Dynamic Allocation of Memory Space

Run-Time Allocation of Space

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

Run-Time Allocation of 1D Array on Stack

- We can allocate one or multi-dimensional array on the run-time stack whose size is unknown at the compilation time (ISO C99).
- But this data is local to a function. So it is not available beyond the life-time of the function.

Run-Time Allocation of 1D Array on Stack

```
#include <stdio.h>
int main(){ // 1Darray1.c
    int size, i;
    printf("Enter the number of elements: ");
    scanf("%d", &size);
    int a[size];
     printf("Enter %d integers: ", size);
    for(i=0; i<size; ++i) scanf("%d", &a[i]);
    printf("Data in reverse order: ");
    for(i=size-1; i>=0; --i) printf("%d ", a[i]);
    putchar('\n');
    return 0;
}
```

Not Permitted

```
#include <stdio.h>
int main() // 1Darray1a.c
{
    int size, a[size], i; // not permitted
              // works if size is initialized
              // with some value.
    printf("Enter the number of elements: ");
    scanf("%d", &size);
    .....
```

Not Permitted

```
$ cc -Wall 1Darray1a.c
$ a.out
Segmentation fault (core dumped)
```

Dynamic 1D Array as Parameter

Parameter passing is identical.

```
#include <stdio.h>
int elmSum(int x[], int size){ // 1Darray2.c
    int sum=0, i;

    for(i=0; i<size; ++i) sum += x[i];
    return sum;
}
int main() // 1Darray2.c
{
    int size, i;
    printf("Enter the number of elements: ");
    scanf("%d", &size);
    int a[size];

    printf("Enter %d integers: ", size);
    for(i=0; i<size; ++i) scanf("%d", &a[i]);
    printf("Input data: ");
    for(i=0; i<size; ++i) printf("%d ", a[i]);
    putchar('\n');
    printf("Sum of elements: %d\n", elmSum(a,size));

    return 0;
}
```

Run-Time Allocation of 2D Array on Stack

```
#include <stdio.h>
int main() // 2Darray1.c
{
    int row, col, i, j;

    printf("Enter the number of rows: ");
    scanf("%d", &row);
    printf("Enter the number of cloumns: ");
    scanf("%d", &col);
    int a[row][col];
    printf("Enter data in row-major order:\n" );
    for(i=0; i<row; ++i)
```

```
        for(j=0; j<col; ++j)
            scanf("%d", &a[i][j]);
    printf("Input Data:\n");
    for(i=0; i<row; ++i){
        for(j=0; j<col; ++j)
            printf("%d ", a[i][j]);
        putchar('\n');
    }

    return 0;
}
```

Run-Time Allocation of 2D Array on Stack

```
int main() // 2Darray1a.c
{
    int row=0, col=0, a[row][col], i, j;
        .......
        // Acceptable, but ....
```

Run-Time 2D Array as Parameter

```
#include <stdio.h>
int sumOfElm(int col, int x[][col], int row){
    int i, j, sum = 0;   // 2Darray2.c

    for(i=0; i<row; ++i)
        for(j=0; j<col; ++j) sum += x[i][j];
    return sum;
}
int main() // 2Darray2.c
{
    int row, col, i, j;
```

```
printf("Enter the number of rows: ");
scanf("%d", &row);
printf("Enter the number of columns: ");
scanf("%d", &col);
int a[row][col];

printf("Enter data in row-major order:\n" );
for(i=0; i<row; ++i)
    for(j=0; j<col; ++j)
        scanf("%d", &a[i][j]);
printf("Input Data:\n");
for(i=0; i<row; ++i){
    for(j=0; j<col; ++j)
        printf("%d ", a[i][j]);
        putchar('\n');
    }
    printf("Sum of elements: %d\n",
                     sumOfElm(col, a, row));

    return 0;
}
```

Not Permitted

```
int sumOfElm(int x[][col], int col, int row){
```

```
$ cc -Wall 2Darray2a.c
2Darray2a.c:3:22: error: col undeclared
       here (not in a function)
2Darray2a.c: In function main:
2Darray2a.c:30:5: error: type of formal
       parameter 1 is incomplete
```

