Unit 1. Concept and Definition of Data Structures

a. Information and its meaning
b. Array in C
c. The array as an ADT
d. One dimensional array
e. Two dimensional array
f. Multi-dimensional array
g. Structure
h. Union
i. Pointer

By: Pravakar Ghimire

Background

- A computer is a machine that manipulates information.
- The study of computer science includes the study of how information is organized in a computer, how it can be manipulated, and how it can be utilized.
- Thus, it is important for a student of computer science to understand the concepts of information organization and manipulation in order to continue study of the field.

Information and its meaning

- The definition of information cannot be given precisely however it can be measured.
- The basic unit of information is a **bit** (contraction of the word "binary digit") and the quantity of information can be interpreted in terms of bits.
- Non-negative integers are interpreted in terms of bits using the ***binary number system*** or ***binary coded decimal*** while negative integers are interpreted using the ***ones complement notation*** or ***twos complement notation***. Also real numbers are interpreted in terms of bits using the ***floating point notation***. Similarly characters are interpreted in terms of a group of 8-bits called a ***byte*** using ***ASCII code*** and the character string as a sequence of characters.
- Thus we can see that information itself has no meaning and any meaning can be assigned to a particular bit pattern, as long as it is done consistently. It is the interpretation of a bit pattern that gives the information a meaning. For e.g., the bit string 00100110 can be interpreted as the number 38 in binary number system or the number 26 in binary coded decimal or the character '&'.
- A method of interpreting a bit pattern is called a ***data type***.

Information and its meaning...

- Every computer has a set of "native" data types. The data types offer a method of interpreting information as memory contents in a computer (The memory is organized into bytes or memory addresses for this). Also the hardware of a computer is wired and constructed with a mechanism for manipulating bit patterns consistent with the objects they represent.
- The declarations that the programmer specifies play a key role in how the contents of computer memory are to be interpreted by the program. The declarations consequently specify what the operation symbols specify that are subsequently used.

Information and its meaning...

- However, when we divorce the "***data type***" from the hardware capabilities of a computer, we can consider a limitless number of data types.
- A data type is then an abstract concept and defined by a set of logical properties. Once such an abstract data type is defined and the legal operations involving that type are specified, we may implement that data type. The implementation may be a ***hardware implementation***, in which the circuitry necessary to perform the required operations is designed and constructed as part of a computer; or it may be a ***software implementation***, in which a program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations.
- Thus, a software implementation includes a specification of how an object of the new data type is represented by objects of previously existing data types, as well as a specification of how such an object is manipulated in conformance with the operations defined for it.

Information and its meaning...

- ***Abstract Data Type***: A useful tool for specifying the logical properties of a data type is the ***abstract data type*** or ***ADT***. An ***ADT*** is just a mathematical concept and is independent of how the data type is to be implemented.
- For example: Let us consider the ADT ***RATIONAL***, which corresponds to the mathematical concept of a rational number. A rational number is a number that can be expressed as the quotient of two integers. The operations on rational numbers that we define are the creation of a rational number from two integers, addition, multiplication, and testing for equality.

Information and its meaning...

- An ADT consists of two parts: **a value definition** and **an operator definition**.
- The value definition defines the collection of values for the ADT and consists of two parts: **a definition clause** and **a condition clause**.
- For example: The value definition for the ADT RATIONAL states that a RATIONAL value consists of two integers (numerator and denominator), the second of which does not equal to zero.
- Note: Array notation (square brackets) is used to indicate the parts of an abstract data type.

Information and its meaning...

- The keywords ***abstract typedef*** introduce a value definition, and the keyword ***condition*** is used to specify any conditions on the newly defined type. In this definition, the condition specifies that the denominator may not be zero. The definition clause is required, but the condition clause may not be necessary for every ADT.

```
/* value definition */
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] != 0;
```

Information and its meaning...

- Immediately following the value definition comes the operator definition. Each operator is defined as an abstract function with three parts: a header, the optional preconditions, and the postconditions.
- For example: The operator definition of the ADT RATIONAL includes the operations of creation *(makerational)*, addition *(add)* and multiplication *(mult)*, as well as a test for equality *(equal)*.

