Dynamic Allocation of Array Space

Request for Run-Time Allocation

- The volume of data may not be known before the run-time.
- It provides flexibility in building data structures.
- The space is allocated in global data area (not on the stack) called heap and so does not evaporate at the end of function call.

```
void *malloc(size_t n)
```

The C library function malloc() is used to request for an allocation of `n` bytes of contiguous memory space in the global data area. After a successful allocation the function returns a generic pointer void * that can be casted to any required type.

```
void *malloc(size_t n)
```

If `malloc()` fails to allocate the requested space, it returns a NULL pointer. The interface of the function is defined in the header file `stdlib.h`, so it is to be included.

Note

The function `malloc()` sends request to the OS for more data space. On request OS supplies multiples of a fixed chunk of memory e.g. multiples of 4 Kbytes. The underlying memory management system of malloc() manages the available memory.

Pointer Variables and Array

Consider the following declarations:

```
int *p, (*q)[5], *r[3], **s ;
```

The variable `p` is of type `int *`. It can store the address of a location of type `int` and follows the `int` pointer arithmetic i.e. `p + i` is `(int)p + i` × `sizeof(int)`.

Pointer Variables and Array

```
int *p, (*q)[5], *r[3], **s ;
```

The variable `q` is of type `int (*)[5]`. It also stores an address but its arithmetic is different from `p`. The value of `q + i` is `(int)q + i` × `5` × `sizeof(int)`. It may be viewed as a pointer to an 1-D array of five (5) locations of type `int`. The arithmetic of `q` is identical to the arithmetic of `a` in `a[3][5]`.

Pointer Variables and Array

```
int *p, (*q)[5], *r[3], **s ;
```

`r` is not a variable but a constant. Its value is the address of the 0^{th} location of the array of three (3) locations, each of type `int *`.
Naturally `r + i` is
`(int)r + i` × `sizeof(int *)`.

Pointer Variables and Array

```
int *p, (*q)[5], *r[3], **s ;
```

Finally the variable `s` is of type `int **`. It can store the address of a location of type `int *`. The meaning of `s + i` is `(int)s + i` × `sizeof(int *)`. It is a pointer to an `int` pointer, so the arithmetic of `r` and `s` are identical.

C Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p, (*q)[5], *r[3], **s;

    printf("sizeof(int): %d\n", sizeof(int));
    printf("sizeof(int *): %d\n", sizeof(int *));
    printf("sizeof(int [5]): %d\n", sizeof(int [5]));
    printf("sizeof(int (*)[5]): %d\n", sizeof(int (*)[5]));
    printf("sizeof(int **): %d\n", sizeof(int **));
    putchar('\n');
```

```
    printf("p: %p\tp+1: %p\n", p, p+1);
    printf("q: %p\tq+1: %p\n", q, q+1);
    printf("r: %p\tr+1: %p\n", r, r+1);
    printf("s: %p\ts+1: %p\n", s, s+1);
    return 0;
} // pointVar.c
```

Output

```
$ cc -Wall pointVar.c
$ ./a.out
sizeof(int):  4
sizeof(int *):  4
sizeof(int [5]):  20
sizeof(int (*)[5]):  4
sizeof(int **):  4

