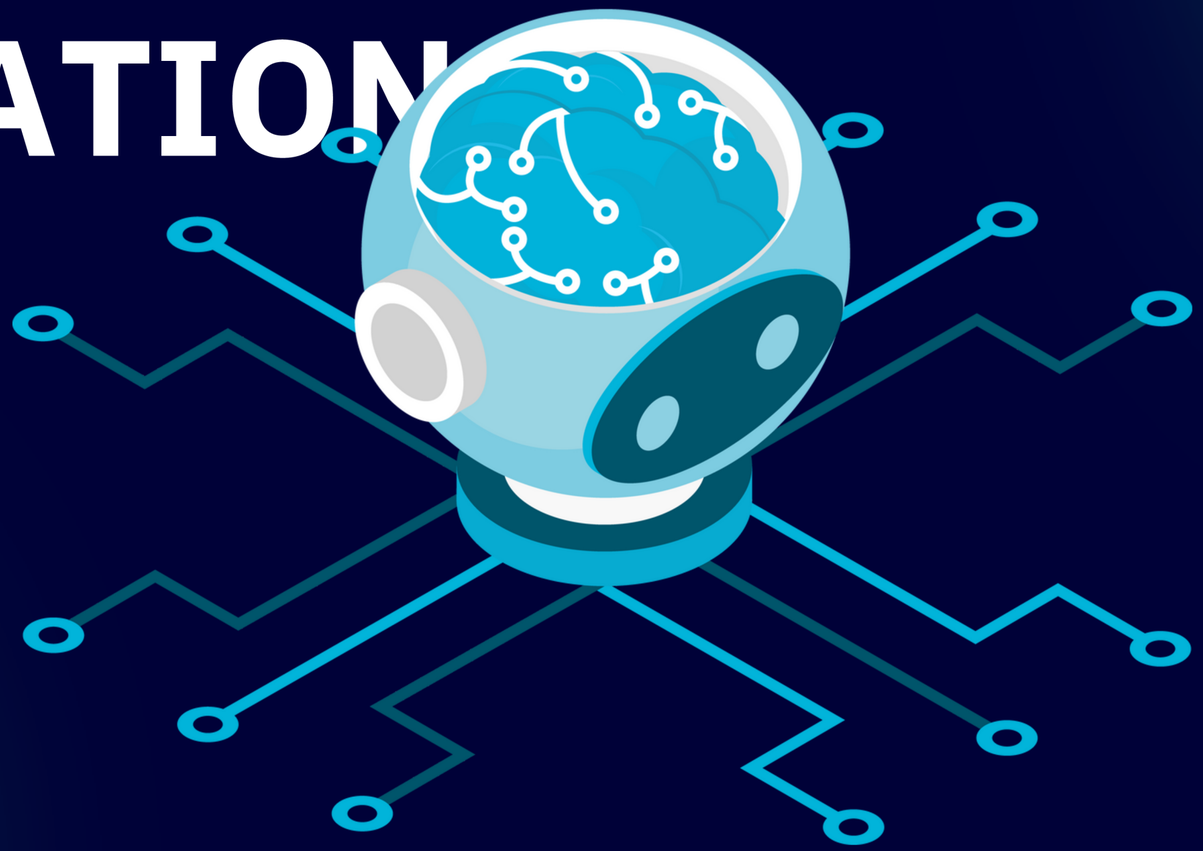
DYNAMIC MEMORY ALLOCATION



Memory Allocation

Memory allocation is the process of reserving space in a computer's memory to store data while a program is running. This space is used to hold variables, data structures, and other information that the program needs to operate.

Two ways of Memory Allocation:

1 **Static Memory Allocation**

Memory is allocated at compile time, before the program starts executing.

2 **Dynamic Memory Allocation**

Memory is allocated during program execution, as needed.

Static Memory Allocation

The size of the memory block is fixed and determined before the program runs.

The size of the memory block cannot be changed during program execution.

Example:

```
int main(void){
        int arr[6]={1,2,3,4,5,6};
}
```

Memory is allocated at compile time, before the program starts executing.

Dynamic Memory Allocation

In the dynamic memory allocation, the memory is allocated to a variable or program at the run time.

The only way to access this memory is through pointer.

Advantages OF Dynamic Memory Allocation are:

1. **Flexibility**

   Enables allocating memory based on runtime needs rather than fixed sizes.

2. **Efficiency for large data sets**

   Dynamically allocates memory as needed, avoiding excessive pre-allocation and managing large datasets more effectively.

