

# CSE 1320

Week of 03/27/2023

Instructor : Donna French

# Enumerated Types

keyword

Enumerated types are scalar types in C and are used to declare a set of integer constants in C.

name of the enumeration

`enum boolean`

`{`

`false, true`

identifiers that are integral constants

`};`

`enum boolean correct;`

a variable named `correct` of type `enum boolean`

# Enumerated Types

```
enum colors
{
    red, orange, yellow, blue, green, indigo, violet
};
```

```
enum colors rainbow;
```

```
enum colors
{
    red, orange, yellow, blue, green, indigo, violet
}
rainbow;
```

# Enumerated Types

```
enum colors
{
    red, orange, yellow, blue, green, indigo, violet
}
rainbow;
```

The default values are assigned starting with 0 and each succeeding identifier is assigned successive integer values.

red	0	green	4
orange	1	indigo	5
yellow	2	violet	6
blue	3		

# Enumerated Types

The default values for the identifiers in an `enum` type can be overridden.

```
enum colors
{
    red=3, orange=6, yellow=6, blue=4, green=5, indigo, violet
}
rainbow;
```

- More than one identifier can be assigned the same value – `orange` and `yellow` are both 6
- `indigo` will be set to 6 since it appears in the list after `green` which was assigned 5.
- `violet` will be set to 7 since it appears in the list after `indigo` which was assigned 6

```
switch (ColorNumber)
{
    case red      : printf("red\n");
    case orange   : printf("orange\n");
    case yellow    : printf("yellow\n");
    case blue      : printf("blue\n");
    case green     : printf("green\n");
    case indigo    : printf("indigo\n");
    case violet    : printf("violet\n");
    default : printf("You have fallen off the rainbow\n");
}
```

```
enum colors
{
    red=3,
    orange=6,
    yellow=6,
    blue=4,
    green=5,
    indigo,
    violet
}
rainbow;
```

## gcc enum1Demo.c

enum1Demo.c: In function 'main':

enum1Demo.c:21: error: duplicate case value

enum1Demo.c:20: error: previously used here

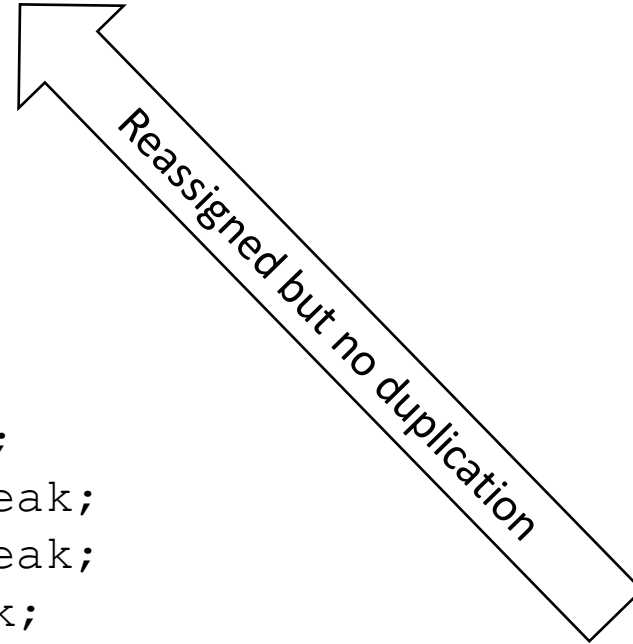
enum1Demo.c:24: error: duplicate case value

enum1Demo.c:20: error: previously used here

```
enum colors
{
    red=3,orange=2,yellow=6,blue=4,green=15,indigo,violet
}
rainbow;

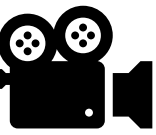
printf("Enter a number between 0 and 7 ");
scanf("%d", &ColorNumber);

switch (ColorNumber)
{
    case red      : printf("red\n"); break;
    case orange   : printf("orange\n"); break;
    case yellow   : printf("yellow\n"); break;
    case blue     : printf("blue\n"); break;
    case green    : printf("green\n"); break;
    case indigo   : printf("indigo\n"); break;
    case violet   : printf("violet\n"); break;
    default       : printf("You have fallen off the rainbow\n");
}
```



frenchdm@omega:~

[frenchdm@omega ~]\$ g





```
enum colors
```

```
{
```

```
    red=3,orange=2,yellow=6,blue=4,green=15,indigo,violet
```

```
} rainbow;
```



# Enumerated Types

An `enum` variable is legally supposed to accept only the values defined in it but compilers are not required to check that assigned value is in the declared list.

Operations with `enum` types are limited. The identifiers are treated as constants of type `int` and can appear anywhere that an `int` constant can.

