{C} Programming

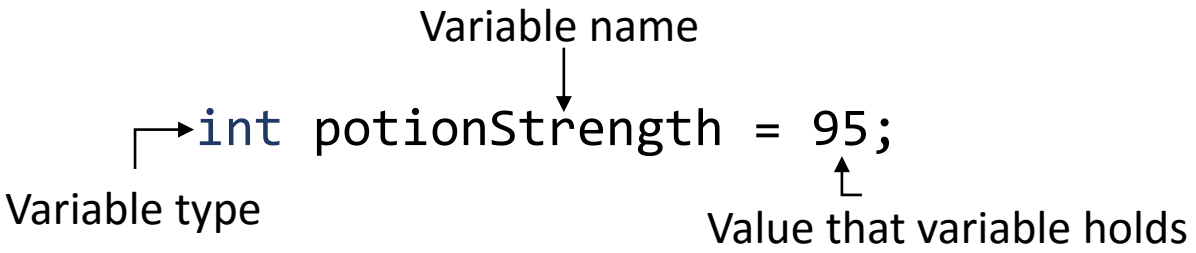
Part 1/2 | Basics

Variables, Conditions, Loops, Arrays, Pointer basics

Variables

A variable is a container (storage area) to hold data.

Eg.



C is strongly typed language. What it means is that the **type of a variable cannot be changed**.

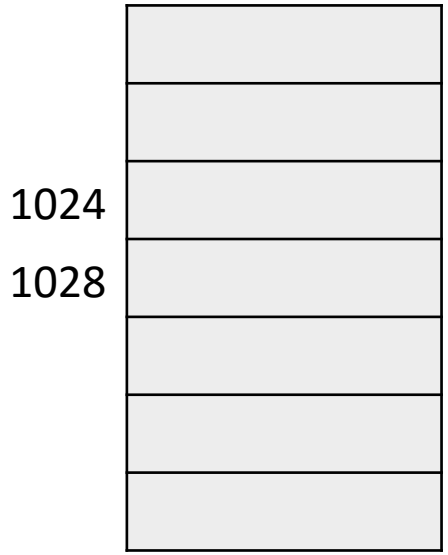
You can use const prefix to declare constant values with specific type:

```
const double PI = 3.14;
```

PI is a constant. That means, that in this program 3.14 and PI is same.

Variables | Example

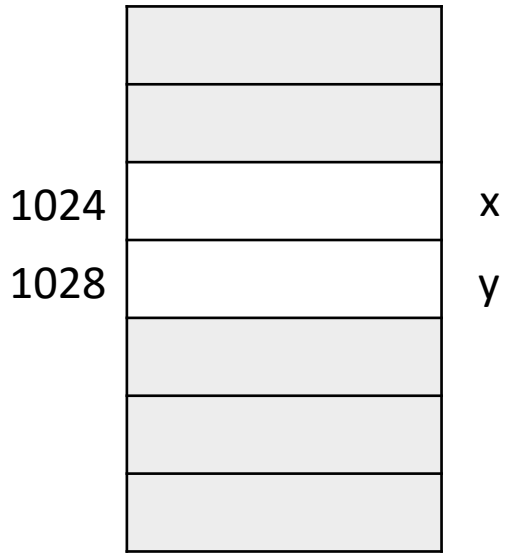
```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

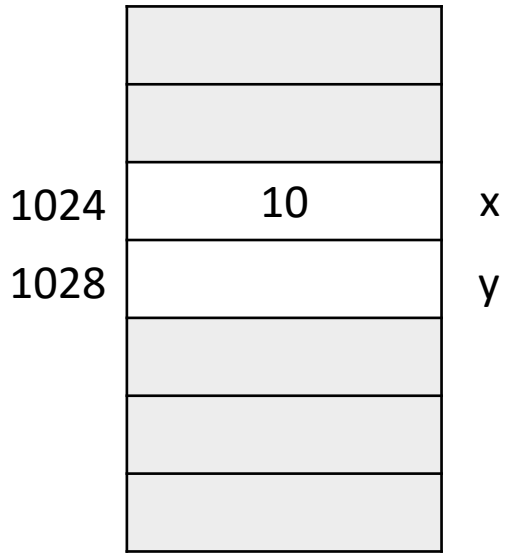
Declare x and y



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

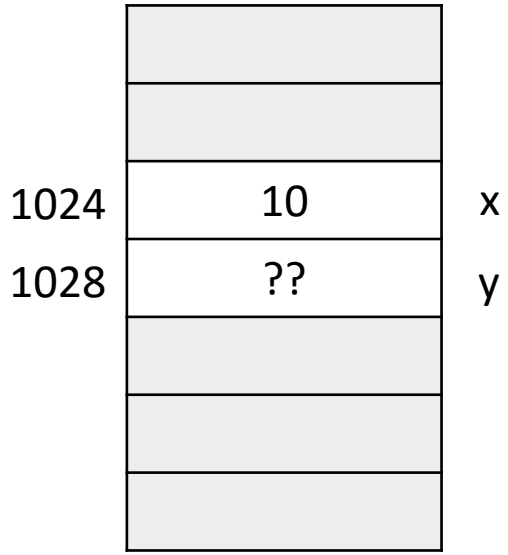
Assign to x the value 10



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

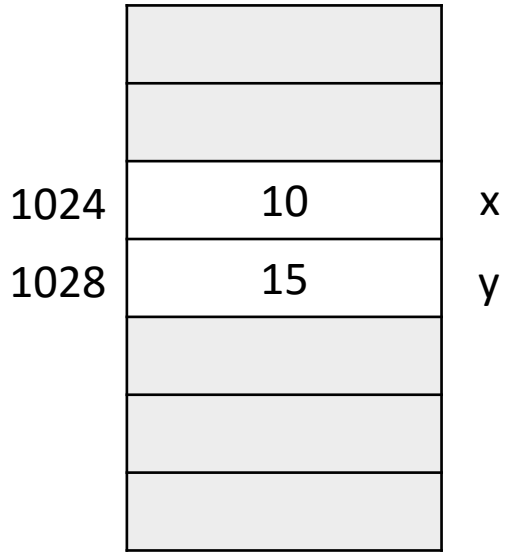
Assign to y the Value x+5



Variables | Example

```c
#include <stdio.h>
int main(void){
        int x, y;
        x = 10;
        y = x + 5;
        printf ("x = %d ", x);
        printf ("y = %d\n", y);
        return 0;
}
```

Program's output:
x = 10 y = 15



Variable sizes

Type	Size	Comment
integer	4 bytes = 2^32 bits	it can take 2^{32} distinct states as: -2^{31},-2^{31}+1, ...,-2, -1, 0, 1, 2, ..., 2^{31}-2, 2^{31}-1
float	4 bytes	Floating point variables has a precision of 6 digits whereas the precision of double is 14 digits.
double	8 bytes	
char	1 byte	
struct	Depends on the structs' fields	

Variable sizes | sizeof();

```c
#include <stdio.h>
int main(void){
        int a, e[10];
        float b;
        double c;
        char d;
        printf("Size of int=%lu bytes\n",sizeof(a));
        printf("Size of float=%lu bytes\n",sizeof(b));
        printf("Size of double=%lu bytes\n",sizeof(c));
        printf("Size of char=%lu byte\n",sizeof(d));
        printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
        return 0;
}
```

What should the code above print?

Variable sizes | sizeof();

```c
#include <stdio.h>
int main(void){
        int a, e[10];
        float b;
        double c;
        char d;
        printf("Size of int=%lu bytes\n",sizeof(a));
        printf("Size of float=%lu bytes\n",sizeof(b));
        printf("Size of double=%lu bytes\n",sizeof(c));
        printf("Size of char=%lu byte\n",sizeof(d));
        printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
        return 0;
}
```

Program Output:

```
Size of int=4 bytes
Size of float=4 bytes
Size of double=8 bytes
Size of char=1 byte
Size of integer type array having 10 elements = 40 bytes
```

If statements

The if keyword introduce the concept of handle some predictable yet unknown event:

Eg. In a pc game, what a character will find If he opens some specific door.

A **true** statement is one that evaluates to a **nonzero** number. A **false** statement evaluates to **zero**.

For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1.

Boolean operators: !(not), &&(and), ||(or)

Relational operators: >, <, >=, <=, ==, !=

A. !(1 || 0)	ANSWER: 0
B. !(1 || 1 && 0)	ANSWER: 0 (AND is evaluated before OR)
C. !((1 || 0) && 0)	ANSWER: 1
D. 5>4	ANSWER: 1
E. (5==5)&&(!(5 != 4))	ANSWER: 0

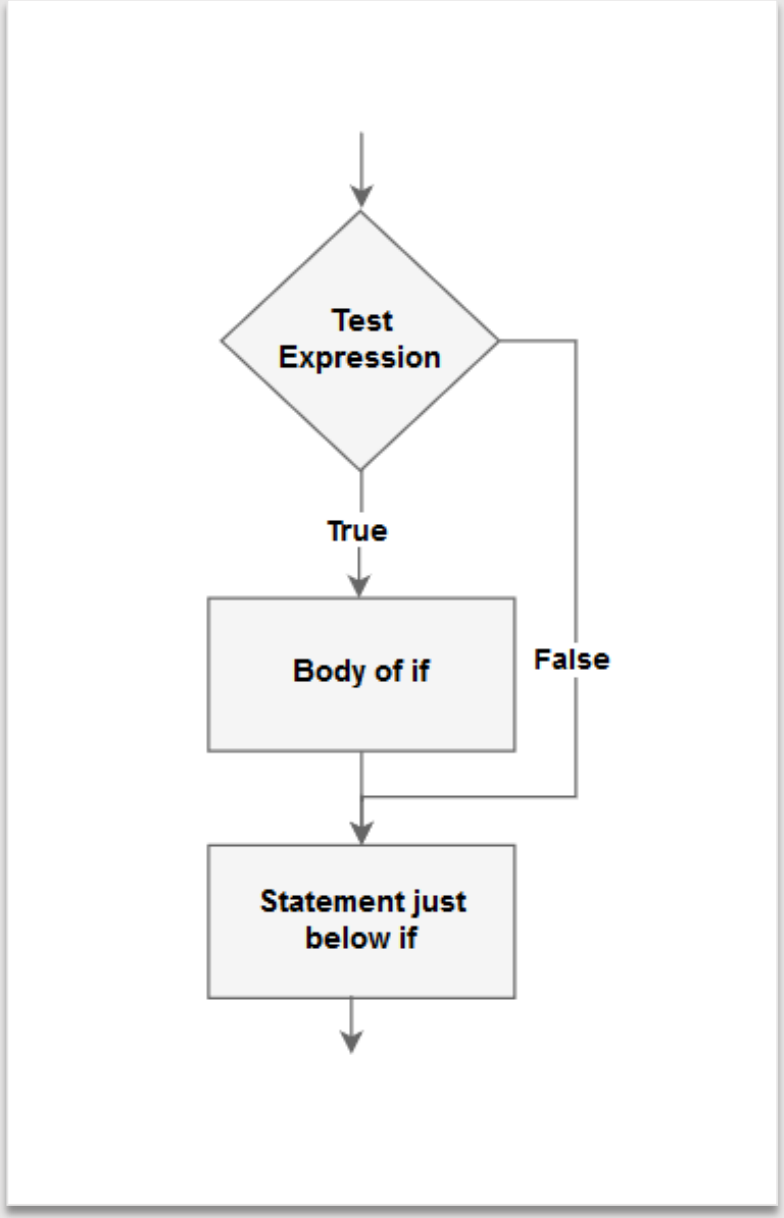
Conditions | if

