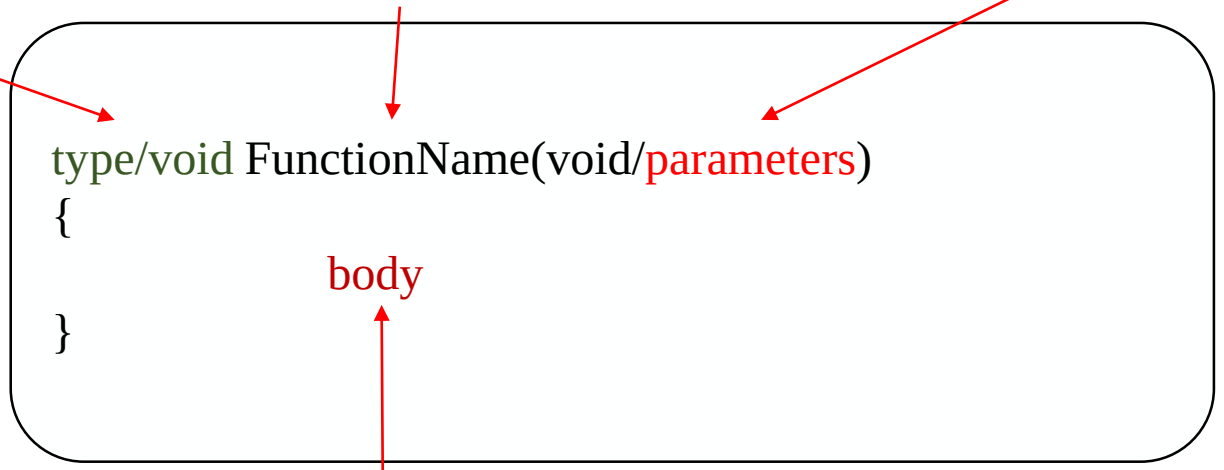
# Functions

# Structure of a Function

A list of parameters refers to the type, order, and number of the parameters of a function. The list of parameters is optional an can be empty.

**Return Type**: the data type of the value that is returned by the function. If no value is returned then the keyword **void** is used.

**Function Name**: The actual name of the function.




The function body contains a collection of statements that define what the function does

*The function name and the parameter list together define the function signature also known as a prototype .

# convert a char(ASCII) to lower case

```
char lower(char c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

טווח ההגדרה של פונקציה הינו מהנקודה בה הוכרז על הפונקציה ממנה ומטה. במקרה שלנו הגדרת הפונקציה sum נעשתה מעל לפונקציית ה- mian ולכן היא מוכרת בטווח ההגדרה של ה- main.

```c
#include <stdio.h>

int sum(int a, int b)
{
  int c=a+b;
  return c;
}

int main()
{
  int x = sum(2,3);
  printf("%d\n",x);
  return 0;
}
```

סדר הופעתן של הפונקציות בקובץ הינו חשוב אם יש תלות בין הפונקציות. למשל כאן כדי שהפונקציה main תוכל לרוץ ולבצע את הדרוש ממנה , עליה להשתמש בתוצאה המוחזרת מ sum . אבל כאן main הוגדרה לפני ה- sum . ולכן בעת שה- complier נתקל ב – main הוא לא מודע עדיין לקיום של sum. בשלב זה ה complier חייב לדעת מה הפונקציה sum מקבלת ומה מחזירה (כדי שיוכל להתריע על אי התאמה בטיפוסים הנשלחים\מוחזר . במקרה זה תתקבל הודעת אזהרה בלבד שכן המהדר לא יוכל לבצע התאמה זו).

```c
#include <stdio.h>
int main()
{
  int x = sum(2,3);
  printf("%d\n",x);
  return 0;
}

int sum(int a, int b)
{
  return a+b;
}
```

warning: implicit declaration of function 'sum'

Output :
5

```c
#include <stdio.h>
int sum(int a, int b);
int main()
{
  int x = sum(2,3);
  printf("%d\n",x);
  return 0;
}

int sum(int a, int b)
{
  return a+b;
}
```

prototype

Output :
5

```c
#include <stdio.h>

int sum(int a, int b)
{
  return a+b;
}

int main()
{
  int x = sum(2,3,4);
  printf("%d\n",x);
  return 0;
}
```

error: too many arguments to function 'sum'

```c
#include <stdio.h>

int sum(int a, int b);

int main()
{
  int x = sum(2,3,4);
  printf("%d\n",x);
  return 0;
}

int sum(int a, int b)
{
  return a+b;
}
```

error: too few arguments to function 'sum'

```c
#include <stdio.h>

int main()
{
  int x = sum(2,3,4);
  printf("%d\n",x);
  return 0;
}

int sum(int a, int b)
{
  return a+b;
}
```

warning: implicit declaration of function 'sum'

Output :
5