/* operator definition */

abstract RATIONAL *makerational(a,b)*

int *a, b;*

precondition b!=0;

postcondition makerational[0] == a;

makerational[1] == b;

abstract RATIONAL *add(a,b)* *e.g p/q + r/d=(p*d) q*r)/(q*d)*

RATIONAL *a, b;*

postcondition add[1] == a[1] * b[1];

add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL *mult(a,b)*

RATIONAL *a, b;*

postcondition mult[0] == a[0] * b[0];

mult[1] == a[1] * b[1];

abstract *equal(a,b)*

RATIONAL *a, b;*

postcondition equal == (a[0] * b[1] == b[0] * a[1]);

Note:

- The header of the operator definition is the first two lines.
- The postcondition specifies what the operation does. In a postcondition, the name of the function is used to denote the the result of the operation. Thus add[0] or mult[0] represents the numerator of the result and add[1] or mult[1] represents the denominator of the result.

TU Exam Question (2066)

- Write a short note on ADT.

Simulation --- --- --- --- --- --- RATIONAL

```
void makerational(struct RATIONAL *, struct
    RATIONAL *);
void add(struct RATIONAL *, struct RATIONAL *);
void mult(struct RATIONAL *, struct RATIONAL *);
void equal(struct RATIONAL *, struct RATIONAL *);

struct RATIONAL
    {
    int numerator;
    int denominator;
    };
void main()
{
struct RATIONAL a,b;
clrscr();

printf("\n Enter numerator and denominator of first
    rational no:");
scanf("%d/%d", &a.numerator, &a.denominator);

printf("\n Enter numerator and denominator of second
    rational no:");
scanf("%d/%d",&b.numerator,&b.denominator);

makerational(&a,&b);

add(&a,&b);

mult(&a,&b);

equal(&a,&b);

getch();
}
```

```
void makerational(struct RATIONAL *p, struct
    RATIONAL *q)
{
    if(p->denominator==0)
    {
    printf("\n The denominator of first rational
    number cannot be zero.");
    printf("\n Enter other rational number:");
    scanf("%d/%d",&p->numerator,&p-
    >denominator);
    }
    if(q->denominator==0)
    {
    printf("\n The denominator of second rational
    number cannot be zero.");
    printf("\n Enter other rational number:");
    scanf("%d/%d",&q->numerator,&q-
    >denominator);
    }
printf("\n The first rational number is:%d/%d", p-
    >numerator, p->denominator);
printf("\n The second rational number is:%d/%d", q-
    >numerator, q->denominator);
printf("\n\n\t**********makerational
    complete**********");
}
```

```
void add(struct RATIONAL *p, struct RATIONAL *q)
{
struct RATIONAL r;
r.denominator = (p->denominator)*(q-
    >denominator);
r.numerator = (p->numerator)*(q-
    >denominator)+(p->denominator)*(q-
    >numerator);
printf("\n The addition of the two rational numbers
    is: %d/%d", r.numerator, r.denominator);
printf("\n\n\t**********addition
    complete**********");
}

void mult(struct RATIONAL *p, struct RATIONAL *q)
{
struct RATIONAL r;
r.denominator = (p->denominator)*(q-
    >denominator);
r.numerator = (p->numerator)*(q->numerator);
printf("\n The multiplication of the two rational
    numbers is: %d/%d", r.numerator,r.denominator);
printf("\n\n\t**********multiplication
    complete**********");
}
```


By: Pravakar Ghimire

```
void equal(struct RATIONAL *p, struct RATIONAL *q)
{
   if((p->numerator)*(q->denominator)==(p->denominator)*(q-
   >numerator))
         {
         printf("\n The two rational numbers are equal.");
         }
   else
         printf("\n The two rational numbers are unequal.");
printf("\n\t*********equality check complete**********");
}
```

Data Structure

- In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- There are various kinds of data structures such as STACKS, QUEUES, LINKED LISTS, TREES, etc. and these different kinds of data structures are suited to different kinds of applications. For example: The data structure ***STACK*** is used to keep the records of visited web pages history in web browsers.