```
if (statement is TRUE){
        Execute all statements inside if body
}
```

The if statement evaluates the expression inside parenthesis.

If test expression is evaluated to true (nonzero), statements inside the body of if are executed.

If test expression is evaluated to false (0), statements inside the body of if are skipped.



If | Example

```c
#include <stdio.h>
int main() {
	int number;
	printf("I am your computer genie!\n");
	printf("Enter a number from 0 to 9:");
	scanf("%d", &number);
	if(number<5){
		printf("That number is less than 5!\n");
	}
	printf("The genie knows all, sees all!\n");
	return(0);
}
```

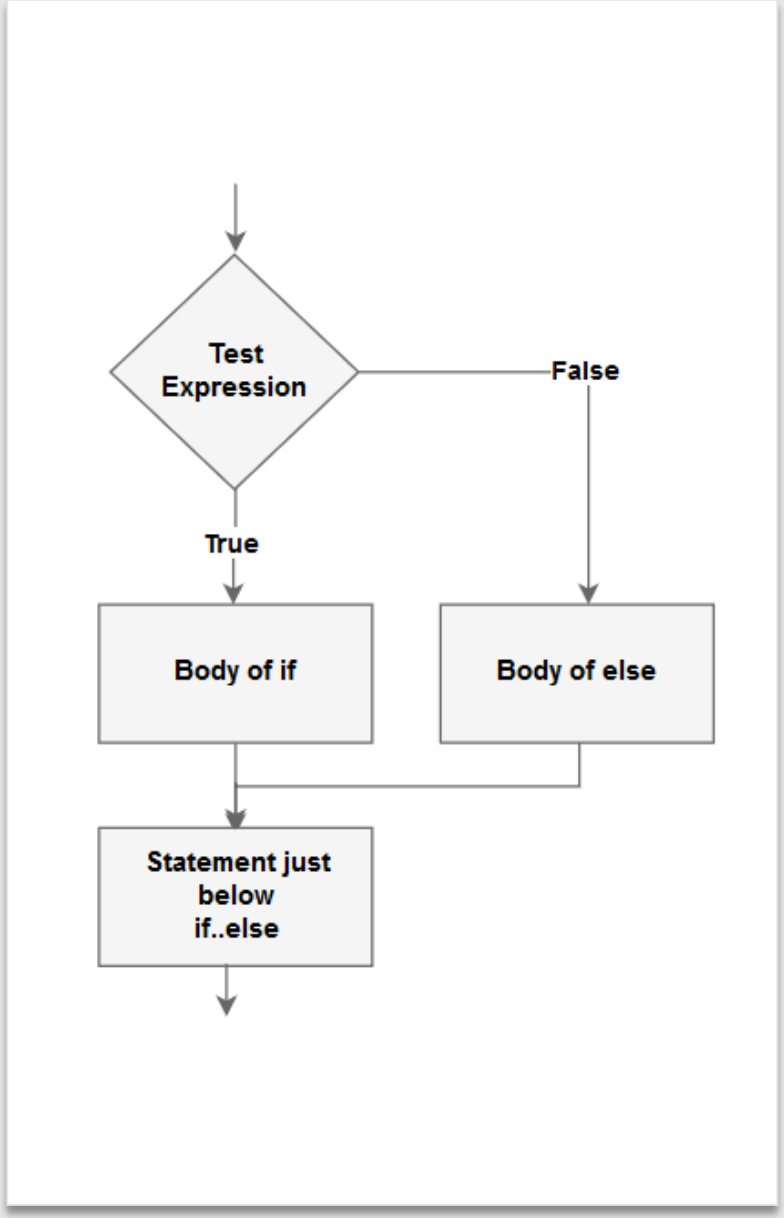
Conditions | if…else

```
if (statement is TRUE){
        statements inside the body of if
}else{
        statements inside the body of else
}
```

The if statement evaluates the test expression inside parenthesis.

If test expression is evaluated to true (nonzero), statements inside the body of if are executed and the statements inside the body of else are skipped.

If test expression is evaluated to false (0), statements inside the body of if are skipped and statements inside the body of if are executed.



If…else | Example

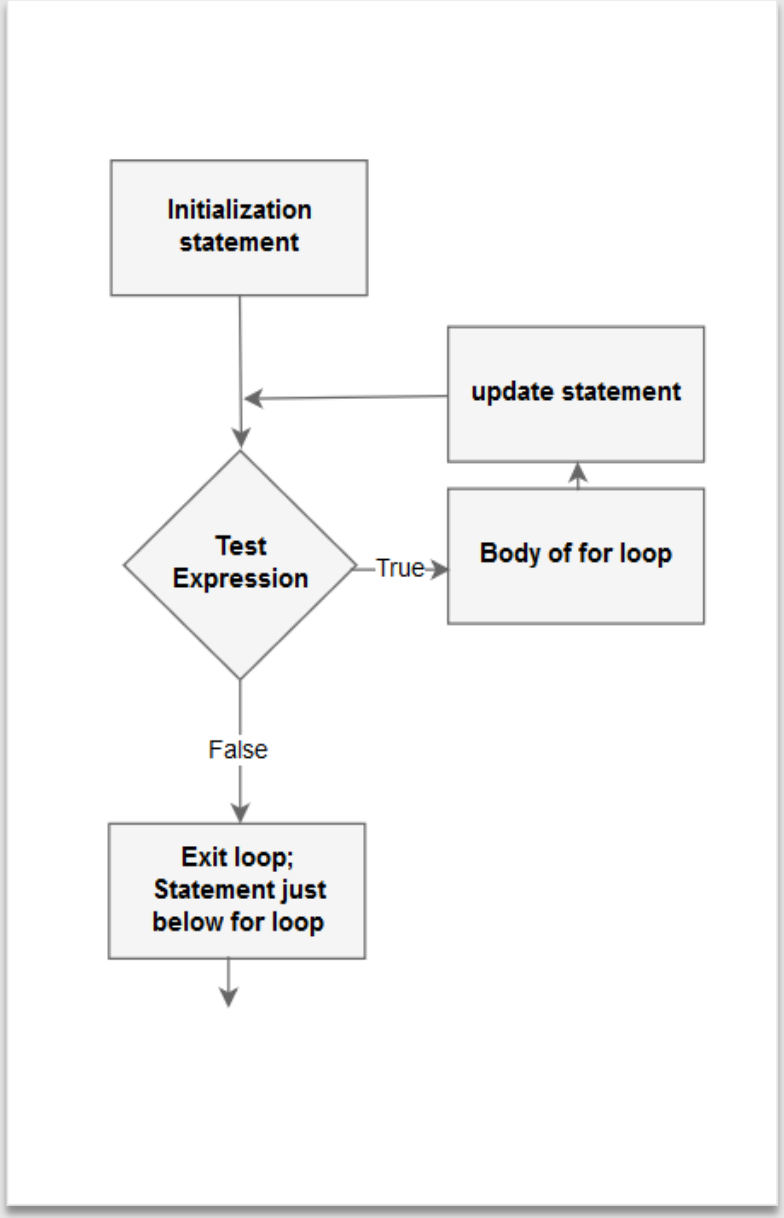
```
#include <stdio.h>
int main(){
        char c;
        printf("Would you like your computer to explode?");
        c=getchar();
        if(c=='Y' || c=='y'){
                printf("OK: Configuring computer to explode now.\n");
                printf("Bye!\n");
        }else{
                printf("Okay. Whew!\n");
        }
        return(0);
}
```

Loops | for

```
for(initStmt; condition; updateStmt){
        stmts to execute while the condition is true;
}
```

The initialization statement is executed once. Then the condition is evaluated. If the condition is true, the code inside of for loop is executed and the update expression is updated.

This process repeats until condition is false.



for | Example

