C programming language

Ecole d'Ingénierie Digitale et d'Intelligence Artificielle (EIDIA)

Cycle Préparatoire formation Ingénieur

Khawla TADIST

Année 2021-2023

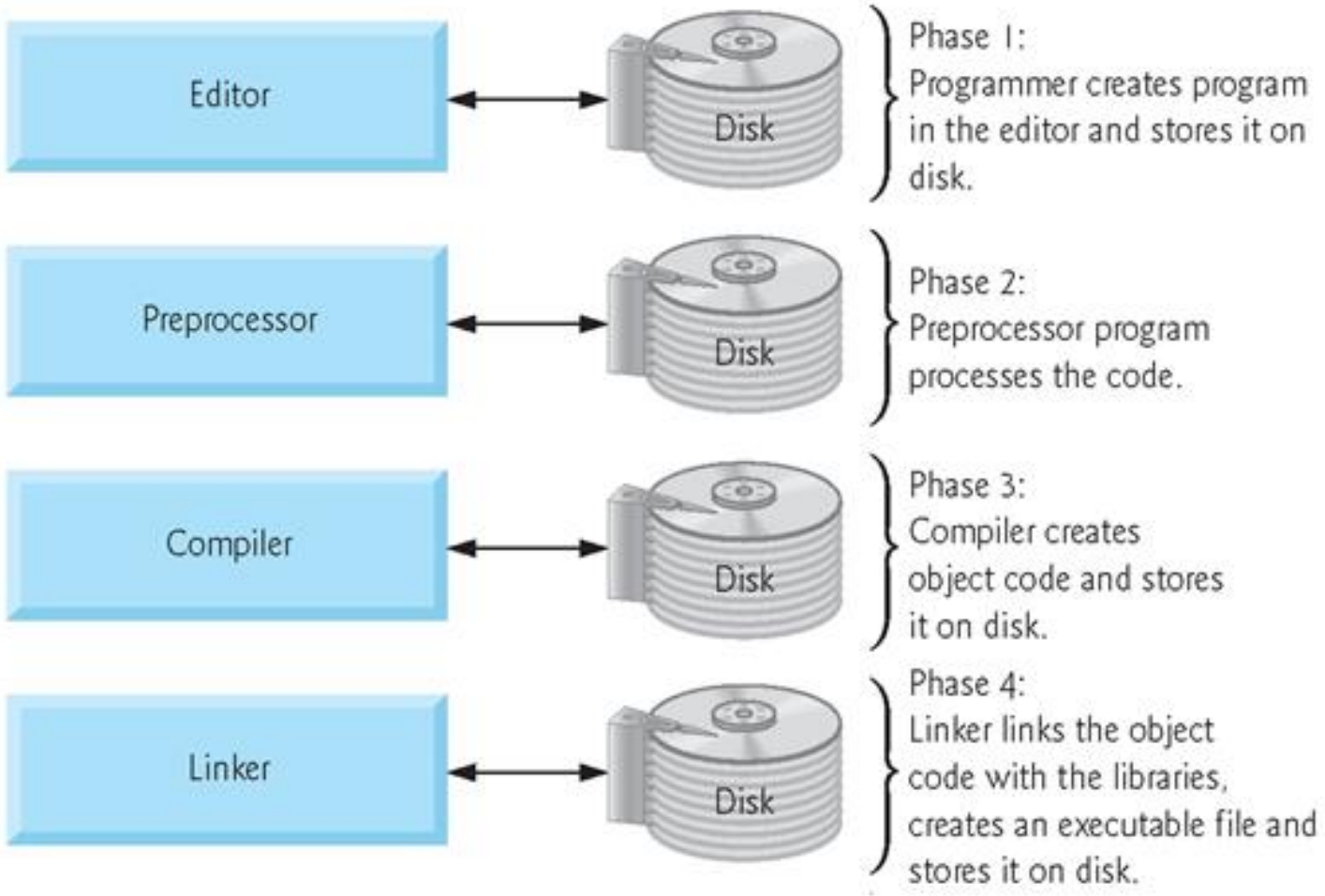
Introduction

- **Low-level** programming language
  - **Very close to the hardware resources** of the computer
    - Direct calls of OS services, direct access to registers and ports
- System programming
  - language (operating system), (embedded) systems…
- A user (programmer) can do almost everything
  - Initialization of the variables, release of the dynamically allocated memory, etc.

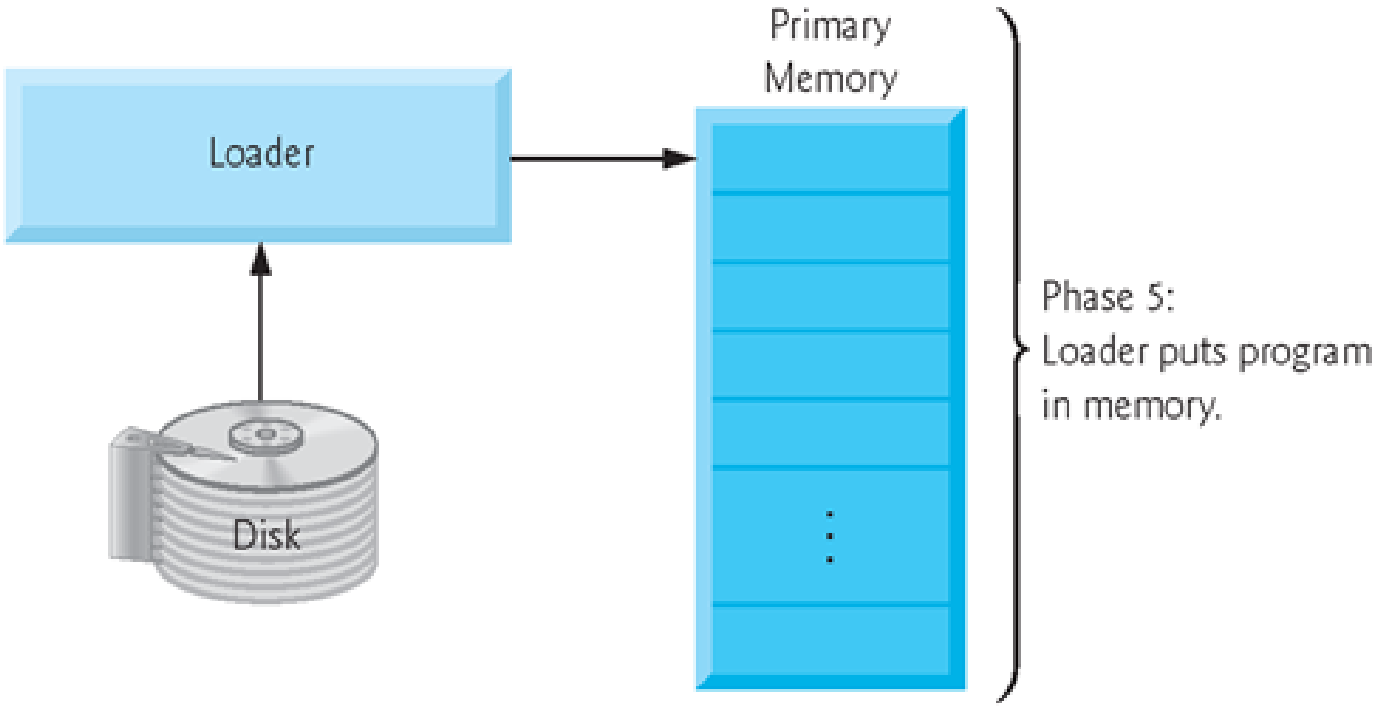
Running a C program

- Editor - code by programmer
- Compiling:
  - **Preprocess** – allows to ***adjust compilation*** according to the particular compilation environment ➔ The output is text (“source”) file,
  - **Compiler** – ***Translates*** source (text) file into machine readable form native (machine) code of the platform, bytecode, or assembler alternatively,
  - **Linker** – ***links with libraries*** and all compiled objects to make executable.
- Running the executable:
  - **Loader** – puts the program in memory to run it,
  - **CPU** – runs the program instructions.

Running a C program



Running a C program



Identifiers in C

- Identifiers are names of variables
  - Rules for the identifiers
    - Characters a–z, A–Z, 0–9 a _
    - Case sensitive,
    - Length of the identifier is not limited,
    - Keywords cannot be used as identifiers

Identifiers in C

- The following identifiers are reserved for the use as keywords, and may not be used otherwise:

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

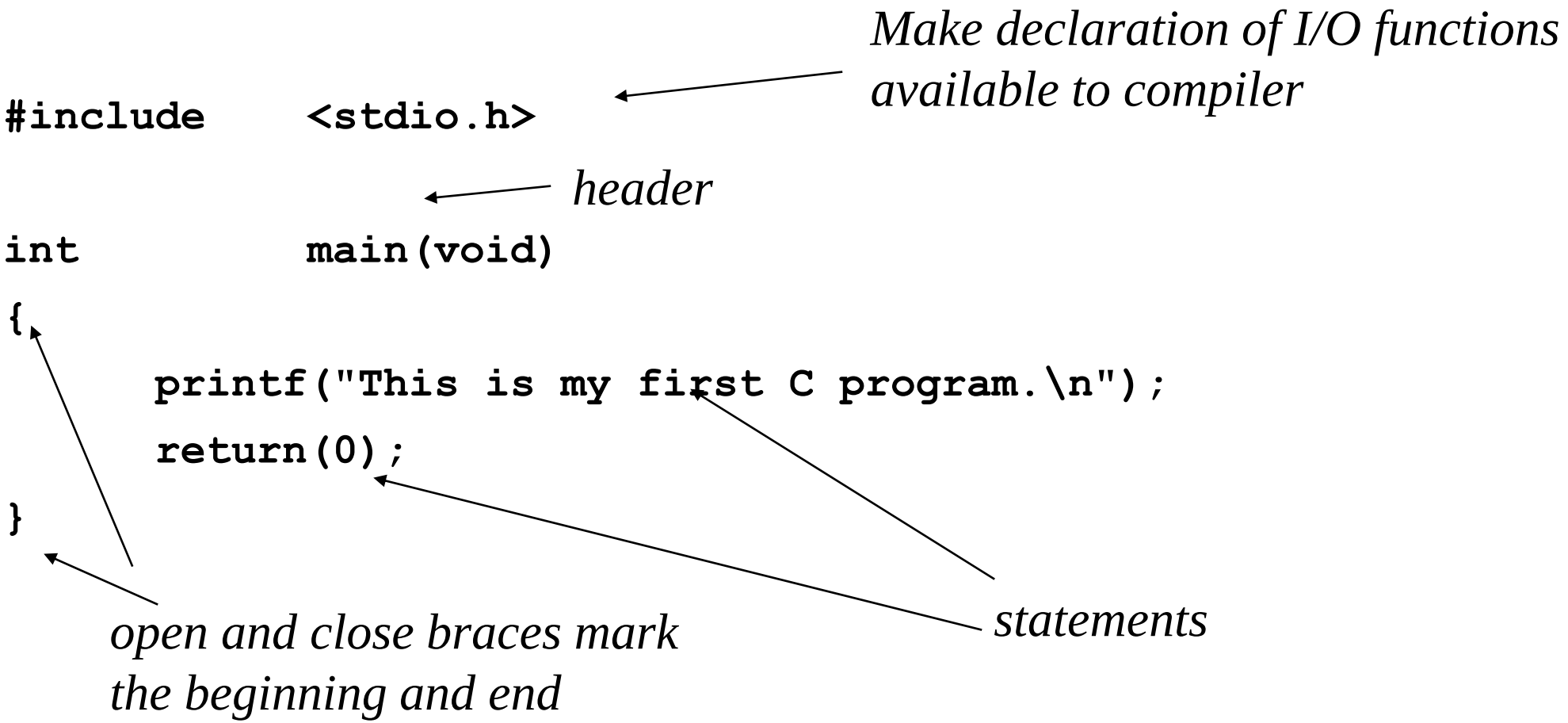
A first program in C

- Each executable program must have **at least one function and the function has to be main()**
  - The run of the program starts at the beginning of the function main(),

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5. printf("This is my first C program.\n");
6.
7. return(0);
8. }
```

  - The form of the main() function is prescribed

A first progam in C



Comments

- A comment is an ***explanatory text*** intended for ***the readers of the program*** and which ***has no impact on his compilation***.
- It consists of any characters placed between the symbols /* and */.
  - Here are some examples of comments:
    1. // My first Program
  - Or :
    1. /* comment extending
    2. on several lines
    3. of source program */.

Values and variables

- Values of the data types are called literals
  - C has 6 type of constants (literals)
    - Integer
    - Rational
    - Characters
    - Text strings
    - Enumerated Enum
    - Symbolic

Values and variables

- **Integer Literals**
  - Integer values are stored as one of the integer type:
    - int,
    - long,
    - short,
    - char
    - and their signed and unsigned variants

Values and variables

- **Literals of Rational Numbers**
  - Rational numbers can be written
    - with floating point – 13.1
    - or with mantissa and exponent – 31.4e-3 or 31.4E-3

Values and variables

- **Character Literals**
  - Format – A character in apostrophe ’A’, ’B’ or ’\n’
  - Value of the single character literal is the code of the character ’0’~ 48, ’A’~ 65

Values and variables

- **String literals**
  - Format – a sequence of character and control characters (escape sequences) enclosed in quotation (citation) marks
    - "This is a string constant with the end of line character \n"
  - String constants separated by white spaces are joined to single constant
    - "String literal" "with the end of the line character\n" is concatenate into "String literal with end of the line character\n"

Values and variables

- **String literals**
  - String literal is stored in ***an array of the type char*** terminated by the null character ’\0’ E.g., String literal “word” is stored as:

‘w’	‘o’	‘r’	‘d’	‘\0’

  - The size of the array must be about 1 item longer to store \0!

Values and variables

- **Constants of the Enumerated Type**
  - Format
    - By default, values of the enumerated type **starts from 0** and each other item **increase the value with one**

Values and variables

- **Constants of the Enumerated Type**
  - Format

```
enum
{
    SPADES,
    CLUBS,
    HEARTS,
    DIAMONDS
};
```

Value of CLUBS ?

  - The enumeration values are usually written in **uppercase**

Values and variables

- **Constants of the Enumerated Type**
  - Type – enumerated constant is the int type
    - Value of the enumerated literal can be used in loops

```
1. enum { SPADES = 0, CLUBS, HEARTS, DIAMONDS, NUM_COLORS };
2. for (int i = SPADES; i < NUM_COLORS; ++i)
3. {
4.  ...
5. }
```

Values and variables

- **Symbolic Constant – #define**
  - Format
    - The constant is **established by the preprocessor** command #define
    - Each #define must be on a new line
      - Example: #define SCORE (Usually written in uppercase)

Values and variables

- **Symbolic Constant – #define**
  - Symbolic constants can express **constant expressions**
    - #define MAX_1 ((10*6) - 3)
  - Symbolic constants **can be nested** #define MAX_2 (MAX_1 + 1)
  - Preprocessor performs the text ***replacement of the define constant*** by its value #define MAX_2 (MAX_1 + 1)
    - It is highly recommended **to use brackets** to ensure correct evaluation of the expression, e.g., the symbolic constant 5*MAX_1 with the outer brackets is 5*((10*6) - 3)=285 vs 5*(10*6) - 3=297.

Values and variables

- **Variable with a constant value modifier (const)**
  - Using the keyword const, a variable can be marked as constant
    - Compiler checks assignment and **do not allow to set a new value** to the variable.
  - A constant value can be defined as follows const float pi = 3.14159265;
  - In contrast to the symbolic constant #define PI 3.14159265
    - ***Constant values have type***, and thus it supports type checking

Expressions

- Expressions prescribe ***calculation value of some given input***
  - Expression is composed of ***operands***, ***operators***, and ***brackets***
  - Expression can be formed of
    - literals
    - variables
    - constants
    - operators
    - function calling
    - brackets

Expressions

- The order of operation evaluation is prescribed by ***the operator precedence and associativity***.
  - Example:
    - 10 + x * y // order of the evaluation 10 + (x * y)
    - 10 + x + y // order of the evaluation (10 + x) + y
      - * has higher priority than +
        - + is associative from the left-to-right

Expressions

- **Operators**
  - Operators are ***selected characters*** (or a sequences of characters) dedicated for ***writing expressions***
    - Binary operators
    - Unary operators
    - Ternary operator

Expressions

- **Operators**
  - Five types of binary operators can be distinguished
    - Arithmetic operators
    - Relational operators
    - Logical operators
    - Bitwise operators
    - Assignment operator

Expressions

- **Operators**
  - Unary operators
    - Indicating positive/negative value
    - Modifying a variable
    - Logical negation
    - Bitwise negation
  - Ternary operator

Expressions

- **Operators**
  - Five types of binary operators can be distinguished
    - ***Arithmetic operators*** – additive (addition/subtraction) and multiplicative (multiplication/division)
    - ***Relational operators*** – comparison of values (less than, greater than. . .)
    - ***Logical operators*** – logical AND and OR
    - ***Bitwise operators*** – bitwise AND, OR, XOR, NOT
    - ***Assignment operator*** – a variables taking a value =

Expressions

- **Operators**
  - Relational operators – comparison of values (less than, greater than, . . .)

Operator	Meaning	Example
==	equals	`x == y`
!=	is not equal to	`1 != 0`
>	greater than	`x+1 > y`
<	less than	`x-1 < 2*x`
>=	greater than or equal to	`x+1 >= 0`
<=	less than or equal to	`-x +7 <= 10`

Expressions

- **Operators**
  - Logical operators – logical AND and OR

Operator	Meaning
&&	AND
||	OR

Expressions

- **Operators**
  - Bitwise operators – bitwise AND, OR, XOR, NOT

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT

Expressions

- **Operators**
  - Bitwise operators – bitwise AND, OR, XOR, NOT
    - The operands are of integer type.
      - Operations are performed bit by bit following binary logic:

b1	b2	~b1	b1&b2	b1 | b2	b1^b2
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Expressions

- **Operators**
  - Unary operators
    - ***Indicating positive/negative value:*** + and − Operator − modifies the sign of the expression
    - ***Modifying a variable :*** ++ and −−
    - ***Logical negation:*** !

Expressions

- **Operators**
  - Unary operators
    - ***Modifying a variable :*** ++ and ––
      - Increment operator ++
        - Pre increment: ++var
        - Post increment: var++
      - Decrement operator --
        - Pre decrement: --var
        - Post decrement: var--

Expressions

- **Operators**
  - Unary operators
    - Pre increment: ++var

Values of x and y ?

```
int x= 4, y=5;
a=++x, b= ++y;          //Pre- increment operators (++x, ++y)
 printf("%d%d",a,b);              // printing value of a, b
```

    - 5, 6

Expressions

- **Operators**
  - Unary operators
    - Post increment: var++

```
int x= 4, y=5;
a=x++, b= y++;     // Post- increment operators (x++, y++)
printf("%d%d",a,b);          // printing value of a, b
```

    - 4, 5

Values of x and y ?

Expressions

- **Operators**
  - Unary operators
    - Pre decrement: --var

Values of x and y ?

