int y;     /* y-coordinate */
};
```

point is the struct tag

- **Given the declarations**

```
struct point p1;
struct point p2;
```

we can access the members of these `struct` variables:

* **the x-coordinate of p1 is `p1.x`**
* **the y-coordinate of p1 is `p1.y`**
* **the x-coordinate of p2 is `p2.x`**
* **the y-coordinate of p2 is `p2.y`**

Using structs and members

- Like other variable types, `struct` variables (e.g. p1, p2) can be passed to functions as parameters and returned from functions as return types.

- The members of a `struct` are variables just like any other and ca be used wherever any other variable of the same type may be used. For example, the members of the `struct point` can then be used just like any other integer variables.

printPoint.c

```
// struct point is a function parameter
void printPoint( struct point aPoint )
{
   printf ("( %2d, %2d )", aPoint.x, aPoint.y);
}

// struct point is the return type
struct point inputPoint( )
{
   struct point inPoint;
   printf("please input the x- and y-coordinates: ");
   scanf("%d %d", &inPoint.x, &inPoint.y);
   return inPoint;
}


int main ( )
{
   struct point endpoint;     // endpoint is a struct point variable
   endpoint = inputPoint( ); // struct assignment.... more later
   printPoint( endpoint );
   return 0;
}
```

printPoint.cpp: in c++, “struct” can be omitted

```
// struct point is a function parameter
void printPoint( point aPoint )
{
   printf (“( %2d, %2d )”, aPoint.x, aPoint.y);
}

// struct point is the return type
point inputPoint( )
{
   point inPoint;
   printf(“please input the x- and y-coordinates: “);
   scanf(“%d %d”, &inPoint.x, &inPoint.y);
   return inPoint;
}


int main ( )
{
   point endpoint;    // endpoint is a struct point variable
   endpoint = inputPoint( ); // struct assignment.... more later
   printPoint( endpoint );
   return 0;
}
```

Initializing A `struct`

- A `struct` variable may be initialized when it is declared by providing the initial values for each member in the order they appear

```
struct point middle = { 6, -3 };
```

defines the `struct point` variable named `middle` and initializes `middle.x = 6` and `middle.y = -3`

`struct` Variants

- **`struct` variables may be declared at the same time the struct is defined**

```
struct point {
  int x, y;
} endpoint, upperLeft;
```

defines the structure named `point` AND declares the variables `endpoint` and `upperLeft` to be of this type.

Not recommended!

`struct` typedef

- **It's common to use a `typedef` for the name of a `struct` to make code more concise.**

```
typedef struct point {
    int x, y;
} POINT;
```

defines the structure named point and defines `POINT` as a `typedef` (alias) for the `struct`. We can now declare variables, parameters, etc., as

`struct point endpoint;` **or as**

`POINT upperRight;`

Unnecessary in C++!

`struct` assignment

- The contents of a `struct` variable may be copied to another `struct` variable of the same type using the assignment (=) operator
- After this code is executed

```
struct point p1;
struct point p2;
p1.x = 42;
p1.y = 59;

p2 = p1;  /* structure assignment copies members */
```

The values of `p2`'s members are the same as `p1`'s members.
E.g. `p1.x = p2.x = 42` and `p1.y = p2.y = 59`

`struct` within a `struct`

- A data element in a `struct` may be another `struct`.
- This example defines a line in the coordinate plane by specifying its endpoints as POINT `structs`

```
typedef struct line
{
  POINT leftEndPoint;
  POINT rightEndPoint;
} LINE;
```

- Given the declarations below, how do we access the x- and y-coodinates of each line's endpoints?

```
struct line line1, line2;
```

`struct` within a `struct`

- **A data element in a `struct` may be another `struct`.**
- **This example defines a line in the coordinate plane by specifying its endpoints as POINT `structs`**

```
typedef struct line
{
  POINT leftEndPoint;
  POINT rightEndPoint;
} LINE;
```

- **Given the declarations below, how do we access the x- and y-coodinates of each line's endpoints?**

```
struct line line1, line2;
```

line1.leftEndPoint.x **line1.rightEndPoint.y**

line2.leftEndPoint.x **line2.rightEndPoint.y**

Arrays of `struct`

- **Since a `struct` is a variable type, we can create arrays of `structs` just like we create arrays of primitives.**
- **Write the declaration for an array of 5 line structures name "lines"**

```
struct line lines[ 5 ];
```

or

```
LINE lines[ 5 ];
```

- **Write the code to print the x-coordinate of the left end point of the 3rd line in the array**

```
printf( "%d\n", lines[2].leftEndPoint.x);
```

Array of `struct` Code

```
/* assume same point and line struct definitions */
int main( )
{
   LINE sides[5];
   int k;

