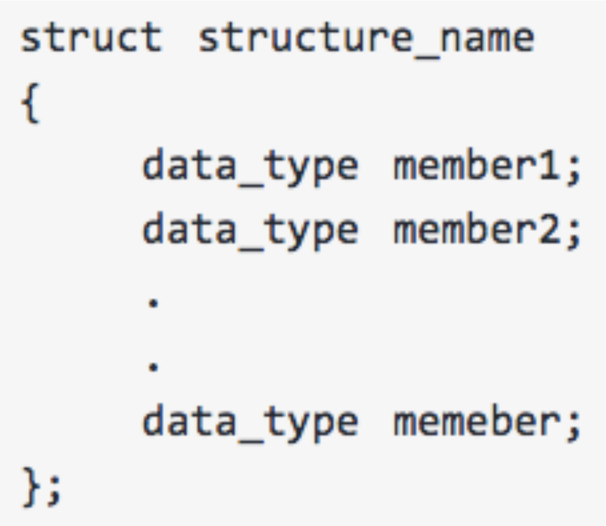
Structures, Unions & Dynamic Memory Allocation

STRUCTURES

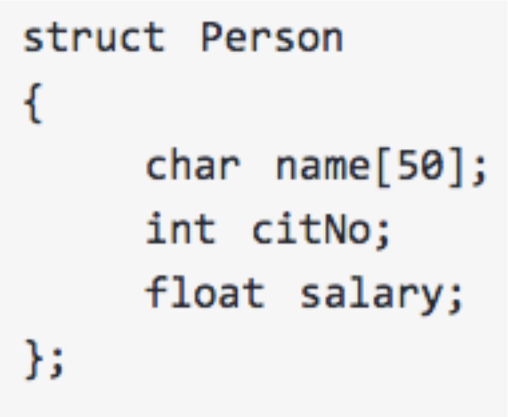
- A structure is a collection of variables under a single name.
    - The variables can be of the same or different data types.
- For example, the homework on student complaints done earlier required you to collect information from the student such as:
    - Student name
    - Registration number
    - Year of study
    - Program of study, etc.
- At the time of doing this homework, you needed to declare a different variable for every piece of information required from every student.
- This same information could have been collected in one go using a single variable *student* of type *struct*.

Defining Structures

- The keyword *struct* is used to create a structure as illustrated below.



```
struct  structure_name
{
      data_type  member1;
      data_type  member2;
      .
      .
      data_type  memeber;
};
```

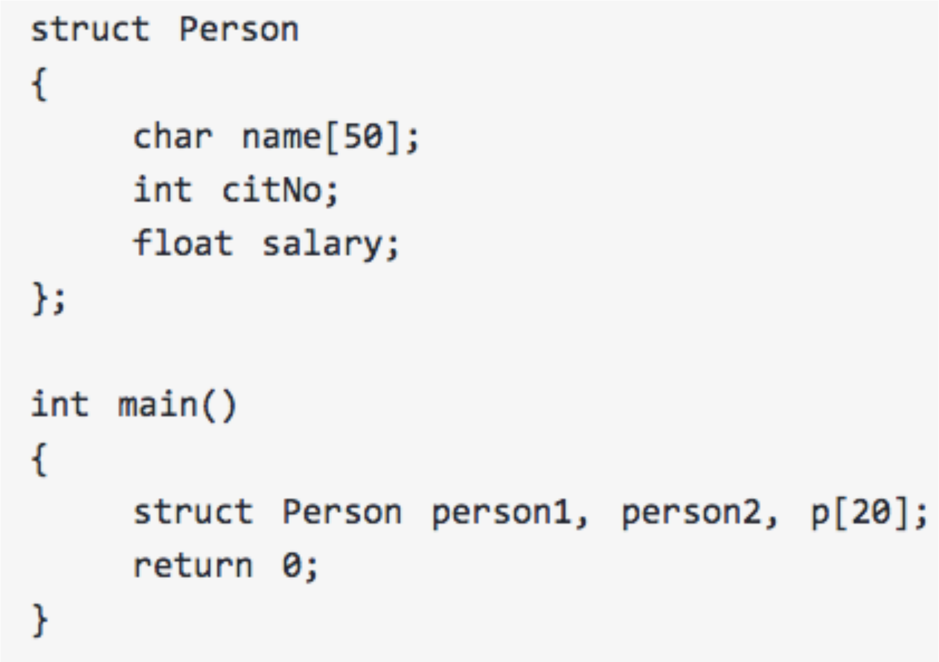


```
struct  Person
{
      char  name[50];
      int  citNo;
      float  salary;
};
```