```c
#include <stdio.h>
int main(void) {
     int x;
     for ( x = 0; x < 10; x++ ){
          printf( "%d\n", x );
     }
     return 0;
}
```

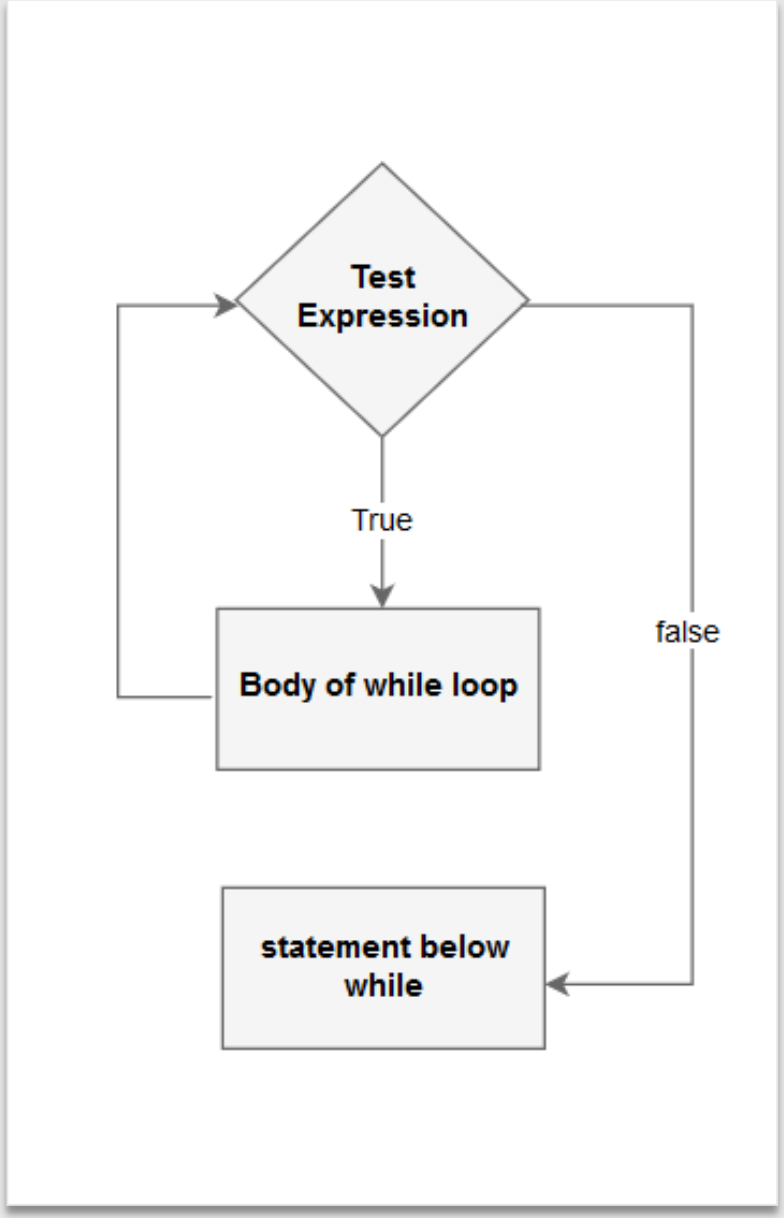
Loops | while

```
while(condition){
	Code to execute while the condition is true
}
```

If the condition is true, the body of the while is executed.

This process repeats until condition is false.

What’s the difference of do..while(); ?

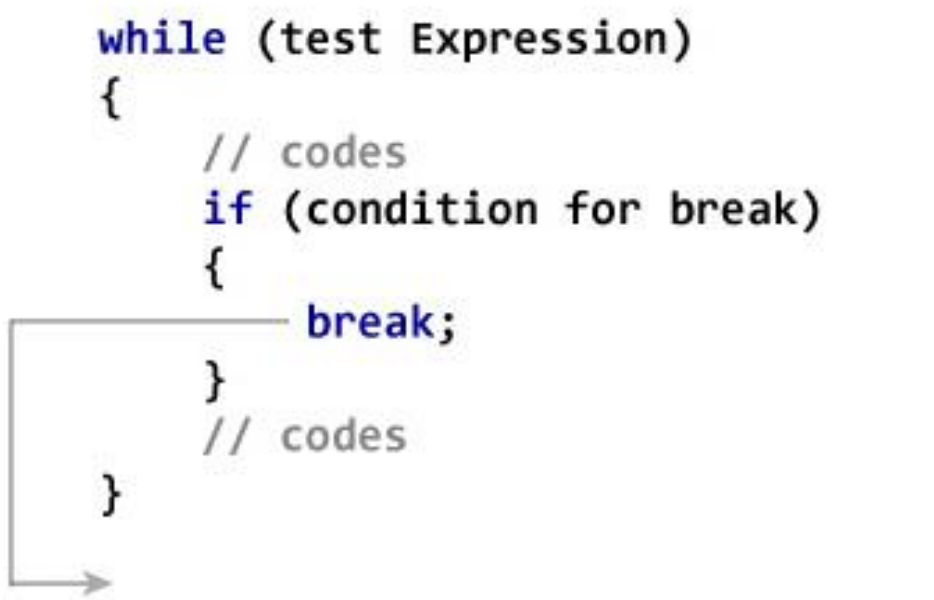


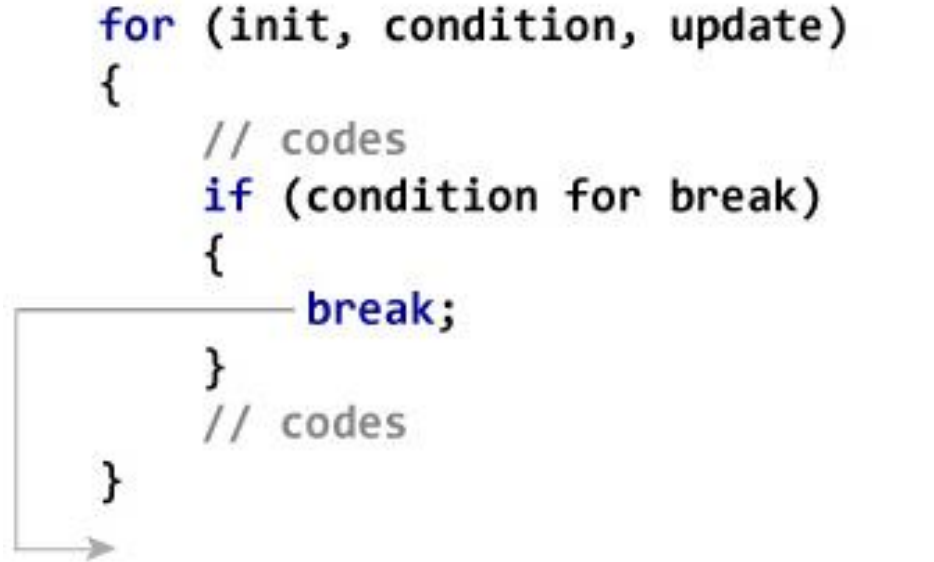
Break & continue I

The break statement **terminates the loop**.

```
for(i=1; i <= 10; ++i){
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);
        if(number < 0.0) {
                break;
        }
}
```

- The for loop stops when a negative number is given by the user





Break & continue II

The continue statement **skips** some statements inside the loop.

```
for(i=1; i <= 10; ++i){
        printf("Enter a n%d: ",i);
        scanf("%lf",&number);
        if(number < 0.0) {
                continue;
        }
        sum+= number;
}
```

- The negative values will not be added to the sum. Why?

```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Switch

```
switch (n)
{
    case constant1:
        // code to be executed if n is equal to constant1;
        break;

    case constant2:
        // code to be executed if n is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if n doesn't match any constant
}
```

Switch | example: main.c (hy240b)

```
switch (event)
{
    case 'A' :
         add_new_movie(movieID,category, year);
         break;
    case ‘R' :
        rate_movie(userID, movieID, score);
        break;
        .
        //other events
        .
    default:
        printf(“Not recognizable event”);
        break;
}
```

Functions

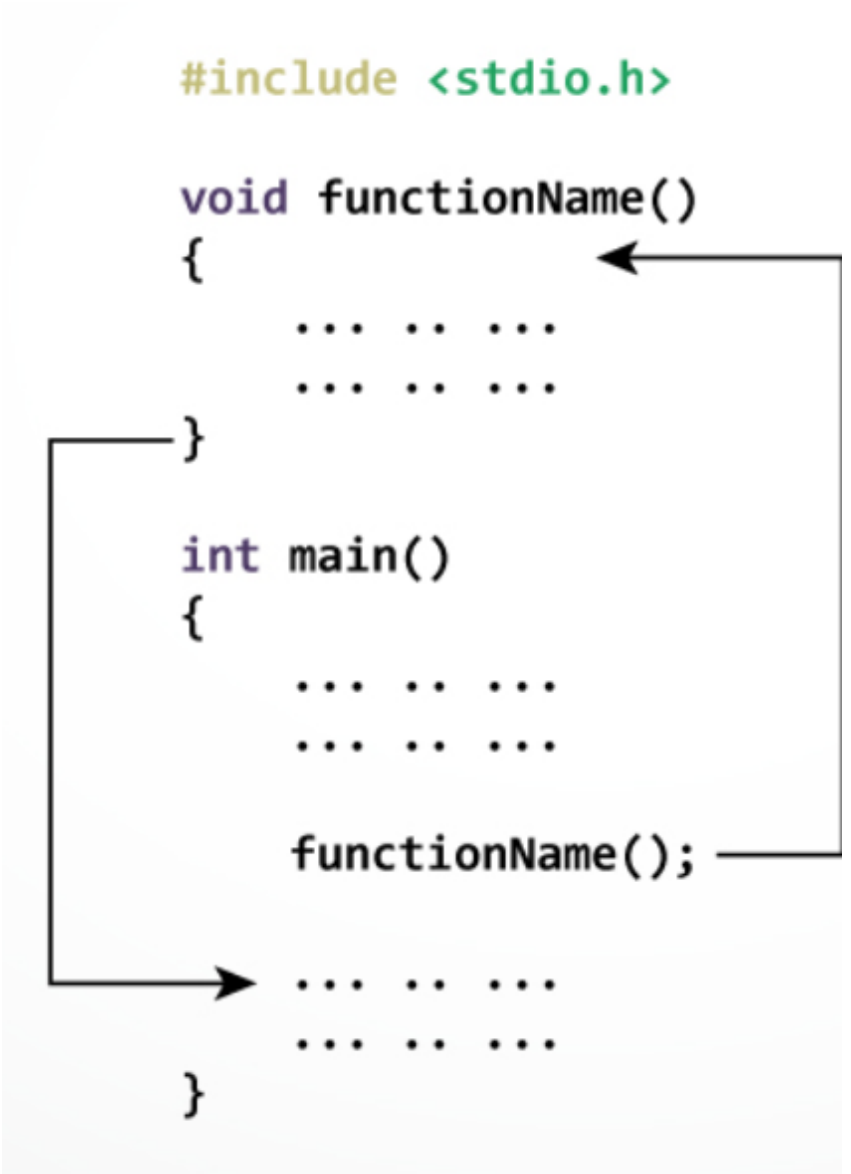
A function is a block of code that performs a specific task

There are two types of functions:

- **Standard library functions:** built-in functions to handle some tasks. Some of them are: printf, scanf etc
- **User-defined functions:** Functions defined by user to make the program easier to understand, reuse the same code or devide the program in smaller modules.

