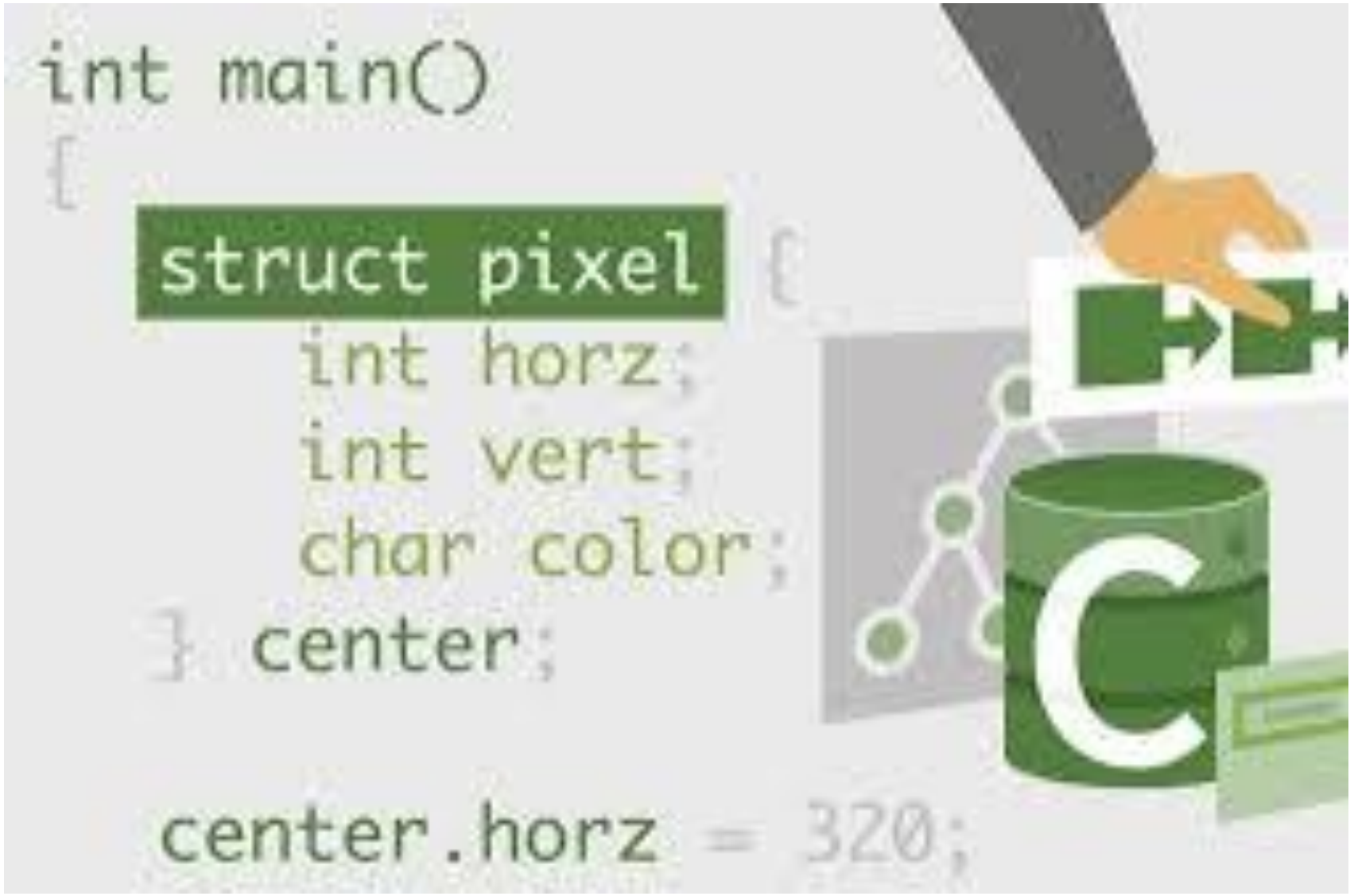
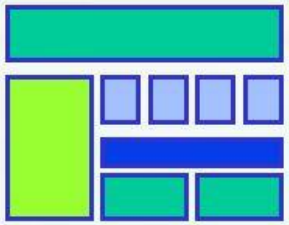
STRUCTURES & MEMORY ALLOCATION

Lesson 05

Structures


int main()
{
struct pixel {
int horz;
int vert;
char color;
} center;
center.horz = 320;

struct examples



- Example:

```
struct StudentInfo{
    int Id;
    int age;
    char Gender;
    double CGA;
};
```

The “StudentInfo” structure has 4 members of different types.