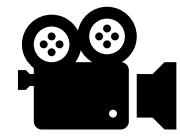
```
enum colors
{
    red, orange, yellow, blue, green, indigo, violet
};
```

```
enum colors color1;
enum colors color2;
enum colors color3;
```

```
color1 = orange;
color2 = green;
color3 = red+orange+yellow+blue+green+indigo+violet;
```

```
printf("color1 = %d\ncolor2 = %d\ncolor3 = %d\n", color1, color2, color3);
```

frenchdm@omega:~



[frenchdm@omega ~]\$ more enum

# Enumerated Types

Other than assignment and equality tests, no other operations must be supported by an ANSI C compiler. Other operations may be supported but their use is not encouraged due to portability issues.

Why use them?

- Improve readability

- Improve maintainability

- Automatic assignment and accounting of values

# Enumerated Types

`#define` **vs** `enum`

1. enums follow scope rules
2. enum variables are automatically assigned values
3. enums have a type - `#defines` do not
4. debug can display enums

# Enumerated Types

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

```
int main(void)
```

```
{
    enum
    {
        STATE_ONE,
        STATE_TWO,
        STATE_THREE
    }
    state;

    state = 4;

    switch (state)
    {
        case STATE_ONE:
            printf("State 1");
            break;
        case STATE_TWO:
            printf("State 2");
            break;
    }
}
```

```
gcc Test.c -g
```

```
gcc Test.c -g -Wall
```

```
Test.c: In function 'main':
```

```
Test.c:15:5: warning: enumeration value 'STATE_THREE' not
handled in switch [-Wswitch]
```

```
15 |         switch (state)
    |         ^~~~~~
```

-Wall enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning)

```
enum DayofWeek
{
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};

int Today;

printf("Enter the day of the week ");
scanf("%d", &Today);

switch (Today)
{
    case Sunday      : printf("Today is Sunday\n"); break;
    case Monday      : printf("Today is Monday\n"); break;
    case Tuesday      : printf("Today is Tuesday\n"); break;
    case Wednesday    : printf("Today is Wednesday\n"); break;
    case Thursday     : printf("Today is Thursday\n"); break;
    case Friday       : printf("Today is Friday\n"); break;
    case Saturday     : printf("Today is Saturday\n"); break;
    default           : printf("That isn't a day of the week\n");
}
```

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

```
enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,Aug, Sep, Oct, Nov, Dec};
```

```
int main(void)
```

```
{
```

```
    int i;
```

```
    for (i = Jan; i <= Dec; i++)
```

```
    {
```


```
        printf("%d ", i);
```


```
    }
```

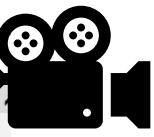
```
    return 0;
```

```
}
```



 frenchdm@omega:~

[frenchdm@omega ~]\$ mo 



# Introduction to Structures

## Aggregate Types

Aggregate types are designed to hold multiple data values

Arrays can hold many data values of the same type

```
int GradeArray[10] = {100,99,98,34,89,99,70,99,88,100};
```

# Introduction to Structures

## Structure

A structure can concurrently hold multiple data values of different types.

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```

# Introduction to Structures

`struct` is a keyword in C - it signals the declaration of a structure

keyword

`struct tshirt`

{

`char size[5];`

`char color[10];`

`char design[100];`

`char fittype;`

`float price;`

`int inventory_level;`

`};`

user defined

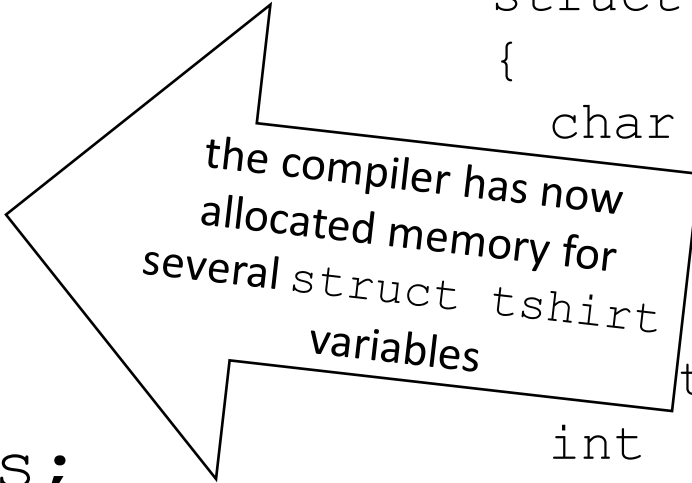
the compiler knows what  
a `struct tshirt` looks  
like but has not created  
one

