CSE 220 – C Programming

Structures, Unions, Enumerations

Custom Data Types:

- Structures
- Union (optional content)
- Enumeration (optional content)

Structure Variables

- Structure: a collection of values
- Members may have different types
- Members have names:
  - To select a member, specify name not position
- Use when we need to store a collection of related data items

Examples

Consider a program that needs to store the following information about a car:

Make, model, engine size, horse power

What variables do you need to declare?

```
char make[30];
char model[10];
float engineSize;
int  horsePower;
```

On Mimir, what is the size of a ***char*** ?

```
1 bytes (8 bits)
2 bytes (16 bits)
4 bytes (32 bits)
I don't know
```

On Mimir, what is the size of a ***int*** ?

```
1 bytes (8 bits)
2 bytes (16 bits)
4 bytes (32 bits)
I don't know
```

On Mimir, what is the size of a ***float*** ?

1 bytes (8 bits)

2 bytes (16 bits)

4 bytes (32 bits)

I don't know

On Mimir, what is the size of a ***char[10]***?

10 bytes (80 bits)

20 bytes (160 bits)

40 bytes (320 bits)

I don't know

Examples

Consider a program that needs to store the following information about a car:

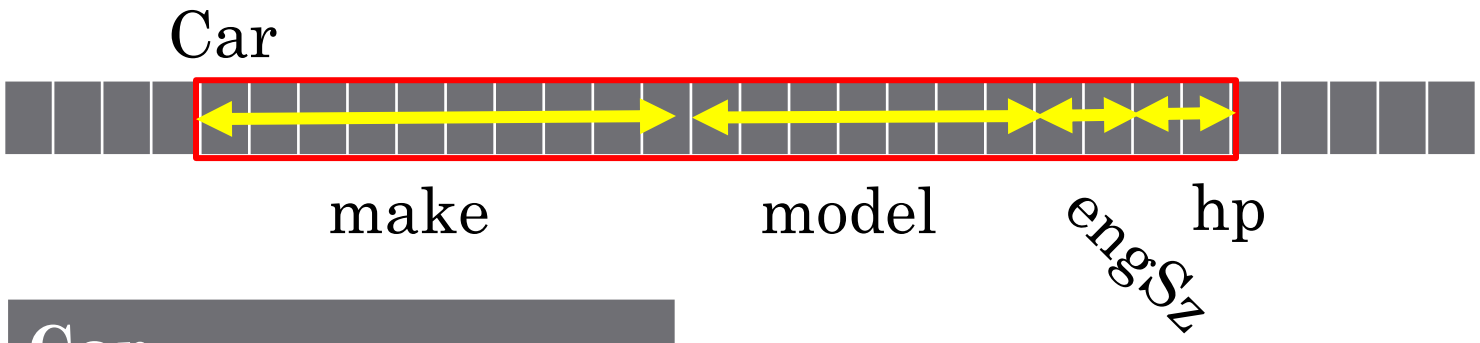
Make, model, engine size, horse power

What if you need to collect this data about 3 cars?

```
char make1[30], make2[30], make3[30];
char model1[10], model2[10], model3[10];
float engineSize1, engineSize2, engineSize3;
int  horsepower1, horsepower2, horsepower3;
```

Example

Better to have one variable containing all pieces of information:



Car

```
char make[30];
char model[10];
float engineSize;
int  horsePower;
```

On Mimir, what is the size of a *struct Car*?

Car

```
char make[30];
char model[10];
float engineSize;
int  horsePower;
```

48 bytes

56 bytes

60 bytes

I don't know

Declaration

- To declare a structure variable, you need to specify the type, the elements and the variable name

```
struct { …. } var;          //Single variable: var

struct { …. } var1, var2;//Multiple variables: var1, var2
```

Example:

```
struct {
    char make[30];
    char model[10];
    float engineSize;
    int horsepower;
} car1, car2, car3;
```

Examples

Declare a struct to represent the following:

Circle

```
float x;
float y;
float radius;
```

```
struct {
   float x;
   float y;
   float radius;
} circle;
```