Array in C

- An array is a group of related data items that share a common name.
- The individual data items are called elements of the array and all of them are of same data type.
- The individual elements are characterized by array name followed by one, two or more subscripts (or indices) enclosed in square brackets.

  E.g. int num[30];
  - This statement tells the compiler that *num* is an array of type *int* and can store 30 integers.
  - The individual elements of *num* are recognized by *num[0]*, *num[1]*,…, *num[29]*.
  - The integer value within square bracket (i.e. [ ]) is called *subscript* or *index* of the array.
  - *Index* of an array always starts from 0 and ends with one less than the size of the array.
- The most important property of an array is that its elements are stored in contiguous memory locations.

Array as an ADT

- We can represent an array as an ADT by providing a set of specifications with a set of operations on the array with no regard to its implementation.
- Let a function ***type(arg)***, returns the type of its argument, ***arg***. Note that such a function does not exist in C, since C cannot dynamically determine the type of a variable. However, in ADT we are not concerned with implementation, but rather with specification, so the use of such a function is permissible.
- Now, let ***ARRTYPE(ub, eltype)*** denotes the array ADT where the precise ADT is determined by the parameters ***ub*** and ***eltype*** where ***eltype*** is a type indicator.

Array as an ADT...

- We can now view any one-dimensional array as an entity of the type ***ARRTYPE***.

/*value definition*/

abstract typedef *<<eltype, ub>> ARRTYPE(ub, eltype);*

condition *type(ub)* == ***int***;

Array as an ADT...

```
/*operator definition*/
abstract eltype extract(a, i)
ARRTYPE(ub, eltype) a;
int i;
precondition 0 <=i <ub;
postcondition extract == a_i
```

```
abstract store(a, i, elt)
ARRTYPE(ub, eltype) a;
int i;
precondition 0 <=i < ub;
postcondition a[i] == elt;
```

Simulation --- --- --- Array as an ADT

```
#define ub 5
void extract(int a[ub], int i);
void store(int a[ub], int i, int elt);
void main()
{
int a[ub]={1,2,3,4,5};
int i;
int elt;
clrscr();
printf("\nWhich array element you want to extract(Oth means first):");
scanf("%d", &i);
extract(a, i);
printf("\nWhat value you want to store at what position (0th means first))?");
printf("\nValue:");
scanf("%d", &elt);
printf("\n Position:");
scanf("%d", &i);
store(a, i, elt);
getch();
}
```

```
void extract(int a[ub], int i)
{
if(i<0 || i>ub-1)
    printf("\n Out of Bound error");
else
    printf("\n The value extracted at %dth position is %d", i, a[i]);
}


void store(int a[ub], int i, int elt)
{

if(i<0 || i>ub-1)
    printf("\n Out of Bound error");

else
    a[i] = elt;
    printf("\n The value stored at %dth position is %d", i, a[i]);
}
```

One dimensional array

- A list of items can be given one variable name using only one subscript (or dimension or index) and such a variable is called a *single-subscripted variable* or a *one-dimensional array*.
- The value of the single subscript or index from 0 to n-1 refers to the individual array elements; where n is the size of the array.
- E.g. the declaration *int a[5];* is a 1-D array of integer data type with 5 elements: *a[0]*, *a[1]*, *a[2]*, *a[3]*and *a[4]*.
- *Note: Generally a single **for** loop is used for input and output in a one-dimensional array.*

One dimensional array...

