12 Pointers

"A lazy person will end up poor."

Pointers are a gift to C programmers. One of the important uses of pointers is the dynamic memory allocation. So pointers work with 'memory'. It necessitates the need to understand jargons related to 'memory' and pointer implementations.

12.1 Memory Overwrite

Whenever we write data into memory, we're actually overwriting the existing data. If we "owned" that memory and if we overwrite it, then there won't be any problem. Otherwise, we would lose any valid data that exist there before. So we must avoid memory overwrite and we should use only the allocated memory.

12.2 Array/Buffer Overflow

If we copy or insert data more into an array of limited size, it is referred as array overflow. Look at the following code:

```
char var1[10];
char var2[5] = "Hello"; /* '\0' is not added as size
                  is given as 5*/
strcpy( var1, var2 );
```

Here, we can find that `var2` ("Hello") is not terminated with a Null terminator ('\0'). So when we copy `var2` to `var1` using `strcpy( )`, the `strcpy( )` routine will copy all the character to `var2` until it finds '\0' in memory. So array overflow may result in memory overwrite!

12.3 Memory Leak

When you repeatedly allocate memory without freeing it, such that all available memory leaks away, it is called as *memory leak*. Too much of memory leak would crash TC, DOS or Windows. So it is more dangerous. For example, the following code would result in memory leak.

```
#include <stdlib.h>
#include <stdio.h>
int main( void )
{
     int x = 1;
```

```
    int *ptr = malloc( sizeof( int ) );
    ptr = &x;
    x = 2;
    *ptr  = 3;
    return(0);
}
```

Here, the variable `ptr` is first initialized with `malloc( )` and once again with address of x. So the value that was returned by `malloc( )` is definitely lost. Now we have memory leak even if we call `free( )` function, because the `free( )` function must be called with the exact value of the pointer returned by `malloc( )`.

The remedy for memory leak is to declare pointer constant. That is,

```
int *const ptr = malloc( sizeof( int ) );
ptr = &x;  /* compiler error */
```

Now, the compiler will generate error. So, we are in safe from memory leak problem.

12.4 Multidimensional array implementation

For the sake of simplicity, let's see two-dimensional implementation only. All of these techniques can also be extended to three or more dimensions.

12.4.1 Version 1

We may allocate an array of pointers, and then initialize each pointer to a dynamically-allocated row.

```
int **array = (int **)malloc(rows * sizeof(int *));
for(i = 0; i < rows; ++i)
        array[i] = (int *)malloc(columns * sizeof(int));
```

I personally prefer this implementation.

12.4.2 Version 2

You may keep the array's contents contiguous with pointer arithmetic as:

```
int **array = (int **)malloc(rows * sizeof(int *));
array[0] = (int *)malloc(rows * columns * sizeof(int));
for(i = 1; i < rows; ++i)
        array[i] = array[0] + i * columns;
```

12.4.3 Version 3

You may also simulate a two-dimensional array with a single, dynamically-allocated one-dimensional array.

```
int *array = (int *)malloc(rows * columns * sizeof(int));
```

12.4.4 Version 4

Here is another version which uses pointers to arrays.

```
int (*array)[NO_OF_COLUMNS] =
        (int (*)[NO_OF_COLUMNS])malloc(rows * sizeof(*array));
```

12.5 Linked List

Linked list is one of the important applications of pointer concepts. Here is the program to create / append, display & reverse a linked list.

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

typedef struct node LNKLIST;

struct node
{
   int data;
   LNKLIST *next;
};

int main( void )
{
   LNKLIST *start = NULL, *p, *q, *temp;
   char opt;
   do
    {
       printf( "\n\t\t Menu"                 \
             "\n\t\t ~~~~"                   \
             "\n\t 1. Create/Append Linked List"\
             "\n\t 2. Reverse Linked List"        \
             "\n\t 3. Display Linked List"        \
             "\n\t 4. Exit"                  \
             "\n Enter your choice "
           );
       opt = getchar( );
       flushall( );

       switch( opt )
       {
           case '1':      /* Create/append Linked List */
                  do
                  {
                     p = start;
                     /* Traverse upto the last node to append */
```

```
                    while( p->next!=NULL )
                        p = p->next;
                    q = (LNKLIST*)malloc(sizeof(LNKLIST));
                    printf( "\nEnter the data: " );
                    scanf( "%d", &q->data );
                    q->next = NULL;
                    if ( start==NULL )
                          start = q;
                       else
                          p->next = q;
                    printf( "Wanna continue? " );
                  } while( tolower( getchar( ) )=='y' );
                  break;

          case '2':    /* Reverse Linked List */
                  p = start;
                  q = p->next;
                  while( q!=NULL )
                   {
                      temp = q->next;
                      q->next = p;
                      p = q;
                      q = temp;
                   }
                  start->next = NULL;
                  start = p;
                  break;

          case '3':  /* Print linked list as [Data | Address] */
                  p = start;
                  printf( "\nstart =%u ", start );
                  while( p!=NULL )
                   {
                       printf( "-> [%d | %u]", p->data, p->next );
                       p = p->next;
                   }
                  getchar( );
       }
    } while( opt!='4' );
   return(0);
} /*--main( )----------*/
```