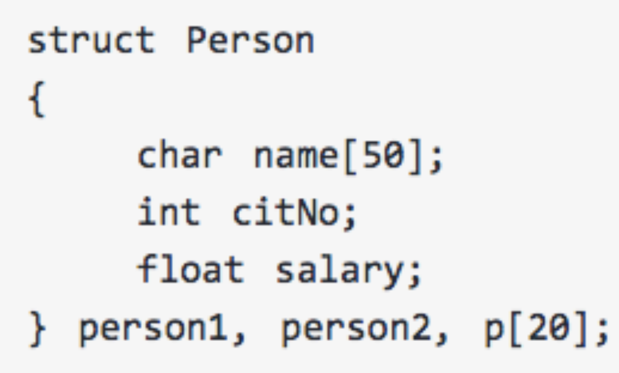
- Based on the above templates, a derived type, struct Person is defined characterized by three points of data: *name[]*, *citNo* and *salary*.
- When a structure is defined (as was done for Person), it creates a user-defined type, founded on primitive data types through the data collected.
- However, no memory can be allocated until variables of the defined type are declared.

Structure Variables

- Structure variables can be created in any of the following two ways:



```
struct Person
{
      char name[50];
      int citNo;
      float salary;
};

int main()
{
      struct Person person1, person2, p[20];
      return 0;
}
```



```
struct Person
{
      char name[50];
      int citNo;
      float salary;
} person1, person2, p[20];
```

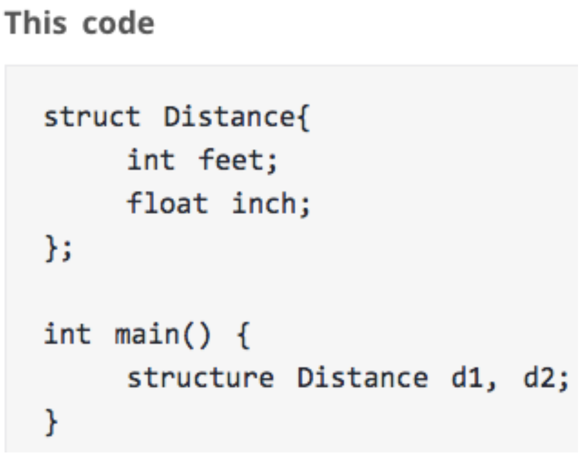
- In both the above definitions, three variables of type *struct Person* are declared as *person1*, *person2* and array variable *p[20]* all with with members *name[50]*, *citNo* and *salary* each with the appropriate primitive data type.
- Nested structures can also be formed by defining a structure within a structure.

Accessing Structure Members

- Two types of operators are used to access members of a structure:
  - ✧ Member operator (.)
  - ✧ Structure pointer operator (->)
- If the name of person1 from the previous structure definition is required, then the member operator can be applied as:

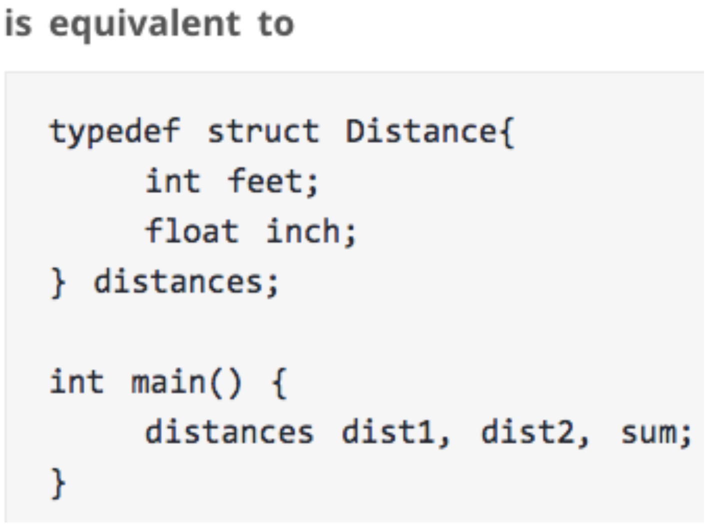
  person1.name

- The keyword typedef can be used to simplify the syntax when declaring structures as shown below:



