Subsections

Dynamic Memory Allocation and Dynamic Structures

Dynamic allocation is a pretty unique feature to C (amongst high level languages). It enables us to create data types and structures of any size and length to suit our programs need within the program.

We will look at two common applications of this:

- dynamic arrays
- dynamic data structure ***e.g.*** linked lists

Malloc, Sizeof, and Free

The Function `malloc` is most commonly used to attempt to ``grab'' a continuous portion of memory. It is defined by:

```
 void *malloc(size_t number_of_bytes)
```

That is to say it returns a pointer of type `void *` that is the start in memory of the reserved portion of size `number_of_bytes`. If memory cannot be allocated a `NULL` pointer is returned.

Since a `void *` is returned the C standard states that this pointer can be converted to any type. The `size_t` argument type is defined in `stdlib.h` and is an ***unsigned type***.

So:

```
    char *cp;
                 cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to `cp`.

```
Also it is usual to use the sizeof() function to specify the number of
bytes:
```

```
    int *ip;
                 ip = (int *) malloc(100*sizeof(int));
```

```
Some C compilers may require to cast the type of conversion. The (int *)
means coercion to an integer pointer. Coercion to the correct pointer type
is very important to ensure pointer arithmetic is performed correctly. I
personally use it as a means of ensuring that I am totally correct in my
coding and use cast all the time.
```

```
It is good practice to use sizeof() even if you know the actual size you
```

want -- it makes for device independent (portable) code.

sizeof can be used to find the size of any data type, variable or structure. Simply supply one of these as an argument to the function.

SO:

```
   int i;
                 struct COORD {float x,y,z};
                 typedef struct COORD PT;

                 sizeof(int), sizeof(i),
                 sizeof(struct COORD) and
                 sizeof(PT) are all ACCEPTABLE
```

In the above we can use the link between pointers and arrays to treat the reserved memory like an array. ***i.e*** we can do things like:

```
   ip[0] = 100;
```

or

```
   for(i=0;i<100;++i) scanf("%d",ip++);
```

When you have finished using a portion of memory you should always free() it. This allows the memory ***freed*** to be aavailable again, possibly for further malloc() calls

The function free() takes a pointer as an argument and frees the memory to which the pointer refers.

Calloc and Realloc

There are two additional memory allocation functions, `Calloc()` and `Realloc()`. Their prototypes are given below:

```
void *calloc(size_t num_elements, size_t element_size};

void *realloc( void *ptr, size_t new_size);
```

`Malloc` does not initialise memory (to ***zero***) in any way. If you wish to initialise memory then use `calloc`. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between `calloc` and `malloc` in that `calloc` takes the number of desired elements, `num_elements`, and element_size, `element_size`, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
    int *ip;
                 ip = (int *) calloc(100, sizeof(int));
```

Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged.

If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You **must**

```
use this new value. If new memory cannot be reallocated then realloc
returns NULL.

Thus to change the size of memory allocated to the *ip pointer above to an
array block of 50 integers instead of 100, simply do:

   ip = (int *) calloc( ip, 50);
```

Linked Lists

Let us now return to our linked list example:

```
  typedef struct {  int value;
                                                   ELEMENT *next;
                                 } ELEMENT;
```

We can now try to grow the list dynamically:

```
 link = (ELEMENT *) malloc(sizeof(ELEMENT));
```

`This will allocate memory for a new link.`

`If we want to deassign memory from a pointer use the free() function:`

```
  free(link)
```

See Example programs (queue.c) below and try exercises for further practice.

Full Program: `queue.c`

A queue is basically a special case of a linked list where one data element joins the list at the left end and leaves in a ordered fashion at the other end.