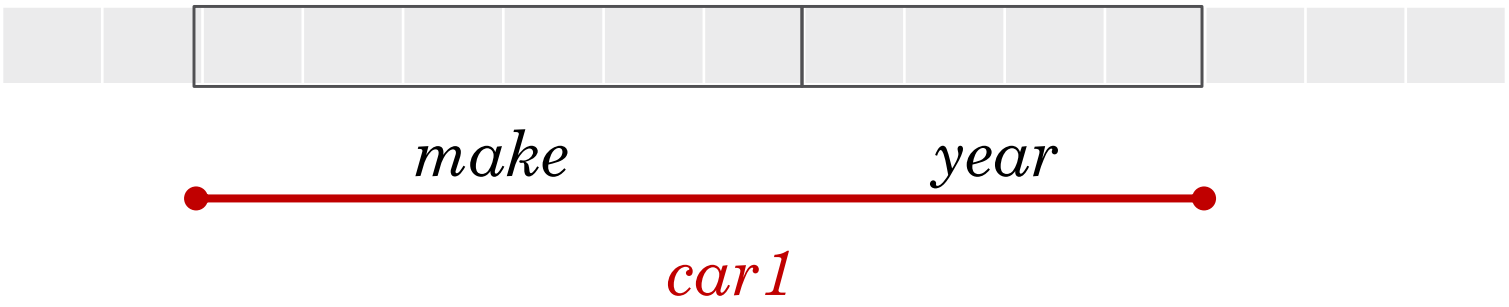
Student

```
char
name[100];
int grade;
```

```
struct {
   char name[100];
   int grade;
} student;
```

Storage

```
struct {
   char make[30];
   int year;
} car1;
```



- Members are stored in memory in the order in which they are declared.

Initialization

```
struct {
     char make[MAX_LEN +1];
     int year;
} car1 = {“Volvo”, 2008},
car2 = {“BMW”, 2010};
```

- Variables can be initialized at the same time of declaration
- Values in initializer must appear in the same order as in the structure

Accessing members

- To access the value of a struct member use: *struct_name.member_name*:

```
car1.year = 2008;
strcpy(car1.make, “Ford”);
printf(“My new car is a %d %s\n”, car2.year, car2.make);
car2.year++;
scanf(“%d”, &car1.year);
```

```
struct {
  char title[101];
  int year;
  float imdb_rating;
} matrix = {"The Matrix", 1999, 8.7}, lotr;

strcpy(lotr.title, "Lord of the Rings");
lotr.imdb_rating = 8.8;
lotr.year = 2001;

char * better_movie;
if (lotr.imdb_rating > matrix.imdb_rating) {
  better_movie = lotr.title;
} else {
  better_movie = matrix.title;
}
printf("%s", better_movie);
```

What does this code output?

The Matrix

Lord of the Rings

lotr.title

I don't know

Copying a struct

- Copy a struct into another:

```
car1 = car2;
```

copies car2.year into car1.year and car2.make into car1.make

  - car1 and car2 must have compatible types (see next slide).
  - Arrays cannot be copied using the = operator, but when inside a struct, they can.

- Cannot use == or != to check if 2 structs are equal or not

Compatible Structures

```
struct {
    char make[M+1];
    int year;
} car1, car2;
…
struct {
    char make[M+1];
    int year;
} car3;
```

car1 and car2 are compatible

car1 and car3 are not compatible

Structure types

```
struct {
    char make[M+1];
    int year;
} car1, car2;
…
struct {
    char make[M+1];
    int year;
} car3;
```

- Repeating structure information: program hard to maintain
- The 2 structs are **not** compatible with each others
- Solution: define a type for the struct or use tag

Structure Tag

```
struct Car {
    char make[M+1];
    int year;
};      //Need a ; here
…
```

Car is a tag, a name given to that particular structure
You have not yet declared a variable having this structure

```
…
struct Car car1;     //Need the keyword struct
struct Car car2 = {“Ford”, 2009};
car1 = car2;         //Valid, since types are
compatible
```

```
struct Movie {
  char title[101];
  int year;
  float imdb_rating;
}

Movie matrix = {"The Matrix", 1999, 8.7};
Movie lotr = {"Lord of the Rings", 2001, 8.8};
Movie better_movie;

if (lotr.imdb_rating > matrix.imdb_rating) {
  better_movie = lotr;
} else {
  better_movie = matrix;
}
printf("%s", better_movie.title);
```

What does this code output?

Lord of the Rings

The Better Movie

Error

I don't know

```
struct Movie {
  char title[101];
  int year;
  float imdb_rating;
};

struct Movie matrix = {"The Matrix", 1999, 8.7};
struct Movie lotr = {"Lord of the Rings", 2001, 8.8};
struct Movie better_movie;

if (lotr.imdb_rating > matrix.imdb_rating) {
  better_movie = lotr;
} else {
  better_movie = matrix;
}
printf("%s", better_movie.title);
```

What does this code output?