```
int x= 4, y=5,a,b;
a=--x, b= --y;                  // Pre- decrement operators (--x, --y)
    printf("%d%d",a,b);                  // printing value of a, b
```

    - 3, 4

Expressions

- **Operators**
  - Unary operators
    - Post decrement: var--

```
int x= 4, y=5,a,b;
a=x--, b= y--;          // Post- decrement operators (x--, y--)
printf("%d%d",a,b);     // printing value of a, b
```

    - 4, 5

Values of x and y ?

Expressions

- **Operators**
  - Unary operators
    - ***Modifying a variable :*** ++ and −−

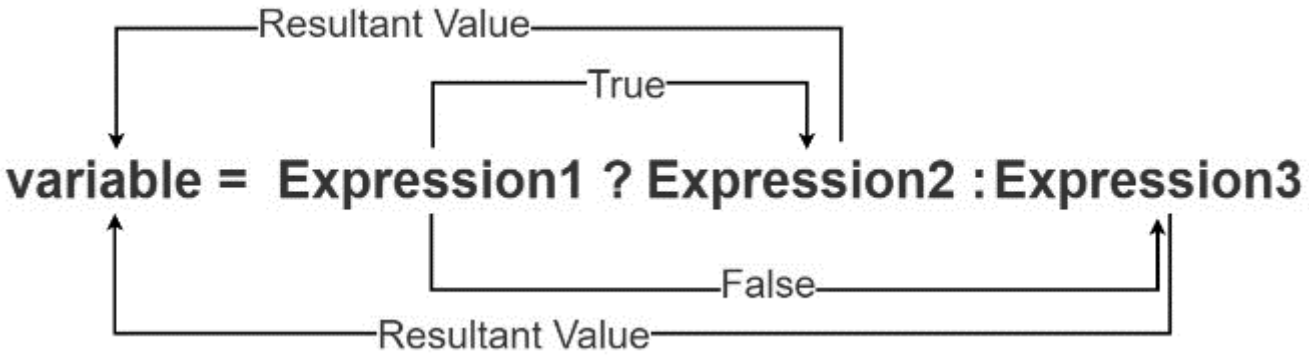
```
main()
{
int i=4,j=5,k=6,l;
l=i++  +  j++  +  ++k ;
printf("%d ",l);
}
```

    - 16

Values of i,j,k ?

Expressions

- **Operators**
  - Ternary operator
    - ***Conditional expression*** ? :



    - Example:
      - bool isnumber5 = number == 5 ? true : false;

Variables, Assignment Operator, and Assignment Statement

- Variables are defined by the **type and name**
  - Name of the variable are usually in lowercase
  - Multi-word variables can be written with underscore _ Or we can use CamelCase
  - Each variable is defined at new line (for organization purposes)

```
1. int n;
2. int number_of_items;
3. int numberOfItems;
```

  - Variables can be defined in the same line

Variables types

Type	Description	Size
char	Signed character. Characters are enclosed in single quotes.	1
double	Double precision number	8
int	Signed integer	4
float	Floating point number	4
long (int)	Signed long integer	4
long long (int)	Signed very long integer	8
short (int)	Short integer	2
unsigned char	Unsigned character	1
unsigned (int)	Unsigned integer	4
unsigned long (int)	Unsigned long integer	4
unsigned long long (int)	Unsigned very long integer	8
unsigned short (int)	Unsigned short integer	2

Variables, Assignment Operator, and Assignment Statement

- Assignment is setting the value to the variable,
  - The value is stored at the memory location referenced by the variable name
- Assignment operator “l_value = expression”
  - The side is the so-called ***l-value – location-value***, must represent ***a memory location*** where the value can be stored.
- Assignment statement must have the assignment operator **=** and **;**

Basic Arithmetic Expressions

- For an operator of the numeric types **int** and **double**, the following operators are defined (also for char, short, and float numeric types).
  - Unary operator for changing the sign −
  - Binary addition + and subtraction −
  - Binary multiplication * and division /

Basic Arithmetic Expressions

- **Implicit conversion**
  - Operands can be integers or reals except for % which **only acts on integers**
  - When the types of the two operands are different, there is a **conversion implicit** in strongest type
    - The / operator returns an **integer quotient** if **both operands are integers**
      - 5 / 2 = ?
        - 2
    - It returns a **real quotient** if at **least one of operands is a real**
      - 5.0 / 2 = ?
        - 2.5

Basic Arithmetic Expressions

- **Implicit conversion**
  - Short and char types are **always converted** to int regardless of the other operands
    - The conversion is generally done according to a hierarchy which does not alter values
      - int → long → float → double → long double

Basic Arithmetic Expressions

- **Implicit conversion**
  - Example1:
    - n * x + p (int n,p; float x)
      - Priority execution of n * x: conversion of n to float
      - Execution of the addition: conversion of p into float
        - 5 * 3.5 + 3

Basic Arithmetic Expressions

- **Implicit conversion**
  - Exemples :
    - n * p + x (int n ; long p ; float x)

n * p + x

n is conversed to long

Multiplied by p

n and p are of type long

The result is converted to float

Added to x

The result is of a type float

Standard Inputs/Outputs

- An executed program within Operating System (OS) environments has assigned (usually text-oriented) **standard inputs** (stdin) and **standard outputs** (stdout)

Standard Inputs/Outputs

- The stdin and stdout streams can be utilized for **communication with a user**
  - Basic function for text-based input is **getchar()** and for the output **putchar()**
    - Both are defined in the standard C library
  - For **parsing numeric** values the scanf() function can be utilized
  - The function printf() provides **formatted output**, (e.g., a number of decimal places)

Formatted Output – printf()

- Numeric values can be printed to the standard output using printf()
  - The first argument is the **format string** that defines how the values are printed
    - The conversion specification starts with the character '%'
      - A text string not starting with % is printed as it is
    - Basic **format strings** to print values of particular types are:
      - char %c
      - int %i, %x, %o,%d
      - float %f, %e, %g, %a
      - double %f, %e, %g, %a

Formatted Intput – scanf()

- Numeric values from the standard input can be read using the scanf() function
- The argument of the function is a **format string**
  - Syntax is similar to printf()
    - **It is necessary to provide a memory address of the variable to set its value from the stdin**

Formatted Intput – scanf()

- Example of readings integer value and value of the double type

```
1.  #include
2.  int main(void)
3.  {
4.  int i;
5.  double d;
6.  printf("Enter int value: ");
7.  scanf("%i", &i); // operator & returns the address of i
8.  printf("Enter a double value: ");
9.  scanf("%f", &d);
10. printf("You entered %i and %f\n", i, d);
11. return 0;
12. }
```

Selection statements

- Selection statements choose one of several flows of control.
  - if (expression) statement
  - if (expression) statement else statement
  - switch (expression) statement

Selection statements

- Selection statements choose one of several flows of control.
  - if (expression) statement
  - if (expression) statement else statement

```
1. If (boolean_expression 1)
2. { statements }
3.  else if ( boolean_expression 2)
4. { statements }
5. else if ( boolean_expression 3)
6.  { statements }
7. else
8. { statements }
```

Selection statements

- Selection statements choose one of several flows of control.
  - switch (expression) statement

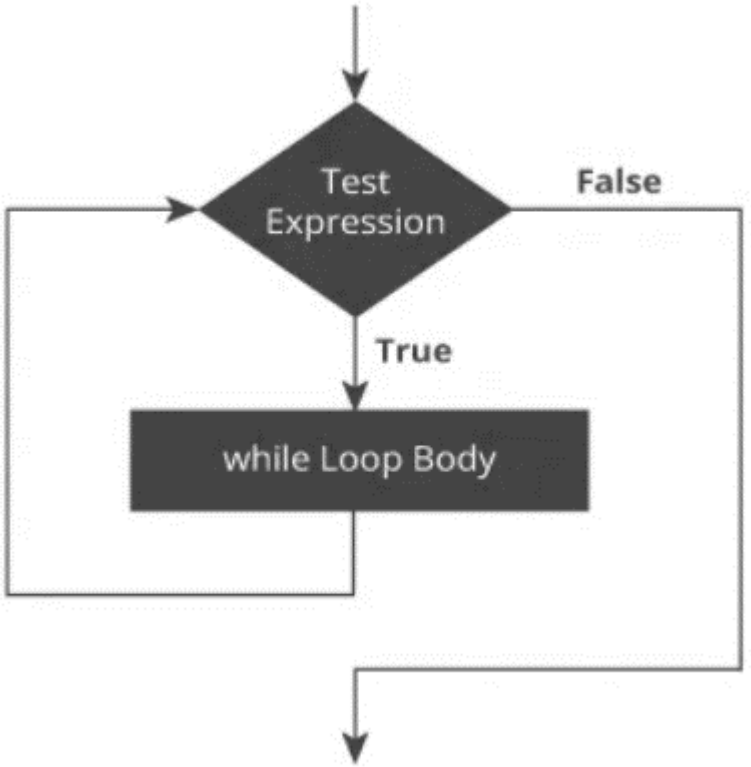
```
switch (expression)
{
   case constant1:
    // statements
    break;
   case constant2:
    // statements
    break;
   .
   .
   default:
    // default statements
}
```

Iteration statements

- Iteration statements specify looping.
  - while (expression) statement
  - do statement while (expression);
  - for (expression; expression; expression) statement.

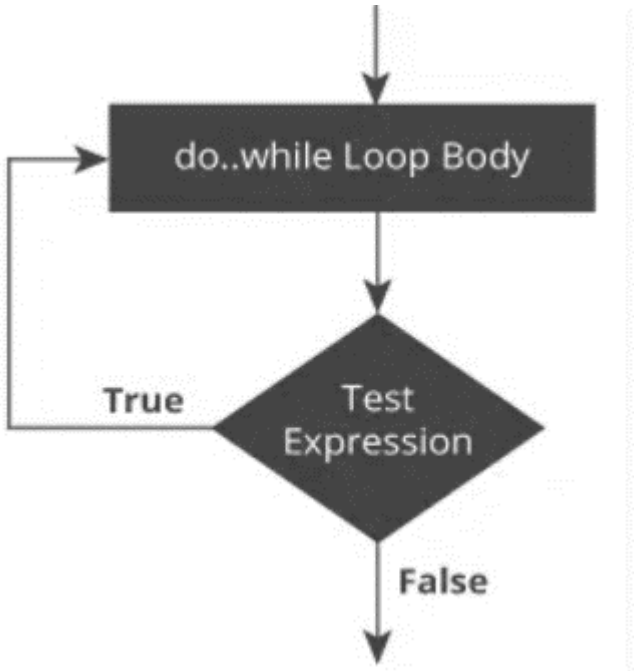
Iteration statements

- Iteration statements specify looping.
  - while (expression) statement
    1. while (testExpression) {
    2. // the body of the loop
    3. }



Iteration statements

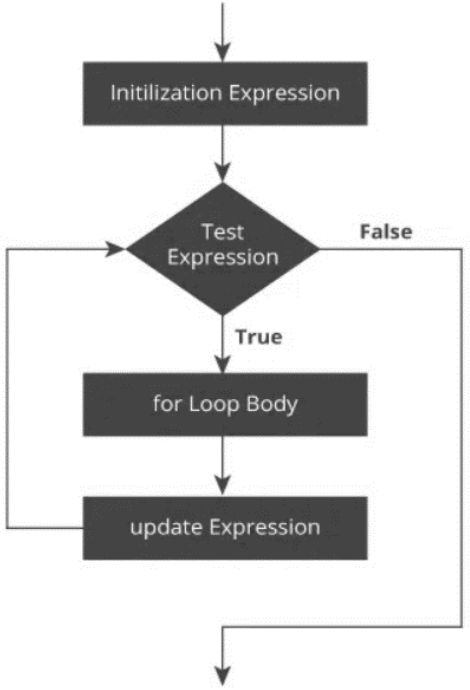
- Iteration statements specify looping.
  - do statement while (expression);
    1. do {
    2. // the body of the loop
    3. }
    4. while (testExpression);



Iteration statements

- Iteration statements specify looping.
  - for (expression; expression; expression) statement.