This code

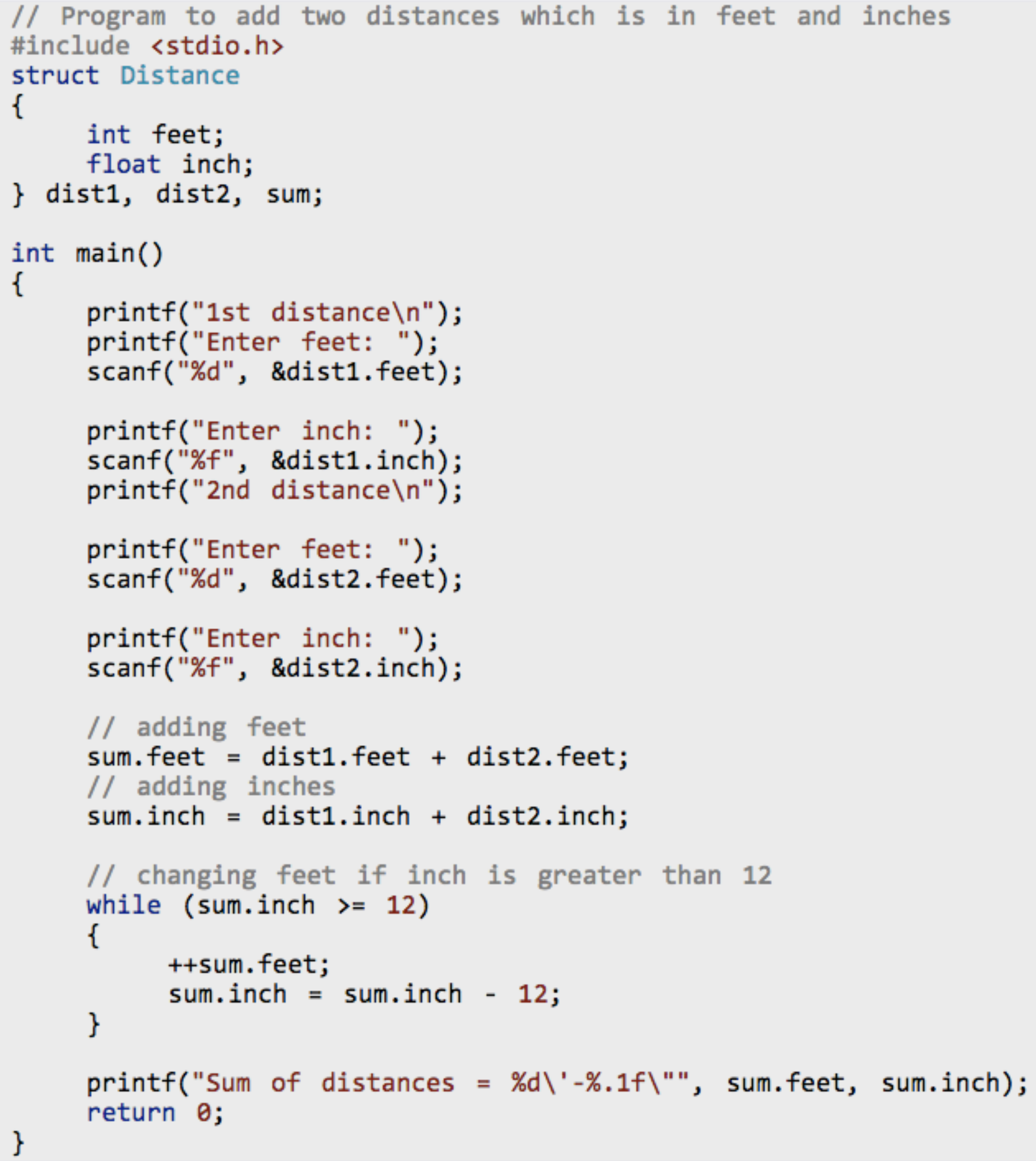
```
struct Distance{
      int feet;
      float inch;
};

int main() {
      structure Distance d1, d2;
}
```



