No Classes in C

- Because C is not an OOP language, there is no way to combine data and code into a single entity.
- C does 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.
- Coming from an objected-oriented programming background, think of how classes are an extension of struct. Classes have data members but allow you to restrict access to them while providing a mechanism to organize and bundle a set of related functions. You can think of a struct as an OOP class in which all data members are public, and which has no methods, not even a constructor.

Struct Definition

- Implementation-wise, a **`struct`** variable represents of block of memory where a heterogeneous set of variables is stored. It is defined by a description of the type of each variable stored along with the offset of each variable from the beginning of the block which is computed by the compiler.

- The general form of a structure definition is

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

- 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 */
};
```

- 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. Note: The ability to return a structure variable provides the option to bundle multiple values in the one return variable provide.
- The members of a **`struct`** are variables just like any other and can 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.

```
// 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 p;
  printf("please input the x-and y-coordinates: ");
  scanf("%d %d", &p.x, &p.y);
  return p;
}

int main ( ) {
  struct point endpoint; // endpoint is a struct point variable
  endpoint = inputPoint( );

  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;

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

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

struct with typedef and enum

- 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_t;
```

- The above code defines the structure named point and defines **`POINT_t`** as a **`typedef`** (alias) for the **`struct`**. We can now declare variables, parameters, etc., as:

```
struct point endpoint;
```
or as:
```
POINT_t upperRight;
```

- The same can be done with enum

```
typedef enum optional_tag {} typename;
typename var1;
```

- There are arguments either way, but I recommend to always defining a struct with typedef. The same goes for enum.

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`**

- So, functionally

```
p2 = p1;
```

- is the same as copying each element

```
p2.x = p1.x;
p2.x = p1.y;
```

- Behind the scenes, **`p2 = p1`** (copying structs) represents copying a block of memory with multiple variables.

struct within a struct

- A data element in a struct may be another struct (similar to composition with classes in Java / C++).

- This example defines a line in the coordinate plane by specifying its endpoints as POINT structs

```
typedef struct line {
  struct point leftEndPoint;
  struct point rightEndPoint;
} LINE_t;
```

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

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

same as

line1.leftEndPoint.x line1.rightEndPoint.x
line2.leftEndPoint.x line2.rightEndPoint.x

Arrays of struct

- Since a **`struct`** is a variable type, we can create arrays of structs just like we create arrays of **`int`**, **`char`**, **`double`**, etc.
- Writing the declaration for an array of 5 line structures name "lines":

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

-or-

```
LINE_t lines[ 5 ];
```

- 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);
```

- Example:

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

  /* write code to initialize all data members to zero */
  for (k = 0; k < 5; k++) {
    lines[k].leftEndPoint.x  = 0;
    lines[k].leftEndPoint.y  = 0;
    lines[k].rightEndPoint.x = 0;
    lines[k].rightEndPoint.y = 0;
  }

  /* call the printPoint( ) function to print
  ** the left end point of the 3rdline */
  printPoint( lines[2].lefEndPoint);
  return 0;
}
```

Arrays of struct

- Structs may contain arrays as well as primitive types

```
struct month {
  int nrDays;
  char name[ 3 + 1];
};
struct month january = { 31, "JAN"};
```

- Note **`january.name[2]`** is '**`N`**'
- Example:

```
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]
printf( "%c\n",allMonths[3].name[1]);
```

P

```
printf( "%s\n",allMonths[3].name);
```

APR

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;
};
```

- 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");
```

- 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

Unions

- A `union` is a variable type thatm like a `struct` may hold different type members of different sizes, ***BUT, unlike `struct`, unions are active for 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.

- A good reference is: http://www.geeksforgeeks.org/difference-structure-union-c/. Their definition of a `union` is: "a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes."

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

union vs struct

- Similarities: (ref: geeksforgeeks website from previous slide)
  1) Both are user-defined data types used to store data of different types as a single unit.
  2) Their members can be objects of any type, including other structures and unions or arrays. A member can also consist of a bit field.
  3) Both structures and unions support only assignment = and sizeof operators. The two structures or unions in the assignment must have the same members and member types.
  4) A structure or a union can be passed by value to functions and returned by value by functions. The argument must have the same type as the function parameter. A structure or union is passed by value just like a scalar variable as a corresponding parameter.
  5) '.' operator is used for accessing members.

- Differences: (ref: geeksforgeeks website from previous slide)

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

Union example #1