p:  0x2c5ff4 p+1:  0x2c5ff8
q:  0x804961c q+1:  0x8049630
r:  0xbfe53600 r+1:  0xbfe53604
s:  0x80482b5 s+1:  0x80482b9
```

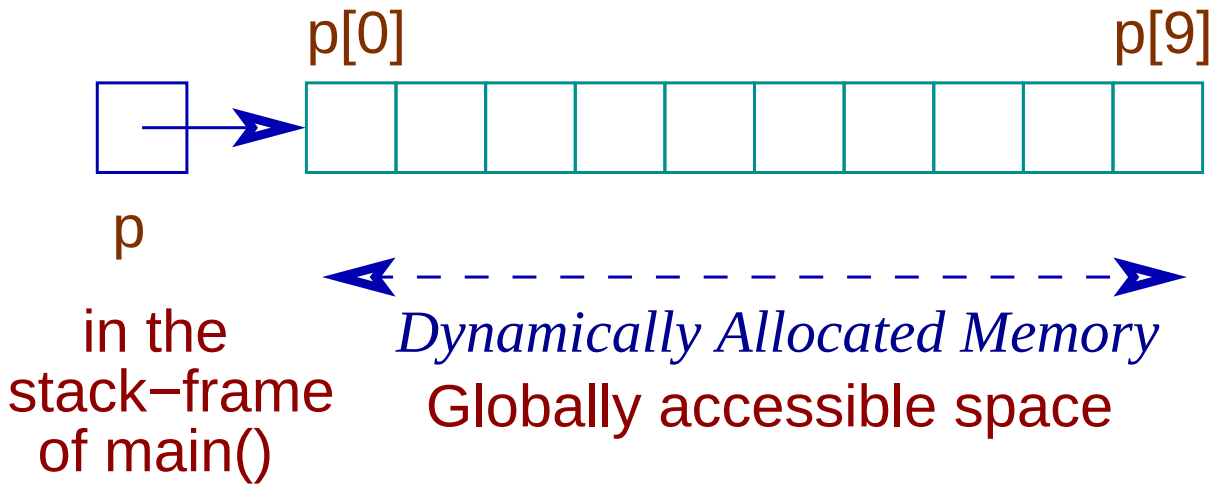
1-D Array of n Elements

We can use a variable of type `int *` and a call to `malloc()` to create an 1-D array of `n` elements of type `int`, where `n` is an input data.

Dynamic 1-D Array

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *p, n, i, val = 1 ;

    printf("Enter a +ve integer: ");
    scanf("%d", &n);
    p = (int *)malloc(n*sizeof(int));
    for(i=0; i<n; ++i) {
        p[i] = val; val = 2*val;
    }
```

```
    for(i=0; i<n; ++i) printf("%d ",p[i]);
    putchar('\n');
    return 0;
} // dynamic1D.c
```


p[0]
p[9]
p
in the
stack−frame
of main()
Dynamically Allocated Memory
Globally accessible space

Output

```
$ cc -Wall dynamic1D.c
$ ./a.out
Enter a +ve integer:  10
1 2 4 8 16 32 64 128 256 512
```