Global Array

```
#include <stdio.h>
#define MAX 100
int data[MAX];
 int sum(int);
int main() // global1.c
{
    int n, i;

    printf("Enter the number of data: ");
    scanf("%d", &n);
    printf("Enter %d data: ", n);
    for(i=0; i<n; ++i) scanf("%d", &data[i]);
     printf("Sum: %d\n", sum(n));
    return 0;
}
int sum(int n){
    if(!n) return 0;
    return data[n-1]+sum(n-1);
 }
```

Note

- The global array `data[MAX]` is available to both `main()` and `sum()`.
- It is not necessary to pass it as a parameter.
- Its size is fixed.

Not Permitted

```
#include <stdio.h>
int n, data[n];
int sum(int);
int main() // global2.c
{ ....
$ cc -Wall global2.c
global2.c:3:8: error: variably modified
                  data at file scope
```

Global Data

- It is necessary to allocate data area at run-time that is available across the function boundary.
- This requirement is met in C by allocating memory space in a region called heap.
- C library provides support for this allocation.

```
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 special global data area called heap. 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 a request to the OS for more heap space. OS supplies in multiples of a fixed chunk of memory block e.g. multiples of 4 Kbytes. The local memory management module of malloc() manages the available memory.

Pointer Variables and Array

We deal with different types of pointers to use the memory allocated by malloc(). Consider the following variable 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`. When `p` is incremented by `i` i.e. `p` $+$ `i` is computed, the value is `(unsigned)p + i` $\times$ `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 `(unsigned)q +` `i` $\times$ `5` $\times$ `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 `int 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 `(unsigned)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 $\texttt{s} + \texttt{i}$ is `(unsigned)s + i` $\times$ `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()   // pointVar.c
{
    int *p, (*q)[5], *r[3], **s;

    printf("sizeof(int): %lu\n", sizeof(int));
    printf("sizeof(int *): %lu\n", sizeof(int *));
    printf("sizeof(int [5]): %lu\n", sizeof(int [5]));
    printf("sizeof(int (*)[5]): %lu\n", sizeof(int (*)[5]))
    printf("sizeof(int **): %lu\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

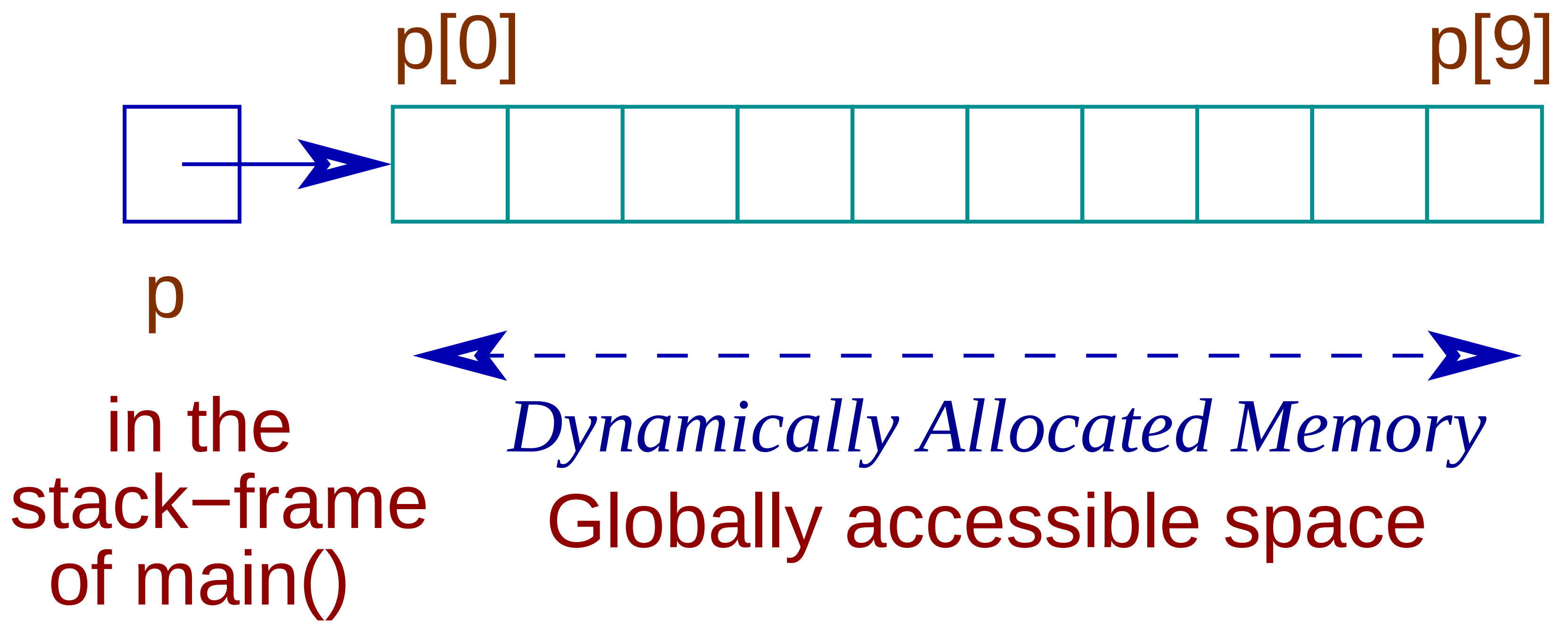
```
$ cc -Wall pointVar.c
$ ./a.out
sizeof(int):  4
sizeof(int *):  8
sizeof(int [5]):  20
sizeof(int (*)[5]):  8
sizeof(int **):  8

