DATA STRUCTURES USING ‘C’

Lecture No.01

Data Structures

Goals of this Course

1. Reinforce the concept that costs and benefits exist for every data structure.

2. Learn the commonly used data structures.
   - These form a programmer's basic data structure “toolkit”.

3. Understand how to measure the cost of a data structure or program.
   - These techniques also allow you to judge the merits of new data structures that you or others might invent.

Need for Data Structures

- Data structures organize data $\Rightarrow$ more efficient programs.
- More powerful computers $\Rightarrow$ more complex applications.
- More complex applications demand more calculations.

Data Structures

- Prepares the students for (and is a prerequisite for) the more advanced material students will encounter in later courses.
- Cover well-known data structures such as dynamic arrays, linked lists, stacks, queues, tree and graphs.
- Implement data structures in C++

Data Structures

- Prepares the students for (and is a prerequisite for) the more advanced material students will encounter in later courses.
- Cover well-known data structures such as dynamic arrays, linked lists, stacks, queues, tree and graphs.
- Implement data structures in C++

Organizing Data

- Any organization for a collection of records that can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

- A solution is said to be *efficient* if it solves the problem within its *resource constraints*.
  - Space
  - Time

- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.

Arrays

- Elementary data structure that exists as built-in in most programming languages.

```
main( int argc, char** argv )
{
    int x[6];
    int j;
    for(j=0; j < 6; j++)
            x[j] = 2*j;
}
```

Arrays

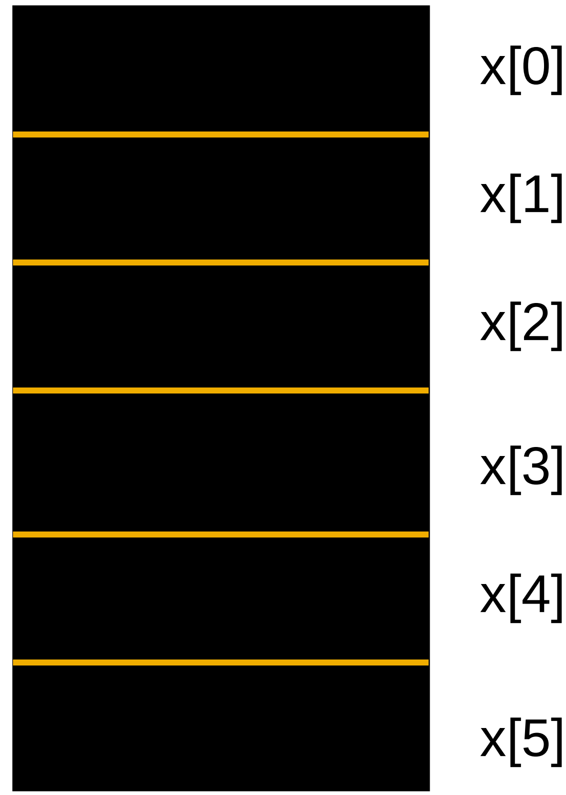
- Array declaration: int x[6];
- An array is collection of cells of the same type.
- The collection has the name ‘x’.
- The cells are numbered with consecutive integers.
- To access a cell, use the array name and an index:

  x[0], x[1], x[2], x[3], x[4], x[5]

Array Layout

Array cells are contiguous in computer memory

The memory can be thought of as an array



What is Array Name?

- ‘x’ is an array name but there is no variable x. ‘x’ is not an *lvalue*.
- For example, if we have the code

```
int a, b;
```

then we can write

```
b = 2;
a = b;
a = 5;
```

But we cannot write

```
2 = a;
```

What is Array Name?

- ‘x’ is an array name but there is no variable x. ‘x’ is not an *lvalue*.
- For example, if we have the code

```
int a, b;
```

then we can write

```
b = 2;
a = b;
a = 5;
```

But we cannot write

```
2 = a;
```

Array Name

- ‘x’ is not an lvalue

```
int x[6];
int n;

x[0] = 5;
x[1] = 2;

x = 3;                  // not allowed
x = a + b;              // not allowed
x = &n;                 // not allowed
```

Dynamic Arrays

- You would like to use an array data structure but you do not know the size of the array at compile time.
- You find out when the program executes that you need an integer array of size n=20.
- Allocate an array using the new operator:

```
int* y = new int[20];  // or int* y = new int[n]
y[0] = 10;
y[1] = 15;              // use is the same
```

Dynamic Arrays

- ‘y’ is a lvalue; it is a pointer that holds the address of 20 consecutive cells in memory.
- It can be assigned a value. The new operator returns as address that is stored in y.
- We can write:

```
y = &x[0];
y = x;          // x can appear on the right
                // y gets the address of the
                // first cell of the x array
```

Dynamic Arrays

- We must free the memory we got using the new operator once we are done with the *y* array.

```
delete[ ] y;
```

- We would not do this to the *x* array because we did not use new to create it.

The LIST Data Structure

- The List is among the most generic of data structures.

- Real life:

  a. shopping list,
  b. groceries list,
  c. list of people to invite to dinner
  d. List of presents to get

Lists

- A list is collection of items that are all of the same type (grocery items, integers, names)

- The items, or elements of the list, are stored in some particular order

- It is possible to insert new elements into various positions in the list and remove any element of the list

Lists

- List is a set of elements in a linear order. For example, data values a_1, a_2, a_3, a_4 can be arranged in a list:

  (a_3, a_1, a_2, a_4)

  In this list, a_3, is the first element, a_1 is the second element, and so on

- The order is important here; this is not just a random collection of elements, it is an *ordered* collection

List Operations

Useful operations

- createList(): create a new list (presumably empty)
- copy(): set one list to be a copy of another
- clear(); clear a list (remove all elments)
- insert(X, ?): Insert element X at a particular position in the list
- remove(?): Remove element at some position in the list
- get(?): Get element at a given position
- update(X, ?): replace the element at a given position with X
- find(X): determine if the element X is in the list
- length(): return the length of the list.

List Operations

- We need to decide what is meant by “particular position”; we have used “?” for this.

- There are two possibilities:

  1. Use the actual index of element: insert after element 3, get element number 6. This approach is taken by arrays
  2. Use a “current” marker or pointer to refer to a particular position in the list.

List Operations

- If we use the “current” marker, the following four methods would be useful:
  - **start()**: moves to “current” pointer to the very first element.
  - **tail()**: moves to “current” pointer to the very last element.
  - **next**(): move the current position forward one element
  - **back**(): move the current position backward one element