Algorithms and Data Structures 2

2. Overview of C

Arrays

- Homogenous, static, memory based data structure.
- The ***dimension*** of an array is its size.
- Position of an element in an array is a ***subscript*** or an ***index.***
- Position of first element is in position 0.
- To declare an array, just use square brackets with a dimension as part of a regular declaration
  - e.g. `int num_array[10];`

Arrays

- An array is a named collection of values of the same type.
- Array declaration in C is:

```
    type arrayname [ array size ];
```

- e.g. an array of 10 integers

```
    int array1[10];
```

- Arrays can be initialised in their declaration:

```
    int array[6] = {1,2,3,4,5,6};

    int array[] = {1,2,3};
```

will set up `array[3]`

Array Declarations

- *type array_name(dimension)*

  e.g. float array_1[10];

  char name[100]; /* string */

  long array_2[50];

- The dimension of an array is often associated with a preprocessor directive

  #define DIRECTIVE_NAME Value

  e.g. #define SIZE 20

Arrays

- Arrays in C are indexed from 0.
- To access elements of array1 use
  `array1[0],array[1],array1[2], ...`
  `...array1[8],array1[9]`

- There is no runtime array bound checking in C

- This means that a program will not terminate if it attempts to access **`array1[10]`**

```
/*program to read in a list of MAX numbers and to write
  them out in reverse order*/

  #include <stdio.h>
  #define MAX 10
  int main()
  {
      int table[MAX];  /* declare array          */

      int index=0;               /* index for the array  */
```

```
     do
    {  printf("Enter an integer : ");
        scanf("%d",&table[index]);
        index++;
    }while (index<MAX);

    /* write out numbers in reverse */
    for (index=MAX-1; index>=0; index--)
                printf("%d\n",table[index]);

    return 0;
  }
```

Using a Break statement with scanf validation

```
int num;
 for (index=0; index<MAX; index++)
{
   if (scanf("%d",&num)!=1)
    /* input invalid integer */
   {
      printf("Input error after %d values\n",index);
       break;     /* exit the loop  */
    }//end if
     table[index]=num;
}//end for
```

- If scanf !==1 the break statement is executed which terminates the FOR statement.

Break and Continue

- Example of code to illustrate the continue statement

```
/* code to print the positive elements in an array */

for (index=0; index<MAX; index++)
{
if (table[index]<0)    /*skip negative elements */
        continue;
    printf("%d,%d\n", index, table[index]);
}
```

- After the continue statement , the rest of the loop statements are ignored and execution immediately resumes at the top of the loop.

- How would this program change if the number of numbers to be read in was not known?
  - User has to indicate when finished input
  - User may attempt to overfill the array
  - Array may not be filled completely

```
/*program to read in a list of numbers and to write them out in reverse
   order*/

   #include <stdio.h>
   #define MAX 10     /* size of array     */
 int main()  {
    int table[MAX];      /* declare array    */
    int num;             /* to hold input    */
    int i,index=0;      /* index for the array */
    int signal;   /* return code for scanf – used for validation*/
    printf("Enter an integer : ");
    signal=scanf("%d",&num); /* read first input and validate data type*/

  while (index < MAX) {
    if (signal != 1) {              /* invalid input */
        printf("Invalid input after %d values\n",index);
         break;
      } //end if
      else if (index>MAX-1) {      /* array full */
          printf("No more room after %d values\n",index);
          break;
      } //end else
       else                             /* valid entry */
         table[index++] = num;


          printf("Enter an integer : ");
         signal=scanf("%d",&num);       /* read next input */
    } //end while

/* write out numbers in reverse */
    for (i=index-1; i>=0; i--)
         printf("%d\n",table[i]);
} //end program
```

Two-Dimensional Arrays

- Consider setting up a table to hold the marks of students in certain modules.
- A 2D array is a data structure that logically looks like a table (with rows and columns). In reality it is an array where every element is another array.
- Table may look like this:

		Module				
		0	1	2	3	… 7
Student	0	52	37	48	59	…
	1	38	40	42	49	...
	2	79	63	65	59	...
	3	53	32	35	46	...
	…	…..				
	99					

- Use a 2-d array to represent the table

```
    #define STUDENTS    100
    #define MODULES     8


int exam_marks[STUDENTS][MODULES];
```

