Arrays and Pointers: Dynamic memory allocation

Recall that we wrote a program to count the number of students who scored above the average. The array size was defined with `#define N 100`, which can handle up to 100 students. Since the array size is 100, a contiguous memory allocation of 100 locations is made, each capable of holding an integer. If the actual student number is lower, say 40, we are wasting 60 places, since these are already blocked by the array declaration.

Here, we show how to allocate space dynamically. Thus, if there are 40 students, we create 40 contiguous places, each of which can hold an integer. We can create contiguous blocks, use them like an array, and free them when they are no longer needed. C provides the functions `calloc()` and `malloc()`, whose prototypes are defined in the standard library `stdlib.h`. The name `calloc` stands for "contiguous allocation" and the name `malloc` stands for "memory allocation." A programmer uses `calloc()` and `malloc()` to dynamically create space for arrays, structures, and unions. We shall discuss structures and unions later.

Both `calloc()` and `malloc()` return a `void*` pointer. If memory cannot be allocated, it should be checked for `NULL`. Note that the `void*` pointer returned by `calloc()` and `malloc()` is the base (starting) address of the blocks created. The memory allocated by `malloc()` is uninitialized, meaning it contains garbage values. On the other hand, the block created by `calloc()` is initialized to zero.

Since these functions return a `void*` (generic pointer), we cast it to the required type to tell the compiler how to interpret and use that memory for storing variable types. Consider the following code, which creates 5 contiguous spaces, each of which can hold an integer variable, and returns the base (starting) address. We use both `calloc()` and `malloc()` to achieve the same result. We can use any one of those blocks to assign marks and find the total marks. Finally, we free the allocated space.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int *a,*b,i,s,n=5,max;
 a=(int *) calloc(n,sizeof(int));
 if(a==NULL)  // check allocation successful
 {
  printf("Problem with calloc:stop\n");
  return 0;
 }
 b=(int *) malloc(n*sizeof(int));
 if(b==NULL)  // check allocation successful
 {
```

```
 printf("Problem with malloc:stop\n");
 return 0;
}
printf("a=%p  b=%p\n",a,b); // print base addresses of spaces created
for(i=0;i<n;i++)
{
 b[i]=rand()%101;   // assign random marks
}
s=0;
for(i=0;i<n;i++)
{
 s+=b[i];
}
printf("Sum of marks = %d\n",s);
   // do other works
   free(a);          //free the spaces with base address at a
   free(b);          //free the spaces with base address at b
   // do other works
return 0;
}
```

Listing 51: C Program "dynmem.c"

Here is the output:

```
a=0x104e69860  b=0x99ac009a0
Sum of marks = 197
```

When the expression `calloc(n,sizeof(int))` is executed with `n=5`, a contiguous 5 spaces are created where each space can hold an integer. The `calloc()` also initializes the spaces with zeros. Further, this function returns the base address, which we explicitly cast using `(int *)` and stored in `int` pointer `a`. The `malloc()` function does a similar job, but the spaces are not initialized. The base address returned by `malloc()` is stored in `int` pointer `b`. Now, if we write `a[i]`, which is equivalent to `*(a+i)`, then we are accessing the content of the location that is obtained by adding $4 \times i$ bytes (assuming that an `int` takes 4 bytes of memory) to the address stored in `a`. Similar is the situation for `a`. Thus, though `a` and `b` are `int` pointer variables, they behave like a one-dimensional `int` array. Next, we assign random marks to the array-like `b` and calculate the total marks. Once we no longer need the allocated spaces, we use the `free()` functions. Executing `free(a)` frees the allocated spaces whose base address is stored in `a`. Note that though `a` and `b` still hold the base addresses, the contents of those allocated spaces are now undefined. It is very important to free the allocated spaces when they

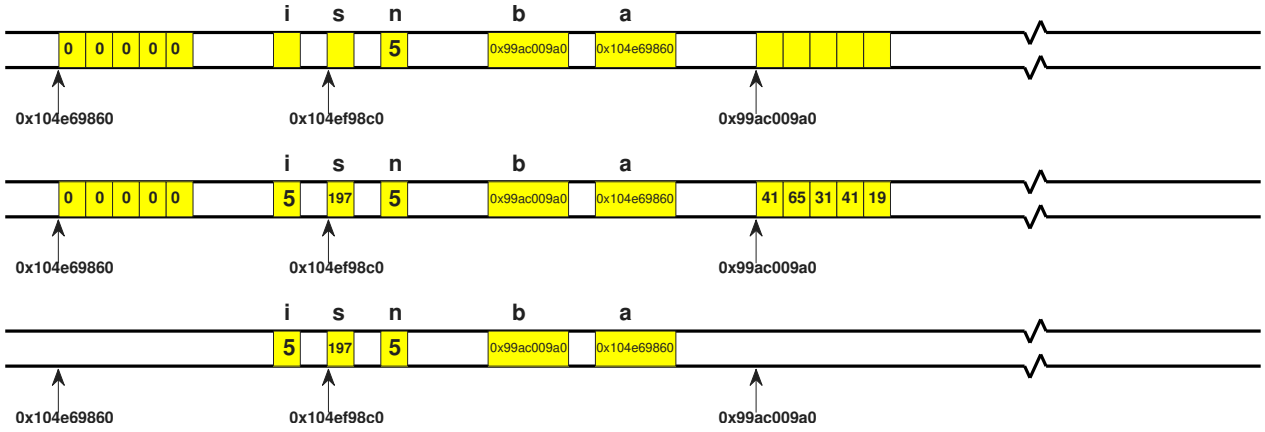

Figure 1: Memory arrangement for Listing 51. The top panel shows the memory allocated by calloc() and malloc(). The middle panel shows the result after the assignment using the rand() function, and the bottom panel shows the configuration after the execution of the free() function.

are no longer needed. The reason is that those spaces will remain allocated until the execution stops. This is true even if these spaces are created in functions other than `main()`. Suppose that we replace the statements

