Engineering School

Computer Architecture and Operating Systems Department

Degree: Artificial Intelligence

Subject: Fundamentals of Programming II

Introduction to C

Content

- Compiling and execution
- Program structure
- Constant and Variable Types
  - Data types
  - Constants
  - Variables
  - Arrays
- Expressions and Operators
  - Assignment statement
  - Arithmetic operators
  - Comparison
  - Logical expressions
- Control Statements
  - Branches
  - Loops
- Functions in C
  - Function arguments
  - Function scope

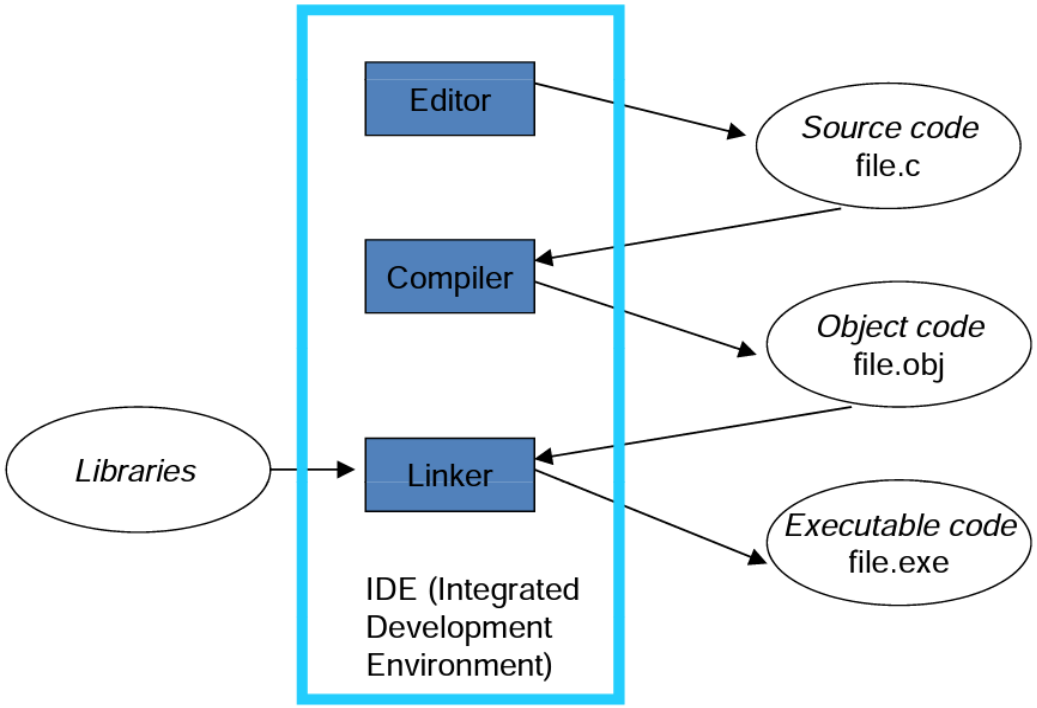
Compilation vs Interpretation

The difference between an interpreted and a compiled language lies in the result of the process of interpreting or compiling.

- **An interpreter interprets command by command executing a corresponding programs and changing our code into python byte code (machine code)**

- **A compiler produces an executable file (first, a code is translated into assembly language and then into machine code)**

Compiling and execution



C Programming

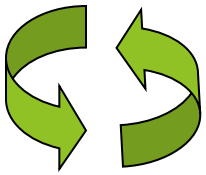
C++

Java

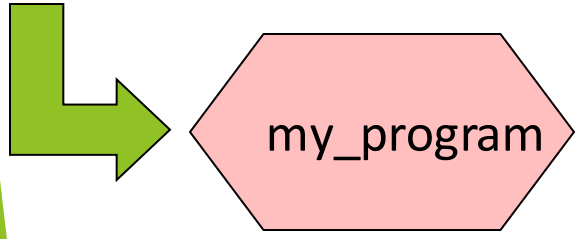
```
L0:  MOV   R1, #a       ; Address of a
     MOV   R2, #b       ; in R1, of b in R2

L1:  LD    R3, (R1)     ; Inport bits in R3
     CMP   R3, #0       ; IF-condition
     BNE   L3           ;

L2:  MOV   R4, #1       ; IF-branch
     JMP   L4           ;

L3:  MOV   R4, #0       ; ELSE-branch

L4:  ST    (R2), R4     ;
     JMP   L1           ;
```