```
union data {
  int x;
  char c[8];
} ;
int i;
union data item;
item.x = 42;
typedef union item DATA_t;

printf("%d, %o, %x", item.x, item.x, item.x );
for (i = 0; i < 8; i++ )
  printf("%x ", item.c[i]);
printf( "%s", "\n");
printf("%s\n", "size of DATA = ", sizeof(DATA_t));
```

Note that x and c occupy the same general area of memory

A second example is shown on the following slide... To name the reference again; http://www.geeksforgeeks.org/difference-structure-union-c/ has a number of additional examples illustrating the differences between unions and structures. These can be worked directly on their website if that may be helpful.

Union example #2

```
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;
};

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;
  }
}
```

struct Storage in Memory

- struct elements are stored in the order they are declared in.
  - a then b in this case:

```
typedef struct tag { char a8; int b16; } T;
```

- The total size reserved for a struct variable is not quite the sum of the size of the elements

`sizeof(T)`	Is NOT necessarily the same as:	`sizeof(char) + sizeof(int)`

- In various systems, byte alignment requirements force some variables types to be aligned to certain memory boundaries (usually some small power 2) to be operated. The size of structure itself may be forced to be a multiple of some word size. These require padding bytes between members in memory and/or at the end of the structure. The padding represents wasted space.

- If a coder reorders the members it may reduce the total number of padding bytes required. A typical rule is to put larger, more constrained types first in the `struct` and place types like `char` last.

- Special compiler directives (options) may allow packing, reducing or eliminating padding, but it may come at a large cost in speed, requiring extra moves at run time, while saving memory since data has to be manipulated

- In the 8bit AVR with single-byte memory accesses, there is no padding.

how to: Printing the Bytes of a Structure to See Padding

Code:

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

typedef struct dummy_tag1 {
  signed char c1;
  int i1;
  signed char c2;
} big_t;

typedef struct dummy_tag2 {
  int i1;
  signed char c1;
  signed char c1;
} small_t;


int main(){

  big_t big =     {1,-1,1};
  small_t small = {-1,1,1};

  unsigned char * ptrByte; //pointer for accessing individual bytes

  ptrByte = (unsigned char *)&big;
  printf("BIG:(%d bytes):\n",sizeof(big_t));
  for (int i=0; i<sizeof(big_t);i++){
    printf("%02x\n",*ptrByte);
    ptrByte++;
  }


  ptrByte = (unsigned char *)&small;
  printf("SMALL(%d bytes):\n",sizeof(small_t));
  for (int i=0; i<sizeof(small_t);i++){
    printf("%02x\n",*ptrByte);
    ptrByte++;
  }

  return 0;
}
```

Compile:

```
$ gcc -Wall -std=c99 ./test.c
```

First Call

```
$ ./a.out
BIG:(12 bytes):
01
00
00
00
ff
ff
ff
ff
01
00
00
00
SMALL(8 bytes):
ff
ff
ff
ff
01
01
5e
57
```

Second Call

```
$ ./a.out
BIG:(12 bytes):
01
00
00
00
ff
ff
ff
ff
01
00
00
00
SMALL(8 bytes):
ff
ff
ff
ff
01
01
c9
50
```

Wasted Space for Padding is highlighted red (platform dependent). The last two bytes of small are garbage values, illustrated by the juxtaposition of two successive runs.

Examining Bytes of a Union

Code:

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

typedef union dummy_tag1 {
  signed char c1;
  int i1;
} T ;

int main(){

  T myUnion;
  unsigned char * ptrByte; //variable for pringting bytes

  printf("sizeof(unsigned char):%d byte\n",sizeof(unsigned char));
  printf("sizeof(int):%d bytes)\n",sizeof(int));
  printf("sizeof(T):%d bytes\n",sizeof(T));

  myUnion.i1 = 0; //clear all the b
  printf("Cleared Bytes of Union Variable:\n");
  ptrByte = (unsigned char *)&myUnion;
  for (int i=0; i<sizeof(T);i++){
    printf("%02x\n",*ptrByte++);    ptrByte++;
  }

  myUnion.c1 = -1;
  printf("After setting member c1 to -1:\n");
  ptrByte = (unsigned char *)&myUnion;
  for (int i=0; i<sizeof(T);i++){
    printf("%02x\n",*ptrByte);    ptrByte++;
  }

  myUnion.i1 = -1;
  printf("After setting member i1 to -1:\n");
  ptrByte = (unsigned char *)&myUnion;
  for (int i=0; i<sizeof(T);i++){
    printf("%02x\n",*ptrByte);    ptrByte++;
  }
  return 0;
}
```

Compile:

```
$ gcc -Wall -std=c99 ./test.c
```

Run

```
$ ./a.out
sizeof(unsigned char):1 byte
sizeof(int):4 bytes)
sizeof(T):4 bytes
Cleared bytes of union variable:
00
00
00
00
After setting member c1 to -1:
ff
00
00
00
After setting member i1 to -1:
ff
ff
ff
ff
```