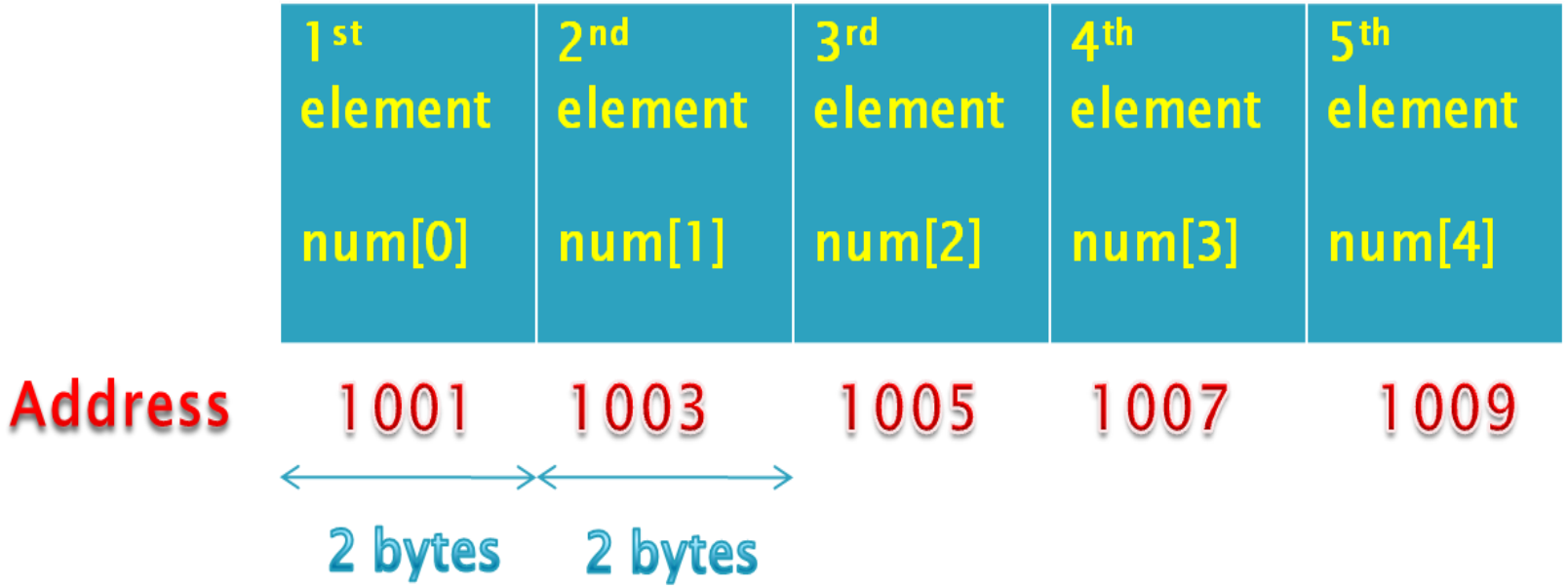

Fig: Memory Representation of a 1-D Array

How to declare array???

- Statically
- Two ways:

```
#define SIZE 100
void main()
{
int array[SIZE];
}
```

```
void main()
{
int array[100];
}
```

- Find out error here:

```
void main()
{
int n=10, i;
int marks[n];
clrscr();
for(i=0;i<n;i++)
        printf("%d\t", marks[i]);
getch();
}
```

Size of an array must be either symbolic constant or integer constant.

Two Dimensional Array

- The two dimensional array is also called matrix.
  - Note: One dimensional array is also called vector.
- An m*n two dimensional array can be thought as a table of values having m rows and n columns.
- The syntax for 2-D array declaration is:

  data_type array_name[row_size][col_size];
- E.g.

  int matrix[2][3];

  Here, matrix is a 2-D array that can contain 2*3=6 elements. This array matrix can represent a table having 2 rows and 3 columns from matrix[0][0] to matrix[1][2].

Two Dimensional Array...

	Column1	Column2	Column3
Row1	matrix[0][0]	matrix[0][1]	matrix[0][2]
Row2	matrix[1][0]	matrix[1][1]	matrix[1][2]

NOTE: ARRAY ELEMENTS ARE STORED ROW WISE IN MEMORY

Two Dimensional Array...

- Consider an array *marks* of size 4*3 with elements having values:

  int marks[4][3]={35,10,11,34,90,76,13,8,5,76,4,1};

- This array can be realized as a matrix having 4 rows and 3 columns as:

35	10	11
34	90	76
13	8	5
76	4	1

- To access a particular element of a 2-D array, we have to specify the array name, followed by two square brackets with row and column number inside it.
- Thus, marks[0][0] accesses 35, marks[1][1] accesses 90, marks[2][2] accesses 5 and so on.
- Note: Generally, a nested ***for*** loop is used for input and output in 2-D arrays.