Compiling and execution

```
#include <stdio.h>
/* The simplest C Program */
int main (int argc, char *argv[])
{
  printf ("Hello World\n");
  return 0;
}
```



```
$ gcc -g my_program.c -o my_program
ex.c: In function `main':
ex.c:6: parse error before `x'
ex.c:5: parm types given both in parmlist and separately
ex.c:10: warning: control reaches end of non-void function
ex.c: At top level:
ex.c:11: parse error before `return'
```



1. Write text of program (source code) using an editor, save as file e.g. my_program.c

2. Run the compiler to convert program from source to an "executable" (binary code):

   $ gcc my_program.c –o my_program

3-N. If any error, compiler gives errors and warnings; edit source file, fix it, and re-compile

N. Run it and see if it works ☺

```
$ ./my_program
Hello World
$
```

Program structure

- A **C** program is composed by one or more functions, and one of them is called **main**.
- The program always starts its execution from function **main**. From this function it is possible to call other functions.
- C functions have
  - Header
  - May include other functions (name and the parameters, if any)
  - Variables declaration
  - The sequence of sentences to be executed
  - Type of return value (may return zero, one or more values)

Program structure

```
#include <libraries.h>

#define CONSTANTS

/* global variables */
/* This is a comment */

void function1( )
{
  // local variables
  // code of the function
  // This is also a comment
}

/* Main program*/
int main(int argc, char *argv[] )
{
  //local variables
  function1();
  return 0;
}
```

```
#include <stdio.h>
/* The simplest C Program */
int main (int argc, char *argv[])
{
  printf ("Hello World\n");
  return 0;
}
```

Can your program have more than one .c file?

#include inserts another file. ".h" files are called "header" files. They contain stuff needed to interface to libraries and code in other ".c" files.

This is a comment. The compiler ignores this.

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char *argv[])
{
  printf("Hello World\n");
  return 0;
}
```

The main() function is always where your program starts running.

Blocks of code ("lexical scopes") are marked by { … }

Return '0' from this function

Print out a message. '\n' means "new line".

Output

- The format of the `printf()` function call is:

$$\texttt{printf(}format\texttt{, }exp_1\texttt{, }exp_2\texttt{, }exp_3\texttt{,...,}\ exp_n\texttt{);}$$

where:

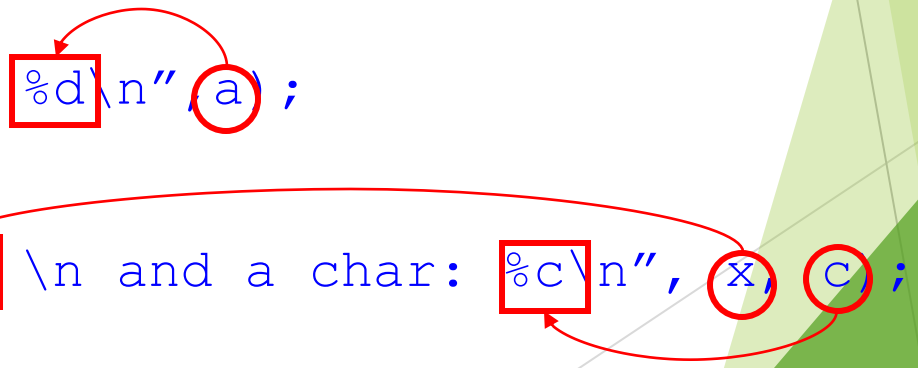
- *format* : string of the data output format
- exp_i : Expression to include inside the format

- #include <stdio.h>

Output

Example:

```
int a=3;
float x=23.0;
char c='A';
printf("Hello World!!\n");

printf("An integer %d\n", a);

printf("A real: %f \n and a char: %c\n", x, c);
```



Content

- Compiling and execution
- Program structure
- Constant and Variable Types
  - Data types
  - Constants
  - Variables
  - Arrays
- Expressions and Operators
  - Assignment statement
  - Arithmetic operators
  - Comparison
  - Logical expressions
- Control Statements
  - Branches
  - Loops
- Functions in C
  - Function arguments
  - Function scope

- Variables

```
<type> <name> ;
```

- Variable type indicates the amount of memory required to store that variable.
- Variable names can include letters, numbers and _ (underscore).
- C is case sensitive.

