Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

Text Book(s):

1. “How To Solve It By Computer”, R G Dromey, Pearson, 2011.
2. “The C Programming Language”, Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Reference Book(s):

1. “Expert C Programming; Deep C secrets”, Peter van der Linden
2. “ The C puzzle Book”, Alan R Feuer

Dynamic memory allocation

Using Memory As You Go

Pointers are an extremely flexible and powerful tool for programming over a wide range of applications. The majority of programs in C use pointers to some extent. C also has a further facility called *dynamic memory allocation that* depends on the concept of a pointer and provides a strong incentive to use pointers in your code. Dynamic memory allocation allows memory for storing data to be allocated dynamically when your program executes. Allocating memory dynamically is possible only because you have pointers available.

An example

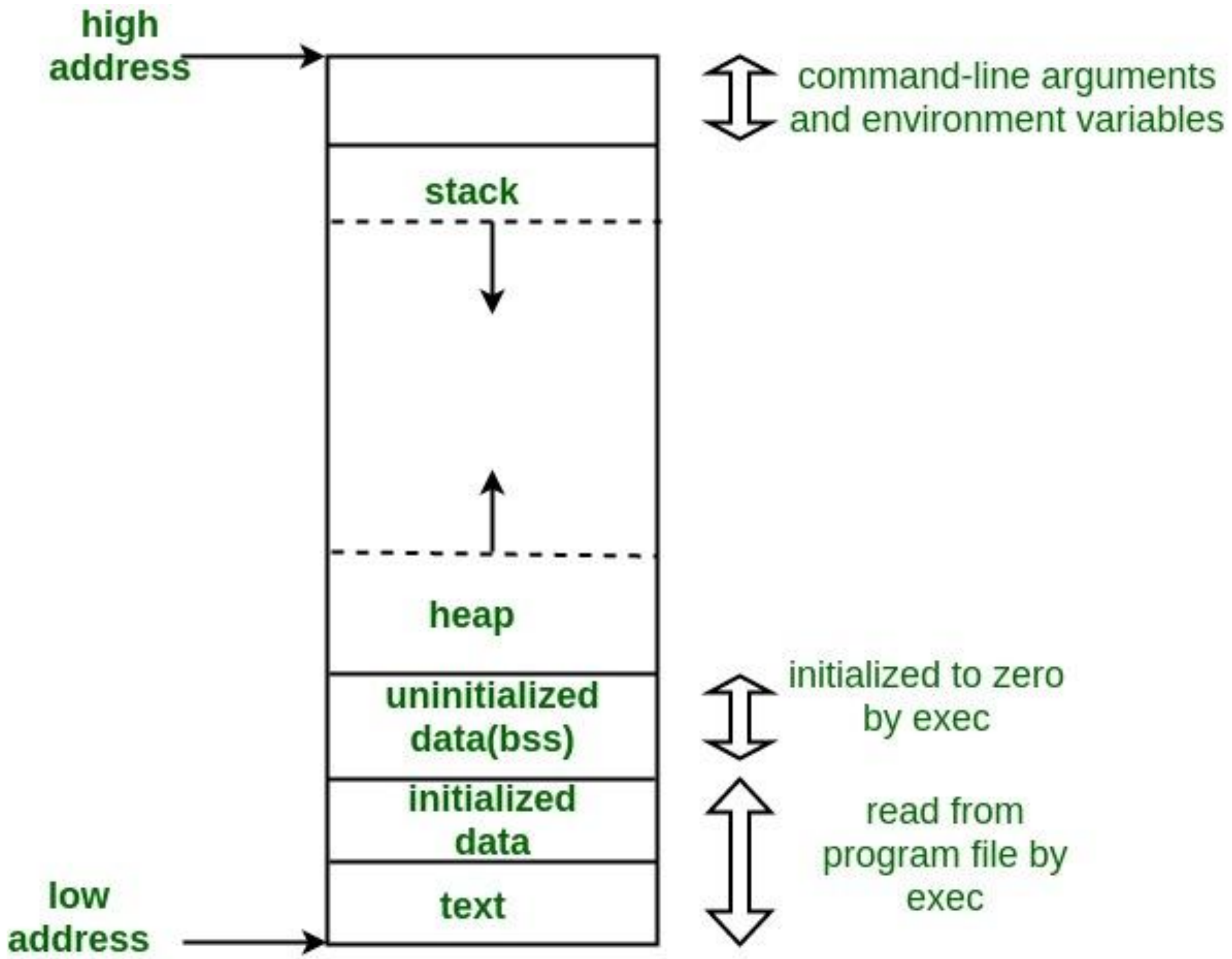
The majority of production programs will use dynamic memory allocation. Your e-mail client does, for example. When you retrieve your e-mail, the program has no prior knowledge of how many e-mails there will be or how much memory each requires. The e-mail client will obtain sufficient memory at runtime to manage the number and size of your e-mails.

When you explicitly allocate memory at runtime in a program, space is reserved for you in a memory area called the *heap. There's another memory area called the stack associated with a program in which space to store function* arguments and local variables in a function is allocated. When the execution of a function ends, the space allocated to store arguments and local variables is freed.