```c
#include <stdio.h>
int isDigit(char);
double atof(char[]);
int main() {
  char s[]="+52.998";
  double x = atof(s);
  printf("%f\n", x);
  return 0;
}
int isDigit(char c) {
  return ((c>='0') && (c<='9'));
}
double atof(char str[]) {
  double val, power;
  int i=0, sign = (str[0]=='-')?-1:1;
  if (str[0] == '+' || str[0]=='-')   i++;
  for (val = 0; isDigit(str[i]); i++)
      val = 10.0 * val + (str[i] - '0');
  if (str[i] == '.')    i++;
  for (power = 1.0; isDigit(str[i]); i++) {
      val = 10.0 * val + (str[i] - '0');
      power *= 10.0;
  }
  return sign * val/power;
}
```

Function prototype definitions

main declaration

isDigit decleration

atof declaration

write a function atof(char str[ ]) which converts a string ‘str’ to a double number.

# C - Storage Classes

In a C program a one should distinguish between **text and data segments,** where text refers to executable program code and data refers to variables.

Nowadays, on modern systems once a program is loaded in memory the text segments contains the program and is usually marked as read-only (to prevents a program from being accidentally modified).

Modern systems use a single text segment to store program instructions, but more than one segment for data, depending upon the storage class of the data being stored there :

1. Text or Code Segment :contains machine code of the compiled program

2. Initialized Data Segments :stores all global, static, constant, and external variables (declared with extern keyword) that are initialized beforehand).

3. Uninitialized Data Segments :Contrary to initialized data segment, uninitialized data or .bss segment stores all uninitialized global, static, and external variables (declared with extern keyword). Global, external, and static variable are by default initialized to zero. This section occupies no actual space in the object file; it is merely a place holder. Object file formats distinguish between initialized and uninitialized variables for space efficiency; uninitialized variables do not have to occupy any actual disk space in the object file.

4. Stack Segment : used for local variables in functions, parameter passing, return address for functions. Local variables have a scope to the block which they are defined in and are created when enters into the block . The data is being added or removed using LIFO manner to a stack (recursive function calls are added to stack).

5. Heap Segment: part of RAM where dynamically allocated variables are stored (malloc and calloc functions are used to allocate dynamic place) .

- A storage class defines the scope (visibility) and life-time of variables/functions within a C Program. They precede the type that they modify.
- There are 4 types of C storage classes:

✓ **auto:**

- default storage class for all local variables

✓ **register:**

- is used to define local variables that should be stored in a register instead of RAM.
- has a maximum size equal to the register size (usually one word)
- can't have the unary '&' operator applied to it (as it does not have a memory location).
- should only be used for variables that require quick access such as counters
- It is only a suggestion and the complier will decide wither to make the value a register value or not, depending on hardware and implementation restrictions.

✓ **static:**

- The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
- The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

✓ **extern:**

- **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When
- using 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.
- When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function.
- *extern* is used to declare a global variable or function in another file.
- extern modifier is usually used when there are two or more files sharing the same global variables or functions.

https://www.tutorialspoint.com/cprogramming/c_storage_classes.htm

# local variables are declared by default as auto

Local variables

Internal block

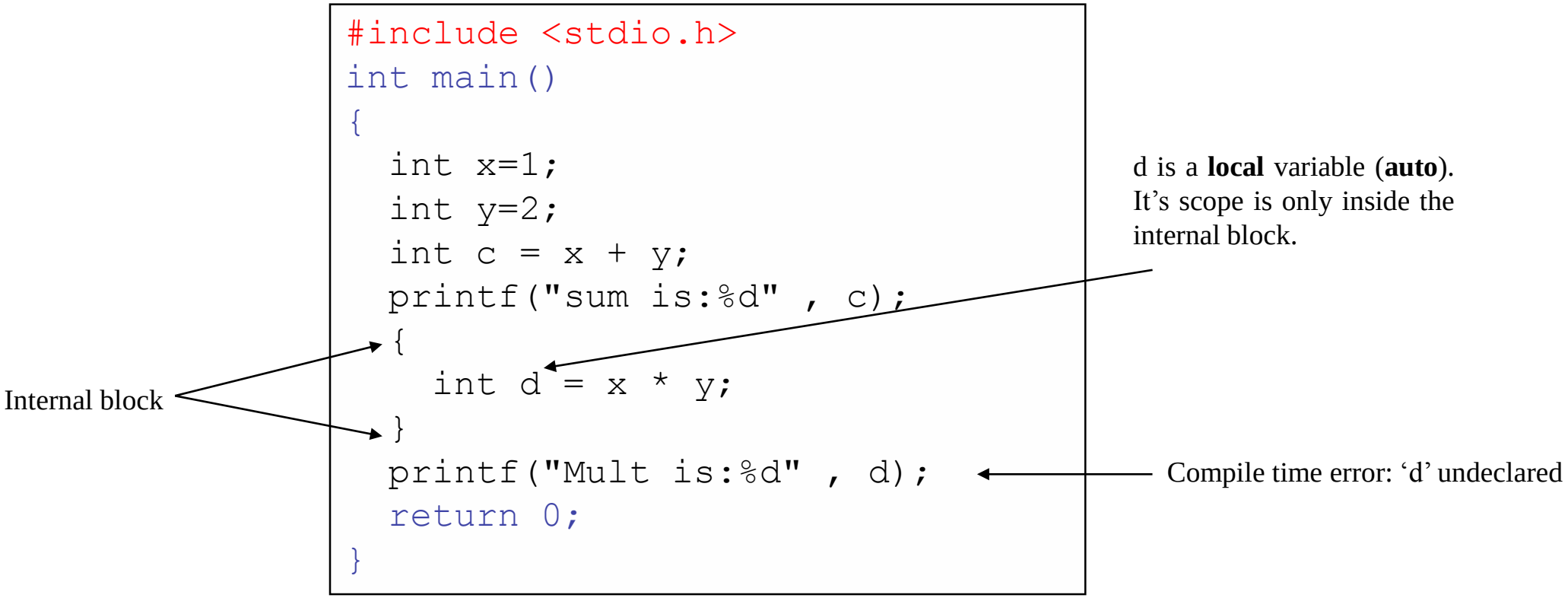
```c
#include <stdio.h>
int main()
{
  int x=1;
  int y=2;
  int c = x + y;
  printf("sum is:%d\n", c);
  {
    int d = x * y;
    printf("Mult is:%d\n", d);
  }
  return 0;
}
```

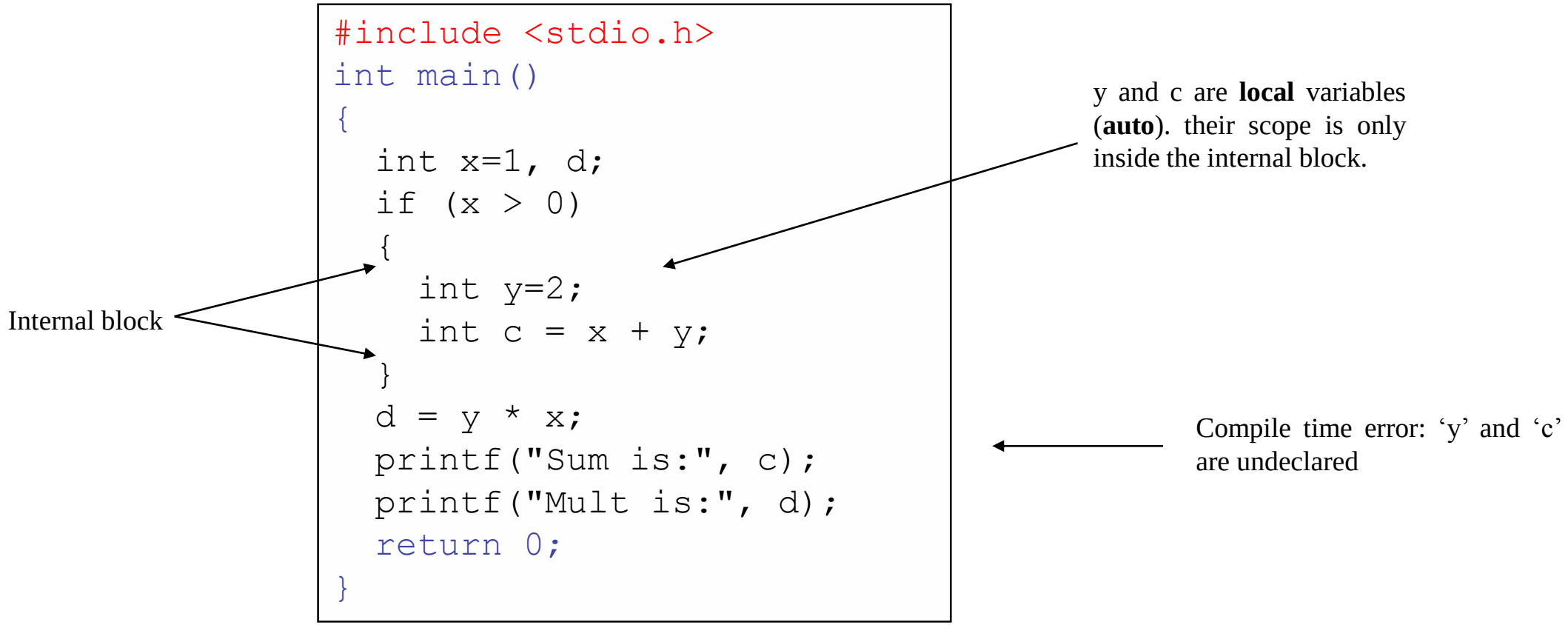
Output :
sum is:3
Mult is:2