Allocation and Type Casting

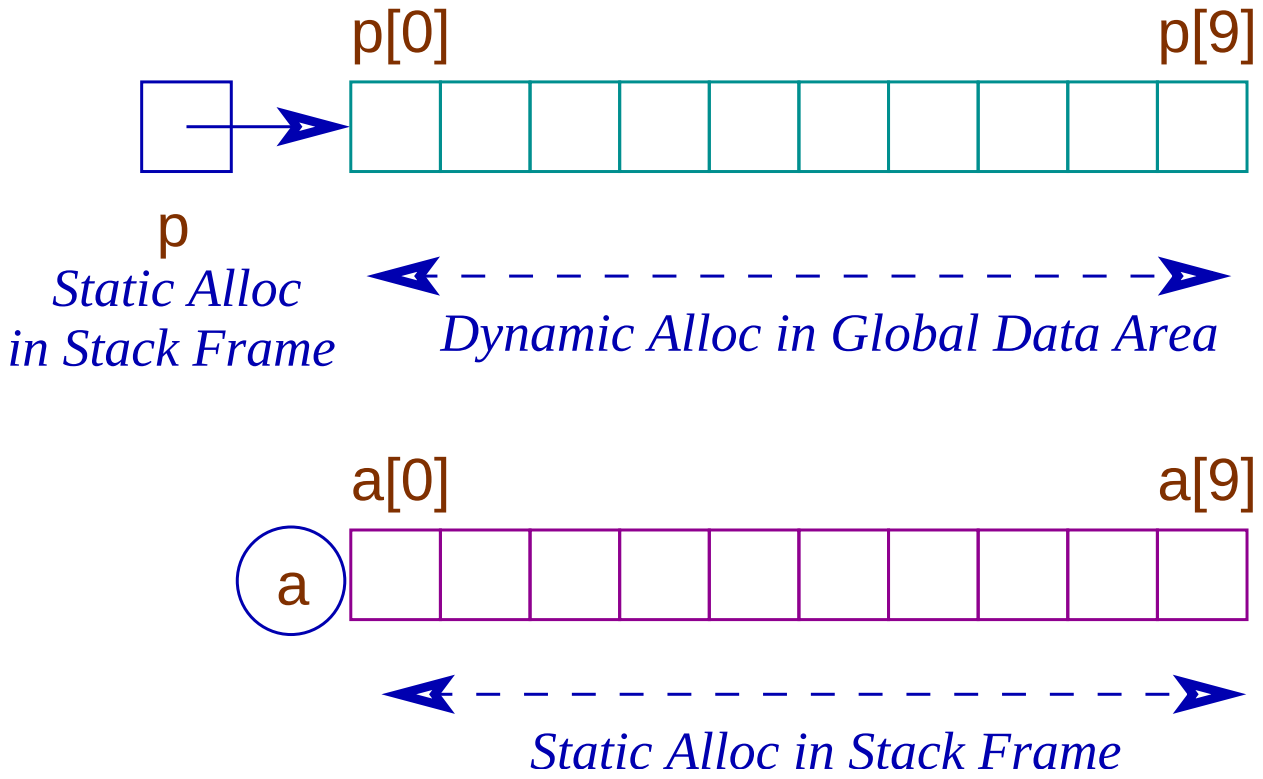
The code
`p = (int *) malloc(n*sizeof(int));`
allocates a contiguous memory area for `n` locations of type `int`. It returns the starting address as a `void *` pointer. The pointer is casted to the type `int *` and is assigned to the variable `p`.

Accessing the Area

The area can be accessed as a 1-D array of `int` due to the equivalence of `p[i]` and `*(p+i)`.

The differences are, `p` is a variable and the array may be lost if `p` is assigned a different address. The space is allocated in the global data area (heap) and is available from other parts of the program.

```
... what(...) {
    int   a[10], *p ;
    p = (int  *) malloc(10*sizeof(int)) ;
```



Allocation in Different Areas

```
#include <stdio.h>
#include <stdlib.h>
int g[5]; // global
int main()
{
    int a[5], *p, n ;
    static int s[5] ;

    scanf("%d", &n);
    int b[n];
    p=(int *)malloc(20) ;
```

```
    printf("g: %p\ts: %p\tp: %p\ta: %p\tb: %p\n",
            g, s, p, a, b) ;
    return 0;
} // memoryArea.c
```

Output

```
$ cc -Wall memoryAlloc.c
$ ./a.out
$ 100
g:  0x804a03c s:  0x804a028 p:
0x8267008 a:  0xbfc4b268 b:  0xbfc4b0a0
```

Note

The allocated space for the local array `a[]` and `b[]` (on stack) are far away from the global array `g[]` and the local static array `s[]` (data area). The dynamically allocated memory pointed by `p` is again at a different region called heap.

Dynamic Allocation of 2-D Array $n \times 5$

We can use the pointer `int (*q)[5]` to allocate space for a 2-D array of n rows and 5 columns.