p:  0x2c5ff4 p+1:  0x2c5ff8
q:  0x80496d0 q+1:  0x80496e4
r:  0xbfe53600 r+1:  0xbfe53608
s:  0x80482b5 s+1:  0x80482bd
```

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()    // dynamic1D.c
{
    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

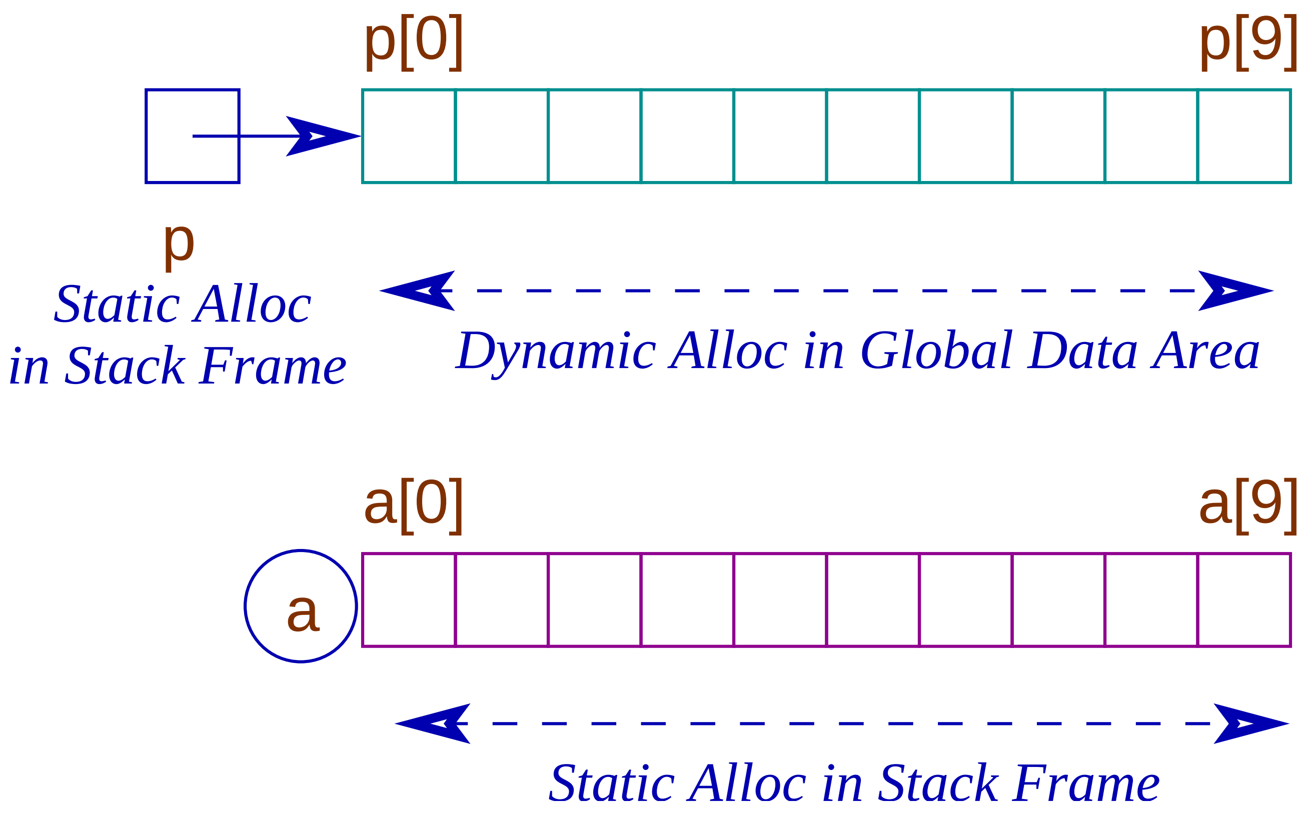
```
p = (int *) malloc(n*sizeof(int));
```

The function `malloc()` allocates a contiguous memory area of size `n`$\times$`sizeof(int)` bytes. 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)`. But unlike an array name, `p` is a variable, so the dynamically created array may be lost if `p` is assigned a different address-value. 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)) ;
```