```c
#include <stdio.h>
int main()
{
  auto int x=1;
  auto int y=2;
  auto int c = x + y;
  printf("sum is:%d\n", c);
  {
    auto int d = x * y;
    printf("Mult is:%d\n", d);
  }
  return 0;
}
```

Output :
sum is:3
Mult is:2

a variable declared within a function or any block is called a **local variable** (auto) i.e. the scope of a local variable is limited to a function or block in which it is being declared, and exists till the end of the function or block where it is declared. Local variable, once declared, it is **not automatically initialized**. We need to explicitly assign value to it others it will contain a garbage value.

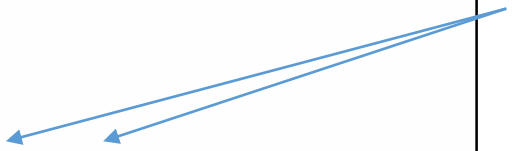

#include <stdio.h>
int main()
{
int x=1;
int y=2;
int c = x + y;
printf("sum is:%d" , c);
{
int d = x * y;
}
printf("Mult is:%d" , d);
return 0;
}
Internal block
d is a **local** variable (**auto**). It's scope is only inside the internal block.
Compile time error: 'd' undeclared


#include <stdio.h>
int main()
{
int x=1, d;
if (x > 0)
{
int y=2;
int c = x + y;
}
d = y * x;
printf("Sum is:", c);
printf("Mult is:", d);
return 0;
}
Internal block
y and c are **local** variables (**auto**). their scope is only inside the internal block.
Compile time error: ‘y’ and ‘c’ are undeclared

```c
#include "stdio.h"
void sum(int a, int b);
void main()
{
  int x, y;

  x = 2;
  y = 3;

  sum(x, y);
}
void sum(int a, int b)
{
  int c;
  c = a + b;
  printf("Sum is: %d\n", c);
}
```



Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

x and y are local variables (so cannot be accessed by any other functions in the program) of main function and are being passed to the function ‘sum’. The values of these variables (not the variables them selves) are being passed to the function ‘sum’. However, there is a main deference between the variables declared in the main function and those from the ‘sum’ function. The variables in the main function are local to it and can be accessed only by it. These local parameters are passed to ‘sum’ function, and are being local (a and b) to ‘sum’ only. These parameters sent to ‘sum’ are called parameter to a function. Parameters to functions can have same names as variables passed from the calling function or different and the compiler treats both of them as different variables even though if they have same names and values (memory address of variables at ‘main’ defer than ‘sum’). This type of passing variable is called **pass by value**.

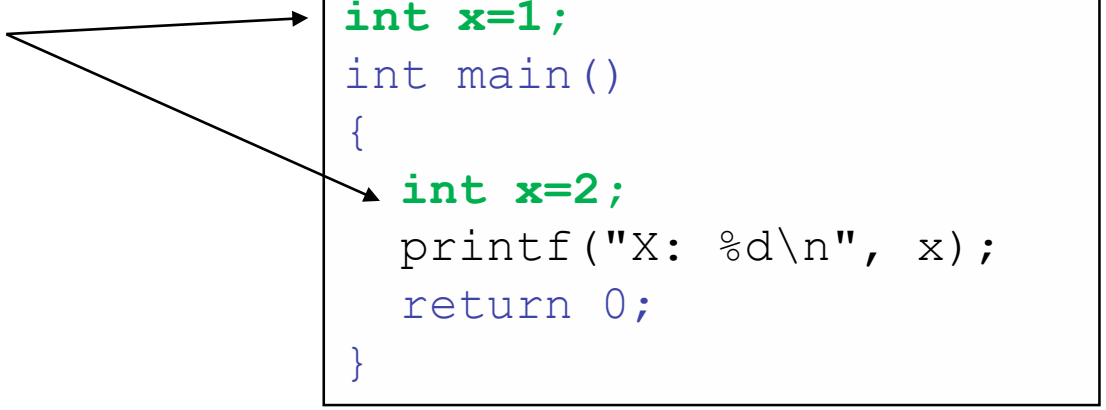
‘y’ and ‘c’ are **global** variables. Their scope cover all the program inside the file they were declared and only for all functions defined under them (here the main function).

```c
#include <stdio.h>
int y;
int c;
int main()
{
 int x=1,d;
 if (x > 0)
 {
   y=2;
   c = x + y;
 }
 d = y * x;
 printf("Sum is:%d\n", c);
 printf("Mult is:%d\n", d);
 return 0;
}
```

Output :
sum is:3
Mult is:2

A variable that is declared outside the main function, and not present in any other function or block (what happen if we declare a variable as a global and create another variable having the same name inside a block or function?). is called a **global variable**. Since this variable is not inside any block or function, it can be accessed by any function, block or expression. Hence the scope of a global variable is not limited to any function or block and it can be accessed by any function or block within the program. Global variables are automatically **initialized,** at creation time, to the initial value defined for its data type.

A program can have same name for local and global variables. In this case the value of local variable inside a function will take preference over the global one.



```c
#include <stdio.h>
int x=1;
int main()
{
  int x=2;
  printf("X: %d\n", x);
  return 0;
}
```

Output :
X: 2

```c
#include <stdio.h>
int x=1;
int y=2;
int sum(void) {
  return x + y;
}
int main(void)
{
  int c;
  c = sum();
  printf("C=%d\n", c);
  return 0;
}
```

Output :
C=3

‘y’ and ‘c’ are global variables and they are available only from the point they were declared onwards. So they are available for use on all functions below them and not know to all functions above them.



```c
#include <stdio.h>
int sum(void)
{
  return x + y;
}

int x=1;
int y=2;

int main(void)
{
  int c;
  c = sum();
  printf("C=%d\n", c);
  return 0;
}
```

Compile time error: ‘x’ and ‘y’ are undeclared

The prototype has been declared inside the main function - the scope of this declaration affects only the main function

version3

```c
#include <stdio.h>
int x=1;
int y=2;
int main(void)
{
  int c;
  int sum(void);
  c = sum();
  printf("C=%d\n", c);
  return 0;
}
int sum(void) {
  return x + y;
}
```

Output :
C=3

version1

```c
#include <stdio.h>
int x=1;
int y=2;
int sum(void);
int main(void)
{
  int c;
  c = sum();
  printf("C=%d\n", c);
  return 0;
}
int sum(void) {
  return x + y;
}
```

Output :
C=3

version2

```c
#include <stdio.h>
int sum(void);
int x=1;
int y=2;
int main(void)
{
  int c;
  c = sum();
  printf("C=%d\n", c);
  return 0;
}
int sum(void) {
  return x + y;
}
```

Output :
C=3

# Static vs Local variable

```c
#include <stdio.h>

int getCounter(void);

int main(void)
{
  int x, y, z;
  x = getCounter();
  y = getCounter();
  z = getCounter();
  printf("X:%d, Y:%d, Z:%d\n", x, y, z);
  return 1;
}

int getCounter(void) {
  int counter = 0;
  counter++;
  return counter;
}
```

Output :
X:1, Y:1, Z:1

'counter' is a local variable of function getCounter. it is visible only from getCounter. And It's life cycle is the getCounter function only. it is re-declared each time we re-enter the getCounter function.