- Syntax:

```
returnType functionName(type1 argument1, type2 argument2, ...)
{
    //body of the function
}
```



Functions | example

```c
#include <stdio.h>
int mult ( int x, int y );

int main(void){
        int x = 3;
        int y = 5;
        printf( "The product of %d and %d is %d\n", x, y, mult( x, y ) );
        return 0;
}

int mult (int x, int y){
        return x * y;
}
```

Recursion

if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

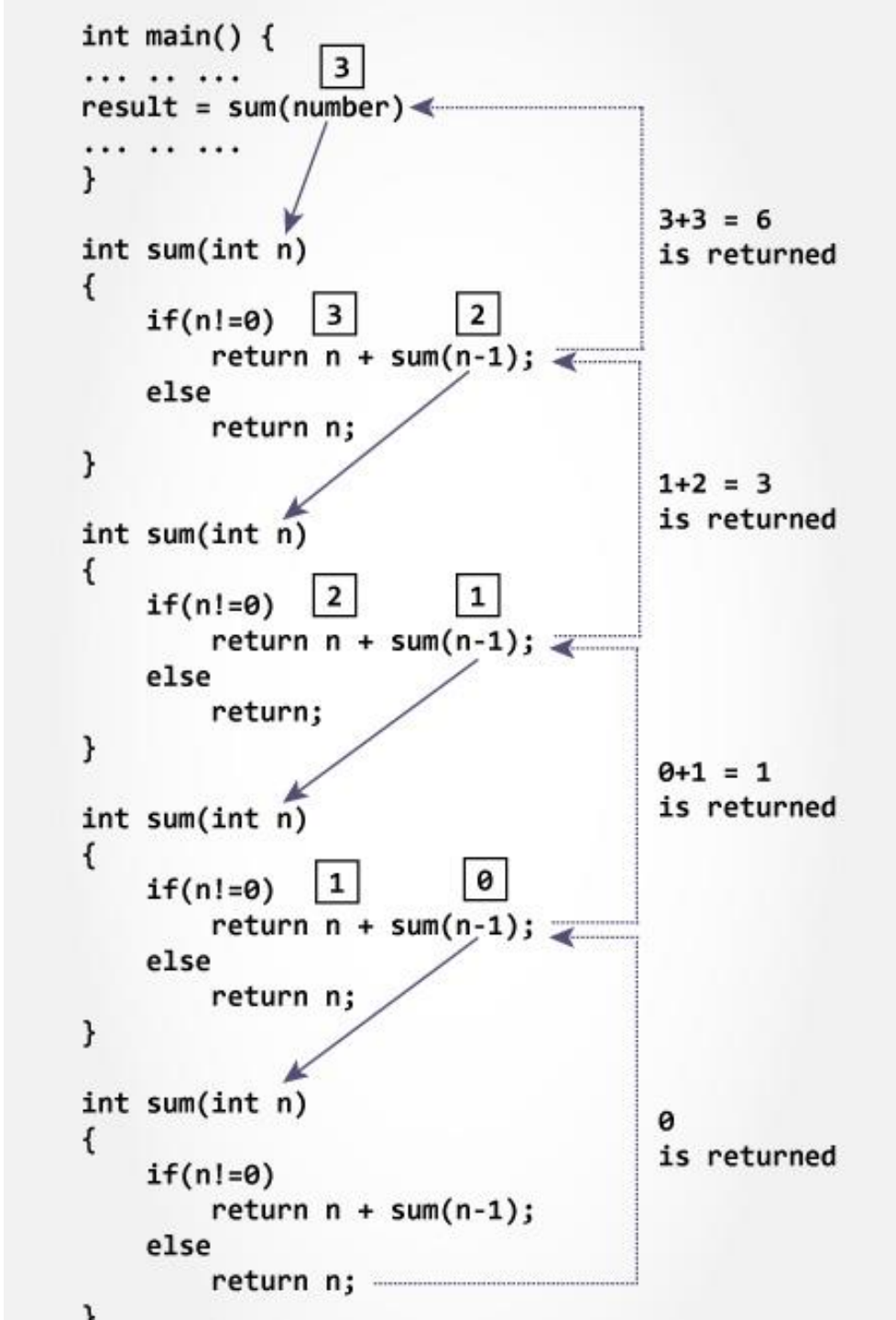
```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement can be used where one branch makes the recursive call and other doesn't.

Recursion | Example

```c
#include <stdio.h>
int sum(int n);
int main(){
        int number, result;
        printf("Enter a positive integer: ");
        scanf("%d", &number);
        result = sum(number);
        printf("sum=%d", result);
}
int sum(int n){
        if (n!=0)
                return n + sum(n-1);
        else
                return n;
}
```



Recursion | factorial

```c
#include <stdio.h>
int factorial(unsigned int i) {
	if(i <= 1) {
		return 1;
	}
	return i * factorial(i - 1);
}

int main() {
	int i = 3;
	printf("Factorial of %d is %d\n", i, factorial(i));
	return 0;
}
```

Arrays | 1D

Arrays is a kind of data structure that can store a **fixed-size sequential collection** of elements of the **same type.**

```
data_type array_name[array_size];
```

Eg. float temperature_in_crete[10]; , int age[5]; , char letters[24];

You can initialize an array one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

If you omit the size of the array, an array just big enough to hold the initialization is created.

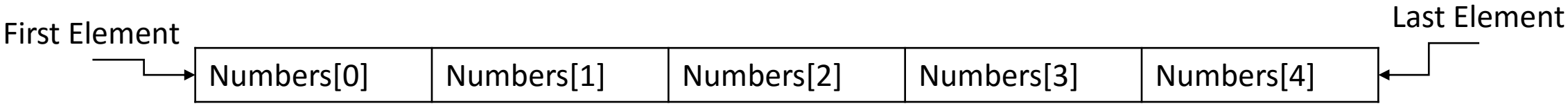
```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Size of array defines the number of elements in an array.

If you have you declare an array int numbers[10]; you can use the array members from numbers[0] to numbers[9]

Arrays | 1D

int Numbers[5];



Suppose, the starting address of Numbers[0] is 1020 and the size of int be 4 bytes. Then, the next address (address of Numbers[1]) will be 1024, address of Numbers[2] will be 1028 and so on.

1020	Numbers[0]
1024	Numbers[1]
1028	Numbers[2]
1032	Numbers[3]
1036	Numbers[4]

Multidimensional Arrays

C programming language allows programmer to create arrays of arrays known as multidimensional arrays.

```
data_type array_name[size_1][size_2]…[sizeN];
```

For example, int c[2][3]={{1,3,0}, {-1,5,9}};

```
#include <stdio.h>
int main () {
        int a[5][2] = {{0,0}, {1,2}, {2,4}, {3,6},{4,8}};
        int i, j;
        for ( i = 0; i < 5; i++ ) {
                for ( j = 0; j < 2; j++ ) {
                        printf("a[%d][%d] = %d\n", i,j, a[i][j] );
                }
        }
        return 0;
}
```

Pointers

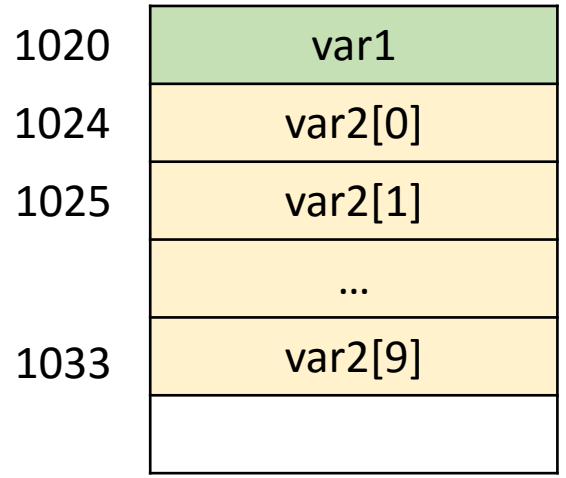
- Every variable is a **memory location** and every memory location has its **address**.
- Address can be accessed using ampersand (&) operator, which denotes an address in memory.

```c
#include <stdio.h>
int main(){
      int var=5;
      printf("Value: %d\n",var);
      printf("Address: %d\n",&var);
      return 0;
}
```

```
sh-4.3$ main
Value: 5
Address: 1278054348
```

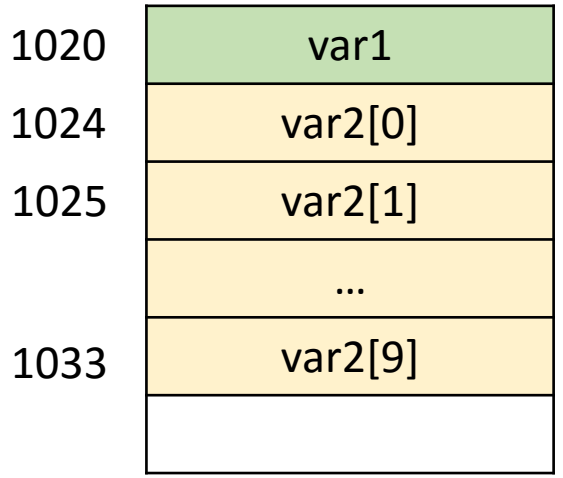
Pointers

```c
#include <stdio.h>
int main () {
        int var1;
        char var2[10];
        printf("Address of var1 variable: %x\n", &var1 );
        printf("Address of var2 variable: %x\n", &var2 );
        return 0;
}
```