`struct tshirt` has 6 members

# Introduction to Structures

`struct tshirt` is now a user-define type that can be used to declare variables of that type

```
struct tshirt MyTShirts;  
struct tshirt YourTShirts;  
struct tshirt TheirTShirts;  
struct tshirt OurTShirts;  
struct tshirt NobodysTShirts;
```



```
struct tshirt  
{  
    char    size[5];  
           color[10];  
           design[100];  
           fittype;  
           price;  
           inventory_level;  
};
```

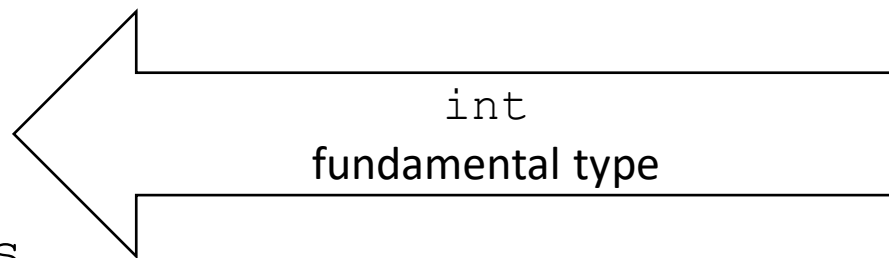
Breakpoint 1, main () at struct1Demo.c:6

```
6          int GradeArray[10] =  
{100,99,98,34,89,99,70,99,88,100};
```

(gdb) step

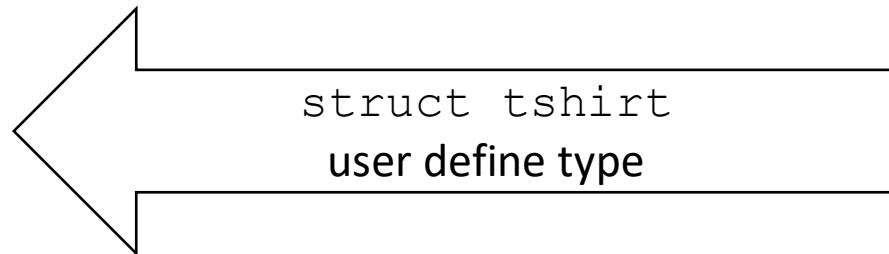
```
18          struct tshirt YourTShirts = {};
```

(gdb) ptype GradeArray  
type = int [10]



(gdb) ptype YourTShirts  
type = struct tshirt {

```
    char size[5];  
    char color[10];  
    char design[100];  
    char fittype;  
    float price;  
    int inventory_level;  
}
```



# Introduction to Structures

A variable in a structure type can be initialized at the same time that the struct is declared.

```
struct tshirt NobodysTShirts;
```

```
struct tshirt YourTShirts = {};
```

```
struct tshirt TheirTShirts = {"S"};
```

```
struct tshirt OurTShirts = {"", "GREEN"};
```

```
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'Y', 14.99, 1987};
```

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```

```

struct tshirt NobodysTShirts;
struct tshirt YourTShirts = {};
struct tshirt TheirTShirts = {"S"};
struct tshirt OurTShirts = {" ", "GREEN"};
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'Y', 14.99, 1987};

```

**18            struct tshirt NobodysTShirts;**

```
(gdb) p NobodysTShirts
```

```

$2 = {
  size = "v\000\000\000",
  color = "\000\000\000\000\000\000\000\000\000\000",
  design = '\000' <repeats 25 times> "\377,
\265\360\000\000\000\000\000\000\302\000\000\000\000\000\000\000\377\265\360\00
0\000\000\000\000\000\206\347\377\377\377\177\000\000\207\347\377\377\377\177\0
00\000\000\000\000\000\000\000\000\000\000\300\313!\311>\000\000\000`\006@\000\
000\000\000\000\203\003@\000\000\000\000\000\000\001\000",
  fittype = 0 '\000',
  price = 1.79079218e-38,
  inventory_level = 4195991
}

```



```
19      struct tshirt YourTShirts = {};
```

```
(gdb) p YourTShirts
```

```
$2 = {
  size = "\000\000\000\000",
  color = "\000\000\000\000\000\000\000\000\000\000",
  design = '\000' <repeats 99 times>,
  fittype = 0 '\000',
  price = 0,
  inventory_level = 0
}
```