```c
#include <stdio.h>

int getCounter(void);

int main(void)
{
  int x, y, z;
  x = getCounter();
  y = getCounter();
  z = getCounter();
  printf("X:%d, Y:%d, Z:%d\n", x, y, z);
  return 1;
}

int getCounter(void) {
  static int counter = 0;
  counter++;
  return counter;
}
```

Output :
X:1, Y:2, Z:3

'counter' is a **static** local variable of function getCounter. it is visible only from the getCounter. And It's life cycle is the entire program. it is declared only once when entering getCounter function for the first time.

Static variables are initialized automatically by the system.

Static variables are initialized with Zero →

```c
#include <stdio.h>
int main()
{
    static int x;
    int y;
    printf("%d %d\n", x, y);
    return 1;
}
```

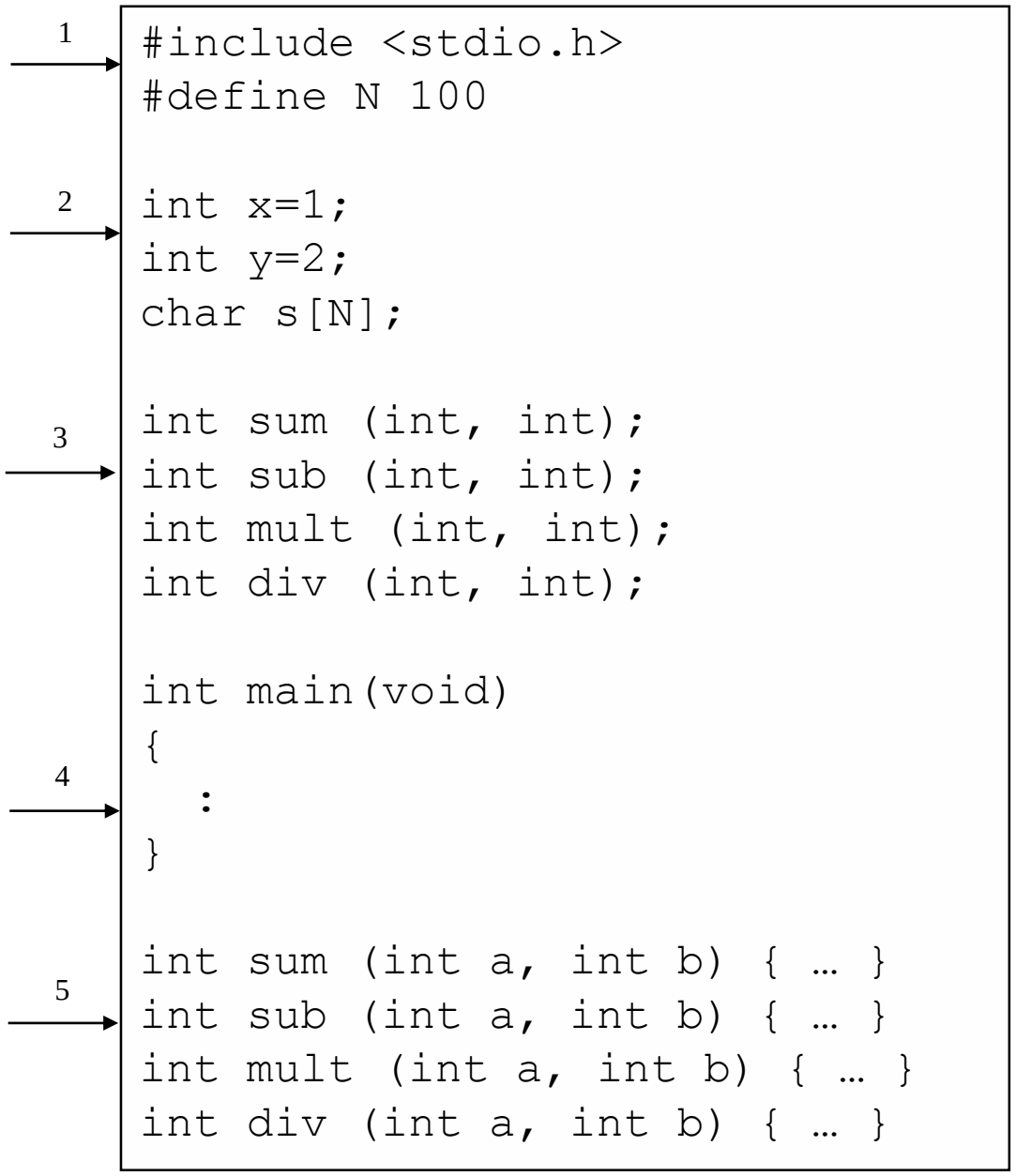
Output :
0 -1217044492

# Recommendation of a Program Structure

**Summarize**

It is considered a good programming practice if a program is being designed using below structure:

1. Declare all includes and defines.
2. Declare all global variables.
3. Declare the functions prototype.
4. Declare the main function.
5. Declare all the rest of functions.



# extern variables and functions

util.c

```
int count = 666;
```

main.c

```
#include <stdio.h>
int main()
{
   printf("%d\n", count);
   return 1;
}
```

error: 'count' undeclared

main.c

```
#include <stdio.h>

extern int count = 555;

int main()
{
 printf("%d\n", count);
 return 1;
}
```

warning: cannot initialize external variable
error: multiple definition of `count'

main.c

```
#include <stdio.h>

extern int count;

int main()
{
   printf("%d\n", count);
   return 1;
}
```

Scope: all the file starting from extern

Output :
666

**gcc -ansi -Wall -pedantic main.c util.c**

main.c

```
#include <stdio.h>
int main()
{
   extern int count;
   printf("%d\n", count);
   return 1;
}
```

Scope: main

Output :
666

```
#include <stdio.h>

extern int count;

int main()
{
   count = 33;
   printf("%d\n", count);
   return 1;
}
```

Output :
33

main.c

```
#include <stdio.h>
int main()  {
    int x = getCount();
    printf("%d\n", x);
    return 1;
}
```

warning: implicit declaration of function ‘getCount

Output :
666

main.c

```
#include <stdio.h>
int getCount();
int main()  {
    int x = getCount();
    printf("%d\n", x);
    return 1;
}

int getCount()
{
    return 999;
}
```

error: multiple definition of `getCount'

util.c

```
int count = 666;
int getCount()
{
  return count;
}
```

main.c

```
#include <stdio.h>
extern int getCount(void);
int main()
{
    int x = getCount();
    printf("%d\n", x);
    return 1;
}
```

Output :
666

**gcc -ansi -Wall -pedantic main.c util.c**

main.c