- The array exam_marks is indexed using the student number and the subject number

- Remember arrays in C are indexed from 0

```
    grade = exam_marks[i][j];
```

accesses the marks for student **`i+1`** in subject **`j+1`**

- Code to print out the values in the marks table:

```
for (pupil=0; pupil<STUDENTS; pupil++)
{
      printf("\nGrades for student %d", pupil+1);

      for (course=0; course<MODULES; course++)
          printf("\nSubject no %d – mark %d",
               course+1, exam_marks[pupil][course]);

}
```

Strings

- A string is any sequence of characters enclosed in double quotation marks. In memory, a string is terminated by the null character ‘\0’.

- Escape Sequences

\n	new line
\t	tab
\r	carriage return
\’	single quote
\”	double quote
\\	display \

Strings and Arrays

- A string is an array of type *char*.
- e.g. char name[20];

  char greetings[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’,’\0’}

  char greetings[] = “Hello”;

- **String Input**
  - scanf(“%s”, name); /* *doesn’t allow white spaces in string* */
  - gets(name); /* *allows white spaces* */
  - printf(“%s”,name);
  - puts(name);

Assigning a String to a Pointer

- A string is an array of characters
- Name of an array is a pointer to the first character of a string
  - therefore a string name is also a pointer to the first character in the string.

String Functions

- To access standard string functions, need to include the preprocessor directive
- #include <string.h>

String Functions (contd..)

- strlen() - returns the length of a string
- strcpy(str1, str2) - copies the contents of string str2 to string str1
- strcat(str1, str2) - append str1 and str2
- strcmp(str1, str2) - compare strings. 0 is returned is strings are equal, negative number is returned if str1 < str2, otherwise a positive number is returned.
- strchr(str, ch) - returns a pointer (or NULL) to the first occurrence of character ch in string str.
- strncat(str1, str2, n) - appends the first n characters of str2 to str1.
- strncmp(str1, str2, n) - compare the first n characters of str2 and str1.
- strrchr(str, ch) - look for last occurrence of character in string.
- strstr(str1, str2) - look for first occurrence of str2 in str1.

Arrays of Strings

- String is an array of characters

- Array is a pointer to its first element

- Then array of strings is
    1. 2D array of characters
    2. an array of pointers to characters.
    3. a ID array of Strings

  e.g. Display the months in reverse character order

  char months[12][4] = {“jan”, “feb”, etc };

  (OR char *months [12] = {“jan”, “feb”, etc })

Arrays of Strings (contd.)

Read 5 character strings from the keyboard and displays the longest:

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

int main()
{
    char strings[5][81];
    int index, index_of_longest;
    int longest_len = 0;

    for (index = 0; index < 5; index++)
    {
       printf("\nEnter string %d: ", index + 1);
       gets(strings[index]);
    }
for (index = 0; index < 5; index++)
        {
          if (strlen(strings[index]) > longest_len)
          {
            longest_len = strlen(strings[index]);
            index_of_longest = index;
          }
        }
        printf("Longest is %s", strings[index_of_longest]);
}
```

Pointers

- Every variable declared in C is stored in a memory location that has a unique address.

- We have already seen the specification of memory addresses via the *scanf* command

  e.g.
  ```
  int num;
  scanf("%d",&num);
  printf("num has value %d and is stored at address %u\n",
    num, &num);
  ```

- A ***Pointer Variable*** is a variable that stores the memory address of another variable. Pointers are declared using an asterisk.

  e.g
  ```
  int *num_pointer;
  ```

Pointers (contd.)

- The unary or monadic operator & gives the ``address of a variable''.
- The indirection or deference operator * gives the ``contents of an object pointed to by a pointer''. ***Note that * is also used to declare a pointer.***
- NOTE: We must associate a pointer to a particular type: You can't assign the address of a short int to a long int, for instance.

IMPORTANT: When a pointer is declared it does not point to anything. You must set it to point somewhere before you use it.