```
[Qualifyer] <type> <name1>,
 <name2>, <name3>=<value>,
          <name4> ;
```

What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value

```
char x;
char y='e';
```

Initial value of x is undefined

Initial value

Name

Type is single character (char)

Symbol	Addr	Value
	0	
	1	
	2	
	3	
x	4	?
y	5	‘e’ (101)
	6	
	7	
	8	
	9	
	10	
	11	
	12	

The compiler puts them somewhere in memory.

C types

Type	Size	Range
-		TRUE or FALSE
char	1 byte	-128 a 127
int	4 bytes	-2147483648 a 2147483647
float	4 bytes	1.2 E-38 a 3.4 E+38
double	8 bytes	1'7 E-308 a 1'7 E+308

Variables

- Simple declaration:
  - `char c;`
  - `unsigned int i;`
- Multiple declaration:
  - `char c,d;`
  - `unsigned int i,j,k;`
- Declaration and initialization:
  - `char c='A';`
  - `unsigned int i=133,j=1229;`

Constants

It allows you to declare a variable as non-changing

The values of variables declared as `const` remain constant throughout the life of the program.

Examples:

```
#define pi 3.141516
#define A 5
#define FileName "/home/user/tmp/test.txt"

const int index = 5;
const char president [16] = "Abraham Lincoln";
const int factor = 35;
const float twoThirds = 2./3.;
```

Variable scope

- Variables can be declared globally or locally
  - **Global** variables can be used from any point in the code and must be declared before main function (**main()**).
  - **Local** variables can only be used inside the function they have been declared. They should be declared after the { symbol.

Example

```
/* Variables declaration */
/* global variables*/
int a=10;
unsigned int b=20;

void function()/* declared function not being used */
{
   printf("a and b globals: %d, %d\n",a,b);  /*a=10,b=20*/
}

int main(int argc, char *argv[])   /* Shows two values */
{
   int b=4;
   /* b is 4*/
   printf("b is local and its value is %d\n",b);
   /* a is 10 */
   printf("a is global and its value is %d",a);
   return 0;
}
```

Arrays

- An **array** is an identifier of a set of data, all of them of the same type.
- Each component of the array is identified by an **index**. The index must be an integer and positive value.
- In **C** the first component of an array is identified by index value 0.
- The last component of an array is identif ied by index value N-1.

Arrays

- Syntax:

```
type name [][]...={ value1, value2...}
```

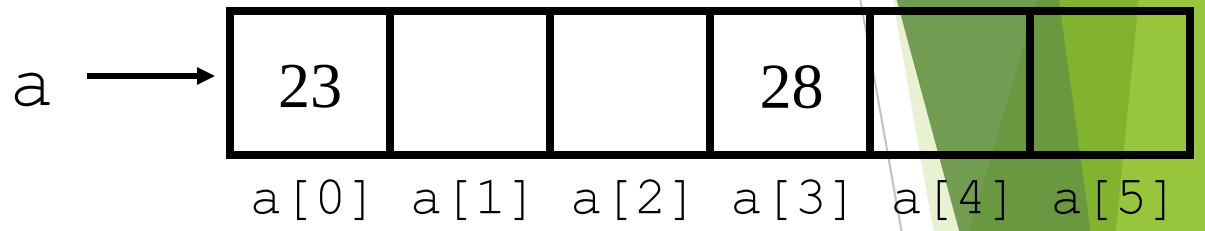
Examples:

```
int vector[10];
int vector[]={1,2,3,4,5,6,7,8};
int numbers[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
char vector[]="program";
char days[5][]={"monday","tuesday","wednesday","thursday","friday"};
char vector[]={'p','r','o','g','r','a','m', '\0'};
```

Vectors

- A vector is an *unidimensional array*.
- Declaration:

```
type name [size];
```



- Access to vectors:

```
/* Access an element*/
   name[index]

/* Access the whole vector*/
   name
```

first element -> 0, last element -> n-1

```
int a[6];
a[0]=23;
a[3]=a[0]+5;
for(i=0;i<6;i++)
  printf("%d", a[i]);
```

Vectors