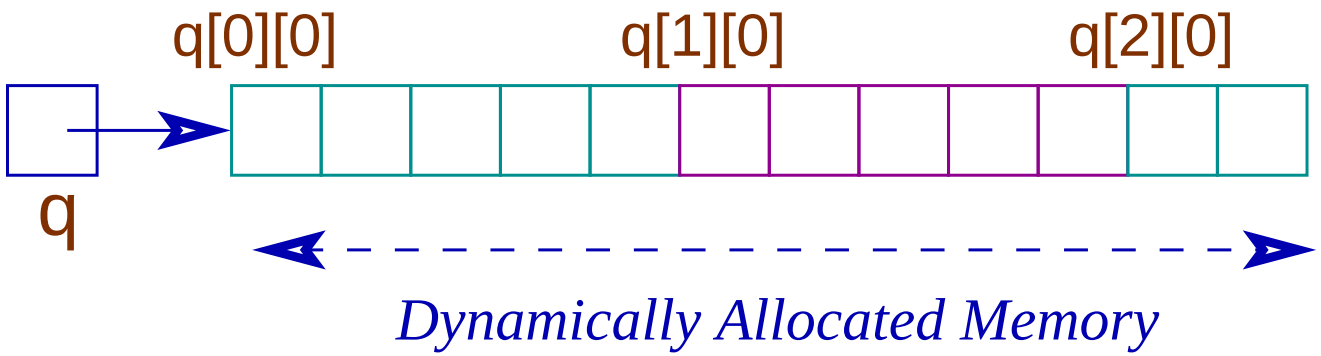
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int  (*q)[5],rows,i,j;

      printf("Enter the number of Rows: ") ;
      scanf("%d", &rows);
      q=(int (*)[5])malloc(rows*5*sizeof(int));
      for(i=0; i<rows; ++i)
          for(j=0; j<5; ++j) q[i][j]=2*i+3*j ;
      for(i=0; i<rows; ++i) {
          for(j=0; j<5; ++j)
             printf("%d ", q[i][j]); printf("\n");
```

```
    }
  return 0;
} // dynamic2D1.c
```

Output

```
$ cc -Wall dynamic2D1.c
$ ./a.out
Enter the number of Rows:  3
0 3 6 9 12
2 5 8 11 14
4 7 10 13 16
```

```
... what(...) {
    int  (*q)[5] ;
    q=(int (*)[5])malloc(rows*5*sizeof(int))
```



$n \times 5$ 2-D Array

- `q` points to the 0^{th} row of 5-element array.
- `q+i` points to the i^{th} row of the 5-element array.
- `*q` is the 0^{th} row, address of `q[0][0]` i.e. `&q[0][0]`.
- `*q+j` is the address of `q[0][j]`, `&q[0][j]`.

$n \times 5$ 2-D Array

- `*(q+i)+j` is the address of `q[i][j]`, `&q[i][j]`.
- `**q` is `q[0][0]`.
- `*(*q+j)` is `q[0][j]`.
- `*(*(q+i)+j)` is `q[i][j]`.

Note

ISO C99 has many features related to array of variable length that we shall not discuss.

Variable Length Automatic Array: An Example

```
#include <stdio.h>
void vla(int r, int c, int a[r][c]);
int main() // varLenArray.c
{
    int row, col;
    printf("Enter row and column numbers: ");
    scanf("%d%d", &row, &col);
    int x[row][col], i, j;
    for(i=0; i<row; ++i)
```