Pointers

```c
#include <stdio.h>
int main () {
        int var1;
        char var2[10];
        printf("Address of var1 variable: %x\n", &var1 );
        printf("Address of var2 variable: %x\n", &var2 );
        printf("Address of var2[0] variable: %x\n", &var2[0] );
        printf("Address of var2[1] variable: %x\n", &var2[1] );
        printf("Address of var2[2] variable: %x\n", &var2[2] );
        printf("Address of var2[3] variable: %x\n", &var2[3] );
        printf("Address of var2[4] variable: %x\n", &var2[4] );
        printf("Address of var2[5] variable: %x\n", &var2[5] );
        printf("Address of var2[6] variable: %x\n", &var2[6] );
        printf("Address of var2[7] variable: %x\n", &var2[7] );
        printf("Address of var2[8] variable: %x\n", &var2[8] );
        printf("Address of var2[9] variable: %x\n", &var2[9] );
        return 0;
}
```




```
sh-4.3$ main
Address of var1 variable: a6d93a2c
Address of var2 variable: a6d93a20
Address of var2[0] variable: a6d93a20
Address of var2[1] variable: a6d93a21
Address of var2[2] variable: a6d93a22
Address of var2[3] variable: a6d93a23
Address of var2[4] variable: a6d93a24
Address of var2[5] variable: a6d93a25
Address of var2[6] variable: a6d93a26
Address of var2[7] variable: a6d93a27
Address of var2[8] variable: a6d93a28
Address of var2[9] variable: a6d93a29
```

{C} Programming

Part 2/2 | Advanced

Pointers, Structs, Memory Allocation, Examples using structs

Εμβέλεια μεταβλητών

- Οι μεταβλητές ενός προγράμματος χωρίζονται σε δύο κατηγορίες:
- Τις **καθολικές μεταβλητές** οι οποίες δηλώνονται στην αρχή κάθε προγράμματος πριν τις συναρτήσεις και μπορούν να προσπελάσονται από όλες τις συναρτήσεις.
- Τις **τοπικές μεταβλητές** κάθε συνάρτησης.

```
int a = 100; // Μία καθολική μεταβλητή
int main (void)
{
   int y = 5; // Μία τοπική μεταβλητή

   …
   return 0;
}
```

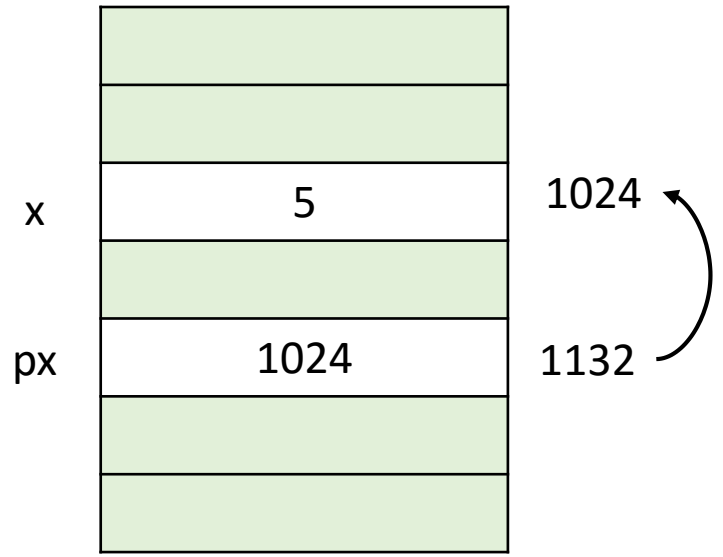
Pointers

Κάθε μεταβλητή είναι μια περιοχή στη μνήμη και κάθε περιοχή στη μνήμη έχει μια διεύθυνση.

- **&x**: Χρησιμοποιούμε αυτόν τον συμβολισμό για να μάθουμε τη διεύθυνση μνήμης που διατηρείται η μεταβλητή x.
- ***px**: Χρησιμοποιούμε αυτόν τον συμβολισμό για να πάρουμε την τιμή της μεταβλητής στην οποία δείχνει ο δείκτης px.

Pointers

```
void main (void) {
      int x;
      int *px;
      x = 5;
      px = &x;
      printf ("x = %d\n", x);
      printf ("px = %u\n", px);
      printf ("*px = %d\n", *px);
}
```



Pointers

Ενας **pointer** είναι μια **μεταβλητή** της οποιάς η τιμή είναι η διεύθυνση μιας άλλης μεταβλητής. Όπως κάθε άλλη μεταβλητή πρέπει να οριστεί πριν χρησιμοποιηθεί:

`type *var-name;`

```
int    *ip;        /* pointer to an integer */
double *dp;        /* pointer to a double */
float  *fp;        /* pointer to a float */
char   *ch         /* pointer to a character */
```

Ο πραγματικό τύπος των pointers, ανεξάρτητα του τύπου που δείχνουν, είναι ένας **long hexadecimal αριθμός που αναπαριστά μια διεύθυνση μνήμης.**

Pointers and arrays

- A difference between a pointer and an array is that a pointer variable **can take different addresses** as value whereas, in case of array it is fixed.

```
#include <stdio.h>
int main(){
   char c[4];
   int i;
   for(i=0;i<4;++i){
      printf("Address of c[%d]=%x\n",i,&c[i]);
   }
   return 0;
}
```

```
sh-4.3$ main
Address of c[0]=83447eb0
Address of c[1]=83447eb1
Address of c[2]=83447eb2
Address of c[3]=83447eb3
```

Pointers and arrays

- int arr[4];

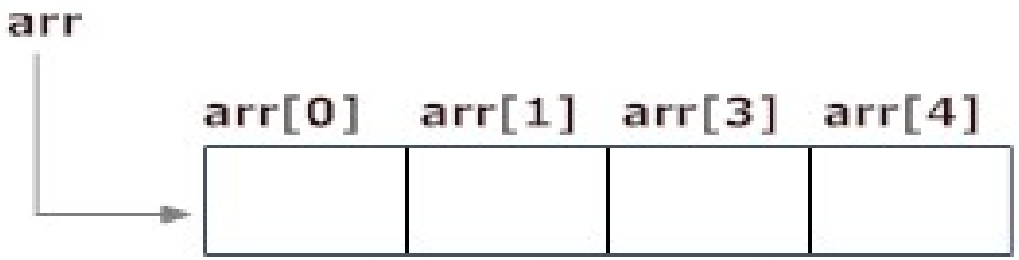


Figure: Array as Pointer

- The name of the array always **points to the first element of an array.**
- Here, **address** of first element of an array is **&arr[0].** Also, arr represents the address of the pointer where it is pointing. Hence**,** **&arr[0] is equivalent to arr.**
- Also, value in address &arr[0] is arr[0] and value in address arr is *arr. Hence, **arr[0] is equivalent to *arr.**

Pointers and arrays

So,

&a[1] is equivalent to (a+1) AND, a[1] is equivalent to *(a+1).

&a[2] is equivalent to (a+2) AND, a[2] is equivalent to *(a+2).

&a[3] is equivalent to (a+1) AND, a[3] is equivalent to *(a+3).

.

.

&a[i] is equivalent to (a+i) AND, a[i] is equivalent to *(a+i).

Arrays and pointers | Example

```c
#include <stdio.h>
int main(){
	int data[5], i;
	printf("Enter elements: ");
	for(i=0;i<5;i++){
		scanf("%d",data + i);
	}
	printf("You entered: ");
	for(i=0;i<5;i++){
		printf("%d ",*(data+i));
	}
	return 0;
}
```

Call by reference | Call by value

Call by Value: If data is passed by value, the data is copied from the variable to a variable used by the function. So if the data passed is modified inside the function, the value is only changed in the variable used **inside the function.**

```
void call_by_value(int x) {
	printf("Inside call_by_value x = %d before adding 10.\n", x);
	x += 10;
	printf("Inside call_by_value x = %d after adding 10.\n", x);
}
int main() {
	int a=10;
	printf("a = %d before function call_by_value.\n", a);
	call_by_value(a);
	printf("a = %d after function call_by_value.\n", a);
	return 0;
}
```

Call by reference | Call by value

Call by Reference: If data is passed by reference, **a pointer to the data is copied** instead of the actual variable as is done in a call by value. Because a pointer is copied, **if the value at that pointers address is changed in the function, the value is also changed in main().**

```
void call_by_reference(int *y) {
        printf("Inside call_by_reference y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside call_by_reference y = %d after adding 10.\n", *y);
}
int main() {
        int b=10;
        printf("b = %d before function call_by_reference.\n", b);
        call_by_reference(&b);
        printf("b = %d after function call_by_reference.\n", b);
        return 0;
}
```

Example

```c
#include <stdio.h>
void swap(int*, int*);
int main(){
        int x, y;
        printf("Enter the value of x and y\n");
        scanf("%d%d",&x,&y);
        printf("Before Swapping\nx = %d\ny =
%d\n", x, y);
        swap(&x, &y);
        printf("After Swapping\nx = %d\ny =
%d\n", x, y);
        return 0;
}
```

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

Structs | Δομές

Πολλές φορές χρειαζόμαστε σύνθετες οντότητες οι οποίες μπορούν να καθορισθούν από ένα σύνολο δεδομένων με διαφορετικό τύπο. Θα μπορούσαμε να ομαδοποιήσουμε αυτά τα δεδομένα και να αναφερόμαστε σε αυτά με κάποιο κοινό όνομα.

Στη c, τα structs είναι αυτή ακριβώς η ομαδοποιήση, δηλαδή μια συλλογή μεταβλητών διαφορετικών τύπων κάτω από το ίδιο όνομα.

```
struct structure_name {
   data_type member1;
   data_type member2;
   .
   .
   data_type memeber;
};
```

Τα μέλη/πεδία της δομής μπορεί να είναι οποιαδήποτε μεταβλητή από τους γνωστούς τύπους της c, πίνακες, δείκτες ακόμα και άλλες δομές.

Structs | Δομές

Για παράδειγμα, αν θέλουμε να αναπαραστήσουμε την πληροφορία για ένα τραγούδι (πχ τίτλος τραγουδιού, καλλιτέχνης, χρονολογία), θα μπόρούσαμε να δημιουργήσουμε μια δομή ως εξής:

```
struct song {
    char title[100];
    char singer[50];
    int year;
}
```

Structs | Δομές

Ο ορισμός ενός struct είναι η δημιουργία ενός **user-defined τύπου**, αλλά δε **δεσμεύεται μνήμη για αυτόν** αφού δεν έχουμε δημιουργήσει κάποια μεταβλητή.