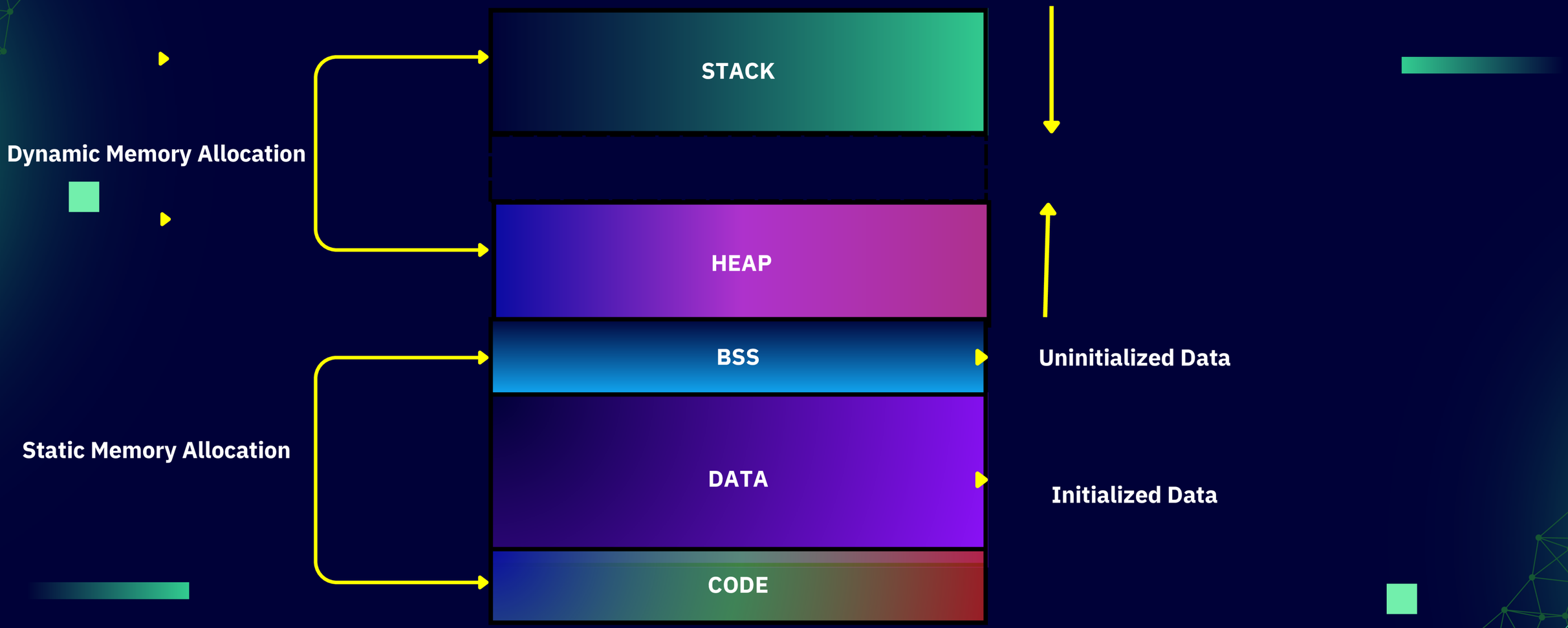
3. **Memory Reusability**

   Frees and reuses memory for different parts of the program as required.

4. **Avoiding Memory Waste (No Fixed Limits)**

   Eliminates unused pre-allocated memory by allocating as needed.