p[0]
p[9]
p
Static Alloc in Stack Frame
Dynamic Alloc in Global Data Area
a[0]
a[9]
a
Static Alloc in Stack Frame

Allocation in Different Areas

```
#include <stdio.h>
#include <stdlib.h>
int g[5]; // global
int main()   // memoryArea.c
{
    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:  0x601060 s:  0x601040 p:  0x1190010
a:  0x7fff24f73180 b:  0x7fff24f73160
```

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.

Simple Use

```
#include <stdio.h>
#include <stdlib.h>
int *createArray(int n){ // dynamic1Da.c
     int *p,i, val=1;

     p = (int *)malloc(n*sizeof(int));
     for(i=0; i<n; ++i) {
         p[i] = val; val = 2*val;
     }
     return p;
}
```

```
int main()
{
    int *p, n, i ;

    printf("Enter a +ve integer: ");
    scanf("%d", &n);
    p = createArray(n);
    for(i=0; i<n; ++i) printf("%d ",p[i]);
    putchar('\n');
    return 0;
} // dynamic1Da.c
```

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 at most 5 columns.

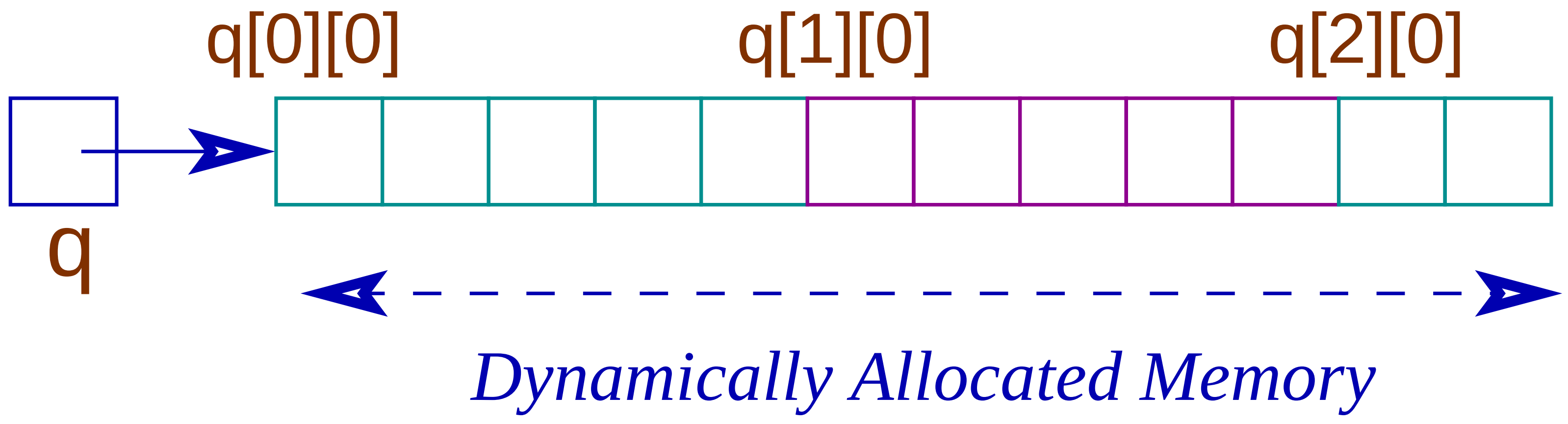
```
#include <stdio.h>
#include <stdlib.h>
int main() // dynamic2D1.c
{
    int  (*q)[5],rows,i,j;

      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]`.

Return Value