Το person είναι ένας καινούργιος τύπος δεδομένων.

```
struct person{
    char name[50];
    int cit_no;
    float salary;
};
```

Μπορούμε να το χρησιμοποιήσουμε για τη δήλωση μεταβλητών αυτού του τύπου πχ: `struct person p1, p2, p[20];`

Ή διαφορετικά:

```
struct person
{
    char name[50];
    int cit_no;
    float salary;
}p1 ,p2 ,p[20];
```

Και με τους δύο τρόπους τα p1, p2 και ο πίνακας p 20 στοιχείων, είναι τύπου struct person.

Structs | Δομές

```
struct point {
        double x;
        double y;
}
struct rectangle {
        struct point p1;
        struct point p2;
}
Int main (){
        struct point my_point = {22.4, -38.9};
        struct rectangle my_rect;
        my_rect.p1.x = 15.3;
        ….
}
```

Εδώ ορίζουμε το struct point για την αναπαράσταση ενός σημείου στο επίπεδο και το struct rectangle για την αναπαράσταση ενός παραλληλογράμμου στο επίπεδο με τις πλευρές παράλληλες στους άξονες

Accessing structures members

Υπάρχουν δύο είδη operators για την πρόσβαση των μελών μια δομής:

- Member operator(.)
- Structure pointer operator(->)

Μπορούμε να έχουμε πρόσβαση στα πεδία της δομής ως εξής:

```
structure_variable_name.member_name
```

- Για παράδειγμα x1.DataStructuresGrade = 5;

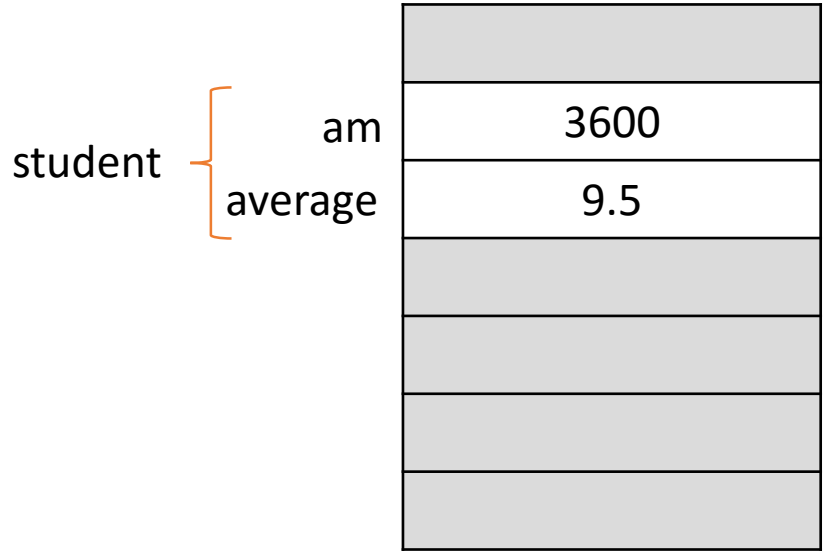
Structs | Examples

Έστω ότι θέλω να δημιουργήσω κάποια δομή η οποία αποθηκεύει το ΑΜ και τον μέσο όρο ενός φοιτητή.

Ποια θα είναι η δομή που θα αναπαραστήσει αυτόν τον φοιτητή;

Structs | Examples

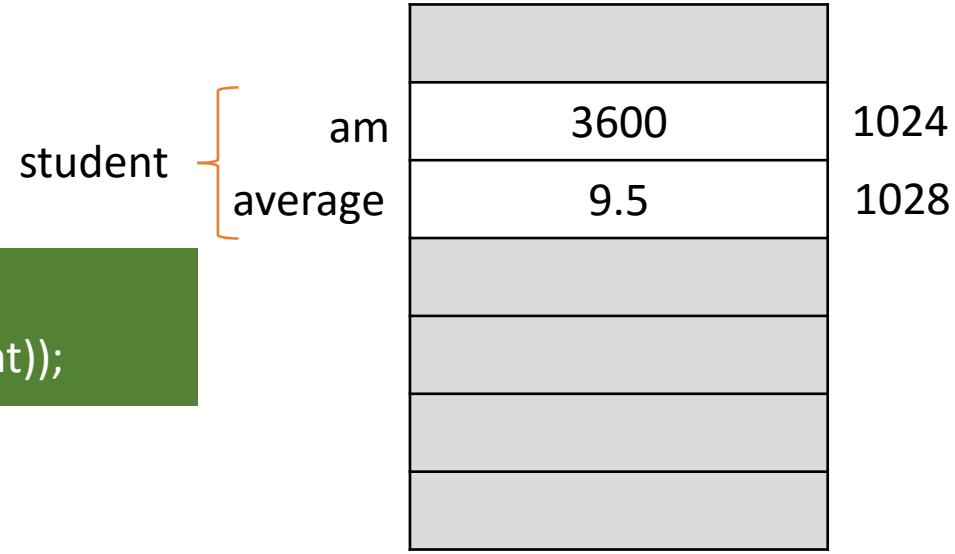
```
#include <stdio.h>
#include <stdlib.h>
struct s {
	int am;
	float average;
};
int main (void){
	struct s student;
	student.am = 3600;
	student.average = 9.5;
	printf ("A.M.:%d -- M.O.: %f", student.am, student.average);

	return 0;
}
```



Structs | Examples

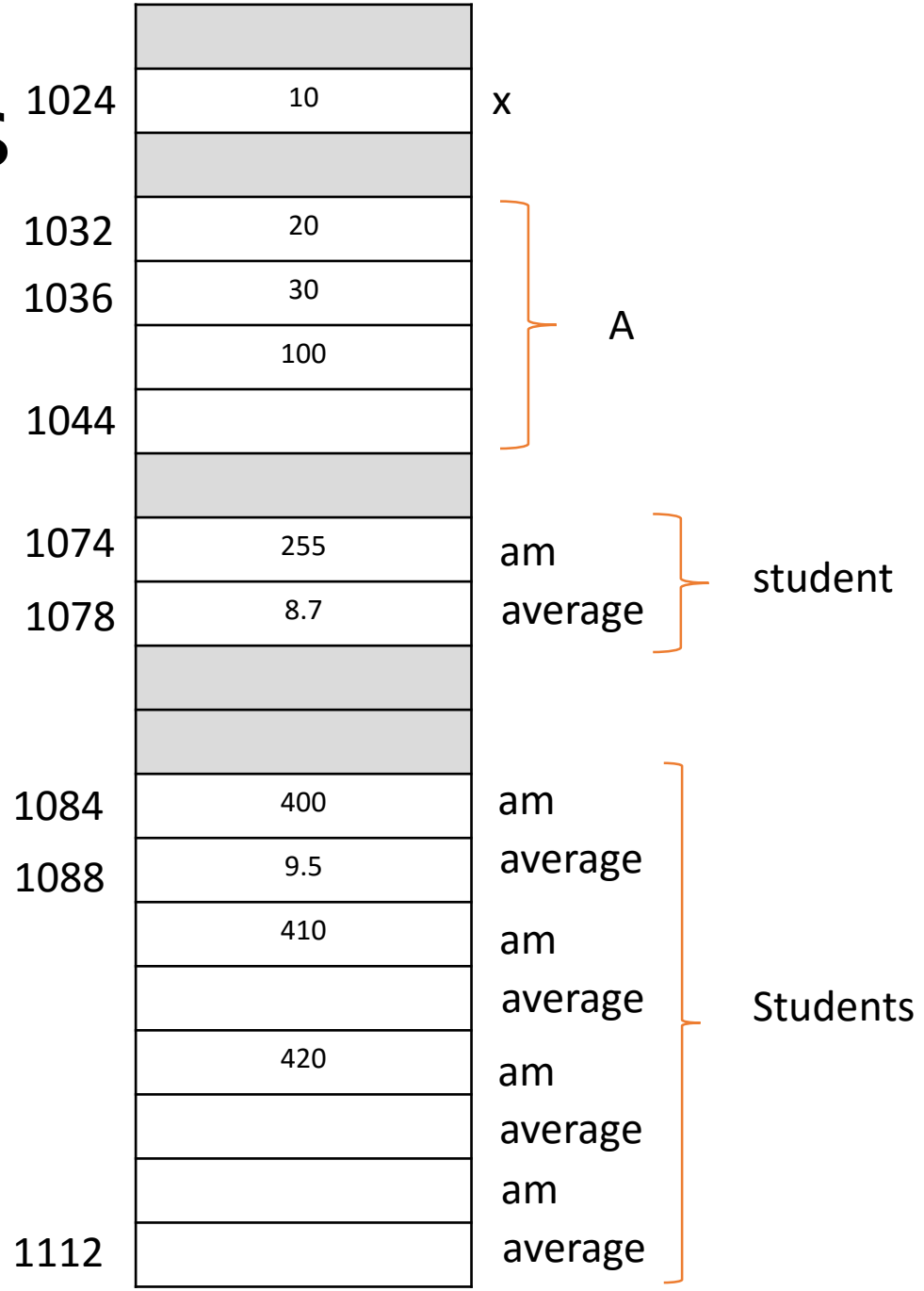
```
#include <stdio.h>
#include <stdlib.h>
struct s {
	int am;
	float average;
};
int main (void){
	struct s student;
	student.am = 3600;
	student.average = 9.5;
	printf ("A.M.:%d -- M.O.: %f", student.am, student.average);

	return 0;
}
```

Τι θα τυπώσει το:
printf ("%d", sizeof(student));



Structs | Examples

```
struct s {
            int am;
            float average;
};
int main (void){
            int x;
            int A[4];
            struct s student;
            struct s Students[4];
            x = 10;
            A[0] = 20;
            A[1] = 30;
            A[2] = 10*x;
            student.am = 255;
            student.average = 8.7;
            Students[0].am = 400;
            Students[0].average = 9.5;
            Students[1].am = 410;
            Students[2].am = 420;
}
```