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```

```
20      struct tshirt TheirTShirts = {"S"};
```

```
(gdb) p TheirTShirts
```

```
$4 = {
  size = "S\000\000\000",
  color = "\000\000\000\000\000\000\000\000\000\000",
  design = '\000' <repeats 99 times>,
  fittype = 0 '\000',
  price = 0,
  inventory_level = 0
}
```

```
21      struct tshirt OurTShirts = {"", "GREEN"};
```

```
(gdb) p OurTShirts
```

```
$4 = {
```

```
  size = " \000\000\000",
```

```
  color = "GREEN",
```

```
  design = "frenchdm@omega ~]$ gcc struct1Demo.c
```

```
  fittype = 89,
  price = 14.98999998,
  inventory_level = 1987
```

```
  price = 14.98999998,
```

```
  inventory_level = 1987
```

```
struct tshirt
```

```
{
```

```
  size[5];
```

```
  color[10];
```

```
  design[100];
```

```
  fittype;
```

```
  price;
```

```
  inventory_level;
```

```
};
```

```
18      struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'Y', 14.99, 1987};
```

```
(gdb) p MyTShirts
```

```
$1 = {
```

```
  size = "XS\000\000",
```

```
  color = "BLUE\000\000\000\000\000\000",
```

```
  design = "DISNEY", '\000' <repeats 93 times>,
  fittype = 89 'Y',
  price = 14.98999998,
  inventory_level = 1987
```

```
  fittype = 89 'Y',
```

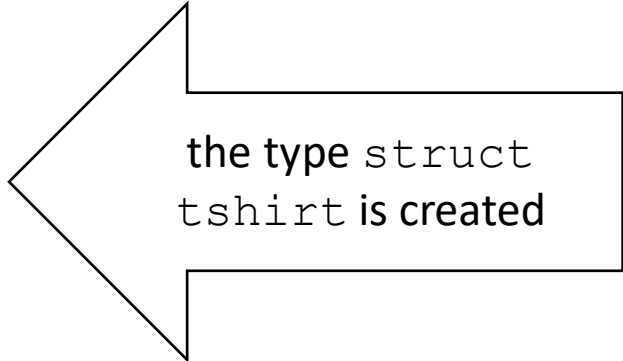
```
  price = 14.98999998,
```

```
  inventory_level = 1987
```

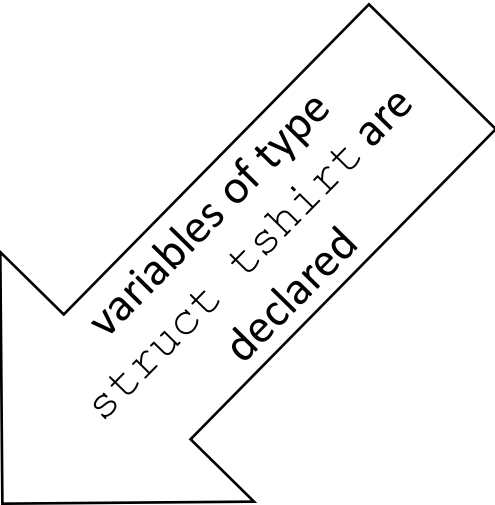
```
}
```

# Introduction to Structures

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```




the type struct  
tshirt is created



variables of type  
struct tshirt are  
declared

```
struct tshirt MyTShirts, YourTShirts;
```

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
}
TheirTShirts, OurTShirts;
```



struct tshirt is declared and two  
variables are created of that type

# Introduction to Structures

```
struct tshirt  
{
```

```
    char size[5];  
    float price;
```

**reusable**

```
    float weight;  
    int inventory_id;  
}
```

```
MyTShirts, YourTShirts;
```

```
struct tshirt NobodysTShirts;  
struct tshirt OurTShirts;
```

```
struct  
{
```

```
    char size[5];  
    float price;
```

**one time use**

```
    float weight;  
    int inventory_id;  
}
```

```
TheirTShirts, OurTShirts;
```

Cannot create more variables based on this structure because the struct was not named; therefore, cannot be reused.

# Introduction to Structures

Restrictions on the types of the members of a structure

- a member of a structure cannot be a function
- a structure may not nest a structure of its own type

a member of a structure cannot be a function

```
struct tshirt
{
    int FunctionX(void);
    char  size[5];
    char  color[10];
    char  design[100];
    char  fittype;
    float price;
    int   inventory_level;
};
```

```
[frenchdm@omega ~]$ gcc struct1Demo.c -g
```

```
struct1Demo.c: In function 'main':
```

```
struct1Demo.c:15: error: field 'FunctionX' declared as a function
```

# a structure may not nest a structure of its own type

```
struct tshirt
{
    char  size[5];
    char  color[10];
    char  design[100];
    char  fittype;
    float price;
    int   inventory_level;
    struct tshirt NobodysTShirts;
}
MyTShirts, YourTShirts;
```

Compiler error because struct tshirt does not exist as a type since it is being declared here

struct2Demo.c: In function 'main':

struct2Demo.c:14: error: field 'NobodysTShirts' has incomplete type

# Introduction to Structures

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
}
```

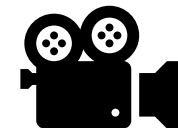
MyTShirts, YourTShirts;