```
1. for (initializationStatement; testExpression; updateStatement)
2. {
3.     // statements inside the body of loop
4. }
```



Arrays in C

- An array is defined as **the collection of similar type of data items** stored at **contiguous memory locations**.
  - Arrays are the **derived data type** in C programming language which can store the primitive type of data such as int, char, double, float, etc.
  - It also has the capability to store the collection of **derived data types**, such as **pointers**, **structure**, etc.
  - The array is the simplest data structure where each data element can be randomly accessed by using its **index number**.

Arrays in C

- C array is beneficial if you have to store **similar elements**.
  - For example, if we want to store **the marks of a student in 6 subjects**, then **we do not need to define different variables** for the marks in the different subject.
    - Instead of that, we can define an array which **can store the marks in each subject** at the contiguous memory locations.

Arrays in C

- **Properties of Array**
  - Each element of an array is of **same data type and carries the same size**, i.e., int = 4 bytes.
  - Elements of the array can be **randomly accessed** since we can calculate the address of each element of the array with:
    - the given base address
    - the size of the data element.

Arrays in C

- **Advantages of C Array**
  1. **Code Optimization:** Less code to the access the data.
  2. **Ease of traversing:** By using the for loop, we can retrieve the elements of an array easily.
  3. **Ease of sorting:** To sort the elements of the array, we need a few lines of code only.
  4. **Random Access:** We can access any element randomly using the array.

Arrays in C

- **Disadvantages of C Array**
  - Fixed Size:
    - Whatever size is define at the time of declaration of the array, **cannot be exceeded**.
    - So, it **does not grow the size dynamically** like LinkedList.

Arrays in C

- **Declaration of C Array**

  data_type array_name[array_size];

  - Example: int marks[5];
    - Here, int is the data_type, marks are the array_name, and 5 is the array_size.

Arrays in C

- **Initialization of C Array**
  - The simplest way to initialize an array is by **using the index of each element**.
    - Consider the following example.
      - marks[0]=80;//initialization of array
      - marks[1]=60;
      - marks[2]=70;
      - marks[3]=85;
      - marks[4]=75;

Arrays in C

- **Initialization of C Array**
  - C array example:

	80	60	70	85	75
Array indices	0	1	2	3	4

Array length: 5
First index: 0
Last index: 4

Output:
80
60
70
85
75

```
#include <stdio.h>
int main()
{
   int i=0;
   int marks[5];//declaration of array
   marks[0]=80;//initialization of array
   marks[1]=60;
   marks[2]=70;
   marks[3]=85;
   marks[4]=75;
   //traversal of array
   for(i=0;i<5;i++)
      printf("%d \n",marks[i]);
   return 0;
}
```

Arrays in C

- There are various ways in which we can **declare an array**:
  - It can be done by specifying its **type** and **size**, by initializing it or **both**.

Arrays in C

1. **Array declaration by specifying size**
   - int arr1[10];
- With recent C/C++ versions, we can also declare an array of **user specified size**
   - int n = 10;
   - int arr2[n];

Arrays in C

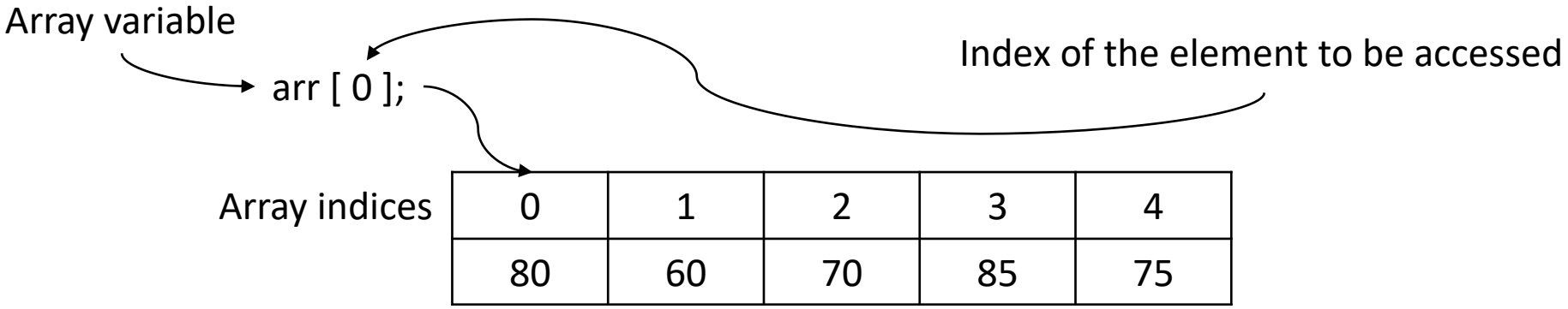
2. **Array declaration by initializing elements**
   - int arr[] = { 10, 20, 30, 40 };
     - Compiler creates an array of size 4.
   - Similar to :
     - int arr[4] = {10, 20, 30, 40};

Arrays in C

3. **Array declaration by specifying size and initializing elements**
   - int arr[6] = { 10, 20, 30, 40 }
     - Compiler creates an array of size 6,
     - Initializes first 4 elements as specified by user
     - And sets two elements as 0.
   - Similar to :
     - int arr[] = {10, 20, 30, 40, 0, 0};

Arrays in C

- **Accessing Array Elements**
  - Array elements are accessed by **using an integer index**.
    - Array index **starts with 0** and goes till **size of array minus 1**.



Arrays in C

- Example:

```
#include <stdio.h>
int main()
{
   int arr[5];
   arr[0] = 5;
   arr[1] = -10;
   arr[2] = 2;
   arr[3] = arr[0];
   printf("%d %d %d %d", arr[0], arr[1], arr[2], arr[3]);
return 0;
}
```

The elements are stored at contiguous memory locations

Output:
5 -10 2 5

Arrays in C

- Example: C program to demonstrate that array elements are stored contiguous locations

```
#include <stdio.h>

int main()

{

    // If arr[0] is stored at address x, then arr[1] is stored at x + sizeof(int)

   // arr[2] is stored at x + sizeof(int) + sizeof(int) and so on.

   int arr[5], i;

   printf("Size of integer in this compiler is %lu\n", sizeof(int));

   for (i = 0; i < 5; i++)

     // The use of '&' before a variable name, yields address of variable.

     printf("Address arr[%d] is %p\n", i, &arr[i]);

    return 0;

}
```

Output:

```
Size of integer in this compiler is 4
Address arr[0] is 0x7ffd636b4260
Address arr[1] is 0x7ffd636b4264
Address arr[2] is 0x7ffd636b4268
Address arr[3] is 0x7ffd636b426c
Address arr[4] is 0x7ffd636b4270
```

Arrays in C

- The *sizeof* operator gives the **amount of storage, in bytes, required to store an object** of the type of the operand.
- This operator allows to avoid specifying machine-dependent data sizes in your programs.

sizeof (type-name)

Arrays in C

- Exercises:
  1. Write a C program for sorting an array of 10 elements
     - Order of initialization: 11,10,5,9,0,1,6,10,8,2
  2. Write a C Program to print the largest and second largest element of the array as well as their position.
     - The program should ask the user to enter the size of the array desired
     - Then it should ask the user to enter the elements of the array
     - Then it will display the largest and the second largest elements
  3. Write a C program to find the average of n numbers using arrays
  4. Write a C program that prints all the numbers of an array then prints the numbers in backward.

Arrays in C

(1)

```
#include<stdio.h>
void main ()
{
   int i, j,temp;
   int a[10] = {11,10,5,9,0,1,6,10,8,2};
   for(i = 0; i<10; i++)
  {
     for(j = i+1; j<10; j++)
    {
       if(a[j] < a[i])
      {
         temp = a[i];
         a[i] = a[j];
         a[j] = temp;
       }
    }
  }
   printf("Printing Sorted Element List ...\n");
   for(i = 0; i<10; i++)
   {
      printf("%d\n",a[i]);
   }
}
```

Output:

0
1
2
5
6
8
9
10
10
11

Arrays in C

(2)

Output:
Enter the size of the array: 5
Enter the elements of the array:
5
4
9
2
1
largest = 9, second largest = 5

```
#include<stdio.h>
void main ()
{
   int arr[100],i,n,largest,sec_largest,pl=0,psl=0;
   printf("Enter the size of the array : ");
   scanf("%d",&n);
   for(i = 0; i<n; i++)
   {
     printf("Element[%d] = ", i);
     scanf("%d",&arr[i]);
    }
    largest = arr[0];
    sec_largest = arr[1];
for(i=0;i<n;i++)
  {
     if(arr[i]>largest)
     {
        sec_largest = largest;
        largest = arr[i];
}
    else if (arr[i]>sec_largest && arr[i]!=largest)
    {
        sec_largest=arr[i];
}
  }
 printf("largest = %d pos = %d, second largest = %d pos = %d",largest,sec_largest);
}
```

Arrays in C

(3)

```
#include <stdio.h>
int main()
{
   int i, n, sum = 0, average;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   int marks[n];
   for(i=0; i<n; ++i)
   {
     printf("Enter number%d: ",i+1);
     scanf("%d", &marks[i]);
    // adding integers entered by the user to the sum variable
    sum += marks[i];
   }
   average = sum/n;
   printf("Average = %d", average);
   return 0;
}
```

Output:
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39

Arrays in C

(4)

Output:
Print all the Numbers :
2
4
6
8
10
12
14
From End to Beginning :
14
12
10
8
6
4
2

```
#include<stdio.h>

int main()

{

    int M[10]={2,4,6,8,10,12,14};

    int j;

    printf("Print all the Numbers : \n");

    for (j = 0; j < 7; ++j)

    {

        printf("M[%d] = %d\n",j,M[j]);

    }

    printf("\nFrom End to Beginning : \n");

    for (j = 6; j >= 0; --j)

   {

      printf("M[%d] = %d\n",j,M[j]);