```
#include <stdio.h>
#include <stdlib.h>
int (*nBy5array(int row))[5]{ // dynamic2D1a.c
    int (*q)[5], i, j;
    q=(int (*)[5])malloc(row*5*sizeof(int));
      for(i=0; i<row; ++i)
          for(j=0; j<5; ++j) q[i][j]=2*i+3*j ;
    return q;
}
```

```
int main()
{
    int  (*q)[5],rows,i,j;

      printf("Enter the number of Rows: ") ;
      scanf("%d", &rows);
      q = nBy5array(rows);
      for(i=0; i<rows; ++i) {
          for(j=0; j<5; ++j)
             printf("%d ", q[i][j]); printf("\n");
      }
    return 0;
} // dynamic2D1.c
```

Parameter Passing

```
#include <stdio.h>
#include <stdlib.h>
#define MAXCOL 50
int sumAll(int q[][MAXCOL], int row, int col){
    int i, j, sum = 0;       // dynamic2D1b.c

    for(i=0; i<row; ++i)
        for(j=0; j<col; ++j)
            sum += q[i][j];
    return sum;
}
```

```
int main()
{
      int  (*q)[MAXCOL],rows, cols, i,j;

      printf("Enter the number of Rows: ") ;
      scanf("%d", &rows);
      printf("Enter the number of Columns <= %d: ",
                                    MAXCOL) ;
      scanf("%d", &cols);

      q=(int (*)[MAXCOL])malloc(rows*MAXCOL*
                                sizeof(int));
      for(i=0; i<rows; ++i)
          for(j=0; j<cols; ++j) q[i][j]=2*i+3*j ;
```