MyTShirts  
has already  
been created  
so OK to use  
here

```
struct
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
    struct tshirt MyTShirts;
}
TheirTShirts, OurTShirts;
```



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5
6     struct tshirt
7     {
8         char size[5];
9         char color[10] = "blue";
10        char design[100];
11        char fittype = 'Y';
12        float price = 11.99;
13        int inventory_level = 100;
14    } MyTShirts, YourTShirts;
15
16    struct tshirt hello;
17
18    struct
19    {
20        char size[5];
21        char color[10];
22        char design[100];
23        char fittype;
24        float price;
25        int inventory_level;
26        struct tshirt MyTShirts;
27    } TheirTShirts, OurTShirts;
28
29    return 0;
30 }
31
```



```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```

```
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'A', 11.10, 20};
```

1-A

```
struct tshirt
{
    char    size[5] = "XS";
    char    color[10] = "BLUE";
    char    design[100] = "DISNEY";
    char    fittype = 'A';
    float   price = 11.10;
    int     inventory_level = 20;
};
```

1-B

```
struct tshirt
{
    char    size[5];
    char    color[10];
    char    design[100];
    char    fittype;
    float   price;
    int     inventory_level;
};
```

```
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", "A", 11.10, 20};
```

```
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'A', 11.10, 20};
```

```
struct tshirt MyTShirts = {'XS', 'BLUE', 'DISNEY', 'A', 11.10, 20};
```

```
struct tshirt MyTShirts = {"XS", "BLUE", "DISNEY", 'A', "11.10", "20"};
```





# Introduction to Structures

## Build a structure for a box



```
struct box
{
    int height;
    int length;
    int depth;
    float weight;
    char size[3]; // XS,S,M,L,XL
    char strength[10]; // how heavy duty
    int code; // USPS assigns codes
    int inventory_level;
};
```

# Introduction to Structures

The individual fields of a structure can be accessed with this syntax

`variable_name.member_name`

```
struct tshirt
{
    char    size[5];           MyTShirts.size
    char    color[10];         MyTShirts.color
    char    design[100];       MyTShirts.design
    char    fittype;           MyTShirts.fittype
    float    price;            MyTShirts.price
    int      inventory_level;   MyTShirts.inventory_level
};

struct tshirt MyTShirts;
```



```
printf("What size is your tshirt? ");  
scanf("%s", MyTShirts.size);
```

```
printf("What color is your tshirt? ");  
scanf("%s", MyTShirts.color);
```

```
printf("What design is your tshirt? ");  
scanf("%s", MyTShirts.design);
```

```
printf("What fit type is your tshirt? ");  
scanf(" %c", &MyTShirts.fittype);
```

```
printf("What is the price of your tshirt? ");  
scanf("%f", &MyTShirts.price);
```

```
printf("How many do you have in stock? ");  
scanf("%d", &MyTShirts.inventory_level);
```

```
printf("Tshirt size      : %s\n", MyTShirts.size);  
printf("Tshirt color     : %s\n", MyTShirts.color);  
printf("Tshirt design     : %s\n", MyTShirts.design);  
printf("Tshirt fit type    : %c\n", MyTShirts.fittype);  
printf("Tshirt price       : %.2f\n", MyTShirts.price);  
printf("Tshirt inventory : %d\n", MyTShirts.inventory_level);
```

Tshirt size	:	M
Tshirt color	:	RED
Tshirt design	:	MARVEL
Tshirt fit type	:	Y
Tshirt price	:	12.99
Tshirt inventory	:	100

# Operations on Structures

Very few operations may operate on a structure as a whole.

The following operations are allowed.

1. The selection operators access a single member from the structure
2. The assignment operator assigns the contents of one structure variable to another.
3. The address operator, `&`, can be used with a structure variable in most interfaces
4. The `sizeof()` operator is usually defined for structures



# Operations on Structures

The selection operators access a single member from the structure

```
scanf("%d", &MyTShirts.inventory);  
printf("Tshirt inventory : %d\n", YourTShirts.inventory);
```

The assignment operator assigns the contents of one structure variable to another.

```
YourTShirts = MyTShirts;
```

The address operator, &, can be used with a structure variable in most interfaces

```
printf("The address of MyTShirts is %p\n", &MyTShirts);  
printf("The address of YourTShirts is %p\n", &YourTShirts);
```

The sizeof() operator is usually defined for structures

```
printf("\n\nThe sizeof() MyTShirts is %d", sizeof(MyTShirts));  
printf("The sizeof() MyTShirts.size is %d\n", sizeof(MyTShirts.size));
```

# Using Structures with Arrays and Pointers

## Arrays of Structures

```
struct tshirt MarvelTShirts[10];  
struct tshirt DisneyTShirts[15];  
struct tshirt DCComicsTShirts[5];
```

