

Introduction to Computing

MCS1101B

Lecture 8

User Defined Datatypes

- Sometimes basic data-types are not sufficient for describing problems conveniently, *e.g., 2D coordinates, complex numbers, student information, etc.*
- You can define your own data-type as per your requirements
- You need to use the keyword **struct** for this purpose
- **struct** is short for **structure**

```
struct new_type {  
    member variable 1;  
    ...  
    member variable n;  
};
```

- **struct new_type** becomes your new user-defined data-type
- **member(s)** can be any existing data-types or user-defined types, such as, int, float, int*, char[10], struct another_type, etc.

Structures

Example:
representing a complex number
 $n = x + i y$

```
struct complex{  
    float x;  
    float y;  
};
```

```
struct complex n;  
n.x = 1.0;  
n.y = 2.0;  
// This^ can represent the complex number  
1.0 + i 2.0 as printed below  
printf("%f + i %f", n.x, n.y);
```

- `struct complex n1={1,2}, n2={2,3}, n3;`
 - Declare and initialize similar to any type
- `n3 = n2;` //copies the value of n2 into n3
- Normal operations **does not work (why?)**, such as
 - `n1+n2, n1-n2`
 - `n1 == n2`

To achieve these, you need to write your own functions

```
struct complex add (struct complex num1, struct complex num2) {  
    struct complex sum;  
    sum.x = num1.x + num2.x;  
    sum.y = num1.y + num2.y;  
    return sum;  
}  
n3 = add (n1, n2); //function call for addition
```

Structures (contd.)

- Normal operations **does not work**
 - $n1+n2$, $n1-n2$
 - $n1 == n2$
- You need to write your own functions and define your own operations
 - Example code for addition of two complex numbers is given \Rightarrow
 - Similarly you can write your own subtraction, multiplication, equality, conjugate, etc.

add is a function that takes two **complex** numbers as input and returns their **complex** sum as output

struct complex

```
add (struct complex num1, struct complex num2)
{
    struct complex sum;
    sum.x = num1.x + num2.x;
    sum.y = num1.y + num2.y;
    return sum;
}
```

```
n3 = add (n1, n2); //function call for addition
```

Renaming Datatypes

You can choose to **rename** (*create an alias*) for *any datatype* using a keyword called **typedef**

-- it is particularly convenient for structures

```
typedef struct complex Q;
```

Another way of writing typedef

```
typedef struct complex{  
    float x;  
    float y;  
}Q;
```

Example: Then, we could write code as follows

You can declare variables as follows: **Q** n1, n2;

Use in functions:

```
Q add (Q n1, Q n2)    {<<function definition>>}
```

Size of structure

Size of a structure variable is (ideally) the sum of the sizes of all its member's sizes, so
 $\text{sizeof}(Q) = \text{sizeof}(\text{float}) + \text{sizeof}(\text{float})$

But this is not always the case in practice:

structure padding -- extra memory added for convenience

`#pragma pack(1)` -- exact size memory as shown above

Consider the following structure

```
typedef struct a_type{  
    char x;  
    int y;  
}atype;
```

`sizeof (atype)` theoretically should be
 $\text{sizeof}(\text{char}) + \text{sizeof}(\text{int}) = 1 + 4 = 5$

But if you run the code, it gives 8 and not 5 due to structure padding (adds extra memory to char type)

But if you use `#pragma pack(1)` at the start of your program, then you can force the size to be 5

Structures and pointers

- Since structures are just another datatype - it is possible to create pointers of it's type
- `struct complex *ptr;` \Rightarrow is able to contain the address of structure variable
 - We could also write `Q *ptr;`
 \Rightarrow since we renamed it as `Q`
- So, `sizeof(ptr)` \Rightarrow ?

Accessing the members using pointers variables

```
Q *ptr; Q v = {10, 20};  
ptr = &v;
```

- `*ptr.real` \Rightarrow will not work
- `(*ptr).real` \Rightarrow will work

Alternatively the arrow operator (`->`) can be used to access members
`printf ("%f", ptr->real);`

Structures examples

Store student record with name, roll number, height, weight, DoB, DoJ

- How do you store information about 100 students?
- What happens if one or more student joins later on?
- What happens if you do not know the number of students beforehand?

```
// A possible implementation
typedef struct _student_info{
    char *name;
    char DoB[10], DoJ[10];
    int roll_no;
    float height, weight;
}student;
```

```
// A single student info
student stud1;
// 100 students info
student stud_arr[100];
```


Array and Structure

Since structures are just another datatype - *it is possible to create an array for the same*

Q `arr[5];` \Rightarrow equivalent to 5 **Q** variables

- Variables are accessed using **indexes** e.g. `arr[1]`, `arr[3]`, etc.
- Can also be accessed using **pointer arithmetic** \leftarrow *remember this?*

- `arr[i].x`, `arr[i].y` \leftarrow to access member variables
- `arr[i]` is the same as `*(arr + i)`
- i.e. `arr + i` is a pointer to `arr[i]`
- So, `(arr+i)->x` will also work

– okay, but how to create array when size is not known beforehand?