is equivalent to

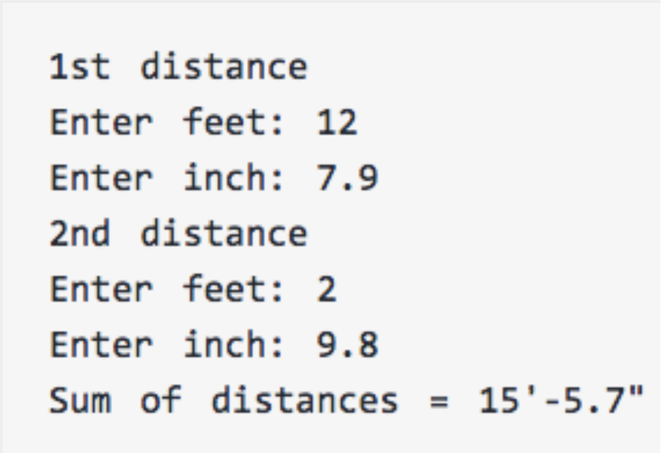
```
typedef struct Distance{
      int feet;
      float inch;
} distances;

int main() {
      distances dist1, dist2, sum;
}
```

Structures Member Operator - Example



```c
// Program to add two distances which is in feet and inches
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;

int main()
{
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);

    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("2nd distance\n");

    printf("Enter feet: ");
    scanf("%d", &dist2.feet);

    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    // adding feet
    sum.feet = dist1.feet + dist2.feet;
    // adding inches
    sum.inch = dist1.inch + dist2.inch;

    // changing feet if inch is greater than 12
    while (sum.inch >= 12)
    {
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }

    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

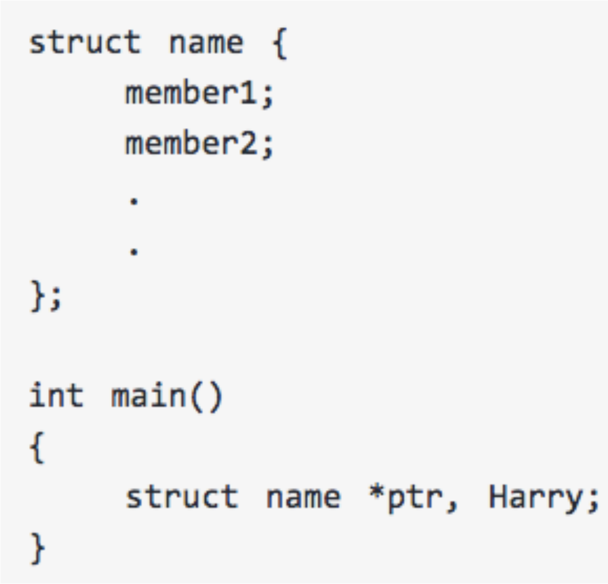
Output



```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

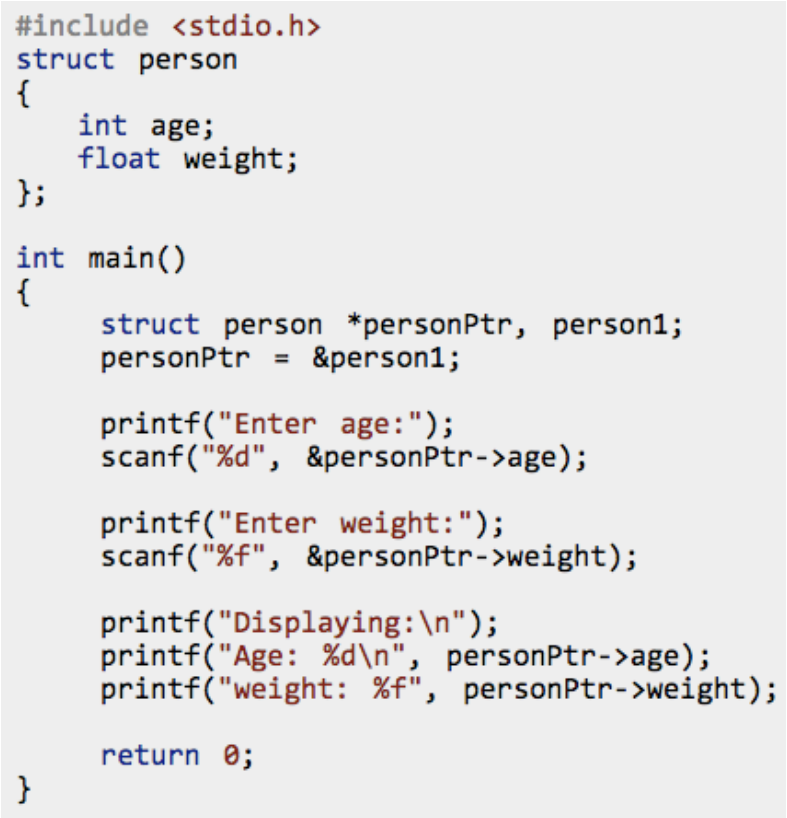
Structures Pointer Operator

- Structures can be accessed using pointers as shown below:



```
struct name {
    member1;
    member2;
    .
    .
};

int main()
{
    struct name *ptr, Harry;
}
```

In the above template, a pointer *ptr* of type *struct name* is created and the pointer can access members of Harry.