```
 int *ptr;
ptr = 100;
```

will generate an error (program crash!!).

The correct use is:

```
 int *ip;
 int num;

ip = &num;
num = 100;
```

Pointers (contd.)

- NOTE: A pointer to any variable type is an address in memory -- which is an integer address. A pointer is definitely NOT an integer.

- The reason we associate a pointer to a data type is to know the size of the data item (i.e. how many bytes the data is stored in).

- When we increment a pointer we increase the pointer by one ``block'' memory.

Pointers are commonly in the implementation of i) linked data structures and ii) passing parameters

Pointers and Arrays

- In C, pointers and arrays are very closely related. ***The name of an array is equivalent to the address of the first element.*** So the name of an array acts as a pointer to the first element of the array.

- So if an array *exam_results[10]* is declared:
  - *exam_results* is the address of the first element
  - *exam_results + 1* is the address of the second element

- The * operator can be used to access the elements of an array.

- Name of an array is a ***constant pointer.*** So expressions such as *exam_results++ is invalid.*

Functions

- Like Java, C programs also subscribe to the notion of ***modularity*** **-** programs can be written as a series of functions.
- Before a function can be used, it must be declared as a ***function prototype.***
- Functions can return single values (int, char, float, etc.) or nothing (void).
- Values can be passed into a function as an argument ***(formal vs actual arguments*****).**

Sample Program that demonstrates the use of functions.

```
#include <stdio.h>

//Function Prototypes
int square(int);
void endmessage();

int main()
{
    int i = 5;
   printf("square of %d is %d\n",i, square(i));
    endmessage();
}
```

```
int square (int num) {
    num = num +10;
   return (num * num)
}

void endemessage() {
   printf("End of program\n");
}
```

Passing arguments/parameters into C functions

- Two methods:
- 1. ***Passing arguments by value***
- Send a copy of the argument to the function. Therefore any changes to argument in the function are not permanent.

- 2. ***Passing arguments by reference***
- Pass address of argument to the function. Therefore any changes are permanent.

- **Using Functions and Passing parameters by value**

```
#include <stdio.h>

/* Function Prototypes */
int square(int);
void endmessage();

int main()
{
    int i = 5;
   printf("square of %d is %d\n",i, square(i));
  endmessage();
}
```

```
int square (int num)
{
   return (num * num)
}

void endemessage()
{
   printf(“End of program\n”);
}
```

•Passing a parameter by reference - Count number of vowels in a string

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

int num_vowels(char *);

int main()
{
  char name[10]; int num=0;
printf("Enter name: ");
gets(name);
num = num_vowels(name);
printf("Number of vowels in %s = %d\n",name, num);
}
```

Formal parameter is a char pointer so the actual parameter must be the address of a char.

```
int num_vowels(char *name)
{   int index; int num = 0;
     for (index=0;index < strlen(name); index++)
      switch(name[index])
        { case 'a','e','i','o','u': num++; break;
        }
     return num;
}
```

Changing the values of arguments

```
#include <stdio.h>

void swap(float *, float *);

int main()
{
          float num1, num2;

    printf("Please enter 2 numbers: ");
    scanf("%f%f", &num1, &num2);

    if (num1 > num2)
      swap(&num1, &num2);
    printf("The numbers in order are %5.1f %5.1f\n",
                    num1, num2);
}
```

```
void swap(float *ptr1, float *ptr2)
{
        float temp;

        temp = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 = temp;
}
```

Passing 1D arrays as Arguments

```
#include <stdio.h>

int find_largest(int *, int);

int main()
{
int values[10] = {22, 4, 75, 3, 23, 6, 2, 5, 17, 19};
int largest;

largest = find_largest(values, 10);
printf("The largest value %d\n", largest);
}
```

```
int find_largest (int *array, int size)
{
int index, largest = 0;
for (index = 0; index < size; index++)
  if  (array[index] > largest) // OR *(array+index) >
largest
    largest = array[index];
return(largest);
}
```

Structures

- An array is an homogenous data structure
- To combine data items of different data types into a data structure - use a ***structure*** or a ***record***.
- To create a structure
  - define a structure template (note that this is only a definition and does not allocation memory)
  - declare a variable to be of the structure type.

- Fields of a structure can be accessed by using the ***member selection (dot) operator***.