<u>The memory in the heap is different in that it is controlled by you. When you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required and free the space you have allocated to allow it to be reused</u>.

Key Differences Between Stack and Heap Allocations

1. In a stack, the allocation and de-allocation is automatically done whereas, in heap, it needs to be done by the programmer manually.
2. Handling of Heap frame is costlier than handling of stack frame.
3. Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.
4. Stack frame access is easier than the heap frame as the stack has small region of memory and is cache friendly, but in case of heap, frames are dispersed throughout the memory so it causes more cache misses.
5. Stack is not flexible, the memory size allotted cannot be changed whereas a heap is flexible, and the allotted memory can be altered.
6. Accessing time of heap is more than a stack.


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Memory layout of a C program

The function malloc() is a general-purpose function that is used to allocate memory for any type of data. The function has no knowledge of what you want to use the memory for, so it returns a pointer of type pointer to void, which is written as void*. Pointers of type void* can point to any kind of data.

<u>However, you can't dereference a pointer of type pointer to void because what it points to is unspecified. Your compiler will always arrange for the address returned by malloc() to be automatically converted to the pointer type on the left of the assignment, but it doesn't hurt to put an explicit cast</u>.

Dynamic memory allocation in C

#include <stdlib.h>
void *malloc(size_t *size***);**

Description
The **malloc**() function allocates *size* bytes and returns a pointer to the allocated memory. *The memory is not initialized*. If *size* is 0, then **malloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

Return Value
The function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. On error, it returns NULL. NULL may also be returned by a successful call to **malloc**() with a *size* of zero.

Note: Assume size_t is same as **int**.

#include <stdlib.h>
void *calloc(size_t *nmemb*, **size_t** *size***)**

Description
The **calloc**() function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero. If *nmemb* or *size* is 0, then **calloc**() returns either NULL, or a unique pointer value that can later be successfully passed to **free**().

Return Value
The function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. On error, the function returns NULL. NULL may also be returned by a successful call to **calloc**() with *nmemb* or *size* equal to zero.

Releasing Dynamically Allocated Memory

When you allocate memory dynamically, you should always release the memory when it is no longer required. Memory that you allocate on the heap will be automatically released when your program ends, but it is better to explicitly release the memory when you are done with it, even if it's just before you exit from the program. In more complicated situations, you can easily have a memory leak.

What is a memory leak?

A *memory leak occurs when you allocate some memory* dynamically and you do not retain the reference to it, so you are unable to release the memory. This often occurs within a loop, and because you do not release the memory when it is no longer required, your program consumes more and more of the available memory on each loop iteration and eventually may occupy it all.

#include <stdlib.h>
void free(void **ptr***);**

Description
The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**(), **calloc**() or **realloc**(). Otherwise, or if *free(ptr)* has already been called before, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

Return Value
The **free**() function returns no value.

#include <stdlib.h>
void *realloc(void **ptr*, **size_t** *size***);**

Description

The **realloc**() function changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will *not* be initialized. If *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*; if *size* is equal to zero, and *ptr* is not NULL, then the call is equivalent to *free(ptr)*. Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc**(), **calloc**() or **realloc**(). If the area pointed to was moved, a *free(ptr)* is done.

Return Value

The **realloc**() function returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or NULL if the request fails. If *size* was equal to 0, either NULL or a pointer suitable to be passed to **free**() is returned. If **realloc**() fails the original block is left untouched; it is not freed or moved.

A simple program to show the use of malloc and realloc

The program.

Write a program that will read an arbitrary number of proverbs from the keyboard and store them in memory that's allocated at runtime. the program should then

1. output the proverbs

The program is here.

Write a program in C to perform the following:

1. Allow the user to key in data for many records
2. Each record should be a structure containing the following information:
    a. Title of the book
    b. Name of the author
    c. Year of publication
    d. Price
3. Display all the information given by the user.
4. You need to use dynamic memory allocation for all the relevant variables
5. Split the program into as many functions as you can
6. Define your own header file and include that in the .c file

The program is here.

Allocating memory for a two-dimensional array

Using a single pointer**:**

A simple way is to allocate memory block of size r*c and access elements using simple pointer arithmetic.

The code is here.

Using an array of pointers

We can create an array of pointers of size r. Note that from C99, C language allows variable sized arrays. After creating an array of pointers, we can dynamically allocate memory for every row.

The program is here.

Using pointer to a pointer

We can create an array of pointers also dynamically using a double pointer. Once we have an array of pointers allocated dynamically, we can dynamically allocate memory for every row like method 2

The program is here.

Pointers to Structures as structure members

Any pointer can be a member of a structure. This includes a pointer that points to a structure. A pointer structure member that points to the same type of structure is also permitted. This is called a self-referential structure.

Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have following limitations.

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

Deletion is also expensive with arrays unless some special techniques are used.

<u>Advantages of linked lists over arrays</u>
1. Dynamic size
2. Ease of insertion/deletion

Drawbacks:
1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.

