a place of mind

THE UNIVERSITY OF BRITISH COLUMBIA

CPSC 259

C Structures

Records (Structures)

- Often, we need to deal with related data (i.e. several *attributes*) about a specific entity, e.g.:
  - an employee who can be identified by a unique employee number, and has the additional (possibly non-unique) attributes: name, street address, city, province, postal code, salary, job title, etc.
- Declare a structure using the struct keyword and structure name
  - attributes are declared within the structure

```
struct Employee {
  int empnum;
  char name[MAXLEN];
  double salary;
};
```

Records (Structures)

Defining and declaring

- The structure definition tells the compiler how it is laid out in memory and details the attribute (member) names
  - does not allocate any memory
- Memory is allocated (e.g. on the stack) when a structure variable is declared

```
struct Employee {
  int empnum;
  char name[MAXLEN];
  double salary;
};
```

```
struct Employee boss1;
```

type name

- Define the structure as a type, so that it can be declared without using the `struct` keyword

```
typedef struct {
  int empnum;
  char name[MAXLEN];
  double salary;
} Employee;
```

```
Employee boss1;
```

struct variables have size and a memory address

address of struct variable is the address of its first attribute

Initialisation and access

- Constant values can be assigned to the members of a structure when the structure variable is declared
  - If no initialisation is explicitly given, C automatically assigns some default values (zero for int/float, '\0' for char and string)

```
Employee lecturer_a = {49239724, "Geoff", 27095.27};
```

- Members can be accessed using the '.' (dot) operator
  - syntax: `variable_name.member`

```
Employee lecturer_a = {49239724, "Geoff", 27095.27};
Employee instructor_b = {72551222, "Cristian", 127028.64};

printf("%s's salary: $%.2f", instructor_b.name, instructor_b.salary);
lecturer_a.salary = 27543.82;
```

Arrays of structures

- Syntax for declaring and accessing is the same as arrays for any other type (using [ ])

```
Employee staff_junior[20];
```

```
staff_junior[0].empnum = 35448722;
strcpy(staff_junior[0].name, "Susan");
staff_junior[0].salary = 32000.00;
```

- Structures can also be declared into dynamic memory

```
Employee* vice_president;
vice_president = (Employee*) malloc(sizeof(Employee));
```

- To access members in dynamic memory, first dereference the pointer to get the structure
  - or, use the "pointing-to" operator (`->`) on the pointer itself

```
(*vice_president).salary += 10000.00;
```

```
vice_president->salary = 150000.00;
```

Dynamic arrays of structures

- Arrays of structures in dynamic memory still follow same rules and syntax for declaration, allocation, and access as for other data types

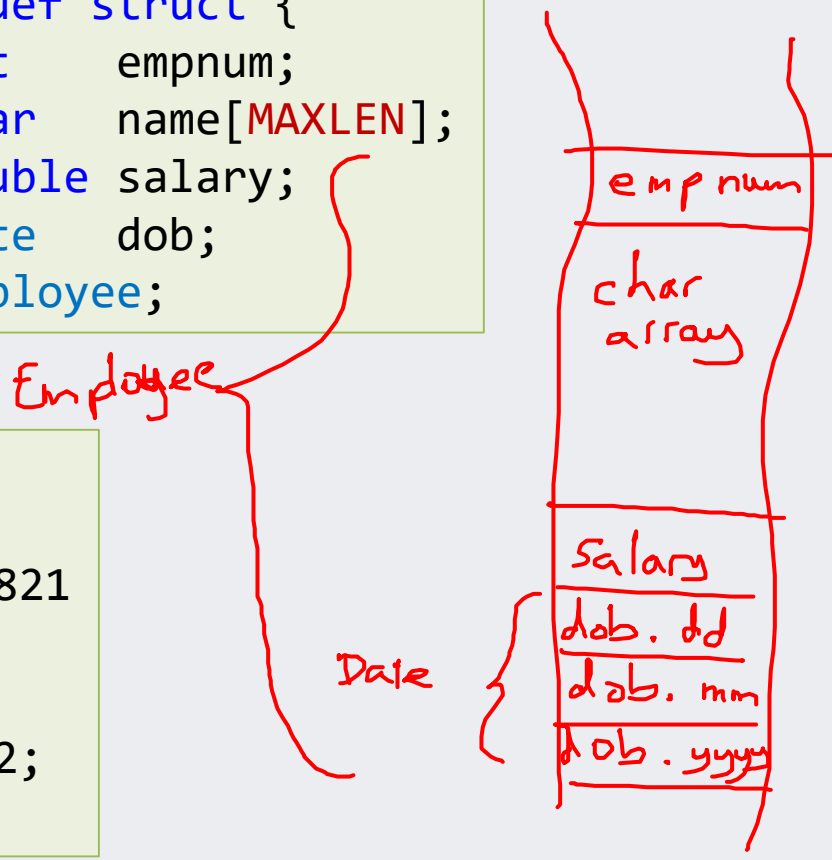
```
Employee* staff_senior;
staff_senior = (Employee*) malloc(num_staff_senior *
                                  sizeof(Employee));

// accessing using array syntax
staff_senior[i].empnum = 100 + i;

// accessing using pointer arithmetic
(staff_senior + i)->salary = 80000;
(*(staff_senior + i)).salary *= 1.05;
```