```
free(a);
free(b);
```

in Listing 51 with

```
free(a);
b=&s;
```

Then the configuration is shown in Fig. 2. Now, since `b` has the address of `int s`, we can no longer access the base address created using `malloc()`. Hence, the spaces created using `malloc()` will remain during the execution of the program.

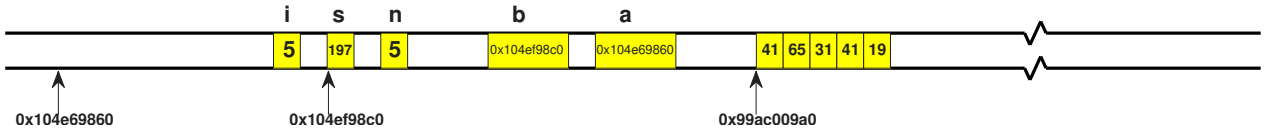

Figure 2: See text for details.

Let us consider another program that uses functions that return a pointer. We have used two functions, both of which accept an integer argument and return an address. The prototypes of these functions are declared at the top of the program (see Listing 52). In the `main()`, there are two `int` variables `i, n` and two `int*` variables `a, b`. Also, `n` is initialized with 5. The configuration before the call to `dynarr()` is shown in the top panel of Fig. 3. Now the function

`dynarr()` is called. The variables `n, i, p` used in the `dynarr()` definition are local to the function and they are shown using cyan background colour. The configuration, just before the function call returns to `main()`, is shown in the second panel of Fig. 3. We have created `n=5` contiguous spaces with `malloc()`, each can hold an integer variable and the base address is stored in `p`. We have also initialized the contiguous spaces created with integer values 10, 20, 30, 40, and 50. Finally, we return the value of `p` to the pointer variable `a` of `main()`. The configuration, when the call returned from `dynarr()` and before calling `constarr()`, is shown in the third panel of Fig. 3. Clearly, the local variables of `dynarr()` disappeared except that the allocated spaces remain intact along with the assigned values.

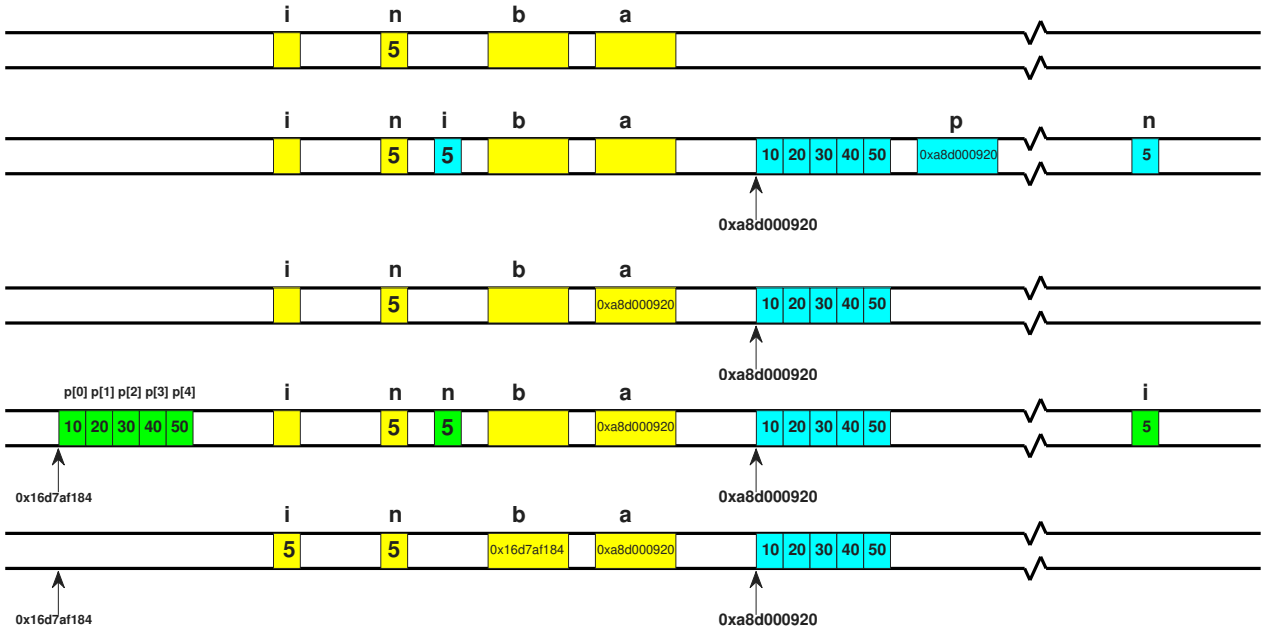

Figure 3: See text for details.