```
        for(j=0; j<col; ++j) x[i][j] = i+j;
    vla(row, col, x);
    return 0;
}
void vla(int r, int c, int a[r][c]){
    int i, j;
    for(i=0; i<r; ++i){
        for(j=0; j<c; ++j) printf("%d ", a[i][j])
        putchar('\n');
    }
}
```

Dynamic Allocation of Using `int *r[3]`

We can allocate a 2-D like structure with three rows and variable number of columns using `int *r[3]`. In fact number of elements in different rows may also be different.

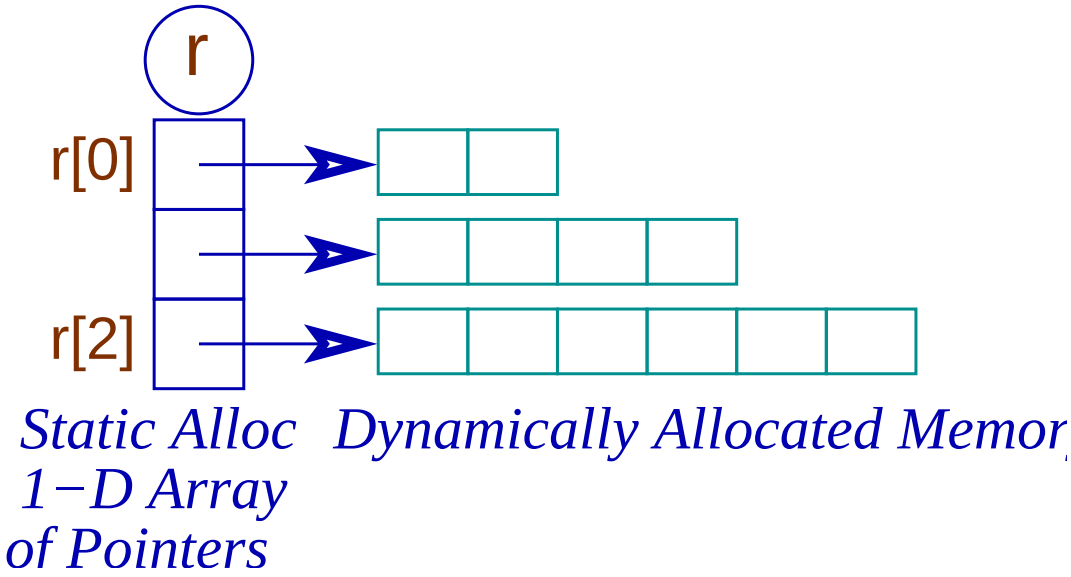
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int  *r[3], i, j;
    for(i=0; i<3; ++i) {
        int col = 2*(i+1) ;
        r[i] = (int *) malloc(col*sizeof(int)) ;
        for(j=0; j<col; ++j) r[i][j] = i+j ;
    }
    for(i=0; i<3; ++i) {
        int col = 2*(i+1) ;
        for(j=0; j<col; ++j)
            printf("%d ", r[i][j]) ; printf("\n");
```

```
    }
    return 0;
} // dynamic2D2.c
```

Output

```
$ cc -Wall dynamic2D2.c
$ ./a.out
0 1
1 2 3 4
2 3 4 5 6 7
```

int *r[3] ;


r
r[0]
r[2]
Static Alloc
Dynamically Allocated Memor
1−D Array
of Pointers

Note

`r[i]` is the i^{th} pointer stores the address of the 0^{th} element of the i^{th} row. So `r[i] + j` is the address of the j^{th} element of the i^{th} row and `*(r[i] + j)`, same as `r[i][j]`, is the j^{th} element of the i^{th} row.

Dynamic Allocation of $r \times c$ Array

We can allocate a 2-D array of variable number of rows and columns, where both the number of rows and the number of columns are inputs.

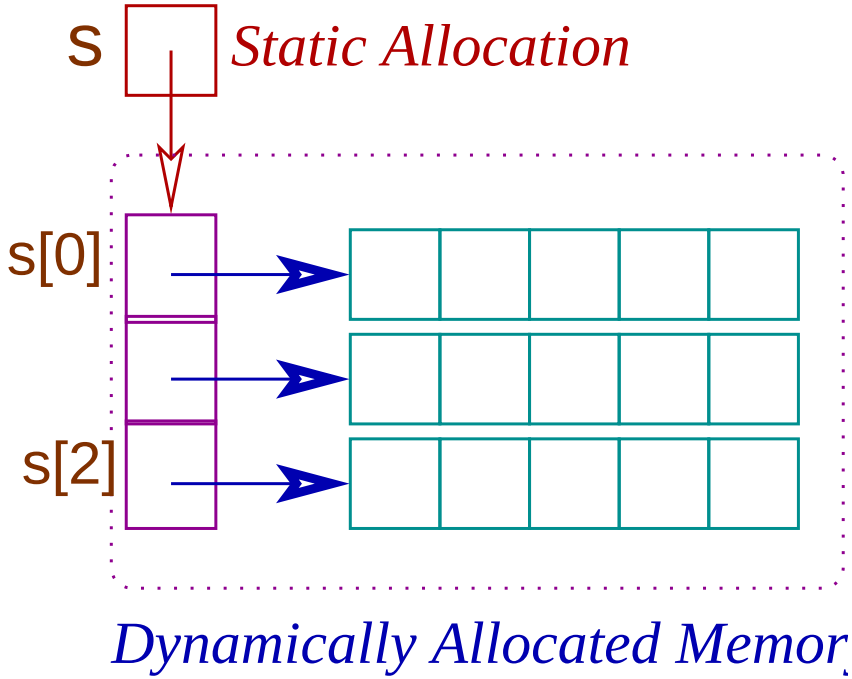
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int  **s, row, column, i, j;

