Pointer Arithmetic

- Assume a 2D array `a[n][n]` has been initialized properly.
- Printing column `k` could be realized by:

```
L1   p = &a[0];
L2   k = 3;
L3   // prints row 0 and column 3
L4   for (i = 0; i < n; i++)
L5       printf("%5d", (*(p+i))[k]);
L6   printf("\n");
```

- Since, `a[0]` is an address `*p` stores an address (line L1).
- `a[0][0]` is the first element of row `a[0]`, similarly, `(*p)[0]` is the first element of row zero of array `a`.

Pointer Arithmetic

2D Arrays & Pointers

```
#include <stdio.h>
int main() {
    int a[10][10];
    int n = sizeof(a[0])/sizeof(a[0][0]);
    int i, j, *ptr;
    int (*p)[10];  // Column p
    for (i = 0; i < n; i++)
        for (j = 0, ptr = a[i];  ptr < a[i] + n; ptr++, j++)
            *ptr = i + j;
    i = 5;
    for (p = &a[0];  p < &a[n]; p++)
        (*p)[i] = 0;
    for (i = 0; i < n; i++) {
        for (ptr = a[i];  ptr < a[i] + n; ptr++)
            printf("%5d", *ptr);
        printf("\n");
    }
}
```

Pointer Arithmetic

2D Arrays & Pointers

- In order to use a pointer as a 2D array, first a memory block should be set aside.
- `void *malloc(size*sizeof(type))` used for this: allocates space for an object whose size in bytes is an argument to malloc.
- On successful allocation it return a void pointer to allocated space, otherwise null pointer is returned.
- Using `malloc` we allocate storage for an array of element of type T in memory and return a pointer to the array.
- Then the pointer can be used as a 2D array.
- After using the space, the space should be deallocated.
- Deallocation is done by calling `free(ptr)`

Pointer Arithmetic

2D Arrays & Pointers

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, j, a[5][5];
    int n = sizeof(a[0])/sizeof(a[0][0]), m = n*n;
    int (*pa)[m];

    pa =  malloc(m*sizeof(int)); // Returns a void pointer
    for (i = 0; i < n; i=i++)
         for (j = 0; j < n; j++)
              pa[i][j] = i*j ;
    printf("2D␣array␣with␣pointer\n");
    for (i = 0; i < n; i++) {
         for (j = 0; j < n; j++)
              printf("%5d", pa[i][j]);
         printf("\n");
    }
}
```

Summary and Pitfalls

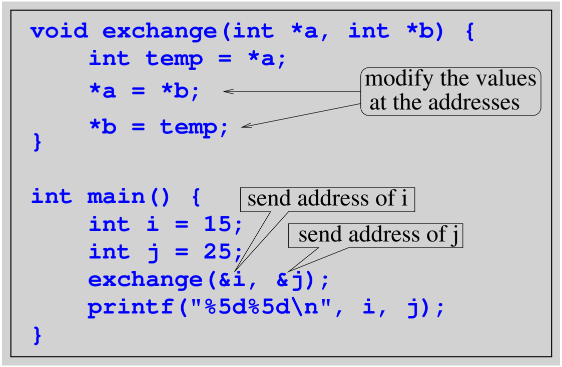
Referencing and Deferencing

```
int i;
int *ip; // declaring a pointer variable of a given type

ip = &i; // assigning address to a pointer variable

/*assigning value to variable to which a pointer
 * variable points to */
*ip = 15;
```

- Use & to get address of a variable.
- Use * To get value of a variable referenced by a pointer
- Use * to declare a pointer variable.

Summary and Pitfalls

Passing Pointer Arguments



```
void exchange(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int i = 15;
    int j = 25;
    exchange(&i, &j);
    printf("%5d%5d\n", i, j);
}
```

Summary and Pitfalls

Some of the Pitfalls

No attempt be made to dereference an unassigned pointer. It causes immediate crash. Eg:

```
int *p;
*p = 25; //physical location needed
```

Using pointer variable before assigning **lvalue**, would cause eventual crash.

```
int x;
int *px;
*px = x; // No address is assigned to px yet.
```

Summary and Pitfalls

Some of the Pitfalls

Following two are not equivalent increments

```
*p += 1; // increment value
*p++; // increments address
```

Following two assignment are legal but have different meanings.

```
   *p = *q;
   p = q;
```