   }

}
```

Arrays in C

- **C Multidimensional Arrays**
  - In C programming, an array of arrays can be created.
    - These arrays are known as **multidimensional arrays**.

Arrays in C

- **C Multidimensional Arrays**
  - Example: float x[3][4];
    - Here, x is a two-dimensional array.
    - The array can hold 12 elements.
    - The array can be considered as a table with 3 rows and each row has 4 columns.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Arrays in C

- Similarly, a three-dimensional array can be declared.
  - For example: float y[2][4][3];
    - y can hold 24 elements.

Arrays in C

- **Initializing a multidimensional array**
  - Initialization of a 2d array:

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

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

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

Arrays in C

- **Initializing a multidimensional array**
  - Initialization of a 3d array
    - A three-dimensional array can be initialized in a similar way like a two-dimensional array.
    - Example:

```
int test[2][3][4] = {
{{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
{{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}
};
```

Arrays in C

1. Write a C program that store the elements entered by user in a 2d array and displays the elements.

   - Output:

```
Enter value for disp[0][0]: 1
Enter value for disp[0][1]: 2
Enter value for disp[0][2]: 3
Enter value for disp[1][0]: 4
Enter value for disp[1][1]: 5
Enter value for disp[1][2]: 6
Two Dimensional array elements:
1 2 3
4 5 6
```

Arrays in C

```
#include<stdio.h>
int main(){
   /* 2D array declaration*/
   int disp[2][3];
   /*Counter variables for the loop*/
   int i, j;
   for(i=0; i<2; i++)
  {
      for(j=0;j<3;j++)
     {
         printf("Enter value for disp[%d][%d]:", i, j);
         scanf("%d", &disp[i][j]);
      }
   }
   //Displaying array elements
   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++)
   {
      for(j=0;j<3;j++)
      {
         printf("%d ", disp[i][j]);}
       }
         printf("\n");
    }
    return 0;
}
```

C Character Array and Strings

- The string in C programming language is actually **a one-dimensional array of characters** which is **terminated by a null character '\0'**.
  - The following declaration and initialization create a string consisting of the word "Hello".

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

C Character Array and Strings

- To hold the null character at the end of the array, the **size** of the character array containing the string **is one more than the number of characters** in the word "Hello".
- '\0' represents the end of the string.
  - It is also referred as **String terminator & Null Character**

C Character Array and Strings

- **String Declaration:**
  - Method 1:

    char address[]={'T', 'E', 'X', 'A', 'S', '\0'};
  - Method 2: The above string can also be defined as:

    char address[]="TEXAS";
    - In the above declaration NULL character (\0) will **automatically be inserted at the end of the string**.

C Character Array and Strings

- **Arrays of strings:**
  - An array of strings is a two-dimensional array of characters in which each row is one string.
    - char names[10][25];
    - char month[5][10] = {“January”, “February”, “March”, “April”, “May”};

C Character Array and Strings

- **String I/O in C programming**
  - printf and scanf
  - puts and gets
    - **Syntax:**
      - printf("%s",str1);
      - puts(str1);
      - scanf("%s",&str1);
      - gets(str1);

C Character Array and Strings

- Read & write Strings in C using Printf() and Scanf() functions

Output:
Enter your Nick name: Sarita
Sarita

```
#include <stdio.h>
#include <string.h>
int main()
{
    /* String Declaration*/
   char nickname[20];
   printf("Enter your Nick name: ");
   /* reading the input string and storing it in nickname
   Array name alone works as a base address of array so we can use nickname instead of &nickname here
   */
    scanf("%s", nickname);
   /*Displaying String*/
   printf("%s",nickname);
   return 0;
}
```

C Character Array and Strings

- Read & write Strings in C using puts() and gets() functions

```
#include <stdio.h>
#include <string.h>
int main()
{
    /* String Declaration*/
    char nickname[20];
    /* Console display using puts */
    puts("Enter your Nick name:");
    /*Input using gets*/
    gets(nickname);
    puts(nickname);
    return 0;
}
```

Output:
Enter your Nick name: Sarita
Sarita

C Character Array and Strings

- **C string functions:**
  - The string cannot be copied by the assignment operator '=':
  - str = "Hello World" is not valid
  - C provides string manipulating functions in the "string.h" library.

C Character Array and Strings

- **C string functions**

Function	Purpose	Example
strcpy	**Copies** a string into another	strcpy(s1, “Hi”);
strcat	**Appends** one string at the end of another	strcat(s1, “more”);
strcmp	**Compares** two strings	strcmp(s1, “Hi”);
strlen	Finds out the **length** of a string	strlen(“Hi”); (returns 2)
strtok	**Breaks** a string into tokens by delimiters	strtok(“Hi, Chaos”, “,”);
strncpy	**Copies first n characters** of one string into another	strncpy(s1, “Test”, 2);
strncmp	**Compares first n characters** of two strings	strncmp(“mo”, “more”, 2);
stricmp	**Compares two strings** without regard to case (**ignores case**)	stricmp(“hi”, “Hi”);
strlwr	Converts a string to **lowercase**	strlwr(“Hi”); (returns hi)
strupr	Converts a string to uppercase	strupr(“Hi”);
strncat	**Appends first n characters** of a string at the end of another	strncat(s1, “more”, 2);
strrev	**Reverses** a string	strrev(s1, “more”);

C Character Array and Strings

- **Example of strlen:**
  - It returns the length of the string **without including end character (terminating char '\0')**.

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[20] = "BeginnersBook";
   printf("Length of string str1: %d", strlen(str1));
   return 0;
}
```

Output:
Length of string str1: 13

C Character Array and Strings

- **Exercises:**
  1. Write a program in C to find the length of a string without using library function.
  2. Write a program in C to print individual characters of string in reverse order.
  3. Write a program in C to count the total number of words in a string

C Character Array and Strings

(1)

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
        char str[100]; /* Declares a string of size 100 */
        int l= 0;
        printf("\n\nFind the length of a string :\n");
        printf("--------------------------------\n");
        printf("Input the string : ");
        gets(str);
        while(str[l]!='\0')
        {
                l++;
        }
        printf("Length of the string is : %d\n\n", l);
}
```

Output:

```
Find the length of a string :
--------------------------------
Input the string : w3resource.com
Length of the string is : 15
```

C Character Array and Strings

(2)

```
#include <stdio.h>
#include <string.h>
void main()
{
        char str[100]; /* Declares a string of size 100 */
        int l,i;
        printf("\n\nPrint individual characters of string in reverse order :\n");
        printf("-------------------------------------------------------\n");
        printf("Input the string : ");
        gets(str);
        l=strlen(str);
        printf("The characters of the string in reverse are : \n");
        for(i=l;i>=0;i--)
        {
                printf("%c ", str[i]);
        }
        printf("\n");
}
```

Output:

```
Print individual characters of string in reverse order :
-------------------------------------------------------
Input the string : w3resource.com
The characters of the string in reverse are : m o c . e c r u o s e r 3 w
```

C Character Array and Strings

(3)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define str_size 100 //Declare the maximum size of the string
void main()
{
        char str[str_size];
        int i, wrd;
        printf("\n\nCount the total number of words in a string :\n");
        printf("-----------------------------------------------------\n");
        printf("Input the string : ");
        gets(str);
        i = 0;
        wrd = 1;
```

Output:

```
Count the total number of words in a string :
-----------------------------------------------------
Input the string : This is w3resource.com
Total number of words in the string is : 3
```

C Character Array and Strings

(3)

```
/* loop till end of string */
while(str[i]!='\0')
{
        /* check whether the current character is white space or new line or tab character*/
        if(str[i]==' ' || str[i]=='\n' || str[i]=='\t')
        {
                wrd++;
        }
        i++;
}
printf("Total number of words in the string is : %d\n", wrd);

}
```

Output:

```
Count the total number of words in a string :
---------------------------------------------------
Input the string : This is w3resource.com
Total number of words in the string is : 3
```

Functions in C

- The function is the fundamental building block of the **modular programming language**
  - Modular program is composed of several modules/source files

Functions in C

- **Advantages:**
  - It provides **modularity** to the program.
  - **Easy code reusability**.
    - You just have to call the function by its name to use it.
  - In case of large programs with thousands of code lines,
    - **Debugging** and **editing** become easier.
  - A function is **independent**:
    - It is “completely” self-contained

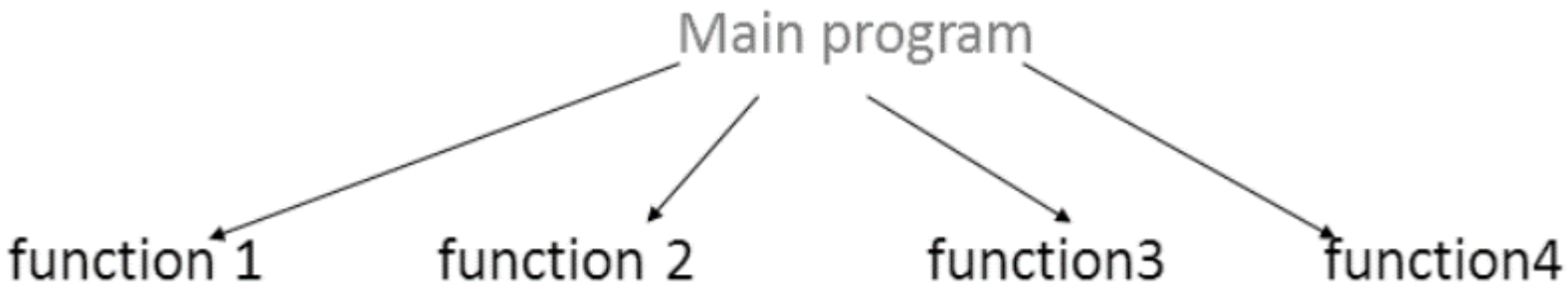
Functions in C

- **Advantages:**
  - It **can be called at any place** of your code and **can be ported** to another program,
  - **Procedural abstraction** - hide internal details,
  - **Factoring of code** - divide and conquer.

Functions in C

- **A function:**
  - receives zero or more parameters,
  - performs a specific task,
  - returns zero or one value,
  - is invoked / called by name and parameters.
- In C, no two functions can have the same name

Functions in C



- **There are two types of functions in C:**
  - Library functions,
  - User-defined functions,

Functions in C

- **Library function**
  - Library functions are the **in-built function** in the C programming system
  - Example:
    - main() - the execution of every C program
    - printf() - for displaying output in C.
    - scanf() - for taking input in C.