Each cell of the array is a structure

```
struct tshirt  
{  
    char    size[5];  
    char    color[10];  
    char    design[100];  
    char    fittype;  
    float   price;  
    int     inventory;  
};
```

# Using Structures with Arrays and Pointers

## Arrays of Structures

```
struct tshirt MarvelTShirts[10];
```

	size	color	design	fittype	price	inventory
MarvelTShirts[0]						
MarvelTShirts[1]						
MarvelTShirts[2]						
MarvelTShirts[3]						
MarvelTShirts[4]						
MarvelTShirts[5]						
MarvelTShirts[6]						
MarvelTShirts[7]						
MarvelTShirts[8]						
MarvelTShirts[9]						

# Arrays of Structures

	size	color	design	fittype	price	inventory
MarvelTShirts[0]						
MarvelTShirts[1]						
MarvelTShirts[2]						
MarvelTShirts[3]						
MarvelTShirts[4]						
MarvelTShirts[5]						
MarvelTShirts[6]						
MarvelTShirts[7]						
MarvelTShirts[8]						
MarvelTShirts[9]						

**MarvelTShirts[0].color**

**MarvelTShirts[5].fittype**

**MarvelTShirts[6].size**

**MarvelTShirts[9].inventory**

# Arrays of Structures

Arrays of structures can be initialize by nesting the initial values for each structure as list elements in the braces enclosing the initial values for the array.

```
struct tshirt MarvelTShirts[10] = {};  
struct tshirt DisneyTShirts[15] = {{ "XS",  
                                     { "S",  
                                       { "M",  
                                         { "L",  
                                           { "XL"  
                                         }  
                                       }  
                                     }  
                                     };  
struct tshirt DCComicsTShirts[5] = {{ "XS", "BLACK", "BATMAN", 'Y', 12.99, 198},  
                                     { "S", "BLUE", "SUPERMAN", 'M', 24.99, 34},  
                                     { "M", "RED", "WONDER WOMAN", 'W', 27.99, 87},  
                                     { "L", "YELLOW", "AQUAMAN", 'M', 26.99, 65},  
                                     { "XL", "GREEN", "GREEN LANTERN", 'Y', 15.99, 81}  
                                     };
```

# Arrays of Structures

Individual elements in an array inside the structure can be accessed the same way as regular arrays.

```
MarvelTShirts[5].color[0] = 'R';  
MarvelTShirts[5].color[1] = 'E';  
MarvelTShirts[5].color[2] = 'D';  
MarvelTShirts[5].fittype = 'Y';  
MarvelTShirts[5].inventory = 123;
```

```
printf("%s\n", MarvelTShirts[5].color);  
printf("%c\n", MarvelTShirts[5].fittype);  
printf("%d\n", MarvelTShirts[5].inventory);
```

RED
Y
123

# Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures.

```
struct tshirt MyTShirts = {"M", "BLUE", "DISNEY", 'W', 29.99, 1};  
struct tshirt *tshirtptr;  
tshirtptr = &MyTShirts;
```

```
printf("MyTShirts.design\t%s\n", MyTShirts.design);  
printf("( *tshirtptr ).design\t%s\n\n", (*tshirtptr).design);
```

MyTShirts.design	DISNEY
(*tshirtptr).design	DISNEY

# Pointers to Structures

In C, it is possible to declare a pointer to any type

This includes pointers to structures in arrays.

```
struct tshirt DCComicsTShirts[5] = {{"XS", "BLACK", "BATMAN", 'Y', 12.99, 198},
                                     {"S", "BLUE", "SUPERMAN", 'M', 24.99, 34},
                                     {"M", "RED", "WONDER WOMAN", 'W', 27.99, 87},
                                     {"L", "YELLOW", "AQUAMAN", 'M', 26.99, 65},
                                     {"XL", "GREEN", "GREEN LANTERN", 'Y', 15.99, 81}
                                     };
```

```
struct tshirt *tshirtarrayptr;
tshirtarrayptr = &DCComicsTShirts[3];
```

```
printf("DCComicsTShirts[3].design\t%s\n", DCComicsTShirts[3].design);
printf("*(tshirtarrayptr).design\t%s\n", (*tshirtarrayptr).design);
```

DCComicsTShirts[3].design	AQUAMAN
(*tshirtarrayptr).design	AQUAMAN



# Pointers to Structures

The () are necessary because the dot selector has precedence over the dereferencing operator \*

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);  
printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
```

Without the (), the compiler complains

```
printf("tshirtptr design\t%s\n\n", *tshirtptr.design);
```

```
error: request for member 'design' in something not a structure or  
union
```

# Pointers to Structures

The concept of a pointer to structure is used so often in C that a special syntax was developed to reference the members of the target structure.

`(*struct_pointer).member` can be written as `struct_pointer->member`

```
printf("tshirtptr design\t%s\n\n", (*tshirtptr).design);
```

```
printf("tshirtptr design\t\t%s\n", tshirtptr->design);
```

```
printf("tshirtarrayptr design\t%s\n", (*tshirtarrayptr).design);
```


```
printf("tshirtarrayptr design\t%s\n", tshirtarrayptr->design);
```

# Passing Structures to and from Functions


Pointers to structures are also used to make structures available to functions.

When a pointer to a structure is passed to a function, the function can access the information in the structure and can modify the information.