- Example:

```
struct StudentGrade{
    char Name[15];
    char Course[9];
    int Lab[5];
    int Homework[3];
    int Exam[2];
};
```

The “StudentGrade” structure has 5 members of different array types.

struct

Structurile grupează date de tipuri diferite.
Componentele unei structuri se numesc **câmpuri**.
La declararea unei structuri se precizează tipurile, identificatorii elementelor componente şi numele structurii.
Forma generală de declarare a unei structuri:

```
struct identificator_tip_structura {
        lista_de_declaratii_membri;
      } lista_identificatori_variabile;
```

în care:

Struct cuvânt cheie (obligatoriu)
identificator_tip_structura numele noului tip (poate lipsi)
lista_de_declaratii_membri listă în de tipuri şi identificatori
lista_identificatori_variabile listă cu identificatorii variabilelor de tip declarat.

Exemplu

Se defineşte structura data, cu câmpurile: zi, luna, an:

```
struct data {
   int zi;
   int luna;
   int an;
}
```

și variabilele:

```
struct data data_nasterii,
data_angajarii;
```

Accesarea componentelor

- fie variabila: **struct data data_nasterii;**
- Accesarea câmpurilor:
- **data_nasterii -> an**
- **data_nasterii -> luna**
- **data_nasterii -> zi**
- sau
- **data_nasterii.an**
- **data_nasterii.luna**
- **data_nasterii.zi**

Accesarea componentelor

- fie variabila: **struct data da_nast[20];**
- Accesarea câmpurilor:
- **da_nast[i] -> an**
- **da_nast[i] -> luna**
- **da_nast[i] -> zi**
- sau
- **da_nast[i].an**
- **da_nast[i].luna**
- **da_nast[i].zi**

Structures within Structures

[*] exemplu_struct_001.cpp

```cpp
#include <stdio.h>
struct student
{
    int student_id;
    char name[20];
    float average;
    struct date
        {
            int day;
            int month;
            int year;
        } dob;
} faf[90];
int n = 3, i;
int main()
{
    for( i = 0; i < n; i++)
        scanf("%d %s %f %d %d %d", &faf[i].student_id, &faf[i].n
        &faf[i].average, &faf[i].dob.day, &faf[i].dob.month, &fa

    for( i = 0; i < n; i++)
        printf("%8d %20s %6.2f %3d %3d %5d\n", faf[i].student_id
        faf[i].average, faf[i].dob.day, faf[i].dob.month, faf[i]
    return 0;
}
```

```
1 Ion 7.67 12 8 1998
2 Maria 8.45 15 1 2000
3 Dora 9.56 29 2 2000
        1                   Ion    7.67  12    8  1998
        2                 Maria    8.45  15    1  2000
        3                  Dora    9.56  29    2  2000


--------------------------------
Process exited after 64.8 seconds with return value 0
Press any key to continue . . .
```

Dynamic memory allocation. Arrays

Memory allocation functions

As you know, you have to declare the size of an array before you use it. Hence, the array you declared may be insufficient or more than required to hold data. To solve this issue, you can allocate memory dynamically.

Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

Memory allocation functions

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

C malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a **pointer** of type **void** which can be casted into pointer of any form.

Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, **ptr** is pointer of cast-type.

malloc()

The **malloc()** function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns **NULL** pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all

bytes to zero.

Syntax of calloc

Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

free()

C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example: maximal element from array

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

int main()
{
    int i, num;
    float *data, max;

    printf("Elements : ");
    scanf("%d", &num);

// Allocates memory for 'num'.
    data = (float*)
    calloc(num, sizeof(float));
    if(data == NULL)
    {
     printf("Error!!! memory not
     allocated.");
     exit(0);
    }
    printf("\n");

    // Stores the numbers
for(i = 0; i < num; i++)
  {
    printf("Number %d: ", i + 1);
    scanf("%f", &data[i]);
  }

for(i = 0; i < num; i++)
printf("%0.0f ", data[i]);
printf("\n");

// Loop to store largest number

max = data[0];
for(i = 0; i < num; ++i)
  {
//if(*data < *(data + i))
//    *data = *(data + i);
if (max < data[i]) max = data[i];
  }

   printf("%0.0f", max);
free(data)
   return 0;
}
```

realloc()

C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

realloc example

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

int main()
{
    int *data, i , n1, n2;
    printf("Enter size of array: ");
    scanf("%d", &n1);

    data = (int*) malloc(n1 * sizeof(int));

    printf("Address of previously allocated memory: ");
    for(i = 0; i < n1; ++i)
         printf("%u\t",data + i);
    // enter data  using pointers
    for(i = 0; i < n1; ++i)
    {
       printf("\nEnter Number %d: ", i + 1);
       scanf("%d", data + i);
    }
```

realloc example

```
 for(i = 0; i < n1; ++i)
    printf("%d\t", *(data + i));

    printf("\nEnter new size of array: ");
    scanf("%d", &n2);
    data = realloc(data, n2 * sizeof(int));
    for(i = 0; i < n2; ++i)
         printf("%u\t", (data + i));
    printf("%d\n");
    for(i = 0; i < n2; ++i)
         printf("%d\t", *(data + i));
    return 0;
}
```