Functions in C

- **Library function - main**
  - The main function is defined as:
    - int main(int argc, char*argv[]) ;

Functions in C

- **Library function - main**
  - The "***main***" function is any function returns an integer and accepts two arguments.
    - The returned integer is used to give an indication of the **execution status of the program**.
      - If the function returns "0", the execution of the program took place **without error**.
      - If not, the program was probably **crashed** for some reason to be determined.

Functions in C

- **Library function - main**
  - The "***main***" function is any function returns an integer and accepts two arguments.
    - The arguments of the "main" function represent the following:
      - The "argc" variable contains the **number of arguments on the command line**.
      - The "argv" variable is **an array of pointers to command line arguments**.

Functions in C

- **Library function**
  - Some of the math.h Library Functions
    - ***sin()*** - returns the sine of a radian angle.
    - ***cos()*** - returns the cosine of an angle in radians.
    - ***tan()*** - returns the tangent of a radian angle.
    - ***floor()*** - returns the largest integral value less than or equal to x.
    - ***ceil()*** - returns the smallest integer value greater than or equal to x.
    - ***pow()*** - returns base raised to the power of exponent.

Functions in C

- **Library function**
  - Some of the conio.h Library Functions
    - ***clrscr()*** - used to clear the output screen.
    - ***getch()*** - reads character from keyboard.
    - ***textbackground()*** - used to change text background.

Functions in C

- **User defined function**
  - Allows programmer to **define their own function** according to their requirement.

Functions in C

- **User defined function**
  - ***Advantages of user defined functions***
    - It helps to decompose the large program into **small segments** which makes programmer easy to **understand, maintain and debug**.
    - If **repeated code** occurs in a program.
      - Function can be used to **include those codes and execute when needed** by calling that function.
    - Programmer working on large project **can divide the workload** by making different functions.

Functions in C

- **Function naming rule in C**
  - Name of function includes only **alphabets, digit and underscore**.
  - First character of name of any function must be an **alphabet or underscore**.
  - Name of function **cannot be any keyword** of c program.
  - Name of function **cannot be global identifier**.
  - Name of function **cannot be exactly same as of name of function** in the same scope.
  - Name of function is **case sensitive**
  - …

Functions in C

- **The General Form a Function**

```
return_type function_name (parameter list)
{
    body of the function
}
```

Functions in C

- **The General Form a Function**
  - The return-type specifies the **type of data** that the function returns.
    - A function **may return any type** (default: int) of data **except an array.**
  - The parameters (formal arguments) list is a **comma-separated list of variable names and their associated types.**
    - The parameters **receive the values of the arguments** when the function is called.

Functions in C

- **The General Form a Function**
  - A function can be without parameters:
    - An **empty parameters list can be explicitly specified** as such by placing the keyword **void inside the parentheses.**

Functions in C

- **Formal argument/parameter list**

  (type varname1, type varname2, . . . , type varnameN)

  - All function parameters must be declared individually, each **including both the type and name.**
    - f(int i, int k, int j)

Functions in C

- **Formal argument/parameter list**
  - f(int i, k, float j) ???
    - /* wrong, k must have its own type specifier */

Functions in C

- **Scope of a function**
  - Each function is a **discrete block of code**.
    - A function's code is **private to that function,**
      - It **cannot be accessed by any statement** in any other function **except through a call to that function**.

Functions in C

- **Scope of a function**
  - Each function is a **discrete block of code**.
    - Variables that are defined **within a function are local variables**
      - A local variable **comes into existence when the function is entered and is destroyed upon exit**

Functions in C

- **Scope of a function**
  - The **formal arguments/parameters** to a function also fall within the function's scope:
    - known throughout the entire function comes **into existence when the function is called and is destroyed when the function is exited**.
  - Even though they **perform the special task of receiving the value of the arguments passed to the function**, they **behave like any other local variable**

Functions in C

- **Returning value**
  - If nothing is returned
    - return;
    - or, until reaches right curly brace (})
  - If something is returned
    - return expression;
  - Only one value can be returned from a C function

Functions in C

- **Returning value**
  - A function **can return only one value**, though it **can return one of several values based on the evaluation of certain conditions**.
    - Multiple return statements can be used within a single function (eg: inside an “if-else” statement…)
  - The return statement not only returns a value back to the calling function, **it also returns control back to the calling function**.

Functions in C

- **Three Main Parts of a Function**
  - Function Declaration (Function prototype),
  - Function Definition,
  - Function Call.

Functions in C

- **Structure of a C program with a Function**
  - **Function prototype** giving the **name**, **return type** and the **type of formal arguments**
  - main( )
  - { …….
    - **Call to the function:**
      - Variable to hold the value returned by the function = Function name with actual arguments.
  - …… }
  - **Function definition:**
    - **Header of function** with **name**, **return type** and the **type of formal arguments** as given in the prototype
    - **Function body** within { } with **local variables declared , statements and return statement**

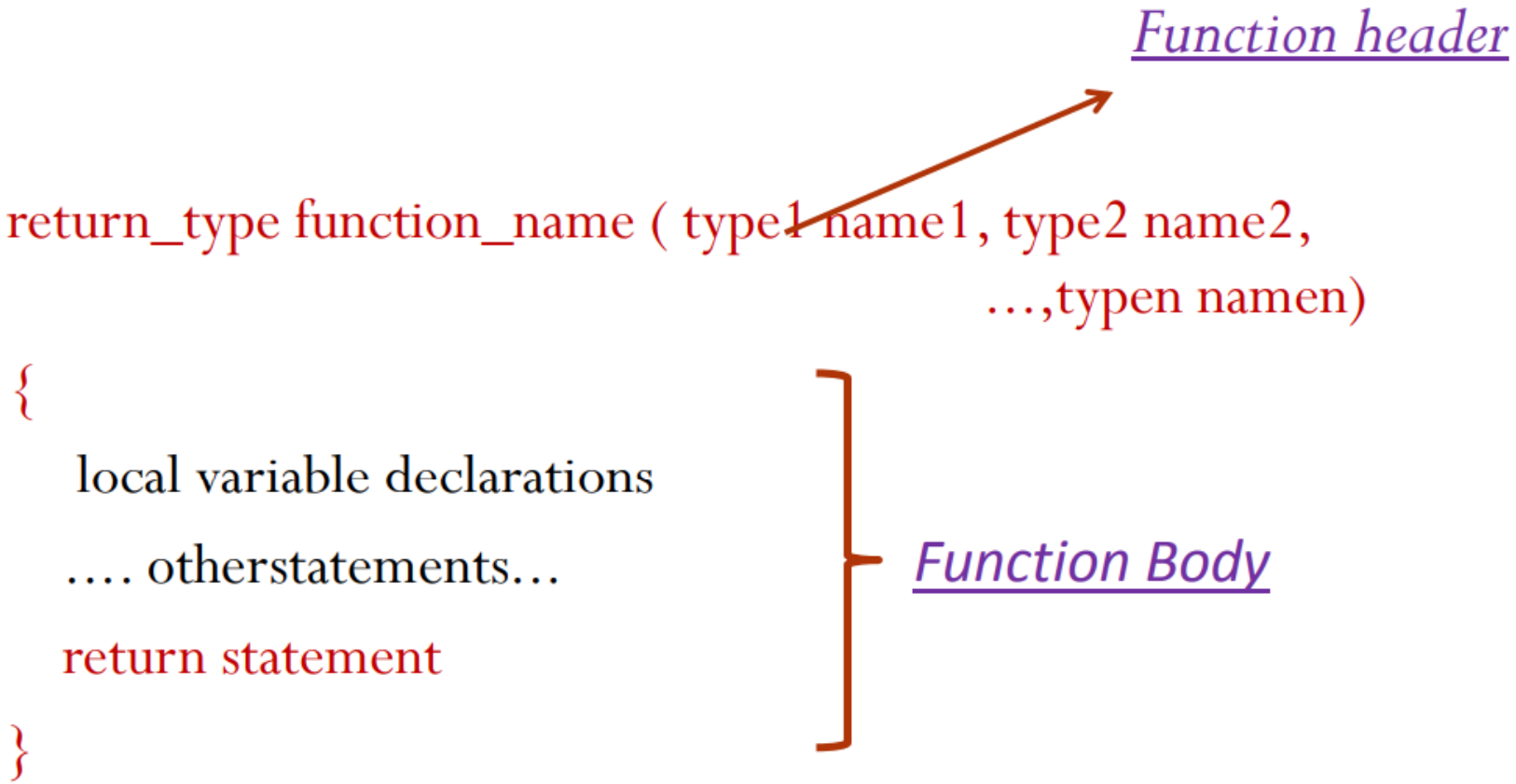
Functions in C

- **Function Prototype**
  - Functions **should be declared before they are used**
  - Prototypes are only needed if **function definition comes after use in program**
  - Function prototypes are **always declared at the beginning of the program** indicating :
    - Name of the function,
    - Data type of its arguments,
    - Data type of the returned value

return_type function_name (type1 name1, type2 name2, ..., typen namen);

Functions in C

- **Function Definition**



Functions in C

- **Function Call**
  - A function is **called from the main( )**
  - A function **can in turn call another function**
    - Function call statements invokes the function which means **the program control passes to that function**
      - Once the function completes its task,
        - **The program control is passed back to the calling environment**

Functions in C

- **Function Call**
  - Variable = function_name (argument list);
  - Or Function_name (argument list);
    - **Function name** and **the type** and **number of arguments must match with that of the function declaration statement** and the **header of the function definition.**

Functions in C

- **Return statement**
  - To return a value from a C function you must explicitly return it with a **return statement**

return <expression>;

  - The expression can be any valid C expression that **resolves to the type defined in the function header**

Functions in C

- **Return statement**

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

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

  - Ex: Function call: int value = add(5,8)
    - Here, add( ) sends back the value of the expression (a + b) or value of c to main( )

Functions in C

- **Function Prototype Examples**
  - double squared (double number);
  - void print_report (int);
  - int get_menu_choice (void);

Functions in C

- **Function Definition Examples**

```
double squared (double number)
{
    return (number * number);
}
```

Functions in C

- **Function Definition Examples**

```
void print_report (int report_number)
{
    if (report_nmber == 1)
        printf(“Printer Report 1”);
    else
        printf(“Not printing Report 1”);
}
```

Functions in C

- **Calling Functions – Two methods**
  - Call by value
  - Call by reference

Functions in C

- **Calling Functions – Two methods**
  - ***Call by value***
    - **Copy** of argument passed
    - Changes in function do **not effect original**
    - Use when function **does not need to modify argument**
      - Avoids accidental changes

Functions in C

- **Calling Functions – Two methods**
  - ***Call by reference***
    - Passes **original argument**
    - Changes in function **effect original**
    - *Only used with trusted functions*

Functions in C

- **Recursion**
  - A recursive function is a function that **calls itself** either directly or indirectly through another function.
  - **Nature of recursion**
    - One or more simple cases of the problem have a straightforward, non-recursive solution.
    - The other cases can be redefined in terms of problems that are closer to the simple cases.