```
int main() /*Multiplication table */
{
   int vector[10],i,num;
   printf(""Introduce one number: \n");
   scanf ("%d\n", &num);

   for (i=0; i<10; i++)
      vector[i]=i*num;

   for (i=0; i<10; i++)
      printf("%d mul %d = %d\n", i, num, vector[i]);

   return 0;
}
```

Matrix

- A matrix is a multidimensional array.
- Matrixes require an index for each dimension.
- Syntax:

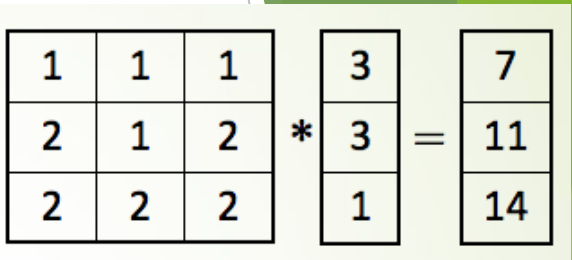
```
type name [size1][size2]...;
```

- Access:

```
name[ind1][ind2]
```

Matrix

```
/* Multiplication of a
matrix and a vector */
#define SIZE 3
int main()
{
  int vector[SIZE] = {3,3,1};
  int matrix[SIZE][SIZE] = { {1,1,1}, {2,1,2}, {2,2,2}};
  int result[SIZE];
  int i,j;

  /* initialization of result vector */
  for (i = 0; i < SIZE; i++)
       result[i] = 0;

  /* calculate the result */
  for (i = 0; i < SIZE; i++)
       for (j = 0; j < SIZE; j++)
              result[i] += matrix[i][j] * vector [j];
  /* print the result */
  for (i = 0; i < SIZE; i++)
       printf(" row %d -> %d \n", i, result[i]);
  return 0;
}
```



1	1	1		3		7
2	1	2	*	3	=	11
2	2	2		1		14

Content

- Compiling and execution
- Program structure
- Constant and Variable Types
  - Data types
  - Constants
  - Variables
  - Arrays
- Expressions and Operators
  - Assignment statement
  - Arithmetic operators
  - Comparison
  - Logical expressions
- Control Statements
  - Branches
  - Loops
- Functions in C
  - Function arguments
  - Function scope

Expressions and operators

- Assignment (=)
- Addition (+, +=, ++)
- Substraction (-, -=, --)
- Multiplication (*, *=)
- Division (/, /=)
- Module (rest) (%, %=)

Assignments

```
n = n + 3       can be expressed as     n += 3
k = k * ( x - 2 ) can be expressed as     k *= x - 2
```

Expressions and operators

```
int a = 1, b = 2,c = 3,r;


r = a + b;
r = r * 2;
r = r % 4;
r = r  / 3
c++;
++c;
r = c++ * 3;
c = 5;
r = ++c * 3;
```

Expressions and operators

```
int a = 1, b = 2,c = 3,r;


r = a + b;          /* r equal 3 */
r = r * 2;          /* r equal 6 */
r = r % 4;          /* r equal 2 */
r = r  / 3          /* r equal 0 */
c++;              /* c equal 4 */
++c;              /* c equal 5 */
r = c++ * 3;      /* r equal 15 and c equal 6 */
c = 5;            /* c equal 5 */
r = ++c * 3;      /* r equal 18 y c equal 6 */
```

Expressions and operators

- Greater than (>)
- Smaller than (<)
- Greater or equal than (>=)
- Smaller or equal than (<=)
- Is equal (==)
- Is different (!=)

Comparisons

These operands return **1** if condition is true and **0** if condition is false.

Expressions and operators

Logic operators

- There are three basic logic operators:
  - AND (**&&**)
  - OR (**||**)
  - NOT (**!**)

Example:

```
if( ( age >= 25 ) && ( age < 65 ) ||
    ( salary < 1250 ) && !fixed )
{
  wantsalaryincrease = TRUE;
}
```

Expressions and operators

Operator	Description
!	Negation
* / %	Multiplication/division/module
+ -	Addition/subtraction
< <= > >=	Relational operators
== !=	Iguality and difference
&&	AND
||	OR
? :	Conditional
= += -= *= /=	Assignments

Expressions and operators

- Examples

  (1 > 2 + 3 && 4)

  1 == 2 != 3

  16 + (2 * 5) - (6 / 2)

  (10 + 2) * (8 - 6) / 2

  20 + 300 – 50 / 10 % 2

Content

- Compiling and execution
- Program structure
- Constant and Variable Types
  - Data types
  - Constants
  - Variables
  - Arrays
- Expressions and Operators
  - Assignment statement
  - Arithmetic operators
  - Comparison
  - Logical expressions
- Control Statements
  - Branches
  - Loops
- Functions in C
  - Function arguments
  - Function scope

Control statements - branches

```
if (condition)
{
     sentence;
}
```

The sentence only will execute if condition is true: Otherwise, it will continue without executing the sentence.

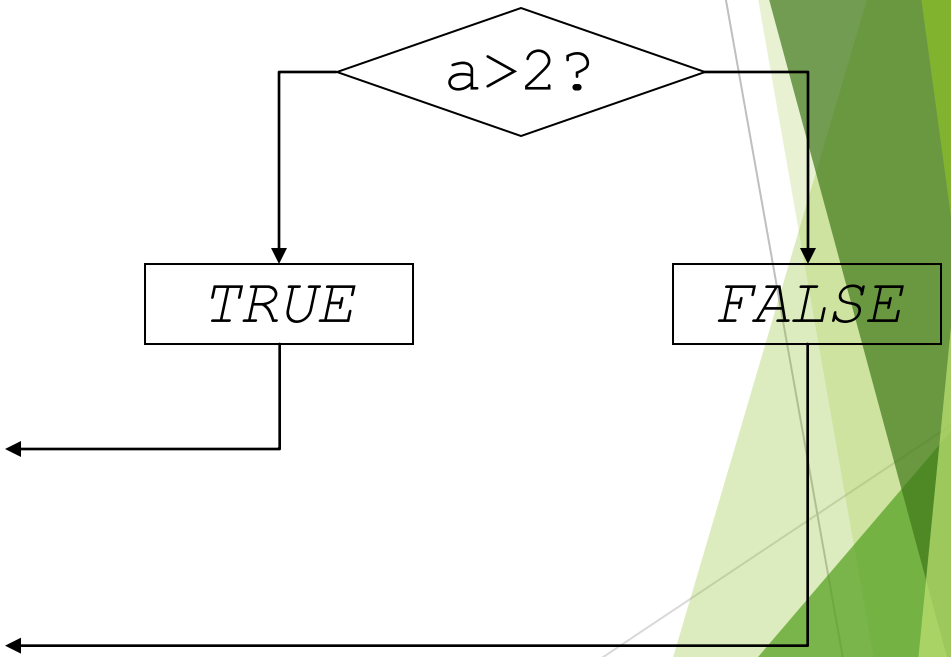
```
if (condition)
{
    sentence1;
}
else
{
    sentence 2;
}
```

If condition is true then it executes sentence 1. If condition is false then it executes sentence 2. In both cases it will continue after sentence 2.

Control statements - branches

if ... else

```
int main()
{
   int a=3, b;

   if(a>2)
   {
    b=100+a;
    printf("part if");
   }
   else
    printf("part else");
}
```



Control statements - branches

if ... else vs switch

```
if (condition)
{
  sentence1;
}
else if (condition)
{
  sentence2;
}
else if (condition)
{
  sentence3;
}
else
{
  sentence4;
}
```