```
#include<stdio.h>
#include<stdlib.h>
int *dynarr(int);
int *constarr(int);
int main()
{
 int *a,*b,i,n=5;
 a=dynarr(n);
 b=constarr(n);
 printf("a=%p  b=%p\n",a,b);
        for(i=0;i<n;i++)
 {
  printf("a[%d]=%d   b[%d]=%d\n",i,a[i],i,b[i]);
```

```
 }
 return 0;
}
int *dynarr(int n)
{
 int *p,i;
 p=(int *) malloc(n*sizeof(int));
 printf("dynarr: p=%p\n",p);
 for(i=0;i<n;i++)
 {
  p[i]=(i+1)*10;
 }
 return p;
}
int *constarr(int n)
{
 int p[5],i;
 printf("constarr: p=%p\n",p);
 for(i=0;i<n;i++)
 {
  p[i]=(i+1)*10;
 }
 return p;
}
```

Listing 52: C Program "dynvfix.c"

```
dynarr: p=0xa8d000920
constarr: p=0x16d7af184
a=0xa8d000920  b=0x16d7af184
a[0]=10   b[0]=1
a[1]=20   b[1]=-1695709560
a[2]=30   b[2]=1
a[3]=40   b[3]=40
a[4]=50   b[4]=50
```

Fourth panel in Fig. 3 shows the configuration just before the `return` statement in `constarr()`. It shows that array `p[5]` is created and initialized with the same values as in the case of `dynarr()`. Then we return the base address `p` which is equivalent to `&p[0]`. Thus, `b` of `main()` is assigned with base address of the local array of `constarr()`. When we print the elements

of the allocated spaces and the local array, then we see that only the elements of the allocated spaces are printed correctly. The reason for this is the following. When we compile, we see the following warning