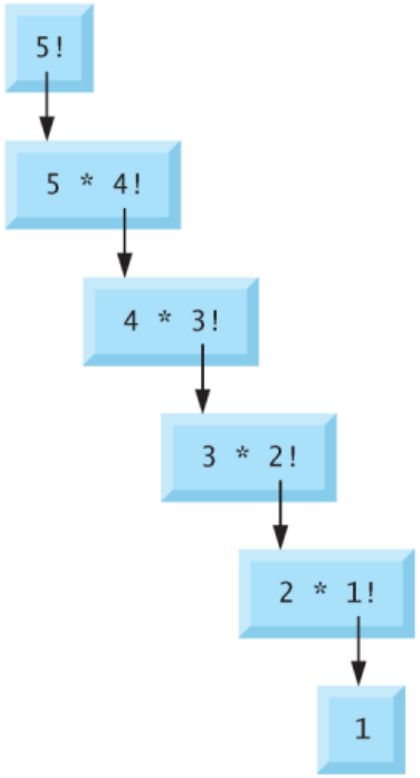
Functions in C

- **Recursively calculating factorial**
  - The factorial of a nonnegative integer n, written n! is the product:
    - n x (n –1) x (n – 2) x … x 1
  - A recursive definition of the factorial function is arrived at by observing the following relationship:
    - n! = n x (n – 1)!

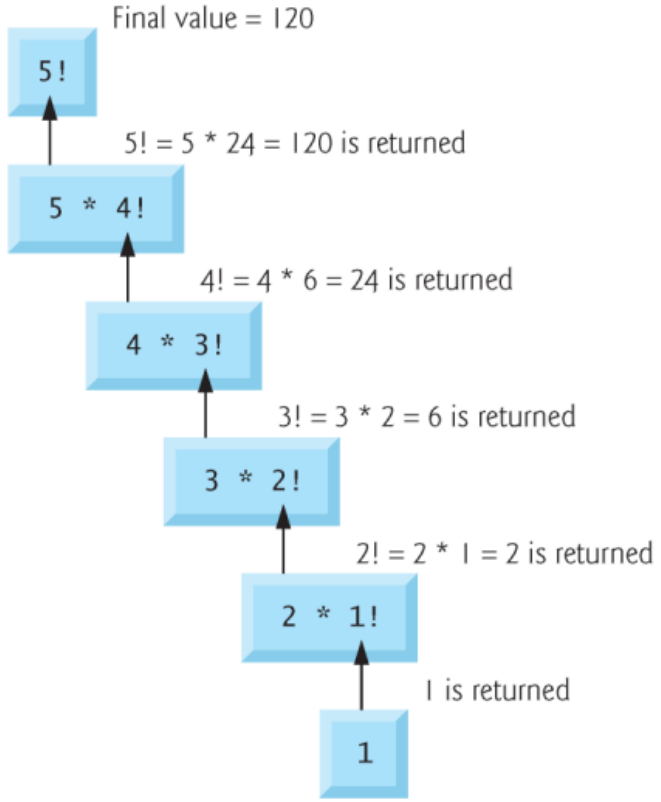
Functions in C

- **Recursively calculating factorial**

Sequence of recursive calls



Values returned from each recursive call



Functions in C

```
#include <stdio.h>
#include <stdlib.h>
int factorial (int n)
{
        if (n==1)
        {
                return 1;
        }
        else
                return n*factorial(n-1);
}
```

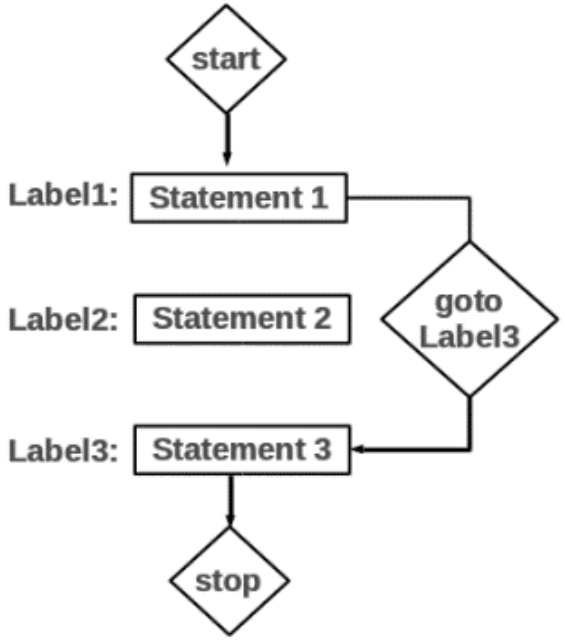
```
int main(int argc, char *argv[]) {
        int a;
        printf("Enter the desired number");
        scanf("%d",&a);
        printf("The factorial of %d is %d", a, factorial(a));
        return 0;
}
```

Jump statements

- Jump statements transfer control unconditionally.
  - goto *identifier*;
  - continue;
  - break;
  - return *expression*;

Jump statements

- **goto *identifier*;**
  - The goto statement is a jump statement which is sometimes also referred to as **unconditional jump statement**.
  - The goto statement can be used to **jump from anywhere to anywhere** within a function.



Jump statements

- **goto *identifier*;**
  - Syntax:

goto label;
..
..
..
label:
  label instructions;
	label:
	  label instructions;
	goto label;
	..
	..
	..

Jump statements

- **goto *identifier*;**
  - Example:

```
#include <stdio.h>

void printNumbers()
{
    int n = 1;
label:
    printf("%d ",n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main() {
    printNumbers();
    return 0;
}
```

```
1 2 3 4 5 6 7 8 9 10
```

Jump statements

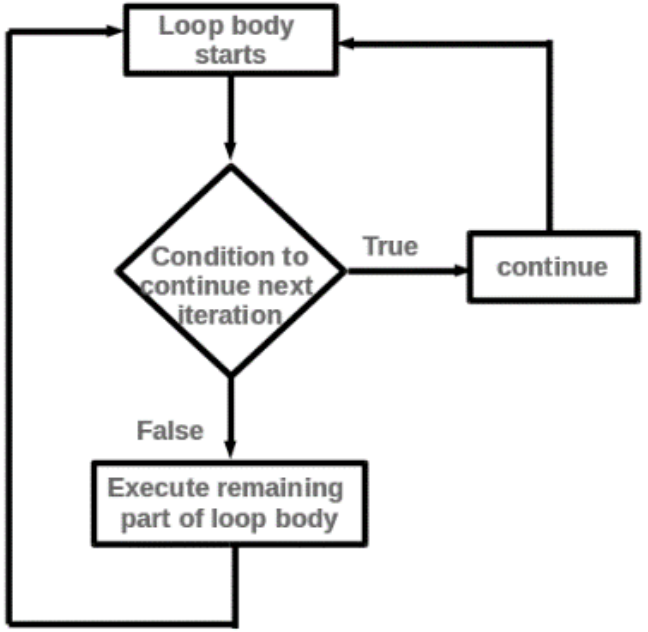
- **goto *identifier*;**
  - Example:

```
26 is even
```

```
#include <stdio.h>

void checkEvenOrNot(int num)
{
    if (num % 2 == 0)
        // jump to even
        goto even;
    else
        // jump to odd
        goto odd;

even:
    printf("%d is even", num);
    // return if even
    return;
odd:
    printf("%d is odd", num);
}

int main() {
    int num = 26;
    checkEvenOrNot(num);
    return 0;
}
```

Jump statements

- **Continue;**
  - A continue statement may appear **only within an iteration statement**.
  - It causes control to pass to the **loop–continuation portion** enclosing such statement.



Jump statements

- Example:

```
// C program to explain the use
// of continue statement
#include <stdio.h>

int main() {
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            printf("%d ", i);
    }

    return 0;
}
```

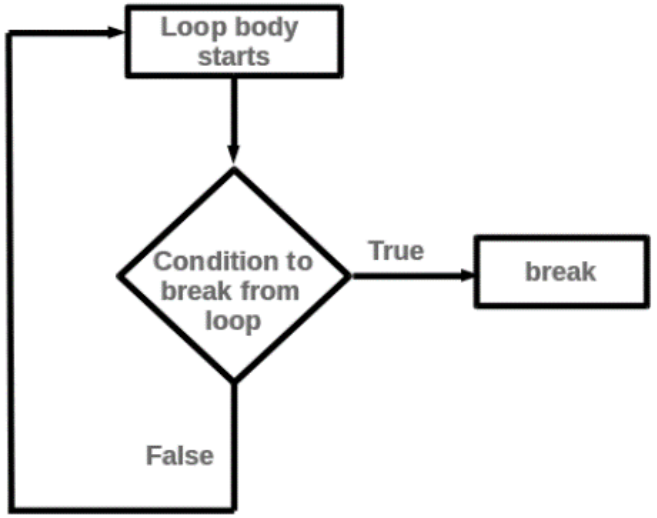
1 2 3 4 5 7 8 9 10

Jump statements

- **break;**
  - A loop control statement that is used to **end a loop** in C or C++ is called a break.
    - The loop iterations cease when the **break statement is reached** from inside a loop,
    - Control instantly moves from the loop **to the first statement after** the loop.

Jump statements

- **break;**
  - Break statements are generally used when **we are unsure of the precise number of loop iterations** or when we wish to **end the loop depending on a condition**.



Jump statements

- break;
  - Example:

```
***
***
***
***
***
```

```
#include <stdio.h>

int main() {
   // nested for loops with break statement
   // at inner loop
   for (int i = 0; i < 5; i++) {
      for (int j = 1; j <= 10; j++) {
         if (j > 3)
            break;
         else
            printf("*");
      }
      printf("\n");
   }

   return 0;
}
```

Jump statements

- **return *expression*;**
  - A function **returns** to its caller by the return statement.
    - When return is followed by an expression, the **value is returned to the caller of the function**.
      - The expression is converted, as by assignment, to the **type returned by the function** in which it appears.