```
switch (variable){
   case value1:
         sentence; break;
   case value2:
         sentence; break;
   default:
         sentence;
   }
```

Loops

while

Syntax:

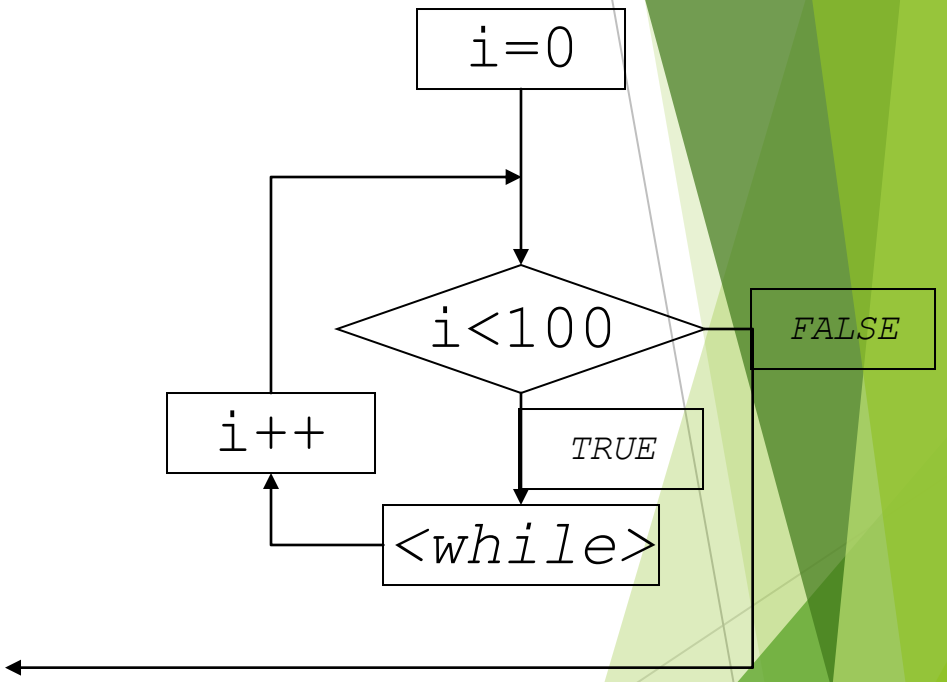
```
while (condition)
{
   sentence;
}
```

While sentence checks the condition before entering the loop.

If condition is not true then the program does not enter the loop.

Loops

```
int main()
{
   int i=0, ac=0;

   while(i<100)
   {
    printf("%d",i*i);
    ac+=i;
    i++;
   }
}
```



Loops

for

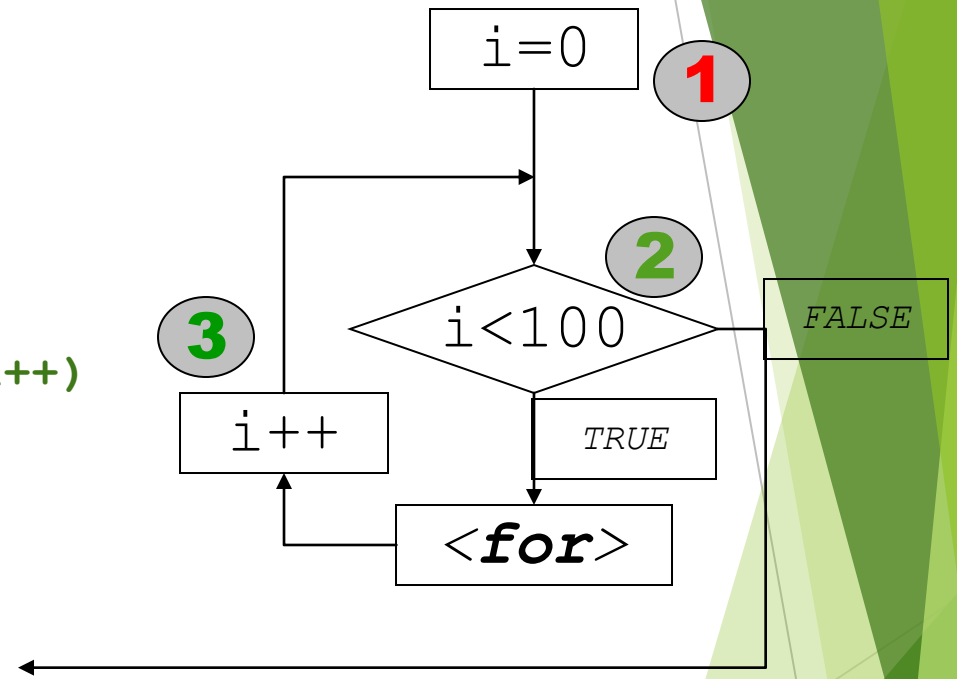
Syntax:

```
for (initialization; condition; increment)
{
   sentence1;
   sentence2;
}
```

The initialization indicates the control variable that guides the loop repetition.

Loops

```
int main()
{
   int i, ac=0;

   for(int = 0; i < 100; i++)
   {
    printf("%d",i*i);
    ac+=i;
   }
}
```



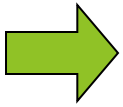
for(**initialization**, **condition_permanence**, **increment**)

1 2 3

Loops

The “for” loop is just shorthand for this “while” loop structure.