```
    printf("Data in the array:\n");
    for(i=0; i<rows; ++i) {
        for(j=0; j<cols; ++j)
           printf("%d ", q[i][j]);
           printf("\n");
    }
    printf("Sum of elements: %d\n",
                         sumAll(q, rows, cols));
    return 0;
} // dynamic2D1.c
```

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

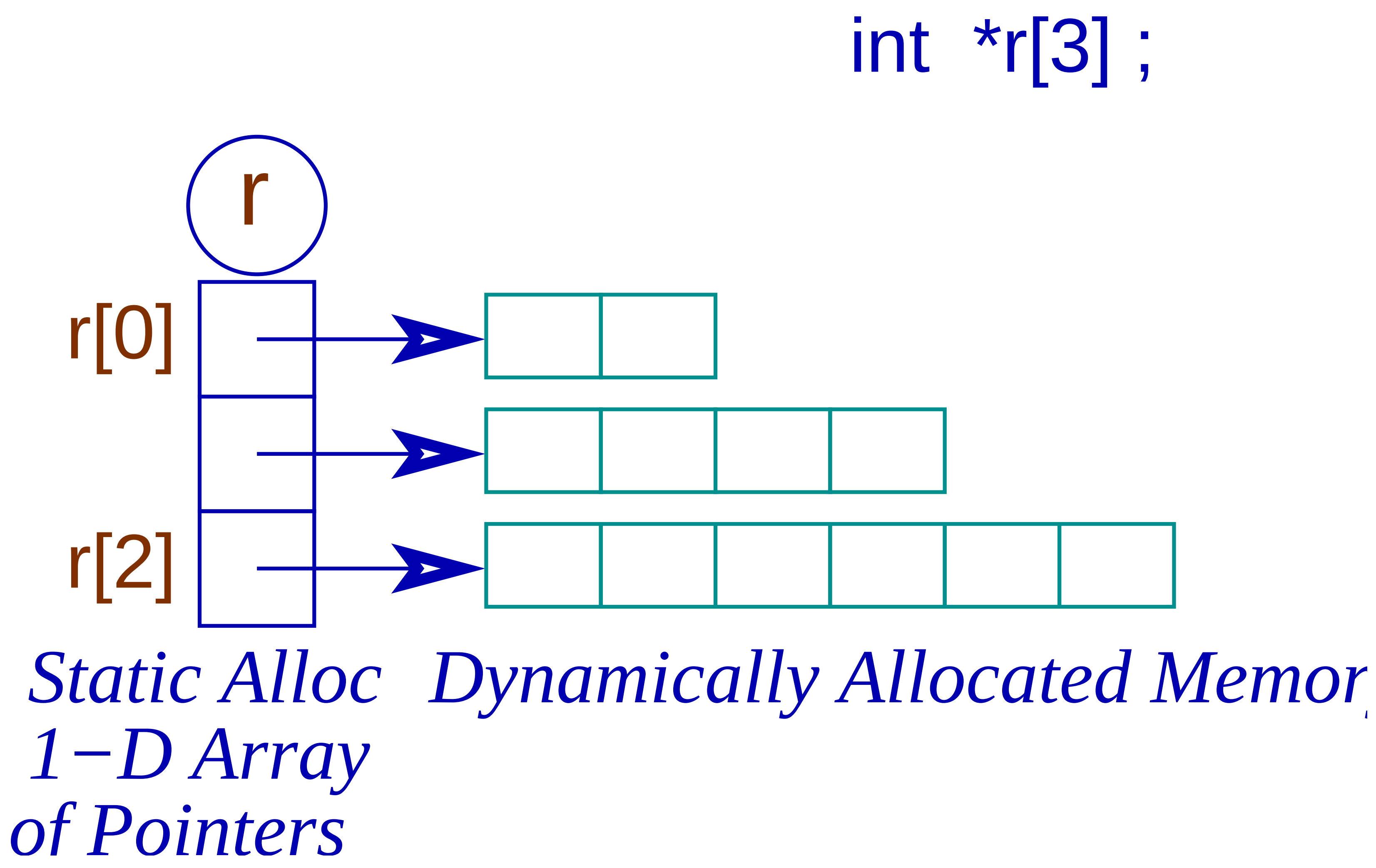
We can allocate a 2D array like structure with at most three rows and variable number of columns using `int *r[3]`. In fact number of elements in different rows may also be different.

```
#include <stdio.h>
#include <stdlib.h>
int main() // dynamic2D2.c
{
    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
1−D Array
of Pointers
Dynamically Allocated Memor

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.

Parameter Passing

```
#include <stdio.h>
#include <stdlib.h>
#define MAXROW 50
void printArray(int *r[], int row, int col){
     int i, j;               // dynamic2D2a.c

     for(i=0; i<row; ++i) {
        for(j=0; j<col; ++j)
            printf("%d ", r[i][j]) ;
        printf("\n");
    }
}
```

```
int main() {
    int  *r[MAXROW], row, col, i, j;

    printf("Enter the number of rows <= %d: ", MAXROW);
    scanf("%d", &row);
    printf("Enter the number of cols: ");
    scanf("%d", &col);
    for(i=0; i<row; ++i) {
        r[i] = (int *) malloc(col*sizeof(int)) ;
        for(j=0; j<col; ++j) r[i][j] = i+j ;
    }
    printArray(r,row, col);
    return 0;
} // dynamic2D2a.c
```