    printf("Enter Row & Column:\n") ;
    scanf("%d%d", &row, &column) ;
    s = (int **) malloc(row*sizeof(int *)) ;
    for(i=0; i<row; ++i) {
        s[i] = (int *) malloc(column*sizeof(int)) ;
        for(j=0; j<column; ++j) s[i][j] = i+j ;
    }
```

```
    for(i=0; i<row; ++i) {
        for(j=0; j<column; ++j)
            printf("%d ", s[i][j]) ;
        printf("\n") ;
    }
    return 0;
} // dynamic2D3.c
```

Output

```
$ cc -Wall dynamic2D3.c
$ ./a.out
Enter Row & Column:
3 5
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
```

int **s ;



Note

`s + i` is the address of the i^{th} element of the pointer array. `*(s + i)`, same as `s[i]`, is the i^{th} element of the pointer array that stores the address of the 0^{th} element of the i^{th} row. `s[i] + j` is the address of the j^{th} element of the i^{th} row. `*(s[i] + j)`, same as `s[i][j]` is the j^{th} element of the i^{th} row.

Related Other Functions

There are other related function for dynamic allocation of memory.
`void *calloc(int numOfElements, int size)`. Here we can specify the number of elements of particular sizes. If there is a structure `student` and we like to allocate space for `n` students, we call
`(student *)calloc(n, sizeof(student))`.

Related Other Functions

As we mentioned earlier, the local memory manager manages the heap area. The library function `void free(void *)` releases the area no longer required by the program to the memory manager for reuse.

Related Other Functions

It may be necessary to change the size of the allocated area. The function `void *realloc(void *p, int n)` changes the size of the memory block pointed by `p` to `n` bytes. If the value of `n` is greater than or equal to the already allocated size, the original data is unaltered but the new memory is uninitialized.

Returning a Dynamic Array

```
#include <stdio.h>
#include <stdlib.h>
int *ret1DArray(int);
int main()
{
    int n, *p, i ;

    printf("Enter a +ve integer: ");
    scanf("%d", &n);
    p = ret1DArray(n);
    printf("\nData are: ");
    for(i=0; i<n; ++i) printf("%d ", p[i]);
```

```
    putchar('\n');
    return 0;
}
int *ret1DArray(int n){
     int *p = (int *)malloc(n*sizeof(int)), i;

     printf("Enter %d data: ", n);
     for(i=0; i<n; ++i) scanf("%d", &p[i]);
     return p;
} // retDynArray1.c
```

Use of realloc()

```
#include <stdio.h>
#include <stdlib.h>
int *ret1DArray(int *);
int main()
{
    int n, *p, i ;

    printf("Enter a +ve integer: ");
    scanf("%d", &n);
    p = ret1DArray(&n);
    printf("\nData are: ");
    for(i=0; i<n; ++i) printf("%d ", p[i]);
```

```
    putchar('\n');
    return 0;
}
#define MAX 5
int *ret1DArray(int *nP){
     int n, max = MAX,
         *p = (int *)malloc(MAX*sizeof(int));

     printf("Enter data, terminate by Ctrl+D: ");
     n = 0;
     while(scanf("%d", &p[n]) != EOF){
       ++n ;
       if(n == max)
         p=(int*)realloc(p,sizeof(int)*(max += MAX));
```

```
    }
    *nP = n ;
    return p;
} // retDynArray2.c
```