   /* write code to initialize all data members to zero */
   for (k = 0; k < 5; k++)
   {
        sides[k].leftEndPoint.x = 0;
        sides[k].leftEndPoint.y = 0;
        sides[k].rightEndPoint.x = 0;
        sides[k].rightEndPoint.y = 0;
   }
   /* call the printPoint( ) function to print
   ** the left end point of the 3rd line */
   printPoint( sides[2].leftEndPoint);
   return 0;
}
```

Arrays within a `struct`

- **Structs may contain arrays as well as primitive types**

```
struct month
{
  int nrDays;
  char name[ 3 + 1 ];
};

struct month january = { 31, "JAN"};
```

Unions

- A `union` is a variable type that may hold different type of members of different sizes, BUT only one type at a time. `All members of the union share the same memory`. The compiler assigns enough memory for the largest of the member types.

- The syntax for defining a `union` and using its members is the same as the syntax for a `struct`.

Formal Union Definition

- **The general form of a `union` definition is**

```
union tag
{
    member1_declaration;
    member2_declaration;
    member3_declaration;
      . . .
    memberN_declaration;
};
```

where `union` is the keyword, `tag` names this kind of `union`, and `member_declarations` are variable declarations which define the members. Note that the syntax for defining a `union` is exactly the same as the syntax for a `struct`.

Without using unions

```
struct square { int length; };
struct circle { int radius; };
struct rectangle { int width; int height; };
enum shapeType {SQUARE, CIRCLE, RECTANGLE };

struct shape
{
   enum shapeType type;
  struct square aSquare;
  struct circle aCircle;
  struct rectangle aRectangle;
};
```

An application of Unions

```
struct square { int length; };
struct circle { int radius; };
struct rectangle { int width; int height; };
enum shapeType {SQUARE, CIRCLE, RECTANGLE };

union shapes
{
   struct square aSquare;
   struct circle aCircle;
   struct rectangle aRectangle;
};

struct shape
{
   enum shapeType type;
   union shapes theShape;
};
```

An application of Unions (2)

```
double area( struct shape s)
{
   switch( s.type ) {
       case SQUARE:
             return s.theShape.aSquare.length
                   * s.theShape.aSquare.length;
       case CIRCLE:
             return 3.14 * s.theShape.aCircle.radius
                   * s.theShape.aCircle.radius;
       case RECTANGLE :
             return s.theShape.aRectangle.height
                   * s.theShape.aRectangle.width;
   }
}
```

Another application of unions

```
struct  matrix4
{
union {
        struct {
            double      _11, _12, _13, _14;
            double      _21, _22, _23, _24;
            double      _31, _32, _33, _34;
            double      _41, _42, _43, _44;

        };
        double m[4][4];
    };
};
```

Union vs. Struct

- **Similarities**
  - Definition syntax virtually identical
  - Member access syntax identical
- **Differences**
  - Members of a `struct` each have their own address in memory.
  - The size of a `struct` is at least as big as the sum of the sizes of the members (more on this later)
  - Members of a union share the same memory.
  - The size of a union is the size of the largest member.

A bit more complex

```
struct month allMonths[ 12 ] = {
  {31, "JAN"}, {28, "FEB"}, {31, "MAR"},
  {30, "APR"}, {31, "MAY"}, {30, "JUN"},
  {31, "JUL"}, {31, "AUG"}, {30, "SEP"},
  {31, "OCT"}, {30, "NOV"}, {31, "DEC"}
};
// write the code to print the data for September
printf( "%s has %d days\n",
  allMonths[8].name, allMonths[8].nrDays);

// what is the value of allMonths[3].name[1]
P
```

Bitfields

- When saving space in memory or a communications message is of paramount importance, we sometimes need to pack lots of information into a small space. We can use `struct` syntax to define "variables" which are as small as 1 bit in size. These variables are known as "`bit fields`".

```
struct weather
{
    unsigned int temperature : 5;
    unsigned int windSpeed : 6;
    unsigned int isRaining : 1;
    unsigned int isSunny : 1;
    unsigned int isSnowing : 1;
};
```

Using Bitfields

- **Bit fields are referenced like any other struct member**

```
struct weather todaysWeather;

todaysWeather.temperature = 44;
todaysWeather.isSnowing = 0;
todaysWeather.windSpeed = 23;
/* etc */

if (todaysWeather.isRaining)
  printf( "%s\n", "Take your umbrella");
if (todaysWeather.temperature < 32 )
  printf( "%s\n", "Stay home");
```

More on Bit fields

- Almost everything about bit fields is implementation (machine and compiler) specific.
- Bit fields may only defined as `(unsigned) ints`
- Bit fields do not have addresses, so the `&` operator cannot be applied to them

- We'll see more on this later