Each node in a list consists of at least two parts:
1) data
2) Pointer (Or Reference) to the next node

In C, we can represent a node using structures. Below is an example of a linked list node with an integer data.

```
// A linked list node
struct Node
{
  int data;
  struct Node *next;
};
```

Inserting a node in a list

A node can be added in three ways

1. At the front of the linked list (As the first node in the list)
2. After a given node.
3. At the end of the linked list (As the last node)

We will look at the program.

Deleting a node from a list

The node which is to be deleted can be:

1. The first node in the list
2. The last node in the list
3. Anywhere in the list

We will look at the program now.

Reversing the nodes in a list

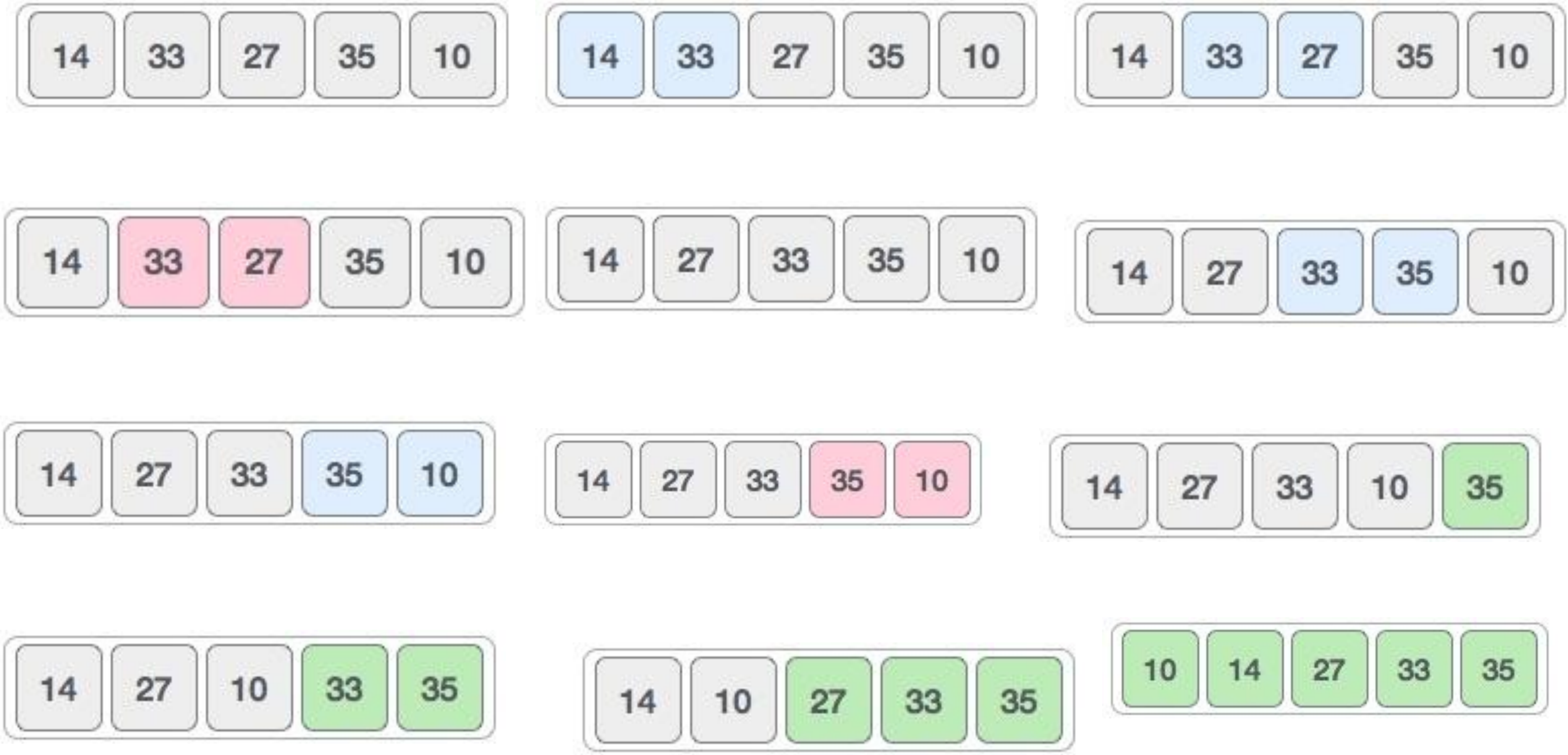
The links have to be rearranged such that the last node becomes the head and the direction of the “next” pointer changes.

Sorting the nodes (Bubble sort)

Sort the nodes in the increasing order of “data”.

- This sorting algorithm is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

An example


14 33 27 35 10
14 33 27 35 10
14 33 27 35 10
14 33 27 35 10
14 27 33 35 10
14 27 33 35 10
14 27 33 35 10
14 27 33 35 10
14 27 33 10 35
14 27 10 33 35
14 10 27 33 35
10 14 27 33 35

Linear Search

Brute force method – Compare the key with data in all the nodes. If found, return true, else false.

The program will be discussed now.

The full program is here.

Implementation of the linked list using double pointer is in this program and this.