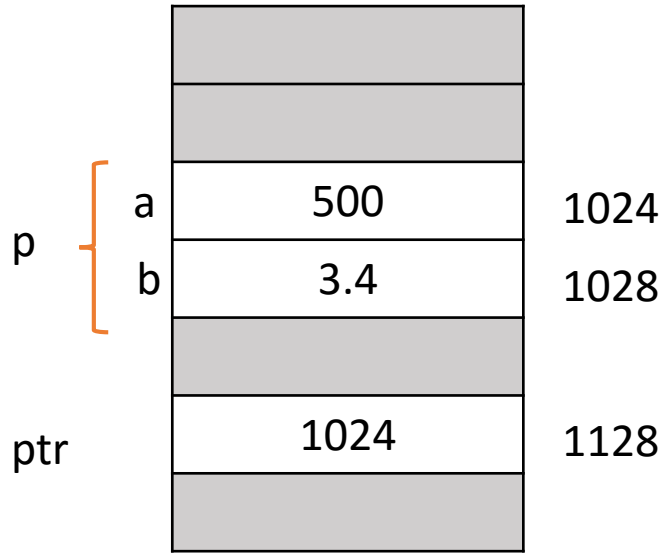
Structs and pointers

Όπως ορίζουμε μεταβλητές με τύπο κάποια δόμή έτσι μπορούμε να ορίσουμε και δείκτες σε δομές:

```
struct song rock_song;
struct song *my_fav_song, *my_least_fav_song;
my_fav_song = & rock_song;
my_least_fav_song = malloc (size(struct song));
```

Έχοντας ορίσει ένα δείκτη σε δομή μπορούμε να αναφερθούμε σε συγκεκριμένο μέλος : (*my_fav_song).title

```c
#include <stdio.h>
struct x{
    int a;
    float b;
};
int main(void){
        struct x p;
        struct x *ptr
        ptr=&p;
        ptr->a = 500;
        ptr->b = 3.4;
        printf("Displaying data: a: %d --- b: %f “, ptr->a, ptr->b);
        return 0;
}
```



Η παρένθεση είναι απαραίτητη λόγω της χαμηλής προτεραιότητας του *

➢ Το (*ptr).a είναι ίδιο με ptr->a και το (*ptr).b είναι ίδιο με ptr->b

Structs

Τα μέλη ενός struct μπορεί να είναι όπως είπαμε οποιοδήποτε τύπου, ακόμα και δείκτες σε δομές του ίδιου τύπου.

Συνήθως τέτοιες δομές (που ‘αναφέρονται’ στον εαυτό τους ή αυτο-αναφορικές δομές) χρησιμοποιούνται για να οργανώσουμε δέδομένα και διευκολύνουν την διαχείριση και επεξεργασία τους.

Συνηθισμένες τετοιες οργανώσεις δεδομένων είναι οι συνδεδεμένες λίστες και τα δυαδικά δέντρα που θα δείτε στο μάθημα.

Ένα παράδειγμα μιας τέτοιας δομής είναι:

```
struct listnode{
        int value;
        struct listnode * next;
}
```

Δυναμική δέσμευση μνήμης

Malloc

- Η συνάρτηση malloc παίρνει ως όρισμα το **μέγεθος της μνήμης** που θα δεσμευθεί. Συνήθως χρησιμοποιούμε τη **sizeof**, π.χ. sizeof(int)
- Επιστρέφει έναν δείκτη στην αρχή της δεσμευμένης μνήμης (ή null αν η malloc δε μπορεί να δεσμεύσει τη μνήμη που της ζητήθηκε! **Προσοχή:** Θα πρέπει να ελεγχετε αν η μνήμη που σας επιστράφηκε είναι διάφορη του Null πριν την χρησιμοποιήσετε!)
- ο δείκτης αυτός είναι τύπου **void** και πρέπει να μετατραπεί στον κατάλληλο τύπο (**casting**)
- Σύνταξη:

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

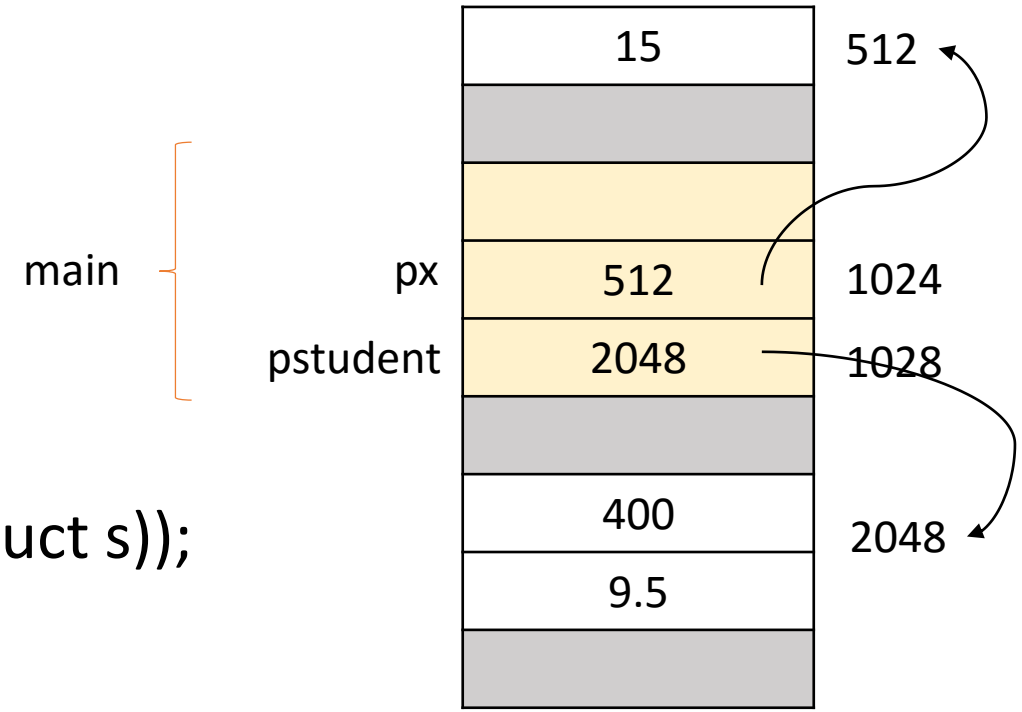
Πχ. ptr=(int*)malloc(100*sizeof(int));

Malloc |example

```
int num , * ptr;
…
ptr = malloc (num * sizeof(int));
if (ptr == NULL){
     printf (“Cannot allocate memory\n”);
     return -1;
}

..
/*Use ptr*/
```

Δυναμική δέσμευση μνήμης

```
int main (void)
{
        int *px;
        struct s *pstudent;
        px = (int *) malloc (sizeof(int));
        pstudent = (struct s *) malloc(sizeof(struct s));
        *px = 15;
        pstudent -> am = 400;
        pstudent -> average = 9.5;
}
```



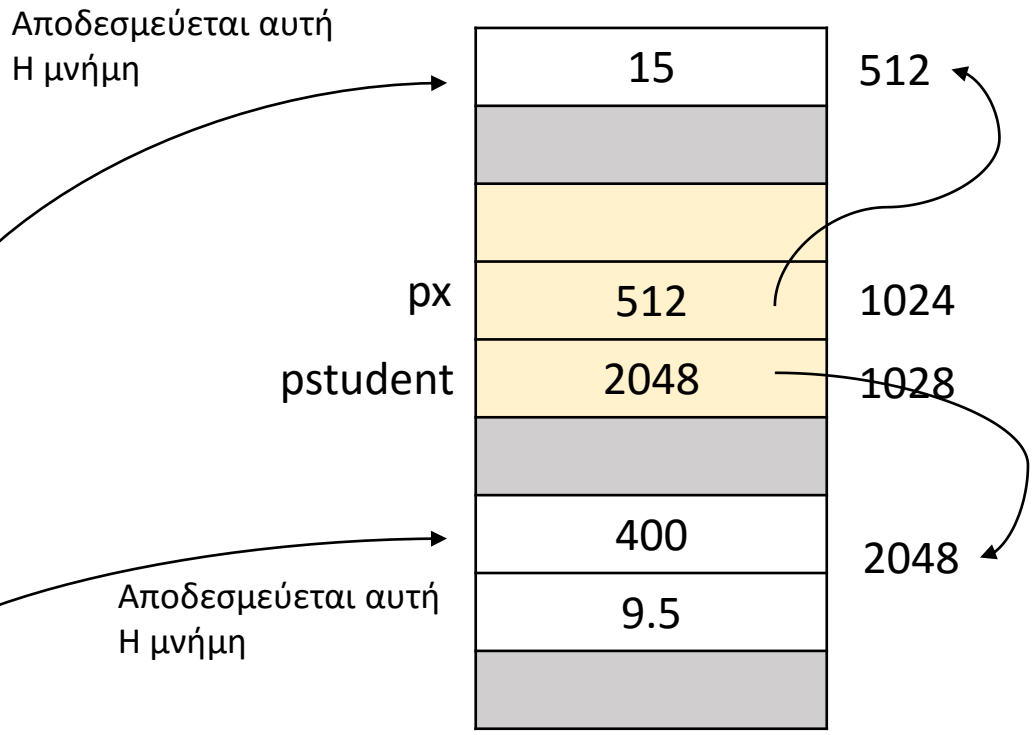
Free

- Η free χρησιμοποιείται για την **επιστροφή** της μνήμης που έχει δεσμευθεί με τη χρήστη της συνάρτησης malloc
- Η συνάρτηση αυτή παίρνει ως όρισμα έναν **δείκτη** στη μνήμη που επιθυμούμε να αποδεσμεύσουμε (ο δείκτης αυτός είναι ίδιος με τον δείκτη που επιστράφηκε από τη malloc)

```
free(ptr);
```