```
#include <stdio.h>
int main()
{
  extern int getCount(void);
  int x = getCount();
  printf("%d\n", x);
  return 1;
}
```

Output :
666

```
#include <stdio.h>
int getCount(void);
int main()
{
  int x = getCount();
  printf("%d\n", x);
  return 1;
}
```

Output :
666

**gcc -ansi -Wall -pedantic main.c util.c**

util.c

```c
#include <stdio.h>
extern int count;
void print(void)
{
   printf("count is %d\n", count);
}
```

main.c

```c
#include <stdio.h>

int count;

int main()
{
   count = 666;
   print();
   return 1;
}
```

warning: implicit declaration of function 'print'

Output :
count is 666

main.c

```c
#include <stdio.h>

int count;

int main()
{
   extern void print(void);
   count = 666;
   print();
   return 1;
}
```

Output :
count is 666

main.c

```c
#include <stdio.h>

int count;
extern void print(void);

int main()
{
   count = 666;
   print();
   return 1;
}
```

Output :
count is 666

f1.c

```
int count = 666;
```

f2.c

```
int count = 555;
```

main.c

```
#include <stdio.h>

extern int count;

int main()
{
   printf("%d\n", count);
   return 1;
}
```

Compile time error: multiple definition of `count'

**gcc -ansi -Wall -pedantic main.c f1.c f2.c**

f1.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

f2.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

main.c

```
#include <stdio.h>

extern int my_sum(int a, int b);

int main()
{
   printf("%d\n", my_sum(1,2));
   return 1;
}
```

Compile time error: multiple definition of `my_sum'

**gcc -ansi -Wall -pedantic main.c f1.c f2.c**

f1.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

f2.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

In general it is not considered a good practice to include source files.

main.c

```
#include <stdio.h>
#include "f1.c"
#include "f2.c"

int main()
{
   printf("%d\n", my_sum(1,2));
   return 1;
}
```

Compile time error: redefinition of ‘my_sum’

**gcc -ansi -Wall -pedantic main.c**

f1.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

f2.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

In general it is not considered a good practice to include source files.

main.c

```
#include <stdio.h>
#include "f1.c"
#include "f2.c"

extern int my_sum(int a, int b);

int main()
{
   printf("%d\n", my_sum(1,2));
   return 1;
}
```

Compile time error: redefinition of ‘my_sum’

**gcc -ansi -Wall -pedantic main.c**

f1.h

```
int my_sum(int, int);
```

f2.h

```
int my_sum(int, int);
```

f1.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

f2.c

```
int my_sum(int a,int b)
{
 return a+b;
}
```

main.c

```
#include <stdio.h>
#include "f1.h"
#include "f2.h"

int main()
{
   printf("%d\n", my_sum(1,2));
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c → Compile time error: undefined reference to `my_sum'**
**gcc -ansi -Wall -pedantic main.c f1.c f2.c → Compile time error: multiple definition of `my_sum'**

f1.c

```c
#include <stdio.h>
extern int arr[3];
void show1()
{
    int i;
    for(i=0;i<3;i++)
      printf("%d\n",arr[i]);
}
```

f2.c

```c
#include <stdio.h>
extern int arr[2];
void show2()
{
    int i;
    for(i=0;i<2;i++)
      printf("%d ",arr[i]);
    putchar('\n');
}
```

main.c

```c
#include <stdio.h>

int arr[3] = {1,2,3};
extern void show1();
extern void show2();

int main()
{
    show1();
    show2();
    return 1;
}
```

**gcc -ansi -Wall -pedantic main.c f1.c f2.c**

Output:

```
1
2
3
1 2
```

f1.c

```c
#include <stdio.h>
extern int arr[3];
void show1()
{
    int i;
    for(i=0;i<3;i++)
      printf("%d\n",arr[i]);
}
```

f2.c

```c
#include <stdio.h>
extern int arr[2];
void show2()
{
    int i;
    for(i=0;i<2;i++)
      printf("%d ",arr[i]);
    putchar('\n');
}
```

main.c

```c
#include <stdio.h>
#include "f1.c"
#include "f2.c"

int arr[3] = {1,2,3};

int main()
{
    show1();
    show2();

    return 1;
}
```

In general it is not considered a good practice to include source files.

**gcc -ansi -Wall -pedantic main.c**

In file included from main.c::
f2.c: error: conflicting types for ‘arr’
f1.c: note: previous declaration of ‘arr’ was here

f1.c

```c
#include <stdio.h>
extern int arr[3];
void show1()
{
    int i;
    for(i=0;i<3;i++)
      printf("%d\n",arr[i]);
}
```

f2.c

```c
#include <stdio.h>
extern int arr[3];
void show2()
{
    int i;
    for(i=0;i<2;i++)
      printf("%d ",arr[i]);
    putchar('\n');
}
```

In general it is not considered a good practice to include source files.

**gcc -ansi -Wall -pedantic main.c**

main.c

```c
#include <stdio.h>
#include "f1.c"
#include "f2.c"

int arr[3] = {1,2,3};
int main()
{
    show1();
    show2();

    return 0;
}
```

Output:

```
1
2
3
1 2
```

util.c

```
static int count = 666;
int getCount()
{
  return count;
}
```

- Once a global variable is declared as static then its scope is only the file where it was declared in i.e. it can be accessed only from the file where it was declared in.
- A global static variable allows declaring a private global variable.

main.c

```
#include <stdio.h>
extern int getCount(void);
int main()
{
   int x = getCount();
   printf("%d\n", x);
   return 1;
}
```

Output :
666

main.c

```
#include <stdio.h>
int main()
{
   printf("%d\n", count);
   return 1;
}
```

main.c

```
#include <stdio.h>
extern int count;
int main()
{
   printf("%d\n", count);
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c util.c**

util.c

```
int count = 666;
static int getCount()
{
  return count;
}
```

- Once a function is declared as static then its scope is only the file where it was declared in i.e. it can be accessed only from the file where it was declared in.
- A static function allows declaring a private function.

main.c

```
#include <stdio.h>
int main()
{
   int x = getCount();
   printf("%d\n", x);
   return 1;
}
```

main.c

```
#include <stdio.h>
extern int getCount();
int main()
{
   int x = getCount();
   printf("%d\n", x);
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c util.c**

אין משמעות להגדרת static אם מבצעים include

util.c

```
static int count = 666;
static int getCount()
{
  return count;
}
```

**gcc -ansi -Wall -pedantic main.c**

In general it is not considered a good practice to include source files. Better if we declare header files instead. In our example better to declare util.h.

main.c

```
#include <stdio.h>
#include "util.c"

int main()
{
   int x = getCount();
   int y = count;
   printf("%d %d\n", x, y);
   return 1;
}
```

util.c

```
int count = 666;
int getCount()
{
  return count;
}
```

main.c

```
#include <stdio.h>

extern int count;
extern int getCount();

int main()
{
   int x = getCount();
   int y = count;
   printf("%d %d\n", x, y);
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c util.c**

util.h

```
int count = 666;
int getCount()
{
  return count;
}
```

main.c

```
#include <stdio.h>

extern int count;
extern int getCount();

int main()
{
   int x = getCount();
   int y = count;
   printf("%d %d\n", x, y);
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c util.h**

util.h

```
int count = 666;
int getCount()
{
  return count;
}
```

main.c

```
#include <stdio.h>
#include "util.h"

int main()
{
   int x = getCount();
   int y = count;
   printf("%d %d\n", x, y);
   return 1;
}
```

**gcc -ansi -Wall -pedantic main.c**
**gcc -ansi -Wall -pedantic main.c util.h**

Don't compile header files without including them as they don't create object files.