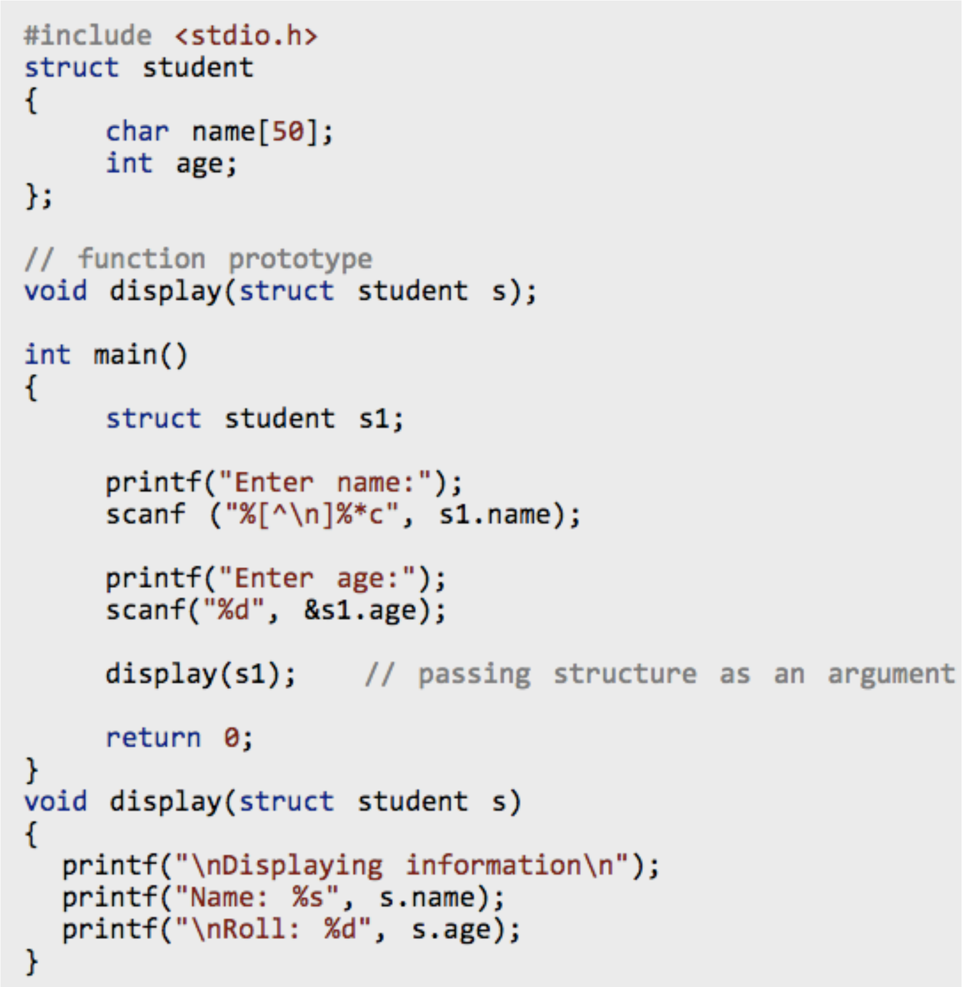


```
#include <stdio.h>
struct person
{
   int age;
   float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age:");
    scanf("%d", &personPtr->age);

    printf("Enter weight:");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```

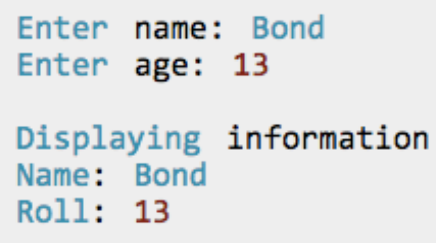
- In the above pointer example, the address of *person1* is stored in the *personPtr* variable by *personPtr = &person1*; by the way,
    - `personPtr->age` is equivalent to `(*personPtr).age`
    - `personPtr->weight` is equivalent to `(*personPtr).weight`

Passing Structures

- Passing structure(s) to a function is not any different from passing any other argument to a function as indicated in the code and output below:



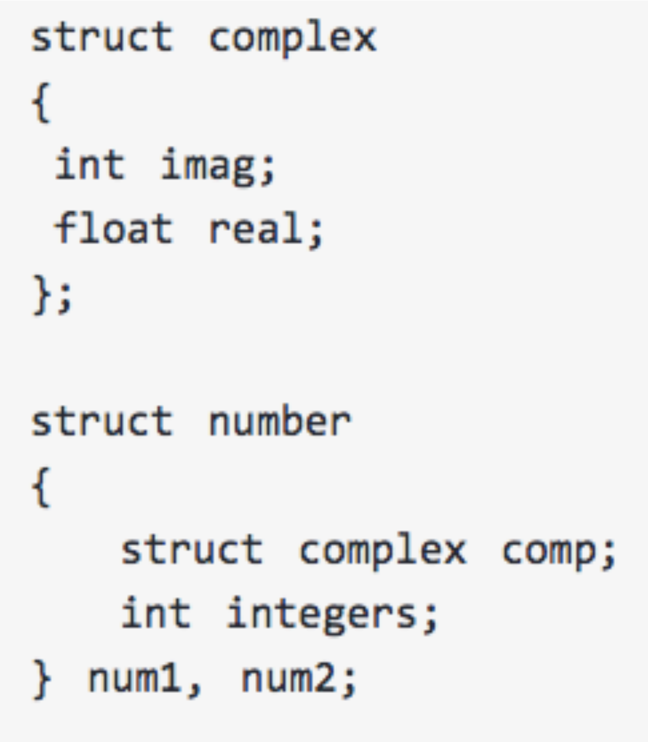
```c
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
void display(struct student s);