Dynamic Memory allocation (DMA)

- This is another way to allocate memory for variables
- It can allocate memory to a variable during the runtime of the program
 - So, you can read/scan the number of elements from the user
 - Then allocate necessary memory
- It works for allocating memory for
 - A single variable of any type
 - An array of any type

- We need a new include library **stdlib.h**
- We will use two functions from this library
 - malloc - **memory allocator**
 - free - frees some allocated memory

Prototype: *void* malloc (int size)*

- It allocates a memory space of the given **size**
- returns an address of the memory, i.e., a pointer but without any specific type
 - Hence a **void***
- You can **typecast** the pointer to your need

DMA (contd.)

To create a int variable using malloc, declare a int pointer variable

```
int *ptr;
```

Allocate memory using malloc (two ways)

```
ptr = (int*) malloc (sizeof(int)); // explicit typecast
```

```
ptr = malloc (sizeof(int)); // implicit typecast
```

Access the values using *ptr

```
*ptr = 10;
```

```
printf ("%d", *ptr); // → prints 10
```

Caution: if you try to access *ptr before allocating memory, the behaviour is undefined

For the structure Q, we can do the same as follows

```
Q *ptr;
```

```
ptr = (Q*) malloc (sizeof(Q));
```

Access: ptr->x, ptr->y

Array and DMA

- To create an array using DMA
- We need to specify the total memory size (in bytes) required for the array

e.g., to get an integer array of size 10, we can write the following code

```
int *arr;  
arr = (int*) malloc (sizeof(int) * 10);
```

Access as `arr[i]` or `*(arr+i)`

If you need to **take the size from the user**, you can do the following:

```
int n;    int *ptr;  
scanf ("%d", &n);  
ptr = (int*) malloc (sizeof(int) * n);
```

To release an allocated memory, you can write

```
free (ptr);
```

- Make sure the ptr is a valid one
- Otherwise, it may result in error

Adding an element in array

- Array has a fixed size
 - Be it allocated using DMA or statically
- Assume you have an array of 10 elements
 - You have inserted 5 elements from 0 to 4 indexes, then you want to insert another element in position 2
 - You have already inserted 10 elements, then you want to add another element

A better solution for such issues:

Linked list

- A clever solution using structures, DMA and pointers
- It requires more space than an array to store the same amount of data

It's a beautiful testimony to the power of C language

- *If time permits, we will talk about it at the end of this course*

Storage issues

- Single variable
 - Can only store a value
- Array of variables
 - Can store multiple values, but size allocation needs to be known first
- Array using DMA - can be allocated later, based on requirements
 - But insertion, deletion, resizing is still an issue
- Linked list is used to alleviate such problems
 - However, it uses more memory compared to arrays to store the same information

← All of these solution works only until program is running, once it is closed all data are lost.

- The solution to this problem is usage of **persistent storage** (you know these as pen drive, ssd, hard disk, etc.)
- But how do you write in such devices?

— We create files.

File

- Stored as sequence of bytes, logically contiguous
 - May not be physically contiguous on disk, but you don't need to worry about that
- Two types of files
 - Text - can only contain ASCII characters
 - Binary - can contain non-ASCII characters
 - Example: image, video, executable, audio, etc.
- Basic operations on file (stdio.h)
 - Open
 - Read
 - Write
 - Close
- A file needs to be open before you can do read or write operations
- Once the works are done on file you need to close the file
- In case, close is not done, some/all contents of the file may be lost

File (contd.)

- **FILE*** is a datatype used to represent a pointer to a file
- To open a file we use a function called **fopen**
 - It takes two parameters
 - Name of the file
 - Mode in which it is to be opened
 - It returns a pointer to the file if the file is opened successfully, otherwise it returns NULL

Example of a file creation for writing

```
FILE *fp;
char filename[] = "a_file.dat"
fp = fopen (filename, "w");
if (fp == NULL)
{
    printf ("unable to create file");
    /* DO SOMETHING */
}
/* WRITE SOMETHING IN FILE */
fclose (fp);
```


File (contd.)

Modes of opening a file

- “r” – Opens a file for reading
 - Error if the file does not already exist
 - “r+” allows write also
 - “w” – Opens a file for writing
 - If file does not already exist, it creates a new file
 - If file already exists, all the previous contents of the file will be overwritten
 - “w+” allows read also
 - “a” – Opens a file for appending (write at the end of the file)
 - “a+” allows read also
- When error occurs, e.g. file failed to open, the rest of your program may not work properly
 - In such case, you may want to exit the program on emergency basis
 - The function **exit()** from `stdlib.h` allows you to do so
 - It can be called from anywhere in the c program and it will terminate the program at once

File (contd.)

```
FILE *fp;
char filename[] = "a_file.dat"
fp = fopen (filename, "w");
if (fp == NULL)
{
    printf("unable to create file");
    /* DO SOMETHING */
    exit(-1);
}
/* WRITE SOMETHING IN FILE */
fclose (fp);
```

- You can pass any integer in the exit function
- This value will be returned as the output of the program
 - Recall that a c function is a collection of functions and functions must return something
 - A negative value (by convention) is treated as some error has happened

Next Class...

- Python preliminaries