```
address of stack memory associated with local variable 'p' returned [-Wreturn-stack-ad
```

It means the local array `p[5]` of `constarr()` is created on the stack, with a lifetime equal to the function's. On the other hand, the allocated spaces in `dynarr()` are created on the heap, whose lifetime is the program's lifetime. Thus, the allocated spaces and the values assigned to the allocated spaces created with `malloc()` or `calloc()` remain with the program's lifetime. The last panel of Fig. 3 shows the configuration after the last `printf()` in the main(). To deallocate the spaces, we need to use the `free()` function.

Merge sort: Finally, we illustrate merge sort algorithm where we show the power of recursion. Also, we use dynamic memory allocation in the algorithm. Merge sort is an efficient, stable sorting algorithm that uses the divide-and-conquer approach to sort a list of elements.

Divide: It recursively divides an array of elements into two (equal or almost equal) halves until each sublist contains only a single element. A list with a single element is considered sorted by definition.

Conquer/Merge: The sorted sublists are then merged back together in sorted order. Here, we compare the first elements of two adjacent sorted sublists, take the smaller element, and adding it to a new, larger sublist. This is repeated until all elements from both the sublists are added to the new sublist. This merging continues until a single, fully sorted list is obtained.

To illustrate the merge process, consider a subarray `A[n_s:n_e]`, where `n_s` is the starting index, `n_e` is the end index. Thus, the subarray has `n_e-n_s+1` elements. Let `n_m` be an index (which could be near the middle of the subarray) such that subarrays `A[n_s:n_m]` and `A[n_m+1,n_e]` are in sorted order. For example, consider the integer array

`A[2:10]={10,23,45,60,6,8,30,41,70`

Here, total number of elements is `10-2+1=9` and the subarray `A[2:5]={10,23,45,60}` and the subarray `A[6:10]={6,8,30,41,70}` are sorted. Thus, here `n_m=5`.

Now we create a dynamic array `W[0:n_e-n_s]` which also has `n_e-n_s+1` elements. Here, dynamic array `W` means we create contiguous memory allocation for `n_e-n_s+1` elements and store the base address in `W`.

First, we compare `A[n_s]` and `A[n_m+1]`. Whichever is smaller, say `A[n_m+1]`, put into `W[0]`. Next, compare `A[n_s]` and `A[n_m+2]`. Whichever is smaller, say `A[n_m+2]`, put into `W[1]`. Next, compare `A[n_s]` and `A[n_m+3]`. Whichever is smaller, say `A[n_s]`, put into `W[2]`. Next, compare `A[n_s+1]` and `A[n_m+3]`, and so on. Eventually one of the subarrays `A[n_s:n_m]` and `A[n_m+1,n_e]` will be exhausted. At that point, the remainder of the elements in the other array are simply copied into array `W`. Finally, we copy `W[0:n_e-n_s+1]` into `A[n_s:n_e]` and the subarray `A[n_s:n_e]` is now sorted.

The merge sort algorithm works as follows: Let `A[0:n]` be an unsorted array. We call the `merge_sort()` function which has the base address of the array to be sorted, an `int` variable

`start` that denotes the starting index, and an `int` variable `end`, which is the final index of the array. Now, inside the `merge-sort()`, we divide the array into two halves and then again call `merge-sort()` with each half so that both of them will be sorted. Next, we merge the two halves using the procedure described in the previous paragraph to obtain the sorted array. Note that the same process repeats for each half until a half consists of only one element, which is already sorted. Below is the code that implements this. Again, we use it to sort students' marks in a class. To see the working of the algorithm, we use static variables in the functions `merge_sort()` and `merge ()`. A static variable in C is a variable that has a lifetime for the entire duration of the program, but its scope is limited to the function or file in which it is declared. It is initialized only once, and its value persists across multiple function calls-unlike a normal (automatic) local variable, which is created and destroyed with each function call.

```
#include<stdio.h>
#include<stdlib.h>
int *assign_mrks(int);
void print_mrks(int*,int);
void merge(int *,int,int,int);
void merge_sort(int *,int,int);
int main()
{
 int i,*a,n; // n no. of students
 printf("Enter the no. of students:");
 scanf("%d",&n);
        a=assign_mrks(n); // assign marks
 printf("Marks of the students are:\n");
 print_mrks(a,n); // print marks
 merge_sort(a,0,n-1); // call recursive function merge_sort
 printf("Marks of the students in sorted order are:\n");
 print_mrks(a,n); // print marks
 return 0;
}

int *assign_mrks(int n)
{
 int i,*p;
 p=(int *)calloc(n,sizeof(int));
 if(p==NULL)
 {
  printf("Error in calloc: stop\n");
```

```
  exit(1);
 }
 for(i=0;i<n;i++)
 {
  p[i]=rand()%101;
 }
 return p;
}
void print_mrks(int *a,int n)
{
 int i;
 for(i=0;i<n;i++)
 {
  printf("%4d",a[i]);
  if((i+1)%10==0)
  {
   printf("\n");
  }
 }
 printf("\n");
}
void merge_sort (int *A, int start, int end) // Array A[start:end]
{
           static int count=0;
    count++;
//    printf("merge_sort pass %d: start=%d end=%d\n",count,start,end);
    int mid;
           if( start < end)   // array is not a singleton
    {
     mid = (start + end )/2;           // divide the current array in 2 halves
               merge_sort (A, start, mid) ;      // sort the 1st half of array .
               merge_sort (A,mid+1, end) ;       // sort the 2nd half of array.
               merge(A,start,mid,end);           // merge both halves
    }
}
void merge(int *A, int start, int mid, int end)
{
 int i,p,q,k,*W; // W is a temporary dynamic array
 static int count=0;
```