```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  i=0;
  while (i < exp) {
    result = result * x;
    i++;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf(“p = %f\n”, p);
  return 0;
}
```



```
float pow(float x, uint exp)
{
  float result=1.0;
  int i;
  for (i=0; (i < exp); i++) {
    result = result * x;
  }
  return result;
}

int main(int argc, char **argv)
{
  float p;
  p = pow(10.0, 5);
  printf(“p = %f\n”, p);
  return 0;
}
```

Loops

do…while

Syntax:

```
do
{
   sentence1;
   ...
   sentenceN;
} while (condition);
```

In this case the condition is verified at the end of the loop.
The loop is executed at least once.

Loops

- **break**

  This sentence is used to finish the execution of a loop. Program continues after the loop.

- **continue**

  It is used inside a loop to avoid the execution of the lasts sentences of some iterations.

Loops

```
int main()
{
  int i;
  for(i=0;i<100;i++)
  {
   if(i%2==0)
      continue; /*Start the next iteration*/
   if(i%17==0)
      break;    /*exit the loop*/
   printf("%d",i);
  }
}
```

Exercises

Write numbers from 1 to 10

Calculate the prime numbers in an interval

Exercises

```
# Python program to print prime numbers in an interval

def prime(x, y):
    prime_list = []
    for i in range(x, y):
        if i == 0 or i == 1:
            continue
        else:
            for j in range(2, int(i/2)+1):
                if i % j == 0:
                    break
            else:
                prime_list.append(i)
    return prime_list

# Driver program
starting_range = 2
ending_range = 10
lst = prime(starting_range, ending_range)
if len(lst) == 0:
    print("There are no prime numbers in this range")
else:
    print("The prime numbers in this range are: ", lst)
```

Exercises

Simple C program to print prime numbers in an interval

```
#include <stdio.h>

// This function is to check
// if a given number is prime
int isPrime (int n)
{
    if (n == 1 || n == 0)
        return 0;
    for (int i = 2; i < n/2+1; i++) {
        if (n % i == 0)
            return 0;
    }
    return 1;
}