# Initializing Local and Global Variables

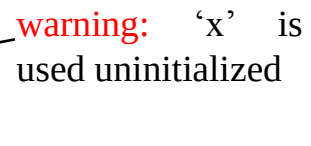
- local variables are not initialized by the system.
- Local uninitialized variables contain garbage values which already available at their memory location.
- Global variables are initialized automatically by the system.
- It is a good programming practice to initialize variables properly, to avoid unexpected results.

| **Global Variables Default Values** | |
|---|---|
| Data Type | Initial Default Value |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

```c
#include <stdio.h>
int x;
int main()
{
  printf("X: %d\n", x);
  return 0;
}
```

Output :
X: 0

```c
#include <stdio.h>
int main()
{
  int x;
  printf("X: %d\n", x);
  return 0;
}
```




Output :
X: -1216770060

**gcc -ansi -Wall -pedantic main.c**

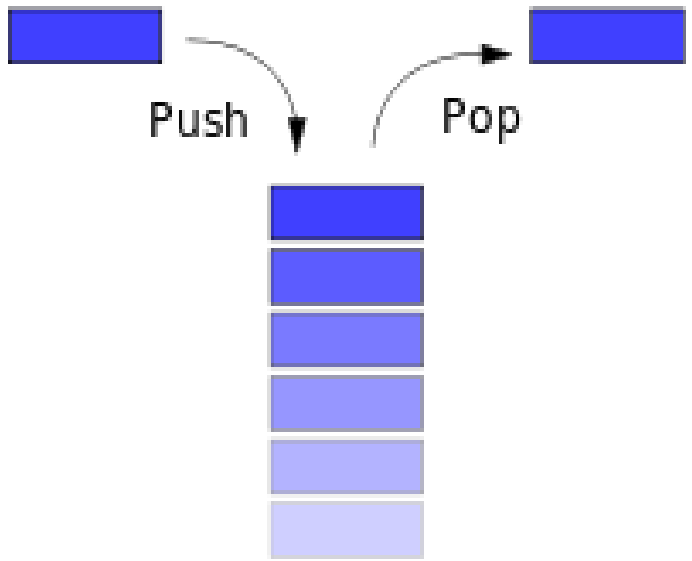
```c
#include <stdio.h>
char c;
int i;
float f;
double d;
int* p1;
char* p2;

int main()
{
   printf("X: %d\n", c);
   printf("X: %d\n", i);
   printf("X: %f\n", f);
   printf("X: %f\n", d);
   printf("X: %d\n", p1);
   printf("X: %d\n", p1);
  return 0;
}
```

```
Output :
X: 0
X: 0
X: 0.000000
X: 0.000000
X: 0
X: 0
```

# Stack
# מחסנית

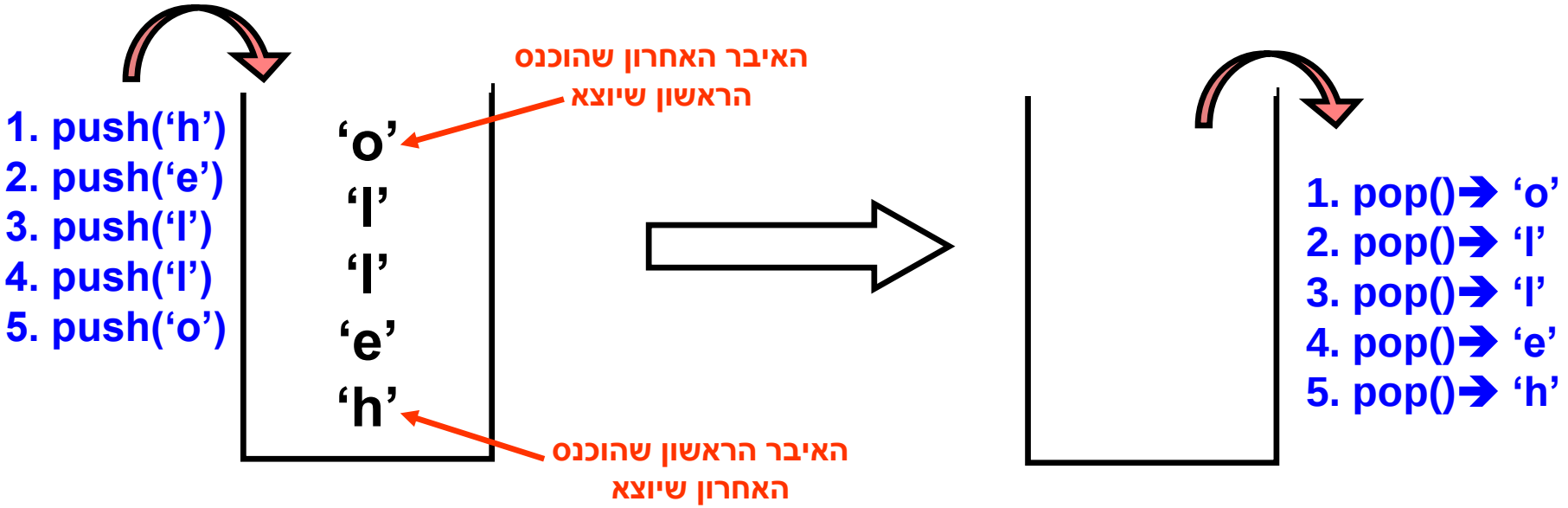
- מבנה נתונים ליניארי
- מחסנית ניתנת למימוש ע"י מערך או רשימה מקושרת
- האיברים במחסנית מנוהלים לפי עיקרון LIFO (last in first out) כלומר האיבר האחרון שהוכנס לרשימה הוא האיבר הראשון שיוצא ממנה.
- כמו בתור, חיסרון המימוש ע"י מערך הוא הקצאה סטטית בזיכרון דבר שגורם למחסנית להיות מלאה באיזשהו שלב




http://en.wikipedia.org/wiki/Stack_(abstract_data_type)

# פעולות בסיסיות שניתן לבצע על מחסנית

- פעולות בסיסיות שניתן לבצע על מחסנית :
  - ✓ push הוספת איבר לראש הרשימה O(1)
  - ✓ pop הסרת איבר מראש הרשימה O(1)