The full listing for `queue.c` is as follows:

```
/*                                                            */
/* queue.c                                                    */
/* Demo of dynamic data structures in C                       */

#include <stdio.h>

#define FALSE 0
#define NULL 0

typedef struct {
    int     dataitem;
    struct listelement *link;
}               listelement;

void Menu (int *choice);
listelement * AddItem (listelement * listpointer, int data);
listelement * RemoveItem (listelement * listpointer);
void PrintQueue (listelement * listpointer);
void ClearQueue (listelement * listpointer);

main () {
    listelement listmember, *listpointer;
    int     data,
            choice;

    listpointer = NULL;
    do {
        Menu (&choice);
        switch (choice) {
            case 1:
                printf ("Enter data item value to add  ");
                scanf ("%d", &data);
                listpointer = AddItem (listpointer, data);
                break;
            case 2:
                if (listpointer == NULL)
                    printf ("Queue empty!\n");
                else
                    listpointer = RemoveItem (listpointer);
                break;
            case 3:
                PrintQueue (listpointer);
                break;

            case 4:
                break;

            default:
                printf ("Invalid menu choice - try again\n");
                break;
        }
    } while (choice != 4);
    ClearQueue (listpointer);
}                               /* main */

void Menu (int *choice) {

    char    local;

    printf ("\nEnter\t1 to add item,\n\t2 to remove item\n\
\t3 to print queue\n\t4 to quit\n");
    do {
        local = getchar ();
        if ((isdigit (local) == FALSE) && (local != '\n')) {
            printf ("\nyou must enter an integer.\n");
            printf ("Enter 1 to add, 2 to remove, 3 to print, 4 to quit\n");
        }
    } while (isdigit ((unsigned char) local) == FALSE);
    *choice = (int) local - '0';
}

listelement * AddItem (listelement * listpointer, int data) {

    listelement * lp = listpointer;

    if (listpointer != NULL) {
        while (listpointer -> link != NULL)
            listpointer = listpointer -> link;
        listpointer -> link = (struct listelement  *) malloc (sizeof (listele
        listpointer = listpointer -> link;
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return lp;
    }
    else {
        listpointer = (struct listelement  *) malloc (sizeof (listelement));
        listpointer -> link = NULL;
        listpointer -> dataitem = data;
        return listpointer;
    }
}

listelement * RemoveItem (listelement * listpointer) {

    listelement * tempp;
    printf ("Element removed is %d\n", listpointer -> dataitem);
    tempp = listpointer -> link;
    free (listpointer);
    return tempp;
}

void PrintQueue (listelement * listpointer) {

    if (listpointer == NULL)
        printf ("queue is empty!\n");
    else
        while (listpointer != NULL) {
            printf ("%d\t", listpointer -> dataitem);
            listpointer = listpointer -> link;
        }
    printf ("\n");
}

void ClearQueue (listelement * listpointer) {

    while (listpointer != NULL) {
        listpointer = RemoveItem (listpointer);
    }
}
```

Exercises

Exercise 12456

Write a program that reads a number that says how many integer numbers are to be stored in an array, creates an array to fit the exact size of the data and then reads in that many numbers into the array.

Exercise 12457

Write a program to implement the linked list as described in the notes above.

Exercise 12458

Write a program to sort a sequence of numbers using a binary tree (Using Pointers). A binary tree is a tree structure with only two (possible) branches from each node (Fig. 10.1). Each branch then represents a false or true decision. To sort numbers simply assign the left branch to take numbers less than the node number and the right branch any other number (greater than or equal to). To obtain a sorted list simply search the tree in a depth first fashion.

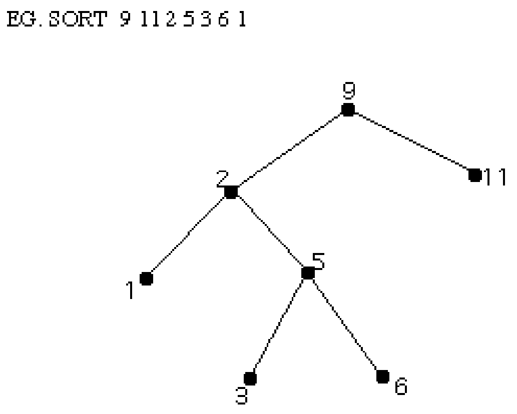

Fig. 10.1 Example of a binary tree sort Your program should: Create a binary tree structure. Create routines for loading the tree appropriately. Read in integer numbers terminated by a zero. Sort numbers into numeric ascending order. Print out the resulting ordered values, printing ten numbers per line as far as possible.

Typical output should be

```
The sorted values are:
 2  4  6  6  7  9 10 11 11 11
15 16 17 18 20 20 21 21 23 24
27 28 29 30
```

Dave Marshall
1/5/1999