GNG1106
Fundamentals of Engineering Computation

Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

In-Class Exercise:

Outline

1. Structure

- Structures are used to group variables or arrays under one name.
- A structure may contain variables of different types (recall that an array contains variables of the same type).
- A structure is used to group data which are naturally bundled together, allowing for better data encapsulation and program modularization. For example:
  - A student's record in a course naturally contains: the student's first name, last name, student number, grade, etc.; one can define a structure that contains all of these data items.
  - an object, say a cube, has attributes color, height, width and length; one can define a structure that contains all of these items.
- Structures are used to develop "data structures and algorithms" (which is a course by itself)

Defining a Structure

- A structure can be viewed as a user-defined type or a “complex type”.

```
struct
studentRecord
{
 int ID;
 char name[200];
 float grade;
};
```

- This essentially creates a new type called “`struct studentRecord`”
- It does not declares a variable! It serves as the “template” for variables declared to have this type.
- It says that a variable with this type will contain three members:
  - an int-type variable called `ID`
  - a char array called `name`
  - a float-type variable called `grade`

- The keyword struct introduces the structure definition
- Notice the ending semi-colon in the definition.
- The variables declared within the `{ }` are the structure's members.
    - any number of members are allowed
    - members can be of any type, including arrays, pointers, and even other structures.
    - members within the same structure definition must have unique names.
- Defining a structure does not reserve any memory.
- Once you use a specific instance of a structure (namely, declare a variable to have the defined structure type), memory is allocated.
- Initialization inside structure definition is not allowed.

Declaring a Variable of Structure Type

```
#define RED 'r'
#define GREEN 'g'
struct cube
{
 char color;

 double height;

 double width;

 double length;

};
```

Suppose that we already have "`struct cube`" defined and have some symbolic constants defined. Declaring a variable of the type "`struct cube`" can be:

```
struct cube myCube;
```

- `myCube` is the variable name.
- The declaration suggests that `myCube` includes four "member variables".
  - a char-typed variable called `myCube.color`;
  - three double-typed variables called `myCube.height`, `myCube.width`, and `myCube.length`

Declaring and Initialization of Structure Typed Variables

Using the previous example:

```
struct cube myCube={GREEN, 0.3, 0.4, 0.5};
```

With this statement, not only the variable `myCube` is declared to have type `struct cube`, its member variables are also assigned values.

- `cube.color` is assigned 'g'.
- `cube.height` is assigned 0.3.
- `cube.width` is assigned 0.4.
- `cube.length` is assigned 0.5.

Highlight

When declaring and initializing a structure typed variable, the list of values must be in the same order as the order of members in the structure definition.

Operations with Structure Typed Variables

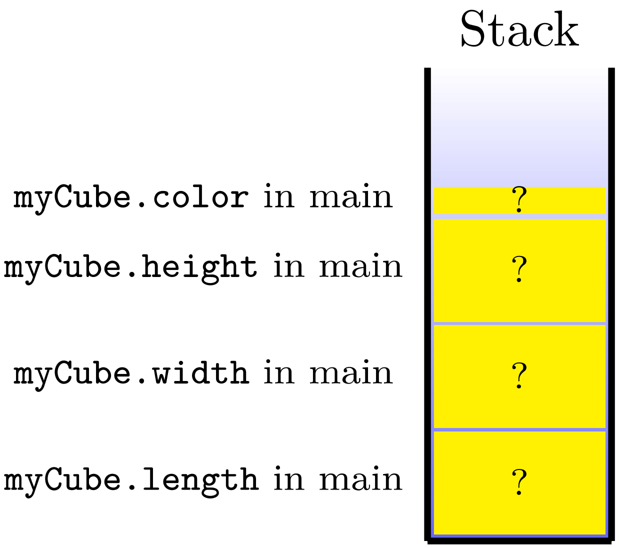
The only operations that can be performed on a structure variable are:

- assigning a structure variable to another structure variable of the same type using "`=`" (content is copied)
- taking the address of a structure variable using the address operator "`&`"
- determining the size of a structure variable using the `sizeof` operator
- accessing a structure's members (using `structureName.memberName`, e.g., `myCube.color`)

Highlight

Structure variables cannot be compared!

- The declaration of a structure-typed variable X inside a function results in the allocation of memory in the stack for all member variables of X.



For example, the declaration

```
struct cube myCube;
```

in `main` will results in a memory allocation in the stack as shown in the picture.

The **typedef** command gives an existing type a new name, via

```
typedef existingType newTypeName;
```

```
typedef int ROUNDNUMBER;
...
ROUNDNUMBER x;
```

- In this code, the int type is given a new name called "`ROUNDNUMBER`".
- Then "`ROUNDNUMBER x;`" does exactly the same thing as "`int x;`".
- After the existing type is given the new name, its original name is still valid.

Highlight

The command **typedef** are often exploited to simplify the declaration of a structure typed variable.

Typedef a structure: basic form

```
struct studentRecord
{
 int ID;

 char name[200];

 float grade;

};
typedef struct studentRecord REC;
```

With this code, the type "`struct studentRecord`" has been renamed as type `REC`, and we can simply declare a variable of type "`struct studentRecord`" by

```
REC rec1;
```

Typedef a structure: a “combined” form

Defining “`struct studentRecord`” type and renaming it can be combined in one command as follows.

```
typedef struct studentRecord
{
 int ID;

 char name[200];

 float grade;

} REC;
```

Typedef a structure: the short form

Since we are going to rename the "**`struct studentRecord`**" type, why bother giving it a name in the first place? C actually allows a further simplified form.

```
typedef struct
{
 int ID;

 char name[200];

 float grade;

} REC;
```

Pointers Pointing to a Structure

- A pointer can be declared to point to a structure.
- After the pointer is assigned the address of a structure-typed variable (or an array of such variables), to access the member variables of the structure-typed variable(s), one can use the following generic procedure as one would expect:
    1. first dereference the pointer to obtain the variable
    2. then use "`.memberVariable`" to access any of its members.
- There is an alternative way to access the member variables of a structured variable to which a pointer points.

- Suppose that the following structure has been defined.

```
typedef struct
{
  float length;
  float width;
}RECTANGLE;
```

The generic way:

```
RECTANGLE A={1, 2}, B[2]={{3, 4}, {5, 6}}, *ptr;
ptr=&A;
printf("%f\n", (*ptr).width);
ptr=B;
printf("%f\n", (*ptr).width);
printf("%f\n", (*(ptr+1)).width);
ptr[1].length=20;
printf("%f\n", ptr[1].length);
```

The alternative way:

```
RECTANGLE A={1, 2}, B[2]={{3, 4}, {5, 6}}, *ptr;
ptr=&A;
printf("%f\n", ptr->width);
ptr=B;
printf("%f\n", ptr->width);
printf("%f\n", (ptr+1)->width);
(ptr+1)->length=20;
printf("%f\n", (ptr+1)->length);
```

Programming Example

Implement and test the following functions.

- a function that prints the information (length and width) of a RECTANGLE variable
- a function that asks the user to fill in the information (length and width) of a RECTANGLE variable
- a function that prints the information (length and width) of an array of RECTANGLE variables

Coding Demonstration