int main()
{
    struct student s1;

    printf("Enter name:");
    scanf ("%[^\n]%*c", s1.name);

    printf("Enter age:");
    scanf("%d", &s1.age);

    display(s1);    // passing structure as an argument

    return 0;
}
void display(struct student s)
{
  printf("\nDisplaying information\n");
  printf("Name: %s", s.name);
  printf("\nRoll: %d", s.age);
}
```



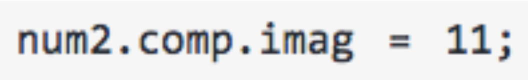
```
Enter name: Bond
Enter age: 13

Displaying information
Name: Bond
Roll: 13
```

Nested Structures

- You can create a structure within a structure if required, hence the nested structure.
- In the example that follows, if you wanted to assign the value 11 to the *imag* of variable *num2*, you could write:



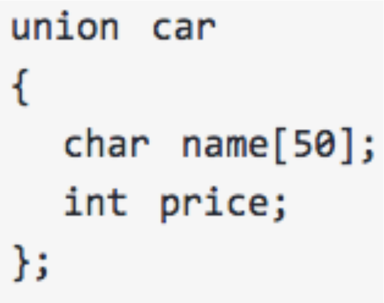
```
struct complex
{
 int imag;
 float real;
};

struct number
{
    struct complex comp;
    int integers;
} num1, num2;
```



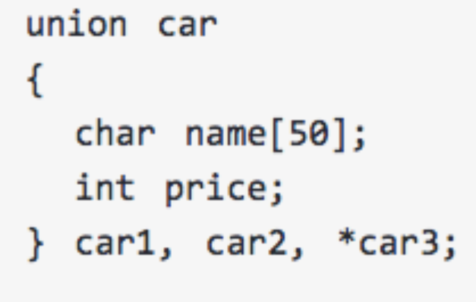
```
num2.comp.imag = 11;
```

UNIONS

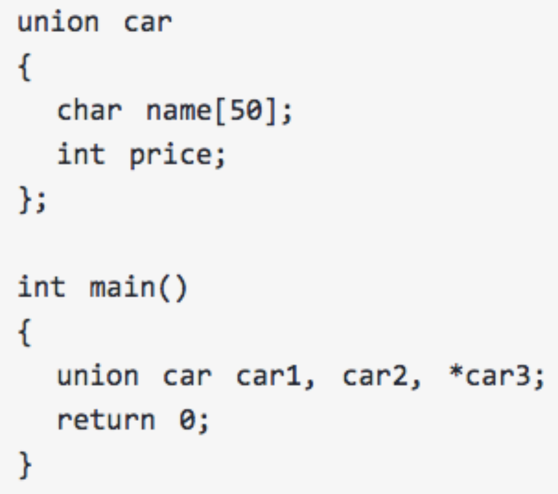
- A union is also a user-defined type like structures.
  - ✧ Union is one of the C keywords and operates in a very similar way to struct.
  - ✧ In the prototype below, the derived type *union car* is defined.