Test

A 2-D array A[4][3] is stored row-wise in memory. The first element of the array is stored at location 80. Find the memory location of A[3][2] if each element of array requires 4 memory locations.

Multi Dimensional Array

- Multidimensional arrays are those having more than two dimensions and more than two pairs of square brackets are used to specify its dimensions.
- Example: A 3-D array requires three pairs of square brackets.
- Syntax for defining multidimensional array is:

  data_type array_name[dim1][dim2]…[dimN];

  Here, dim1, dim2,…,dimN are positive valued integer expressions that indicate the number of array elements associated with each subscript. Thus, total no. of elements=dim1*dim2*…*dimN

- E.g.

  int survey[3][5][12];

  Here, survey is a 3-D array that can contain 3*5*12=180 integer type data. This array survey may represent a survey data of rainfall during last three years (2009,2010,2011) from months Jan. to Dec. in five cities. Its individual elements are from survey[0][0][0] to survey[2][4][11].

- Note: Generally, more than two nested ***for*** loops are used for input and output in a multidimensional array.

TU Exam Question (2065)

- **What are the differences between two dimensional array and multidimensional array?**

Classwork

Pointers

- **The & operator**

  The operator & (called "address of" operator) immediately preceding a variable returns the base address of the variable associated with it in memory.

Pointers...

- "A pointer is a variable that contains an address which is a location of another variable in memory"
- The data type of the pointer variable and data type of another variable whose address the pointer variable holds should be same.
- Declaration Syntax: *data_type *pointer_name;*

  This declaration tells the compiler three things about the variable *pointer_name*:

  1. The asterisk (*) tells that the variable *pointer_name* is a pointer variable
  2. *pointer_name* needs a memory location
  3. *pointer_name* holds the address of type *data_type*
- Example:

  *int *p;*
- This signifies that *p* is a pointer variable and it can store the address of an integer variable (The address of float variable cannot be stored in it).

Pointers...

- Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement such as:

  pointer_name = &*variable_name;*

  which causes the pointer_name to point to variable_name.

- Example:

```
int *p, num;
p=&num;
```

- Here, *p* contains address of *num* (or *p* points to *num*). This is called *pointer initialization*.
- Note: Before a *pointer* is initialized, it should not be used.

Pointers...

- **The * (star or asterisk) operator**
  - When * is used with variable declaration before a variable's name, it becomes a pointer variable, not a normal variable. E.g. int *p;
  - When * is used in front of pointer variable, it indirectly references ***the value at that address*** stored in the pointer. In this case * is also called *indirection* or *dereference* operator.

- **What is NULL pointer???**
  - A *null* pointer is a special pointer value that points nowhere or nothing.
  - The predefined constant *NULL* in stdio.h is used to define *null* pointer.
  - Example:

```
#define NULL 0
void main()
{
int *ptr=NULL;
        if(ptr==NULL)
        printf("Change NULL to 1 and there will be warnings!!!");
getch();
}
```

Structure

- A structure is a collection of data items of different data types under a single name.
- The individual data items are called members of the structure.
- The syntax for structure definition is:

```
struct structure_name
        {
        data_type member_variable1;
        data_type member_variable2;
        ……..……;
        data_type member_variableN;
        };
```

- A structure definition creates a new data type that is used to declare structure variables. Once ***struct structure_name*** is declared as a new data type, then variables of that type can be declared as:

```
struct structure_name structure_variable;
```

- The member variables of a structure are accessed using the dot (.) operator.

Structure...

- The members of a structure variable can be initialized using the syntax:

  struct structure_name structure_variable={value1, value2, … , valueN};

  where, value1 is initialized to the first member, value2 to the second member and so on.

- **<u>Copying structure variables</u>**

  Two variables of the same structure type can be copied by using the assignment operator "=".

  E.g. ***structure_variable1= structure_variable2***

- Note: Two structure variables cannot be compared directly by using the comparison operators "==" and "!=". We may do so by comparing individual structure members.

Structure...