# Stack Implementation using Array

stack.c

```c
#include <stdio.h>
#define SIZE 3
static int sp = 0;
static double stack[SIZE];
void push(double x) {
  if (sp < SIZE) {
    stack[sp++] = x;
    printf("%f was added to the stack.\n", x);
  }
  else {
    printf("Stack is full. %f was not added.\n", x);
  }
}

double pop(void) {
  if (sp > 0)  {
     double c = stack[--sp];
     printf("%f was removed from the stack.\n", c);
     return c;
  }
  else {
     printf("Stack is empty\n");
     return (double)0;
  }
}
```

stack.h

```c
void enqueue(double);
double dequeue(void);
```

main.c

```c
#include "stack.h"

int main(void)
{
  push(1.0);
  push(2.0);
  push(3.0);
  push(4.0);

  pop();
  push(4.0);

  return 1;
}
```

```
student@ubuntu:~/Desktop/test$ readelf -s stack.o

Symbol table '.symtab' contains 15 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS stack.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     4 OBJECT  LOCAL  DEFAULT    4 sp
     6: 00000008    24 OBJECT  LOCAL  DEFAULT    4 stack
     7: 00000000     0 SECTION LOCAL  DEFAULT    5
     8: 00000000     0 SECTION LOCAL  DEFAULT    7
     9: 00000000     0 SECTION LOCAL  DEFAULT    8
    10: 00000000     0 SECTION LOCAL  DEFAULT    6
    11: 00000000    95 FUNC    GLOBAL DEFAULT    1 push
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    13: 0000005f    84 FUNC    GLOBAL DEFAULT    1 pop
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
student@ubuntu:~/Desktop/test$
```

readelf is tool which allows reading binary files. It allows investigating **symbols**, **sections**, and **segments** areas of a program.

**gcc -ansi -Wall -pedantic  main.c stack.c**

**Output :**

```
1.000000 was added to the stack.
2.000000 was added to the stack.
3.000000 was added to the stack.
Stack is full. 4 was not added.
3.000000 was removed from the stack.
4.000000 was added to the stack.
```

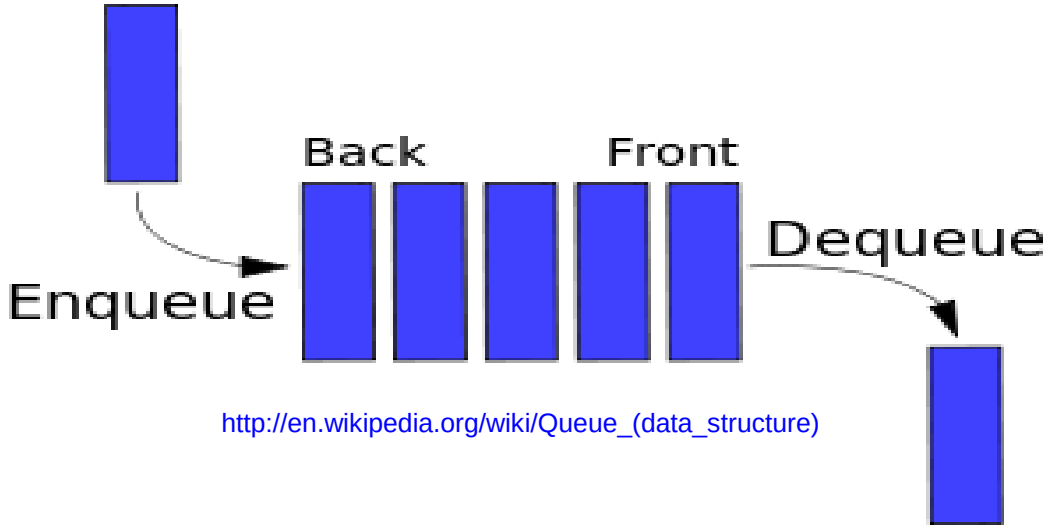
# Queue
# תור

- כמו מערך ורשימה מקושרת תור הוא מבנה נתונים ליניארי לכן ניתן למימוש ע"י שימוש במערך או רשימה מקושרת .
- האיברים בתור מנוהלים לפי עיקרון FIFO (first in first out) כלומר האיבר הראשון שהוכנס לרשימה הוא האיבר הראשון שיוצא ממנה.
- חיסרון המימוש ע"י מערך הוא הקצאה סטטית בזיכרון דבר שגורם לתור להיות מלא באיזשהו שלב.

דוגמאות לתור (תהליך ההדפסה במחשב
תור המתנה בבנק)


# פעולות בסיסיות שניתן לבצע על תור

- פעולות בסיסיות שניתן לבצע על תור :
  - ✓enqueue הוספת איבר לסוף התור O(1)
  - ✓dequeue הסרת איבר מראש התור O(1)
  - ✓empty מחזירה אמת אם התור ריק O(1)


http://en.wikipedia.org/wiki/Queue_(data_structure)

# Queue Implementation using Array

queue.c

Version 1

```
#include <stdio.h>
#define SIZE 3
int front, rear, counter;
double arr[SIZE];
void enqueue(double x) {
  if (counter < SIZE) {
    if ( rear == SIZE)
       rear = 0;
    arr[rear++] = x;
    counter++;
    printf("%f was inserted to the Queue.\n", x);
  } else
    printf("Queue is full!\n");
}
double dequeue(void) {
  double ret = 0;
  if (counter > 0) {
    ret = arr[front++];
    if(front == SIZE)
      front = 0;
    counter--;
    printf("%f was removed from the Queue.\n", ret);
  }
  return ret;
}
```

queue.h

```
void enqueue(double);
double dequeue(void);
```

main.c

```
#include "queue.h"

int main(void)
{
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  enqueue(4.0);

  dequeue();
  enqueue(4.0);

  return 1;
}
```

```
student@ubuntu:~/Desktop/test$ gcc -ansi -Wall -pedantic -c main.c queue.c
student@ubuntu:~/Desktop/test$ readelf -s main.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000    89 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND enqueue
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND dequeue
student@ubuntu:~/Desktop/test$ readelf -s queue.o

Symbol table '.symtab' contains 17 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS queue.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000004     4 OBJECT  GLOBAL DEFAULT  COM front
    10: 00000004     4 OBJECT  GLOBAL DEFAULT  COM rear
    11: 00000004     4 OBJECT  GLOBAL DEFAULT  COM counter
    12: 00000008    24 OBJECT  GLOBAL DEFAULT  COM arr
    13: 00000000   120 FUNC    GLOBAL DEFAULT    1 enqueue
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    16: 00000078   101 FUNC    GLOBAL DEFAULT    1 dequeue
student@ubuntu:~/Desktop/test$
```

**Complie time error:**

```
undefined reference to `enqueue‘
undefined reference to `enqueue‘
undefined reference to `enqueue‘
undefined reference to `enqueue‘
undefined reference to `dequeue‘
undefined reference to `enqueue'
```

queue.c is not compiled So enqueue and dequeue are not declared

**gcc -ansi -Wall -pedantic main.c**

queue.c

```
#include <stdio.h>
#define SIZE 3
int front, rear, counter;
double arr[SIZE];
void enqueue(double x) {
  if (counter < SIZE) {
     if ( rear == SIZE)
        rear = 0;
     arr[rear++] = x;
     counter++;
     printf("%f was inserted to the Queue.\n", x);
  } else
    printf("Queue is full!\n");
}
double dequeue(void) {
  double ret = 0;
  if (counter > 0) {
     ret = arr[front++];
     if(front == SIZE)
       front = 0;
     counter--;
     printf("%f was removed from the Queue.\n", ret);
  }
  return ret;
}
```

**gcc -ansi -Wall -pedantic main.c queue.c → a.out**

queue.h

```
void enqueue(double);
double dequeue(void);
```

main.c

```
#include "queue.h"

int main(void)
{
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  enqueue(4.0);

  dequeue();
  enqueue(4.0);

  return 1;
}
```

1. gcc –c main.c
2. gcc –c queue.c

→ main.o , queue.o

3. gcc main.o queue.o

→ a.out

queue.o must be linked with main.o

**Output :**

```
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.
Queue is full!
1.000000 was removed from the Queue.
4.000000 was inserted to the Queue.
```

```
#include "queue.h"
int main(void) {
  counter=5;
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Compile time error:**
error: ‘counter’ undeclared

```
#include "queue.h"
int main(void) {
  int counter=5;
  counter++;
  printf("%d\n", counter);
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
6
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.