Nested structures

- A structure can be a member of another structure

```
typedef struct {
  int dd;
  int mm;
  int yyyy;
} Date;
```

```
typedef struct {
  int    empnum;
  char   name[MAXLEN];
  double salary;
  Date   dob;
} Employee;
```




```
int main() {
  Employee instructor;
  instructor.empnum = 62234821
  instructor.dob.dd = 10;
  instructor.dob.mm = 11;
  instructor.dob.yyyy = 1962;
};
```

Passing structs as parameters

- By default, structs behave like ordinary variable parameters and are passed by value (a copy gets placed on call stack)
  - This can be inefficient with large structures or frequently-called functions

```
printEmp(new_boss);
```

```
void printEmp(Employee emp) {
  printf("Employee number: %d\n", emp.empnum);
  printf("Employee name: %s\n", emp.name);
  printf("Employee salary: $%.2f\n\n", emp.salary);
}
```

- Thus we can also pass structures by reference

```
printEmp(&new_boss);
```

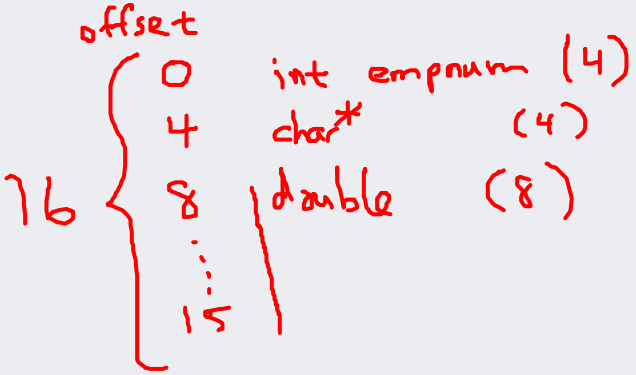
See `employee_records.c`

```
void printEmp(Employee* emp) {
  printf("Employee number: %d\n", (*emp).empnum);
  printf("Employee name: %s\n", (*emp).name);
  printf("Employee salary: $%.2f\n\n", (*emp).salary);
}
```

Structure variables

What does a struct variable look like in memory?

- What is the size of the `Employee` structure, given that `sizeof(int)`=4, `sizeof(char*)`=4, and `sizeof(double)`=8?




```
typedef struct {
  int    empnum;
  char*  name;
  double salary;
} Employee;
```

a) 12 bytes
b) 16 bytes
c) 20 bytes
d) 32 bytes
e) We can't estimate the size since we don't know how many characters are in the `name` field

minimum size of struct is sum of attribute sizes.

Structure variables

What's in a pointer attribute?

- What is stored in the `name` field of the `boss` variable?

```
typedef struct {
  int    empnum;
  char*  name;
  double salary;
} Employee;
```

```
Employee boss;
```

a) The `name` field eventually contains a character string of some currently unknown length, so the size of `boss` will change

b) The `name` field eventually contains a character string of some currently unknown length, but the size of `boss` will not change

c) The `name` field is a pointer to another area of memory that eventually holds a character string of some currently unknown length, and the size of `boss` will not change