Dynamic Allocation of $r \times c$ Array

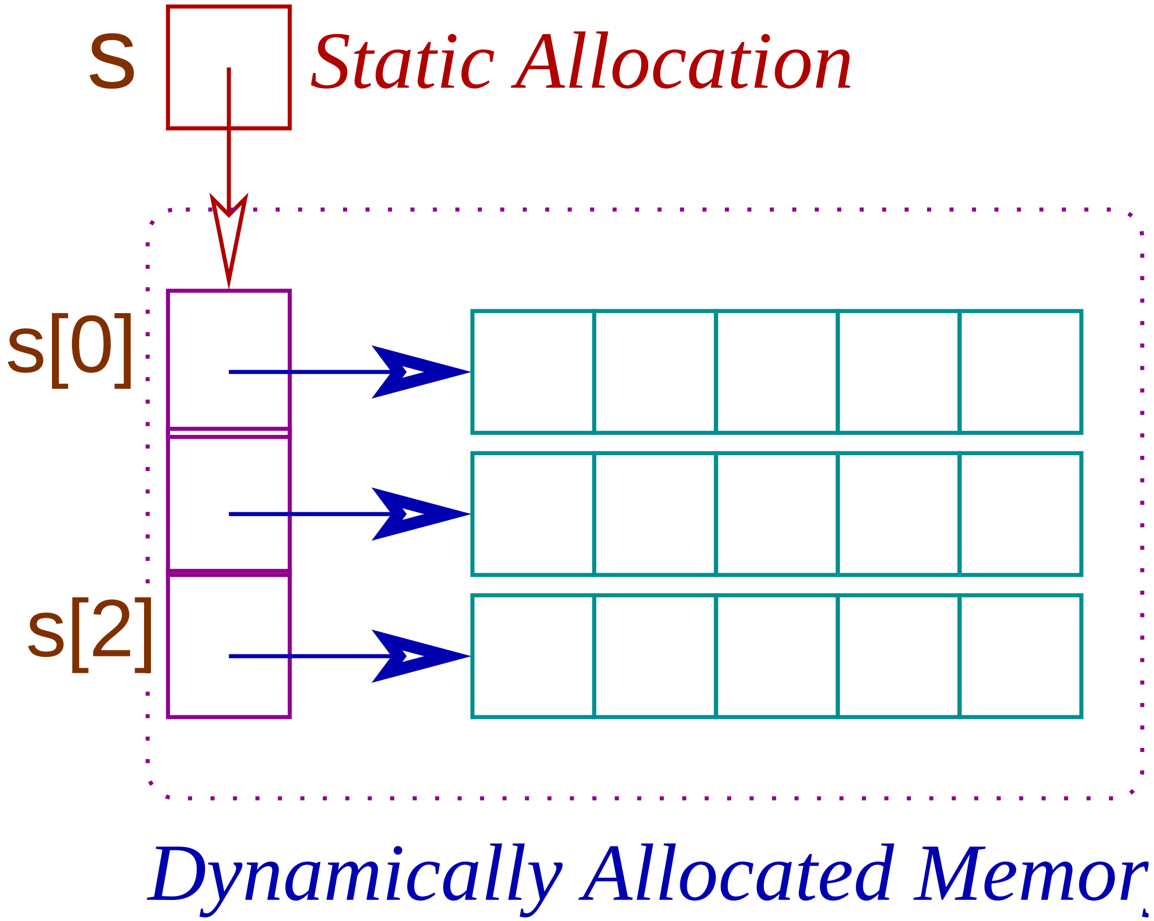
We can allocate a 2-D array of variable number of rows and columns, where both the number of rows and the number of columns are inputs.

```
#include <stdio.h>
#include <stdlib.h>
int main() // dynamic2D3.c
{
    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.

Parameter Passing

```
#include <stdio.h>
#include <stdlib.h>
void printArray(int **s, int row, int col){
     int i, j ;   // dynamic2D3.c

     for(i=0; i<row; ++i) {
         for(j=0; j<col; ++j)
            printf("%d ", s[i][j]) ;
     printf("\n") ;
    }
}
```

```
int main(){  // dynamic2D3.c
    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 ;
    }
    printArray(s, row, column);
    return 0;
} // dynamic2D3a.c
```

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.

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
```