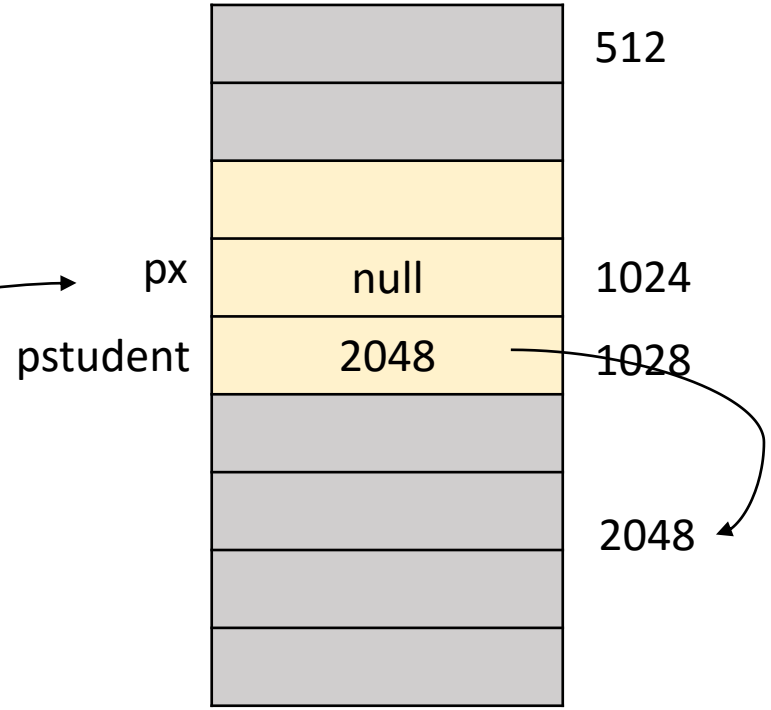
Free

```
void main (void){
        int *px;
        struct s *pstudent;

        px = (int *) malloc (sizeof(int));
        pstudent = (struct s *) malloc(sizeof(struct s));

        *px = 15;
        pstudent -> am = 400;
        pstudent -> average = 9.5;
        free (px);
        free (pstudent);
        px=NULL;
        pstudent=NULL;
}
```



Free

```
void main (void){
        int *px;
        struct s *pstudent;

        px = (int *) malloc (sizeof(int));
        pstudent = (struct s *) malloc(sizeof(struct s));

        *px = 15;
        pstudent -> am = 400;
        pstudent -> average = 9.5;
        free (px);
        free (pstudent);
        px=NULL;
        pstudent=NULL;
}
```



Παράδειγμα με χρήση πινάκων και δομών

- Έστω ότι η γραμματεία του csd θέλει να διατηρήσει διάφορα στοιχεία για τους φοιτητές. Συγκεκριμένα θέλει να αποθηκεύσει τον αριθμό μητρώου και τον μέσο όρο βαθμολογίας.
- Για την αποθήκευση αυτών των στοιχείων κάθε φοιτητή χρησιμοποιείται η εξής δομή:

```
struct student{
        int am;
        float average;
};
```

- Τα στοιχεία των φοιτητών διατηρούνται στον πίνακα CSD.
- Ο πίνακας StudentArray είναι ένας πίνακας από δομές struct student, δηλαδή:

```
struct student StudentArray[MAX_STUDENTS]
```

Παράδειγμα με χρήση πινάκων και δομών

- Θέλουμε να δημιουργήσουμε ένα σύστημα διατήρησης αυτών των δεδομένων, το οποίο θα υποστηρίζει της εξής λειτουργίες:
- Εισαγωγή νέου φοιτητή (InsertNewStudent)
- Αναζήτηση στοιχείων φοιτητή (SearchStudent)
- Ενημέρωση μέσου όρου φοιτητή (UpdateStudentAverage)

Επίσης, το σύστημα θα πρέπει να έχει κατάλληλο μενου για την προσπέλαση των λειτουργιών.

main.c

```
#include <stdio.h>
#define MAX_STUDENTS 1000
struct student {
        int am;
        float average;
};
struct student StudentArray[MAX_STUDENTS]; //Υποθέτουμε ότι τα πεδία am και average είναι αρχικοποιημενα με -1
int students_count = 0; // Μετρητής του πλήθους των καταχωρημένων φοιτητών
void InsertNewStudent (float average);
void SearchStudent (int am);
void UpdateStudentAverage (int am, float average);
int main (void){
        int choice = 0, am;
        float average;
        while (choice != 4) {
                printf (“1. Εισαγωγή νέου φοιτητή \n”);
                printf (“2. Αναζήτηση στοιχείων φοιτητή \n”);
                printf (“3. Ενημέρωση στοιχείων φοιτητή \n”);
                printf (“4. Έξοδος \n”);
                scanf (“%d”, &choice);
```

```
        switch (choice) {
                case 1 : printf (“Δώσε τον μέσο όρο του νέου φοιτητή: ”);
                         scanf (“%d”, &average);
                         InsertNewStudent (average);
                         break;
                case 2 : printf (“Δώσε το Α.Μ. του φοιτητή: ”);
                         scanf (“%d”, &am);
                         SearchStudent(am);
                         break;
                case 3 : printf (“Δώσε το Α.Μ. του φοιτητή: ”);
                         scanf (“%d”, &am);
                         printf (“Δώσε τον νέο μέσο όρο του φοιτητή”);
                         scanf (“%d”, &average);
                         UpdateStudentAverage (am, average);
                         break;
                case 4 : printf (“Έξοδος από το πρόγραμμα”);
                         break;
                default : printf (“Λάθος επιλογή”);
                         break;
}}} // End of main
```

Εισαγωγή νέου φοιτητή

```
void InsertNewStudent (float average){

      StudentArray[students_count].am = students_count;
      StudentArray[students_count].average = average;

      printf (“Ο νέος φοιτητής γράφτηκε επιτυχώς στο τμήμα με
A.M. %d και μέσο όρο %lf\n”, students_count,
StudentArray[students_count].average);
      students_count ++;
}
```

Αναζήτηση φοιτητή

```
void SearchStudent (int am){


      if (CSD[am].am != -1)
            printf (“Ο μέσος όρος του φοιτητή με Α.Μ.:%d
είναι: %lf\n”, am, StudentArray[am].average);
      else
            printf (“Δεν υπάρχει φοιτητής με Α.Μ.:%d\n”, am);
}
```

Ενημερωση μέσου όρου φοιτητή

```
void UpdateStudentsAverage (int am, float average)
{
     if (StudentArray[am].am != -1)
          StudentArray[am].average = average;

     printf (“Ο μέσος όρος του φοιτητή με Α.Μ.:%d
είναι: %lf\n”, CSD[am].am, CSD[am].average);

}
```

Παράδειγμα με δείκτες σε δομές

- Στο προηγούμενο παράδειγμα χρησιμοποιήσαμε τη δομή:

  struct s StudentArray[MAX_STUDENTS]

- Εάν η δομή struct student περιέχει πολλά στοιχεία, τότε σπαταλάτε άσκοπα μεγάλο μέρος της μνήμης είτε στον **StudentArray** αποθηκεύεται ένας φοιτητής είτε MAX_STUDENTS φοιτητές

- Better: ο πίνακας StudentArray να περιέχει δείκτες προς δομές struct student

main.c

```
#define MAX_STUDENTS 1000
struct s {
        int am;
        float average;
};
int students_count = 0; // Μετρητής του πλήθους των καταχωρημένων φοιτητών
void InsertNewStudent (float average, struct s ** StudentsArray);
void DeleteStudent (int am, struct s ** StudentsArray);
void SearchStudent (int am , struct s ** StudentsArray);
void UpdateStudentAverage (int am, float average , struct s ** StudentsArray);
int main (void){
        int choice = 0, am;
        float average;
        struct s * StudentArray[MAX_STUDENTS];
        while (choice != 5) {
                printf (“1. Εισαγωγή νέου φοιτητή \n”);
                printf (“2. Αναζήτηση στοιχείων φοιτητή \n”);
                printf (“3. Ενημέρωση στοιχείων φοιτητή \n”);
                printf (“4. Διαγραφή φοιτητή \n”);
                printf (“5. Έξοδος\n”);
                scanf (“%d”, &choice);
```

```
        switch (choice) {
                case 1 : printf (“Δώσε τον μέσο όρο του νέου φοιτητή: ”);
                         scanf (“%d”, &average);
                         InsertNewStudent (average, CSD);
                         break;
                case 2 : printf (“Δώσε το Α.Μ. του φοιτητή: ”);
                         scanf (“%d”, &am);
                         SearchStudent(am, CSD);
                         break;
                case 3 : printf (“Δώσε το Α.Μ. του φοιτητή: ”);
                         scanf (“%d”, &am);
                         printf (“Δώσε τον νέο μέσο όρο του φοιτητή”);
                         scanf (“%d”, &average);
                         UpdateStudentAverage (am, average, CSD);
                         break;
                case 4 : printf (“Δώσε το Α.Μ. του φοιτητή: ”);
                         scanf (“%d”, &am);
                         DeleteStudent(am, CSD);
                case 4 : printf (“Έξοδος από το πρόγραμμα”);
                         break;
                default : printf (“Λάθος επιλογή”);
                         break;
}}} // End of main
```

Εισαγωγή νέου φοιτητή

```
void InsertNewStudent (float average, struct **StudentsArray){


        StudentsArray[students_count]=(struct s*) malloc (sizeof(struct s));
        StudentsArray[students_count]->am = students_count;
        StudentsArray[students_count]->average = average;
        printf (“Ο νέος φοιτητής γράφτηκε επιτυχώς στο τμήμα με Α.Μ. %d\n”,
students_count);
        students_count ++;
}
```

Αναζήτηση φοιτητή

```
void SearchStudent (int am, struct **StudentsArray)
{
	if (StudentsArray[am] != null)
		printf (“Ο μέσος όρος του φοιτητή με Α.Μ.:%d είναι: %f\n”, am, StudentsArray[am]->average);
	else
		printf (“Δεν υπάρχει φοιτητής με Α.Μ.:%d\n”, am);
}
```

Ενημερωση μέσου όρου φοιτητή

```
void UpdateStudentsAverage (int am, float average, struct
**StudentsArray)
{
	if (StudentsArray[am] != null)
		StudentsArray[am]->average = average;
}
```

Διαγραφή φοιτητή

```
void DeleteStudent (int am, struct **StudentsArray)
{
     if (StudentsArray[am] != null)
          free(StudentsArray[am]);
          StudentsArray[am]) = null;
}
```