Structures

- A structure is a collection of one or more variables, possibly of different types, grouped together under one name for convenience.
- e.g. a point has an x and y coordinate, both integers

```
    struct point {
      int x;
      int y;
    };
```

- `point` is called the structure tag, which is optional, but should always be included

Structures

- `struct {...}` defines a type
  - the type is `struct point`
- Variables of the type defined by the structure can be declared as follows:

```
          struct point a, b;
```

- or variables can be declared with the definition of the type as follows:

```
          struct point {
                int x;
                int y;
    } a,b;
```

Structures

- Structure members can be variables of
  - the basic data types (char, float, int, pointer, etc…)
  - arrays or other structures

- Structure variables can be initialised in their declaration.

  e.g. `struct point a = {2,5};`

Accessing Structure members

- There are two ways to access the members of a structure variable
  - the structure member operator
  - the structure pointer operator

Structure Member Operator

- To access the members of a structure variable directly use the **structure member operator** - (`.`)
  - also called the dot operator
- For example - to access the coordinates of a point `a`

- `a` is declared as a structure variable

```
struct point a;
```

- To print the coordinates of the point `a`

```
printf ("%d,%d", a.x, a.y);
```

Sample Program that creates and displays single student structure.

```
#include <stdio.h>

//Global Structure
struct student {
	char id[10];
	char courseCode[10];
	int year;
};
```

```
void main ()
{
    struct student aStudent;

    printf("Enter student id: ");
    scanf("%s", aStudent.id);
    printf("Enter course code : ");
    fflush(stdin);
    gets(aStudent.courseCode);
    printf("Enter year (1, 2, 3 or 4): ");
    scanf("%d", &aStudent.year);

    printf("Student details are : %s, %s %d\n",
                aStudent.id, aStudent.courseCode,
aStudent.year);

}
```

Structure Pointer Operator

- The members can also be accessed via pointers to the structure using the **structure pointer operator** - `->`
  - also called the arrow operator

- `aptr` is declared as a pointer to a structure of type `point`

```
    struct point *aptr;
```

- The pointer is assigned to point to a structure variable `a`

```
    aptr = &a;
```

- To print the coordinates of the point `a` using the pointer

```
printf ("%d,%d", aptr->x,aptr->y);
```

Structure Pointer Operator

- **`aptr->x`** is equivalent to **`(*aptr).x`**

- Note the brackets - the dot operator has a higher precedence than the contents of operator . Best keep it simple do not mix notations.

*

Structures

- Structures of structures are allowed
- A structure of structures can represented by a structure as follows

```
struct triangle{
     struct point pt1;
     struct point pt2;
     struct point pt3;
};
```

Structures

- Example - to set the coordinates of the points of the triangle `t1` to coordinates with values (2,3) (1,5) (0,4)

- Declare `t1` as a variable of type `struct triangle`

```
    struct triangle t1;
```

- Assign the members of the structure the appropriate values

```
          t1.pt1.x = 2;
          t1.pt1.y = 3;
          t1.pt2.x = 1;
    etc....
```

Array of Structures

- e.g.

```
struct bank_customer customers[10];
```

To access individual elements:

Give array name, array subscript, field

e.g. `customers[0].balance = 0;`

```
struct bank_customer {
        long accountNumber;
        char name[20];
        char accountType[10];
        int balance;
};
```

Passing Structures to Functions

- Structures can be passed to functions

  by passing structure members,

  by passing the entire structure or

  by passing a pointer to the structure

- Structure members or entire structures are passed ‘call by value’ (a copy)
  - the values cannot be changed in the calling function

- Structure members can be changed when a pointer to the structure is passed to a function
  - simulated ‘call by reference’

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

/* function prototypes */
int inputOrder (struct order *);
void displayOrder(struct order );

/* structure template*/
struct order
   {
         int orderNum;
         char productName[20];
         int quanity;
         float price;
   };
```

```
/* type definition */
typedef struct order ORDER;


/* OR if I want to declare a global variable to be of
type order
 struct order
        {
                int orderNum;
                char productName[20];
                int quanity;
                float price;
        }order1;
*/
```

```
int main()
{
    int theOrderNum;

    struct order order1;
    /* OR
    ORDER order1;
    */

    theOrderNum = inputOrder(&order1);
    printf("the order number is %d\n", theOrderNum);
    displayOrder(order1);
}
```

for this program, I'm assuming that `order1` is not global but instead is local to the main function.