- **Array of Structure**

```
struct structure_name
        {
        data_type member_variable1;
        data_type member_variable2;
        ……..……;
        data_type member_variableN;
        };
```

- An array of structure can be declared as:

```
struct structure_name structure_variable[100];
```

Structure...

- **Pointer to Structure**

```
struct structure_name
        {
        data_type member_variable1;
        data_type member_variable2;
        ……..……;
        data_type member_variableN;
        };
```

- A structure type pointer variable can be declared as:

```
struct structure_name *structure_variable;
```

- However, this declaration for a pointer to structure does not allocate any memory for a structure but allocates only for a pointer, so that to access structure's members through pointer ***structure_variable***, we must allocate the memory using ***malloc()*** function.
- Now, individual structure members are accessed by using arrow operator -> as:

```
structure_variable->member_variable1
structure_variable->member_variable2
…          …          …          …
structure_variable->member_variableN
```

Passing whole structure to function

- Whole structure can be passed to a function by the syntax:

  function_name(structure_variable_name);

- The called function has the form:

```
return_type function_name(struct structure_name structure_variable_name)
        {
        ……;
        }
```

- Note: In this call, only a copy of the structure is passed to the function, so that any changes done to the structure members are not reflected in the original structure.

Passing structure pointer to function

- In this case, address of structure variable is passed as an actual argument to a function.
- The corresponding formal argument must be a structure type pointer variable.
- Note: Any changes made to the members in the called function are directly reflected in the calling function.
- Syntax in calling function:

function_name(&structure_variable_name);

- Syntax in called function:

```
return_type function_name(struct structure_name * structure_variable_name)
{
… … … … …;
}
```

Passing array of structure to function

- The name of the array of structure is passed by the calling function which is the base address of the array of structure.
- Thus, any changes made to the array of structure by the called function are directly reflected in the original structure.
- Syntax in calling function:

function_name(structure_variable_name);

- Syntax in called function:

```
return_type(struct structure_name structure_variable_name[])
{
… … … … …;
}
```

Union

- Unions are similar to structure in the sense that they are also used to group together data items of different data types.
- The distinction lies in the fact that all members within a union share the same memory space whereas each member within a structure is allocated a unique memory space.
- In case of union, the compiler allocates a memory space that is large enough to hold the largest variable type in the union.
- Since same memory space is shared by all the members of a union, only one variable can reside in the memory at a time, and when another variable is set in the memory, the previous variable is replaced by the new one.

Model Question (2008)

- **Explain the difference between structure and union.**

Classwork

Self-Referential Structures

- A self-referential structure contains a pointer member that points to a structure of the same structure type.
- For example, the definition

```
struct node
        {
        int data;
        struct node *nextPtr;
        };
```

defines a type, *struct node*.

- A structure of type *struct node*, has two members: an *int* member *data* and another pointer member *nextPtr*.
- The member *nextPtr* points to (i.e. holds address of) another structure of type *struct node* i.e. a structure of the same type as the one declared here and hence is the term "self-referential structure".

Self-Referential Structures...

- The member *nextPtr* is referred to as a link i.e. *nextPtr* can be used to "tie" a structure of type *struct node* to another structure of the same type.
- Self-referential structures can be linked together to form useful data structures such as lists, queues, stacks and trees.
- The following figure represents two self-referential structure objects linked together to form a list:

Self-Referential Structures...

- A NULL pointer is assigned to the *nextPtr* member of the last self referential structure to indicate that the *nextPtr* doesn't point to another structure.
- The NULL pointer indicates the end of a data structure just as the null pointer indicates the end of a string.
- Note: Not setting the *nextPtr* in the last node of a list to NULL can lead to runtime errors.

// An example of a linked list

```
#define NULL 0
struct node
    {
    int data;
    struct node *nextPtr;
    };
void main()
{
struct node node1,node2;
clrscr();
node1.data=100;
node1.nextPtr=&node2;
node2.data=200;
node2.nextPtr=NULL;
printf("\n");
printf("|%d|%u|-----> |%d|%u|",node1.data,node1.nextPtr,node2.data,node2.nextPtr);
printf("\n Address of node2=%u",&node2);
getch();
}
```