Memory Layout In C



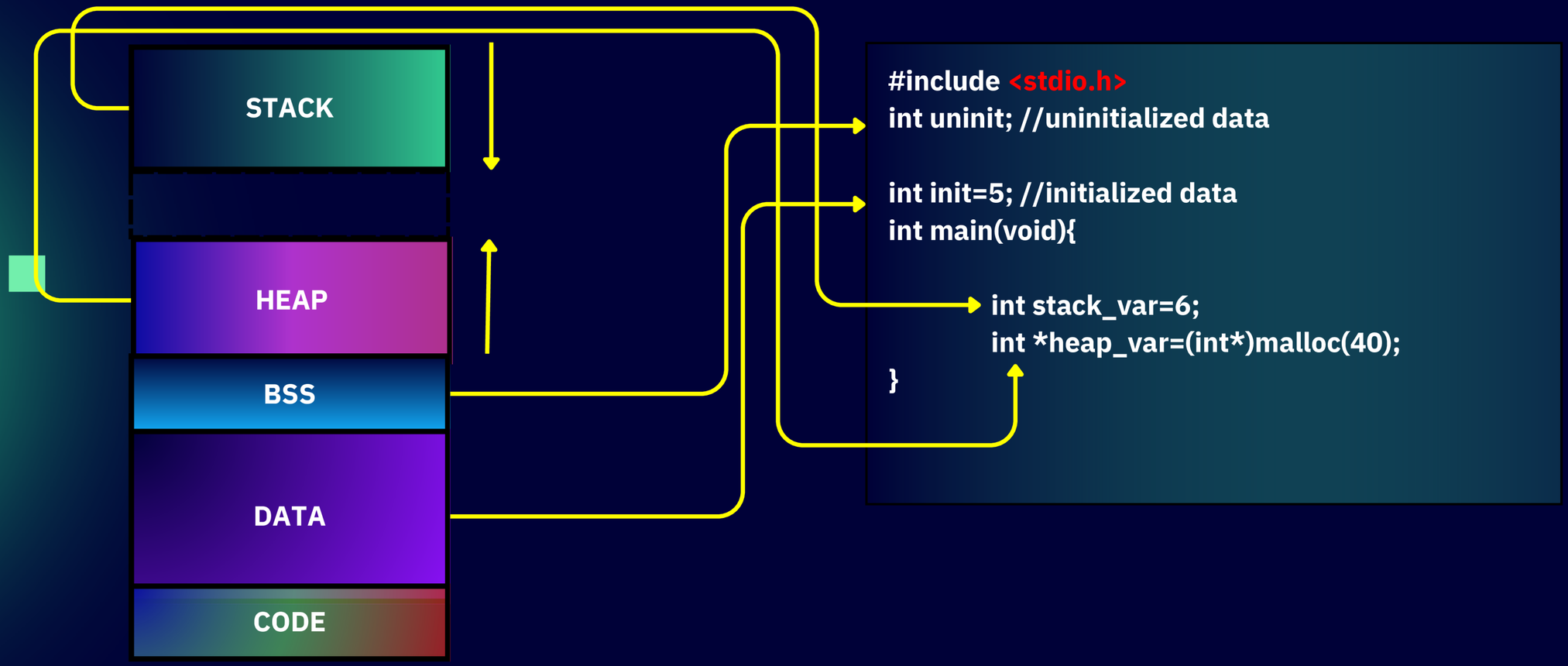
Memory Layout In C

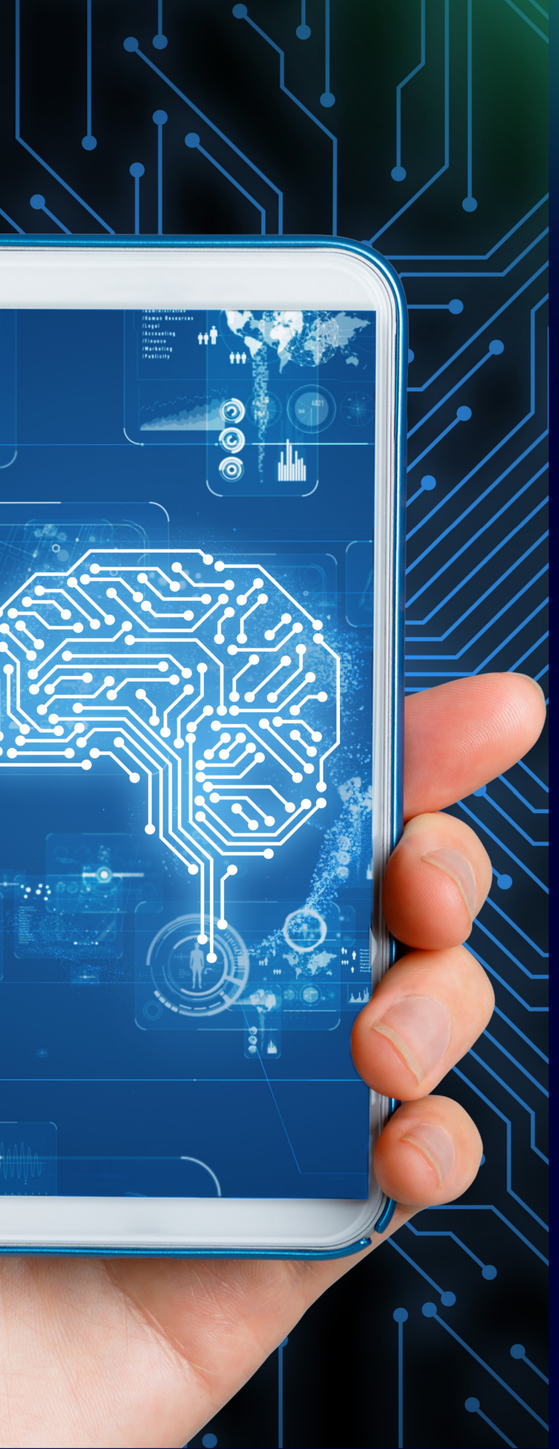

```c
#include <stdio.h>
int uninit; //uninitialized data

int init=5; //initialized data
int main(void){

        int stack_var=6;
        int *heap_var=(int*)malloc(40);
}
```