```
struct tshirt *tshirtptr = &MyTShirts;  
struct tshirt *tshirtarrayptr = &DCComicsTShirts[3];  
  
UpdateInventory(tshirtarrayptr);
```

```
struct tshirt MyTShirts = {}; 
struct tshirt MarvelTShirts[10] = {};

struct tshirt DisneyTShirts[15] = { {"XS"},
    {"S"},
    {"M"},
    {"L"},
    {"XL"}
};

struct tshirt DCComicsTShirts[5] = {{ "XS", "BLACK", "BATMAN", 'Y', 12.99, 198},
    {"S", "BLUE", "SUPERMAN", 'M', 24.99, 34},
    {"M", "RED", "WONDER WOMAN", 'W', 27.99, 87},
    {"L", "YELLOW", "AQUAMAN", 'M', 26.99, 65}, 
    {"XL", "GREEN", "GREEN LANTERN", 'Y', 15.99, 81}
};

struct tshirt *tshirtptr = &MyTShirts;

struct tshirt *tshirtarrayptr = &DCComicsTShirts[3];
```

## Function call passing the pointer to the structure

```
struct tshirt *tshirtarrayptr = &DCComicsTShirts[3];  
  
UpdateInventory(tshirtarrayptr);
```

## Function receiving the pointer to the structure

```
void UpdateInventory(struct tshirt *TShirtPointer)  
{  
    printf("Enter the new inventory level for the %s design TShirt ",  
           TShirtPointer->design);  
    scanf("%d", &TShirtPointer->inventory);  
  
    return;  
}
```

# Unions

A union is an aggregate data type

A union is like a structure

- it can hold members of different types
- the rules for declaring unions are the same as structures
  - a list of members is declared
  - members of a union cannot be a function and it may not contain a union of its own type
- arrays of unions and pointer to unions are allowable
- the -> member selection method is used with unions

# Unions

A union is different from a structure

- will only contain one of its members at any instant (instead of all members)
- storage is allocated for the largest member (instead of all members)
- when a value is assigned to any member, it will overwrite any previously stored information (values are not retained for all members)

# Unions

```
union tag  
{  
    member list  
};
```

union is a keyword

tag is optional



# Unions

```
union MyNumberUnion
{
    short ashort;
    int bint;
    long clong;
};
```

```
union MyCharUnion
{
    char a5[5];
    char b10[10];
    char c100[100];
};
```

```
union MyNumberUnion MyNU = {};
```

```
union MyCharUnion MyCU = {};
```

```
(gdb) ptype MyNU
type = union MyNumberUnion {
    short int ashort;
    int bint;
    long int clong;
}
```

```
(gdb) ptype MyCU
type = union MyCharUnion {
    char a5[5];
    char b10[10];
    char c100[100];
}
```

# Unions

```
union MyNumberUnion
{
    short ashort;
    int bint;
    long clong;
};
```

```
(gdb) p sizeof(MyNU)
$3 = 8
```

```
union MyCharUnion
{
    char a5[5];
    char b10[10];
    char c100[100];
};
```

```
(gdb) p sizeof(MyCU)
$4 = 100
```

```
union MyNumberUnion MyNU = {};  
union MyCharUnion MyCU = {};
```

# Unions

```
union MyNumberUnion
{
    short ashort;
    int bint;
    long clong;
};
```

```
union MyCharUnion
{
    char a5[5];
    char b10[10];
    char c100[100];
};
```

```
union MyNumberUnion MyNU = {};
union MyCharUnion MyCU = {};
```

<b>30</b>	<b>MyNU.ashort = 1;</b>
\$8	= {ashort = 1, bint = 1, clong = 1}
<b>31</b>	<b>MyNU.bint = 10;</b>
\$9	= {ashort = 10, bint = 10, clong = 10}
<b>32</b>	<b>MyNU.clong = 100;</b>
\$10	= {ashort = 100, bint = 100, clong = 100}

# Unions

```
union MyNumberUnion
{
    short ashort;
    int bint;
    long clong;
};
```

```
union MyCharUnion
{
    char a5[5];
    char b10[10];
    char c100[100];
};
```