```
int inputOrder (struct order *anOrder)
{
        printf("Enter order num : ");
        scanf("%d", &anOrder->orderNum);
        printf ("in inputOrder %d\n", (*anOrder).orderNum);
        return (*anOrder).orderNum;
}

void displayOrder(struct order anOrder)
{
    printf("The order num is : %d\n", anOrder.orderNum);
}
```

Function Example - enter 1 exam record and find average result for that exam

```
#include <stdio.h>
#define SIZE 5

/* Structure template */
struct exam
{
   char examName[20];
   char course[10];
   float results[SIZE];
};

/* Type definition */
typedef struct exam EXAM;
```

```
/* Function Prototype */
void inputExam(EXAM *);
float getAverage(EXAM);
```

Example - Arrays, Structures, Functions

Write a program that can store up to 100 bank customer records. Ask the user how many records to store and then input that amount of records. Display all users that are in dept to the bank and calculate the total amount of money held in the bank.

Input : 100 bank records - i.e. Array of bank records (structures)

Num_Customers (integer)

Ouput : Individual records from array,

Total (integer)

Steps :

1. Ask user to input *Num_Customers*
2. Input that number of customer records and store in array
3. Search though array and display those records with a negative balance
4. Search though array and get the total of the balances.

```
#include <stdio.h>
#include <string.h>
#define SIZE 100

struct bank_customer
   {long account_num;
    char name[3];
    float balance;
        };

int getBankSize();
void inputCustomers(int, struct bank_customer *);
void displayCustomer(int, struct bank_customer *);
int getTotal(int, struct bank_customer *);
int main()
{
           struct bank_customer bank[SIZE];
          float total=0.0;
          int numCustomers;

numCustomers=getBankSize();
inputCustomers(numCustomers, bank);
displayCustomer(numCustomers, bank);
total=getTotal(numCustomers, bank);
printf("Total balance in bank = %5.2\n", total);
} //end main
```

```
int getBankSize()
{
   int sizeOfBank;
   do
     {printf(“Enter the number of bank customers (<100)”);
        scanf(“%d”,&sizeOfBank);
   }while (sizeOfBank < 0 || sizeOfBank > SIZE);

     return (sizeOfBank);
} //end getBankSize
```

```
void inputCustomers(int num, struct
          bank_customer *bank)

{

    int index;

    for (index=0; index<num; index++)

    {printf("Enter account number for customer %d: \n",
index+1);

      scanf("%ld",&bank[index].account_num);

          etc.

   } //end for

} //end inputCustomers
```

```
void displayCustomer(int num,
                        struct bank_customer *bank)
{
  int i;
  for (i=0; i < num; i++)
   {
         if (bank[i].balance < 0)
         {printf("Account Num : %ld\n",
         bank[i].account_num);
         printf("Name : %s\n", bank[i].name);
         printf("Balance : %5.2f\n", bank[i].balance);
         } //end if
   } //end for
} // end displayCustomer
```

```
int getTotal(int num, struct bank_customer *bank)
{
        int index; float total = 0.0;
        for (index=0; index<num; index++)
                total+=bank[index].balance;
        return  total;
} //end getTotal
```

Implmenting a Linked List in C

- C allows programmers to reserve or unreserve memory while a program is running. This means that nodes in linked lists can be created and variables deleted during run-time.

- To do this we need the preprocessor directive:

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

  and the functions :

```
        malloc();
        free();
```

- Need to dynamically create each node using *malloc*
- Need to use pointers to connect each node together

```
struct LinearNode {
  struct data *element;
  struct LinearNode *next;
}
```

```
struct LinearNode *front = NULL;  //pointer to first
element in list
```

- We use `malloc()` and `free()` to create and destroy nodes as we need them.

Enumerated Data Types

- An enumerated data type is used when to create a new data type that can have specific values.

```
e.g. /* Prototype definition */
   enum TypeOfResult {honours, pass, fail}

/* Variable Declaration */
enum TypeOfResult ExamResult;
```

[Aside: The system stores the values as integer constants starting at 0, using values makes the program more readable]

•Variable can then be assigned to any of the values

```
    i.e. ExamResult = honours;


    if (ExamResult == honours)
```

•Note that an enumerated value cannot be read directly from the keyboard.

Including a Structure Template from a File

- To allow more than one program to use a particular structure template, need to create a header file that contains the structure and then include the structure as a preprocessor directive.

  e.g.
  ```
  #include <stdio.h>
  #include <bank.h>
  ```