Memory Layout In C





FUNCTIONS USED IN DYNAMIC MEMORY ALLOCATION

In C, **dynamic memory allocation functions** are part of the Standard Library (**stdlib.h**). Below are the details of the functions available in this library:

malloc()

Allocates a block of memory of specified size (in bytes) and returns a pointer to it.

calloc()

Allocates memory for an array of num elements, initializes it to zero, and returns a pointer to the memory.

realoc()

Resizes a previously allocated memory block to a new size.

free()

Deallocates memory that was previously allocated by malloc, calloc, or realloc.

malloc()

malloc() function is used for allocating block of memory at runtime. This function reservers a block of memory of given size and returns a pointer of type void. It is a built-in function declared in the header file **<stdlib.h>**.

Syntax:

```
(void*) malloc(size_t size);
```

Example of malloc()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* ptr = (int*)malloc(3 * sizeof(int)); // Allocating memory
for 3 integers

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Assign values
    ptr[0] = 10;
    ptr[1] = 20;
    ptr[2] = 30;

    // Print values
    for (int i = 0; i < 3; i++) {
        printf("ptr[%d] = %d\n", i, ptr[i]);
    }

    free(ptr); // Free memory
    return 0;
}
```

Output:

ptr[0] = 10

ptr[1] = 20

ptr[2] = 30

calloc()

calloc() allocates memory for an array of num elements and initializes all bytes to zero. It is a built-in function declared in the header file **<stdlib.h>**.

Syntax:

```
(void*) calloc(size_t num, size_t size);
```

Example of calloc()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* ptr = (int*)calloc(3, sizeof(int)); // Allocating memory for 3
integers, initialized to 0

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Print default values
    for (int i = 0; i < 3; i++) {
        printf("ptr[%d] = %d\n", i, ptr[i]); // Outputs 0 for all elements
    }

    free(ptr); // Free memory
    return 0;
}
```

Output:

ptr[0] = 0

ptr[1] = 0

ptr[2] = 0

realloc()

realloc() resizes a previously allocated memory block (ptr) to the specified new size.. It is a built-in function declared in the header file **<stdlib.h>**.

Syntax:

```
(void*) realloc(void* ptr, size_t size);
```

Example of realloc()

```
#include <stdio.h>
#include <stdlib.h>ö
int main() {
    int* ptr = (int*)malloc(2 * sizeof(int)); // Allocate memory for 2
integers
   if (ptr == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
   // Assign values
   ptr[0] = 10;
   ptr[1] = 20;
   // Resize memory to hold 4 integers
    ptr = (int*)realloc(ptr, 4 * sizeof(int));
   if (ptr == NULL) {
      printf("Memory reallocation failed!\n");
      return 1;
   }
   // Assign additional values
   ptr[2] = 30;
   ptr[3] = 40;
   // Print all values
   for (int i = 0; i < 4; i++) {
      printf("ptr[%d] = %d\n", i, ptr[i]);
   }
   free(ptr); // Free memory
   return 0;
}
```

Output:

ptr[0] = 10

ptr[1] = 20

ptr[2] = 30

ptr[3] = 40

free()

free() deallocates memory that was previously allocated using malloc, calloc, or realloc. It is a built-in function declared in the header file **<stdlib.h>**.

Syntax:

```
(void*) free(void* ptr);
```

Example of free()

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int* ptr = (int*)malloc(3 * sizeof(int)); // Allocating memory for 3 integers
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
    // Assign values
    ptr[0] = 1;
    ptr[1] = 2;
    ptr[2] = 3;

    // Print values
    printf("Before freeing memory:\n");
    for (int i = 0; i < 3; i++) {
        printf("ptr[%d] = %d\n", i, ptr[i]);
    }

    free(ptr); // Free memory
    printf("Memory freed successfully.\n");

    return 0;
}
```

Output:

Before freeing memory:
ptr[0] = 1
ptr[1] = 2
ptr[2] = 3
Memory freed successfully.

Conclusion

In conclusion, dynamic memory allocation is a cornerstone of modern programming, offering unmatched flexibility, efficient resource management, and the ability to create and manage complex data structures. By mastering these techniques, developers can build software that is not only more efficient and robust but also scalable and adaptable to diverse and changing requirements.

Any Queries??

Thank YOU!!