What will be the result if we define double counter = 5?

```
#include "queue.h"
counter=5;
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
warning: ‘counter’ data definition has no type
warning: type defaults to ‘int’ in declaration of ‘counter’
Queue is full!
Queue is full!
Queue is full!

```
#include "queue.h"
int counter=5;
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
Queue is full!
Queue is full!
Queue is full!

queue.c

```
#include <stdio.h>
#include "queue.h"
extern int front, rear, counter;
extern double arr[SIZE];
void enqueue(double x) {
  if (counter < SIZE) {
    if ( rear == SIZE)
       rear = 0;
    arr[rear++] = x;
    counter++;
    printf("%f was inserted to the Queue.\n", x);
  } else
    printf("Queue is full!\n");
}
double dequeue(void) {
  double ret = 0;
  if (counter > 0) {
    ret = arr[front++];
    if(front == SIZE)
      front = 0;
    counter--;
    printf("%f was removed from the Queue.\n", ret);
  }
  return ret;
}
```

Version 2

queue.h

```
#define SIZE 3
void enqueue(double);
double dequeue(void);
```

main.c

```
#include "queue.h"
int front=0, rear, counter;
double arr[SIZE];
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  enqueue(4.0);

  dequeue();
  enqueue(4.0);

  return 1;
}
```

```
student@ubuntu:~/Desktop/test$ gcc -ansi -Wall -pedantic -c main.c queue.c
student@ubuntu:~/Desktop/test$ readelf -s main.o

Symbol table '.symtab' contains 16 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000     4 OBJECT  GLOBAL DEFAULT    4 front
    10: 00000004     4 OBJECT  GLOBAL DEFAULT  COM rear
    11: 00000004     4 OBJECT  GLOBAL DEFAULT  COM counter
    12: 00000008    24 OBJECT  GLOBAL DEFAULT  COM arr
    13: 00000000    89 FUNC    GLOBAL DEFAULT    1 main
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND enqueue
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND dequeue
student@ubuntu:~/Desktop/test$ readelf -s queue.o

Symbol table '.symtab' contains 17 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS queue.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000   120 FUNC    GLOBAL DEFAULT    1 enqueue
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND counter
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND rear
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND arr
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    15: 00000078   101 FUNC    GLOBAL DEFAULT    1 dequeue
    16: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND front
student@ubuntu:~/Desktop/test$
```

**Output :**

```
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.
Queue is full!
1.000000 was removed from the Queue.
4.000000 was inserted to the Queue.
```

**gcc -ansi -Wall -pedantic main.c queue.c → a.out**

```
#include "queue.h"
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Complie time error:**
undefined reference to `counter‘
undefined reference to `front‘
undefined reference to `rear‘
undefined reference to `arr'

```
#include "queue.h"
int front, rear, counter;
double arr[SIZE];
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.

```
#include "queue.h"
int main(void) {
  int front, rear, counter;
  double arr[SIZE];
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Complie time error:**
undefined reference to `counter‘
undefined reference to `front‘
undefined reference to `rear‘
undefined reference to `arr'

queue.c

```
#include "queue.h"
static int front=0, rear=0, counter=0;
static double arr[SIZE];
void enqueue(double x) {
  if (counter < SIZE) {
     if ( rear == SIZE)
        rear = 0;
     arr[rear++] = x;
     counter++;
     printf("%f was inserted to the Queue.\n", x);
  } else
    printf("Queue is full!\n");
}
double dequeue(void) {
  double ret = 0;
  if (counter > 0) {
     ret = arr[front++];
     if(front == SIZE)
       front = 0;
     counter--;
     printf("%f was removed from the Queue.\n", ret);
  }
  return ret;
}
```

Version 3

queue.h

```
#include <stdio.h>
#define SIZE 3
void enqueue(double);
double dequeue(void);
```

main.c

```
#include "queue.h"
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  enqueue(4.0);

  dequeue();
  enqueue(4.0);

  return 1;
}
```

```
student@ubuntu:~/Desktop/test$ readelf -s main.o

Symbol table '.symtab' contains 12 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     0 SECTION LOCAL  DEFAULT    5
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000    89 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND enqueue
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND dequeue
student@ubuntu:~/Desktop/test$ readelf -s queue.o

Symbol table '.symtab' contains 17 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS queue.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    4
     5: 00000000     4 OBJECT  LOCAL  DEFAULT    4 front
     6: 00000004     4 OBJECT  LOCAL  DEFAULT    4 rear
     7: 00000008     4 OBJECT  LOCAL  DEFAULT    4 counter
     8: 00000010    24 OBJECT  LOCAL  DEFAULT    4 arr
     9: 00000000     0 SECTION LOCAL  DEFAULT    5
    10: 00000000     0 SECTION LOCAL  DEFAULT    7
    11: 00000000     0 SECTION LOCAL  DEFAULT    8
    12: 00000000     0 SECTION LOCAL  DEFAULT    6
    13: 00000000   120 FUNC    GLOBAL DEFAULT    1 enqueue
    14: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND printf
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    16: 00000078   101 FUNC    GLOBAL DEFAULT    1 dequeue
student@ubuntu:~/Desktop/test$
```

**Output :**

```
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.
Queue is full!
1.000000 was removed from the Queue.
4.000000 was inserted to the Queue.
```

**gcc -ansi -Wall -pedantic main.c queue.c → a.out**

```
#include "queue.h"
int main(void) {
  counter = 5;
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Complie time error:**
error: ‘counter’ undeclared

```
#include "queue.h"
int main(void) {
  int counter = 5;
  counter++;
  printf("%d\n", counter);
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output :**
6
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.

```
#include "queue.h"
int counter = 5;
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.

```
#include "queue.h"
counter = 5;
int main(void) {
  enqueue(1.0);
  enqueue(2.0);
  enqueue(3.0);
  return 1;
}
```

**Output:**
warning: ‘coutner’ data definition has no type
warning: type defaults to ‘int’ in declaration of ‘counter’
1.000000 was inserted to the Queue.
2.000000 was inserted to the Queue.
3.000000 was inserted to the Queue.


END