```
 count++;
 p=start;   // p holds the starting position of 1st array half
 q=mid+1;   // q holds the starting position of 2nd array half
        W=(int *)calloc(end-start+1,sizeof(int));
 if(W==NULL)
 {
  printf("Problem in W calloc:stop\n");
  exit(1);
 }
 k=0;
 for(i=start;i<end+1;i++)
 {
  if(p>mid)      // check if first half exhausted
  {
   W[k]=A[q];    // fill with 2nd half
   k++;
   q++;
  }
  else if(q>end) //check if 2nd half exhusted
  {
   W[k]=A[p];
   k++;
   p++;
  }
  else if(A[p]<A[q])  // element of 1st half < element of 2nd half
  {
   W[k]=A[p];
   k++;
   p++;
  }
  else         // element of 2nd half < element of 1st half
  {
   W[k]=A[q];
   k++;
   q++;
  }
 }
// printf("merge pass %d\n",count);
 // copy the temporary array to the original array
```

```
 for(k=0;k<end-start+1;k++)
 {
  A[start+k]=W[k];
//  printf("%4d ",W[k]);
 }
// printf("\n");
 free(W);    // free the temporary allocation
}
```

Listing 53: C Program "merge.c"

To see the working of the algorithm, we run it with 5 students. The output is obtained when we uncomment the `printf()` statements in `merge_sort()` and `merge()` functions.

```
Enter the no. of students:5
Marks of the students are:
  41  65  31  41  19
merge_sort pass 1: start=0 end=4
merge_sort pass 2: start=0 end=2
merge_sort pass 3: start=0 end=1
merge_sort pass 4: start=0 end=0
merge_sort pass 5: start=1 end=1
merge pass 1
  41   65
merge_sort pass 6: start=2 end=2
merge pass 2
  31   41   65
merge_sort pass 7: start=3 end=4
merge_sort pass 8: start=3 end=3
merge_sort pass 9: start=4 end=4
merge pass 3
  19   41
merge pass 4
  19   31   41   41   65
Marks of the students in sorted order are:
  19  31  41  41  65
```

Figure 4 shows the working of the merge sort algorithm. It illustrates the working with 5 students. Below is the output with 57 students.

```
Enter the no. of students:57
```

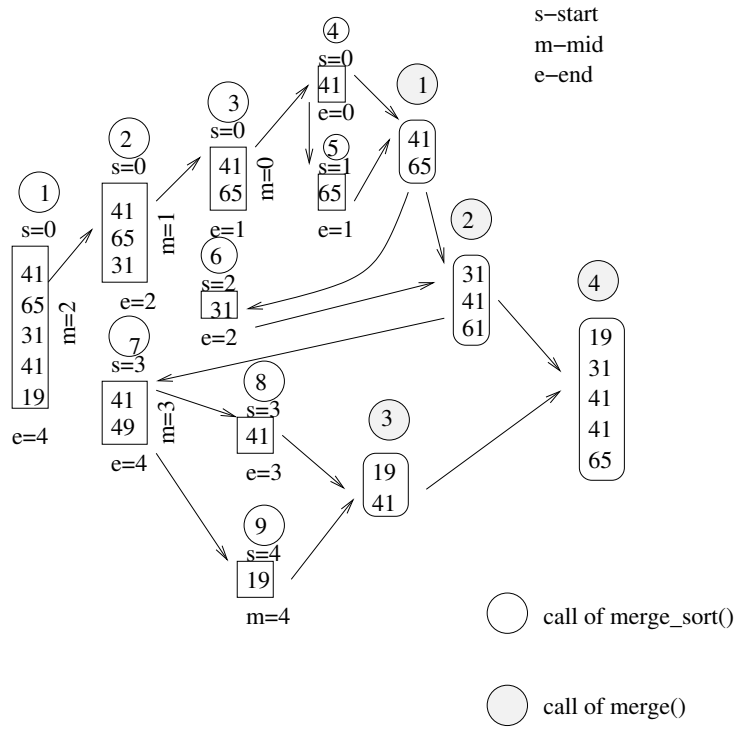

Figure 4: Flow diagram of merge-sort algorithm

```
Marks of the students are:
  41  65  31  41  19  15  72  11  78  69
  37  23  29  63  75   4   5  49  75  99
  27  61  62  17  79  61  22  13  49  71
  61   8  81  67  80  47  83  88  30  12
  74  78  33  23  58  83  76  59  77  85
  25  18  98  16 100  79  88
Marks of the students in sorted order are:
   4   5   8  11  12  13  15  16  17  18
  19  22  23  23  25  27  29  30  31  33
  37  41  41  47  49  49  58  59  61  61
  61  62  63  65  67  69  71  72  74  75
  75  76  77  78  78  79  79  80  81  83
  83  85  88  88  98  99 100
```

Return to Lecture Topics