```
union MyNumberUnion MyNU = {};
union MyCharUnion MyCU = {};
```

```
38 strcpy(MyCU.a5, "Friday");
```

```
$11 = {a5 = "Frida", b10 = "Friday\000\000\000",
      c100 = "Friday", '\000' <repeats 93 times>}
```

```
39 strcpy(MyCU.b10, "HappyFriday");
```

```
$12 = {a5 = "Happy", b10 = "HappyFrida",
      c100 = "HappyFriday", '\000' <repeats 88 times>}
```

```
40 strcpy(MyCU.c100, "Hello there world. How are you?");
```

```
$13 = {a5 = "Hello", b10 = "Hello ther",
      c100 = "Hello there world. How are you?", '\000'
      <repeats 67 times>}
```

# Unions

```
union MyNumberUnion
{
    short ashort;
    int bint;
    long clong;
};
```

```
union MyCharUnion
{
    char a5[5];
    char b10[10];
    char c100[100];
};
```

```
union MyNumberUnion MyNU = {};
union MyCharUnion MyCU = {};
```

MyNU.ashort	address	0x7fffffffffe7a0
MyNU.bint	address	0x7fffffffffe7a0
MyNU.clong	address	0x7fffffffffe7a0
MyCU.a5	address	0x7fffffffffe730
MyCU.b10	address	0x7fffffffffe730
MyCU.c100	address	0x7fffffffffe730

# Unions

```
struct Request
{
    char program_name;
    int service_code;
    char message[100];
} A;
```

```
struct Reply
{
    char message[100];
    int error;
    long result;
} Z;
```

```
union RequestReply
{
    struct Request A;
    struct Reply Z;
};

union RequestReply Service1 = {};
union RequestReply Service2 = {};
```

# Unions

```
(gdb) ptype Z
type = struct Reply {
    char message[100];
    int error;
    long int result;
}

(gdb) ptype A
type = struct Request {
    char program_name;
    int service_code;
    char message[100];
}
```

```
(gdb) p Service1
$7 = {
    A = {
        program_name = 0
        '\000',
        service_code = 0,
        message = '\000'
        <repeats 99 times>
    },
    Z = {
        message = '\000'
        <repeats 99 times>,
        error = 0,
        result = 0
    }
}
```

```
(gdb) p Service2
$6 = {
    A = {
        program_name = 0
        '\000',
        service_code = 0,
        message = '\000'
        <repeats 99 times>
    },
    Z = {
        message = '\000'
        <repeats 99 times>,
        error = 0,
        result = 0
    }
}
```

# Unions

## CAUTION

There is no automatic mechanism to determine which member of a union is in use at any given time. It is up to the programmer to keep track.

Any member of a union can be accessed at any time; however, the contents of a member may not be meaningful if another member was most recently assigned.



# Typedefs

The `typedef` storage class is used to associate an identifier with a type.

```
typedef old_type new_type;
```

- a `typedef` declaration does not cause any storage to be allocated
- `typedef` is a keyword
- `typedef` appears in declarations

# Typedefs

A `typedef` is similar to a `#define` - they can both define data types

## Differences

`typedef`

- processed by the compiler

- only used to define data types

`#define`

- processed by the preprocessor

- can be used to define constants, macros, and other entities as well as data types

# Typedefs

<code>typedef short MyShort;</code>	<code>(gdb) ptype x</code>
<code>typedef int MyInt;</code>	<code>type = short int</code>
<code>typedef long MyLong;</code>	
	<code>(gdb) ptype y</code>
<code>MyShort x = 0;</code>	<code>type = int</code>
<code>MyInt y = 1;</code>	
<code>MyLong z = 2;</code>	<code>(gdb) ptype z</code>
	<code>type = long int</code>

# Typedefs

```
typedef short Velma;  
typedef int   Daphne;  
typedef long  Fred;  
typedef char  Shaggy[100];  
typedef char  Scooby;
```

```
Velma Dinkley;  
Daphne Blake;  
Fred Jones;  
Shaggy Rogers;  
Scooby Doo;
```

```
Dinkley = 10;  
Blake = 1;  
Jones = 3;  
Doo = (Dinkley*Jones+Jones);  
strcpy(Rogers, "Scooby Snacks");
```

```
printf("%s %c%c%c%c\n", Rogers, (Dinkley*9-1), (Blake+'T'), (Jones*Dinkley+'/' ), Doo);
```

```
(gdb) ptype Dinkley  
type = short int  
(gdb) ptype Blake  
type = int  
(gdb) ptype Jones  
type = long int  
(gdb) ptype Rogers  
type = char [100]  
(gdb) ptype Doo  
type = char
```

```
Scooby Snacks YUM!
```