Structure variables

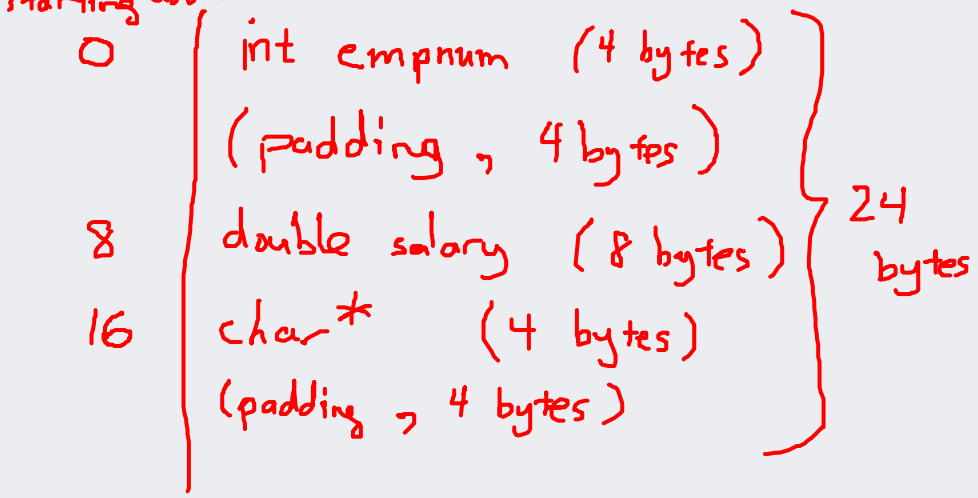
What does a struct variable look like in memory?

- What is the size of the `Employee` structure, given that `sizeof(int)`=4, `sizeof(char*)`=4, and `sizeof(double)`=8?

a) 12 bytes
b) 16 bytes
c) 20 bytes
d) 32 bytes
e) Something else 24 bytes

Order reversed from a previous iClicker

```
typedef struct {
  int    empnum;
  double salary;
  char*  name;
} Employee;
```



Structure alignment

```
typedef struct {
  int     empnum;
  char*   name;
  double salary;
} Employee;
```

offset size
0 4
4 4
8 8

size: 16 bytes

```
typedef struct {
  int     empnum;
  double salary;
  char*   name;
} Employee;
```

offset size
0 4 (4)
8 8
16 4 (4)

size: 24 bytes

- The size difference is due to alignment of variables in memory

- Alignment
  - starting address of variables must be an exact multiple of their size
  - applies to:
    - individual variables
    - `struct` members
    - `struct` variables (aligned to size of largest attribute)
  - simplifies memory access (outside of course scope)

See Lab 2 take-home, non-coding question

Structure variables

A longer example

- Suppose we want to define a structure for holding airline flight information

```
struct Flight {
  int  flightnumber;
  char source[32];
  char destination[32];
};
```

- We can declare and initialize a local record in the following ways:

```
struct Flight AC = {101, "Vancouver", "Calgary"};
```

```
struct Flight AC;
AC.flightnumber = 101;
strcpy(AC.source, "Vancouver");
strcpy(AC.destination, "Calgary");
```

Structure variables

A longer example

- Suppose we want to define a structure for holding airline flight information

```
struct Flight {
  int  flightnumber;
  char source[32];
  char destination[32];
};
```

- Using a pointer to declare and assign values to exactly one flight

```
struct Flight* dynamicAC;
```

```
dynamicAC = (struct Flight*) malloc(sizeof(struct Flight));
dynamicAC->flightnumber = 301;
strcpy(dynamicAC->source, "Montreal");
strcpy(dynamicAC->destination, "Toronto");
```

Structure variables

A longer example

- Suppose we want to define a structure for holding airline flight information

```
struct Flight {
  int  flightnumber;
  char source[32];
  char destination[32];
};
```

- Dynamically allocate more than one plane, but use only one pointer

```
dynamicAC2 = (struct Flight*) malloc(3*sizeof(struct Flight));
```

```
dynamicAC2[1].flightnumber = 402;
strcpy(dynamicAC2[1].source, "Toronto");
strcpy(dynamicAC2[1].destination, "San Francisco");
```

Structure variables

A longer example

- Use a (local) array of pointers to Flight structures
  - each pointer can point to zero, one, or more dynamically allocated Flight structs

```
struct Flight* dynamicWJ[10];
dynamicWJ[0] = NULL; // zero flights
dynamicWJ[1] = NULL;
dynamicWJ[7] = (struct Flight*) malloc(5*sizeof(struct Flight));
dynamicWJ[8] = (struct Flight*) malloc(1*sizeof(struct Flight));
dynamicWJ[9] = (struct Flight*) malloc(100*sizeof(struct Flight));
```

See pointers_airplanes_dma_handout.pdf and pointers_airplanes_dma.c

- Dynamic 2D array

```
struct Flight** dynamicAA;
dynamicAA = (struct Flight**) malloc(20*sizeof(struct Flight*));
dynamicAA[0] = (struct Flight*) malloc(5*sizeof(struct Flight));
```

Readings for this lesson

- Thareja
  - Chapter 5.1 – 5.5 (Structures)

- Next class: Algorithm complexity
  - Thareja, Chapter 2.8 – 2.11