- As was the case with structures, creation of a union without variable declaration is quite pointless since no memory allocations and hence data storage can made. Thus, the exact same methods of structure variables are used with unions too.



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



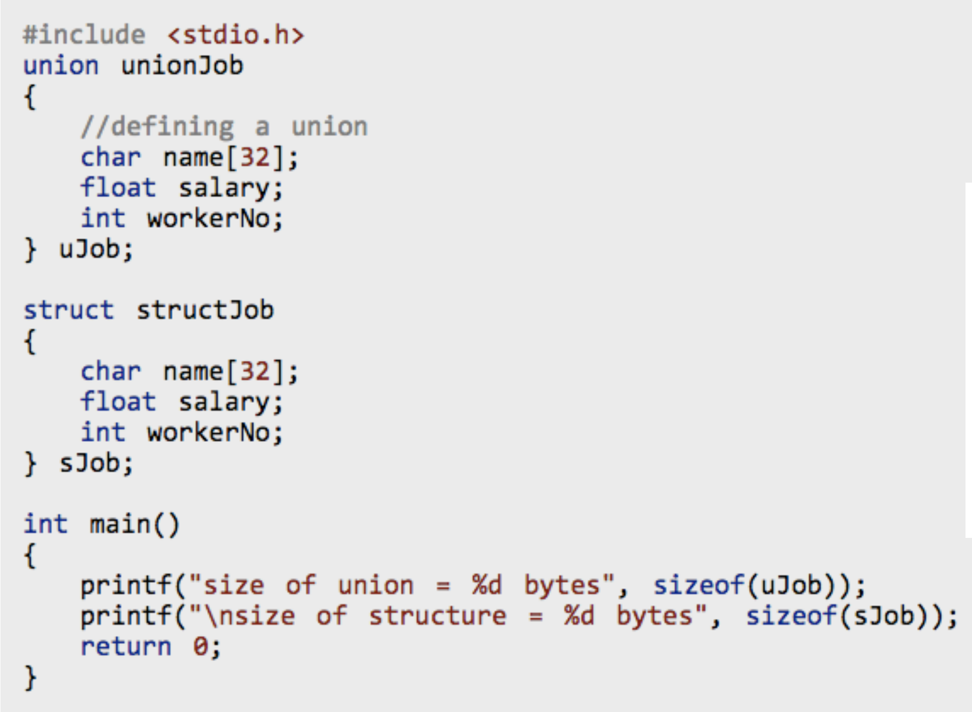
```
union  car
{
   char  name[50];
   int  price;
}  car1,  car2,  *car3;
```



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

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

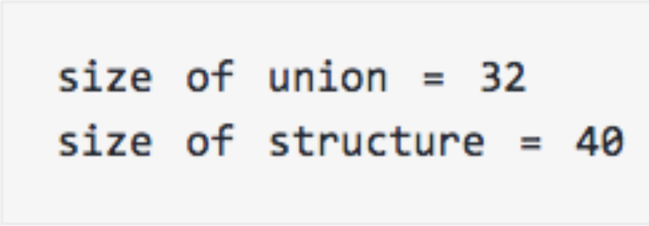
Unions & Structures Differences

- Two main differences arise between unions and structures:
  - ✧ Memory allocations – unions are sized based on the largest memory requirement within its member definitions while structures will sum up the total memory requirements of all the members in the structure.
  - ✧ As a result, all members of a structure can be accessed in one go if required while the union only allows access to one member of the union at any one time.



```c
#include <stdio.h>
union unionJob
{
    //defining a union
    char name[32];
    float salary;
    int workerNo;
} uJob;

struct structJob
{
    char name[32];
    float salary;
    int workerNo;
} sJob;

int main()
{
    printf("size of union = %d bytes", sizeof(uJob));
    printf("\nsize of structure = %d bytes", sizeof(sJob));
    return 0;
}
```

Output



```
size of union = 32
size of structure = 40
```