// Driver code
int main()
{
    int N = 10;
    // check for the every number from 1 to N
    for (int i = 1; i <= N; i++) {
        if (isPrime(i)) {
            printf("%d ", i);
        }
    }
    return 0;
}
```

Content

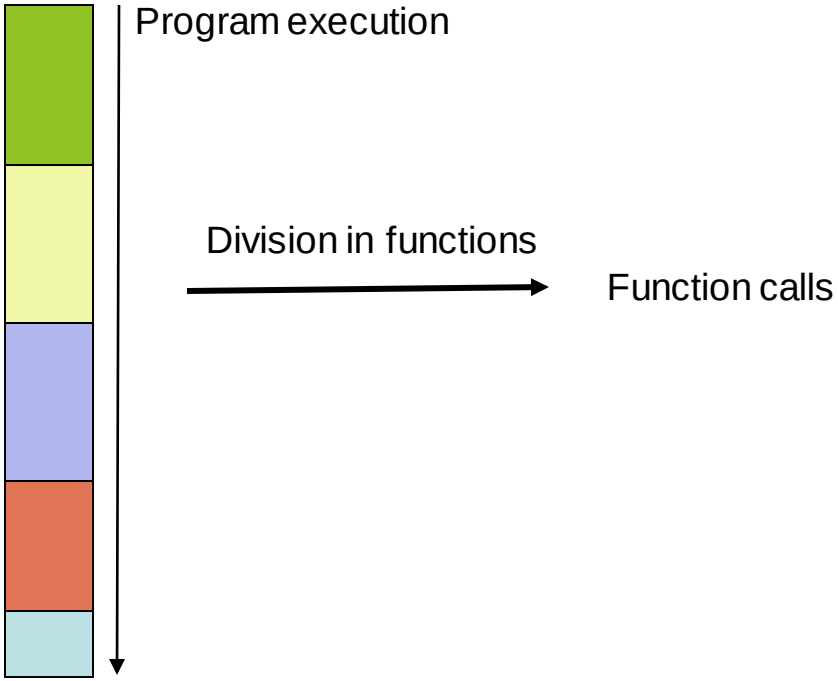
- Compiling and execution
- Program structure
- Constant and Variable Types
  - Data types
  - Constants
  - Variables
  - Arrays
- Expressions and Operators
  - Assignment statement
  - Arithmetic operators
  - Comparison
  - Logical expressions
- Control Statements
  - Branches
  - Loops
- Functions in C
  - Function arguments
  - Function scope

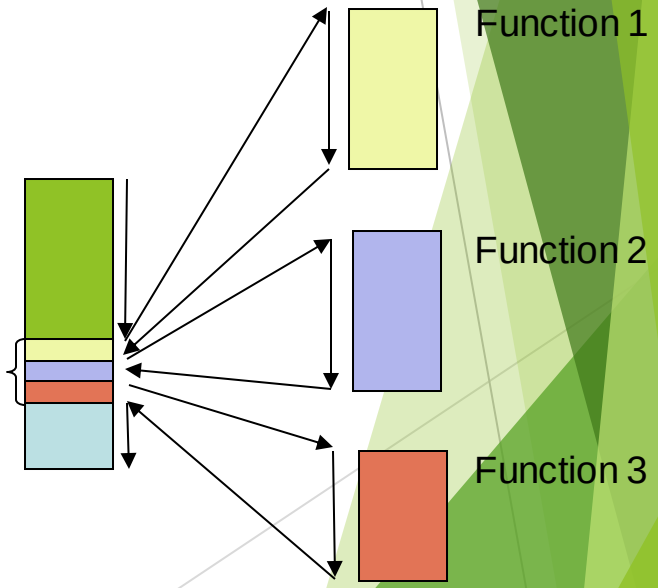
Functions in C

Divide programs in a set of smaller modules that can be better managed (implement, debug, re-use)

Advantages:

- Code simplification
- Clearer code and well structured
- Easier to find errors





Functions in C

A function is a block of code that performs a calculation and returns a value. It allows complicated programs to be parceled up into small blocks, each of which is easier to write, read, and maintain.

```
return_type name (argument1, argument2 ...)
{
   ...
   function body
   ...
}
```

By default functions return integers (int).

If no value is returned, void must be especified.

Functions in C

- Declaration of the function: Function header or prototype

```
int power(int n);
```

- Definition of the function: Function code

```
int power(int n)
{
   int ret = 1;
   ret = n*n;
   return (ret);
}
```

A function must be declared before it can be used.

Functions in C

Example

```
#include <stdio.h>
int a, b, c;
int addition (void);

int addition(void)
{
   int r;
   r = a + b;
   return (r);
}

int main()
{
  a = 2;
  b = 3;
   c = addition ();
   printf("%d + %d = %d",a, b, c);
  return 0;
}
```

Functions in C

- **Passing parameters by value**

  Any change in the parameter, inside the function, does not affect the original value of the parameter.

```
int addition (int, int, int);

void addition(int x, int y, int z)
{
  z = x + y;
  printf("%d + %d = %d", x, y, z);
}

int main()
{
  int a=2, b=3, c=25;
  addition (a, b, c);
  printf("%d + %d = %d", a, b, c);
  return 0;
}
```

```
/*Declaration*/
int foo (int x);

/*Call*/
value = foo(a);
```

Functions in C

- **Passing parameters by reference**

  What is passed to the function is not just the value, but the memory address. The modification inside the function is maintained after exiting the function.

- To pass the address it is necessary to indicate that the parameter is an address.

```
int foo (int  *x); /*Declaration*/

value = foo(&a);  /*Call*/
```

Functions in C

```
int addition (int, int, int);

void addition(int x, int y, int z)
{
  z = x + y;
  printf("%d + %d = %d", x, y, z);
}

int main()
{
  int a=2, b=3, c=25;
  addition (a, b, c);
  printf("%d + %d = %d", a, b, c);
  return 0;
}
```

What should be changed?

Functions in C

```
int addition (int, int, *int);

void addition(int x, int y, int * z)
{
  *z = x + y;
  printf("%d + %d = %d", x, y, *z);
}

int main()
{
  int a=2, b=3, c=25;
  addition (a, b, &c);
  printf("%d + %d = %d", a, b, c);
  return 0;
}
```

What should be changed?