Lord of the Rings

The Better Movie

Error

I don't know

Structure Type Definition (optional content)

```
typedef struct {
    char make[M+1];
    int year;
} Car;      //Need a ; here
…
Car car1;          /* Don't need the keyword
struct, since struct defined with typedef */
Car car2 = {"Ford", 2009};
car1 = car2;
```

Arguments and Return value

Structs can be passed as parameters to a function and as return values:

```
void printCarInfo(struct Car car) {
        printf("%d: %s\n", car.year, car.make);
}
struct Car newerCar(struct Car car1, struct Car car2) {
        if (car1.year > car2.year) {
                return car1;
        }
        return car2;
}
```

Structs are passed by value (a copy is made and sent to the function)

Pointers to Structs

```
//Define struct type
struct Circle { float x; float y; float radius;};
struct Circle c, *ptr;

//Declare a circle, and a pointer
ptr = &c;

//Access using the pointer:
(*ptr).x = 2.0;
(*ptr).y = 3.0;

//Alternative way to access members (optional content):
ptr -> x = 2.0;
ptr -> y = 3.0;
```

Nested Structures

- May declare nested structures:

```
struct Engine {
    float size;
    int horsepower;
} ;
struct Car {
    char make[M+1];
    int year;
    struct Engine engine;
} ;
```

```
struct Car car1;
car1.engine.size = 3.0f;
car1.engine.horsepower = 245;
```

Array of Structs

- Array whose elements are structures:

```
struct Student {
   char name[M+1];
   int grade;
} ;
struct Student cse_students[50];
struct Student honors_students[ ] =
       { {“Jim”, 85}, {“Dalia”, 95}, {“Katie”, 85}};
printInfo(&honors_students[1]);
//Pass a pointer to the ith element
```

Array of Structs

- Define printInfo

```
void printInfo(struct Student *stdPtr) {
     printf("Name: %s\tGrade: %d\n",
           (*stdPtr).name,
           (*stdPtr).grade);
}
```

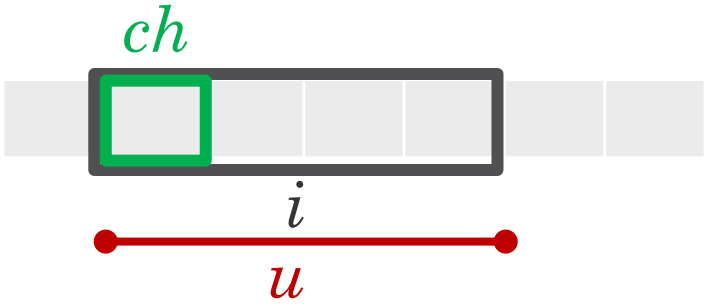
Custom Data Types:

- Structures
- Union (optional content)
- Enumeration (optional content)

Unions (optional content)

- Consist of one or more members, possibly of different types.
- All members share the same space
- Store one member at a time
- Allocates enough space to fit the largest

```
union {
    char ch;
    int i;
} u;
```



Example (optional content)

- Use to build mixed data structures

```
struct Truck {
    char make[30];
    int year;
    float bedLength;
    int towCap;
};
```

```
struct Car {
    char make[30];
    int year;
    int seats;
};
```

```
struct Vehicle {
  char type;
  union {
     struct Truck truck;
     struct Car car;
  } details;
};
```

Enumerations (optional content)

- Enumerated type: type whose values are listed

```
enum {red, yellow, green, blue,
black, orange} c1, c2;
```

- Enumeration Tag:

```
enum Color {red, yellow, green, blue,
black, orange};
enum Color c1, c2;
//Declare 2 variables of type Color
```

- Enumeration typedef:

```
typedef enum {PASS, FAIL} Grade;
Grade g1, g2;
//Declare 2 variables of type Grade
```

Use as integers (optional content)

- Enum variables are stored as integers:

```
enum {red, yellow, green} c1;
enum {red = 0, yellow = 1, green= 2} c2;
enum {car = 10, truck = 20, suv = 30,
bus} vt;
c1 = green;                 //c1 = 2
vt = truck;                 //vt = 20
int val = vt + 5;           //val = 25
int y = bus;                //y = 31
```