Functions in C

1. Write a C program with a function that calculates the average of 3 numbers.
2. Write a C program with a function that swaps the values of two variables.
3. Write a program in C to find the sum of the series 1!/1+2!/2+3!/3+4!/4+5!/5 using the function.

Functions in C

(1)

```
#include<stdio.h>
float average(float, float, float);
int main( )
{
float a, b, c;
printf("Enter three numbers please\n");
scanf("%f %f %f ",&a, &b, &c);
printf("Avg of 3 numbers = %.3f\n", average(a, b, c) );
return 0;
}
float average(float x, float y, float z) //local variables x, y, z
{
float r; // local variable
r = (x+y+z)/3;
return r;
}
```

Function prototype

Function call

Function header

Function Body

Output:
Enter three numbers please
3, 5, 4
Avg of 3 numbers = 4.000

Functions in C

(2)

```
#include<stdio.h>
void swap(int, int );
void main( )
{
        int a=10, b=20;
        swap(a, b);
        printf(" %d %d \n", a, b);
}
void swap (int x, int y)
{
        int temp = x;
        x= y;
        y=temp;
}
```

Output:
10

Functions in C

(3)

```
#include <stdio.h>
int fact(int);
void main()
{
   int sum;
   sum = fact(1)/1+fact(2)/2+fact(3)/3+fact(4)/4+fact(5)/5;
   printf("\n\n Function : find the sum of 1!/1+2!/2+3!/3+4!/4+5!/5 :\n");
   printf("-------------------------------------------------------\n");
   printf("The sum of the series is : %d\n\n",sum);
}
```

```
int fact(int n)
   {
      int num=0,f=1;
      while(num<=n-1)
      {
         f =f+f*num;
         num++;
      }
   return f;
   }
```

Function : find the sum of 1!/1+2!/2+3!/3+4!/4+5!/5 :

The sum of the series is : 34

Pointers

- A variable in a program is something with a name, the value of which can vary.
- The way the compiler and linker handles this is that it assigns a **specific block of memory** within the computer to **hold the value of that variable**.
- The size of that block depends on the range over which the variable is allowed to vary.
  - For example, on computer's the size of an integer variable is 2 bytes, and that of a long integer is 4 bytes.

Pointers

- When we declare a variable we inform the compiler of two things:
  - The name of the variable
  - The type of the variable.
- For example, we declare a variable of type integer with the name k by writing:
  - int k;

Pointers

- On seeing the “int” part of this statement the compiler sets aside 2 bytes of memory to hold the value of the integer.
- It also sets up a symbol table.
  - In that table it adds the symbol k and the relative address in memory where those 2 bytes were set aside.
  - Thus, later if we write: k = 2; we expect that, at run time when this statement is executed, the value 2 will be **placed in that memory location reserved for the storage of the value of k**.
  - In C we refer to a variable such as the integer k as an “**object**”.

Pointers

- In a sense there are two “values” associated with the object k.
  - One is the value of the integer stored there (2 in the example),
  - The other the “value” of the memory location, i.e., the address of k.

Pointers

- In a sense there are two “values” associated with the object k.
  - Also referred to the two values with the:
    - **rvalue** (right value),
    - **lvalue** (left value) respectively.

Pointers

- In some languages
  - The **lvalue** is the value permitted on the left side of the assignment operator '='
    - The address where the result of evaluation of the right side ends up.
  - The **rvalue** is that which is on the right side of the assignment statement, the 2 above.
    - **rvalues** cannot be used on the left side of the assignment statement.
      - Thus: 2 = k; **is illegal**.

Pointers

- Consider:

```
int j, k;
k = 2;
j = 7; <-- line 1
k = j; <-- line 2
```

Pointers

- **Here, the compiler interprets:**
  - The j in line 1 as the address of the variable j (its lvalue) and creates code to copy the value 7 to that address.
  - In line 2, however, the j is interpreted as its rvalue (since it is on the right hand side of the assignment operator ‘=‘).
    - That is, here the j refers to the value stored at the memory location set aside for j, in this case 7.
    - So, the 7 is copied to the address designated by the lvalue of k.

Pointers

- **In all of these examples**
  - All copying of rvalues from one storage location to the other is done by copying 2 bytes.
  - Had we been using long integers, we would be copying 4 bytes.

Pointers

- Let's say that we have a reason for wanting a **variable designed to hold an lvalue** (an address).
- The size required to hold such a value depends on the system.
  - On older desktop computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes.
  - Computers with more memory would require more bytes to hold an address.
  - The actual size required is not too important so long as we have **a way of informing the compiler that what we want to store is an address**.

Pointers

- Such a variable is called a pointer variable.
- In C, when we define a pointer variable by preceding its name with an asterisk.
  - In C, we also give our pointer a type which, in this case,
    - Refers to **the type of data stored at the address** we will be storing in our pointer.
  - For example, consider the variable declaration:
    - int *ptr;

Pointers

- ptr is the name of our variable (just as k was the name of our integer variable).
- The ‘*’ informs the compiler that we want a pointer variable,
  - To set aside however many bytes is required to store an address in memory.
- The int says that we intend to use our pointer variable to store the address of an integer.

Pointers

- Such a pointer is said to “point to” an integer.
- However, note that when we wrote int k; we did not give k a value.
  - ptr **has no value**, that is we have not stored an address in it in the above declaration.
  - In this case, again if the declaration is outside of any function, it is initialized to a value guaranteed in such a way **not point to any C object or function**.
  - A pointer initialized in this manner is called a **“null” pointer**.

Pointers

- To make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer.
  - That macro goes under the name NULL.

Pointers

- Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, **guarantees that the pointer has become a null pointer**.
- Similarly, just as one can test for an integer value of zero, as in if(k == 0), we can test for a null pointer using:
  - if (ptr == NULL).

Pointers

- Using the variable ptr:
  - Suppose that we want to store in ptr the address of the integer variable k.
  - To do this we use the unary **&** operator and write: ptr = &k;
    - What the & operator does is retrieve the lvalue (address) of k, even though k is on the right hand side of the assignment operator '=',
    - Then copies that to the contents of the pointer ptr.
    - Now, ptr is said to "point to" k.

Pointers

- The “dereferencing operator” is the asterisk and it is used as follows: *ptr = 7;
  - It will copy 7 to the address pointed to by ptr.
  - Thus if ptr “points to” (contains the address of) k, the above statement will set the value of k to 7.
  - That is, when we use the ‘*‘ this way we are referring **to the value of that which ptr is pointing to**, not the value of the pointer itself.

Pointers

- Similarly, we could write: printf(“%d\n”,*ptr);
  - To print to the screen the integer value stored at the address pointed to by ptr;.
  - One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

Pointers

```
#include <stdio.h>

int main(void)

{

int j, k;

int *ptr;

j = 1;

k = 2;

ptr = &k;

printf("\n");

printf("j has the value %d and is stored at %p\n", j, (void *)&j);

printf("k has the value %d and is stored at %p\n", k, (void *)&k);

printf("ptr has the value %p and is stored at %p\n", ptr, (void *)&ptr);

printf("The value of the integer pointed to by ptr is %d\n", *ptr);

return 0;

}
```

Pointers

- **To retain:**
  - A variable is declared by giving it a type and a name
    - int k;
  - A pointer variable is declared by giving it a type and a name
    - int *ptr
      - Where the asterisk tells the compiler that the variable named ptr is a **pointer variable** and the type tells the compiler **what type the pointer** is to point to (integer in this case).

Pointers

- **To retain:**
  - Once a variable is declared, we can **get its address** by preceding its name with the unary & operator, as in **&k**.
  - We can “dereference” a pointer, i.e. refer to **the value of that which it points to**, by using the unary ‘*‘ operator as in *ptr.
  - An “**lvalue**” of a variable is the value of its address, i.e. where it is stored in memory.
  - The “**rvalue**” of a variable is the value stored in that variable (at that address).

Pointers

- **Exercises**
  - Write a program in C to add two numbers using pointers.
  - Write a program in C to find the maximum number between two numbers using a pointer.

Pointers and arrays

- Let us consider why we need to identify the type of variable that a pointer points to, as in:
  - int *ptr;
- One reason for doing this is so that later ptr “points to” something :
  - *ptr = 2;

Pointers and arrays

- The compiler will **know how many bytes to copy** into that memory location pointed to by ptr.
- If ptr was declared as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied.
  - Similarly for floats and doubles the appropriate number will be copied.

Pointers and arrays

- For example, consider a block in memory consisting of ten integers in a row.
  - That is, 20 bytes of memory are set aside to hold 10 integers.

Pointers and arrays

- Now, let's say we point our integer pointer ptr at the first of these integers.
- Furthermore lets say that integer is located at memory location 100 (decimal).
  - What happens when we write:
    - ptr + 1;

Pointers and arrays

- Because the compiler “knows” this is a pointer (i.e. its value is an address) and that it points to an integer,
  - It adds 2 to ptr instead of 1, so the pointer “points to” the next integer, at memory location 102.
    - Similarly, if the ptr was declared as a pointer to a long, it would add 4 to it instead of 1.
    - The same goes for other data types such as floats, doubles, or even user defined data types such as structures.

Pointers and arrays

- This is obviously not the same kind of “addition” that we normally think of.
- In C, it is referred to as addition using “**pointer arithmetic**”, a term which we will come back to later.

Pointers and arrays

- Similarly, since ++ptr and ptr++ are both equivalent to ptr + 1
  - Incrementing a pointer using the unary ++ operator, either pre- or post-, **increments the address it stores by the amount sizeof(type)**
    - Where “type” is the type of the object pointed to. (i.e. 2 for an integer, 4 for a long, etc.).

Pointers and arrays

- Since a block of 10 integers located contiguously in memory is, by definition, an array of integers,
  - This brings up an interesting relationship between arrays and pointers.

Pointers and arrays

- **Consider the following:**
  - int my_array[] = {1,23,17,4,-5,100};
    - Here we have an array containing 6 integers.
  - We refer to each of these integers by means of a subscript to my_array.
    - Using my_array[0] through my_array[5].
  - But, we could alternatively access them via a pointer as follows:
    - int *ptr;
    - ptr = &my_array[0];
      - Point our pointer at the first integer in our array

Pointers and arrays

- And then we could print out our array either using the array notation or by **dereferencing our pointer**. The following code illustrates this:

```
#include <stdio.h>
int my_array[] = {1,23,17,4,-5,100};
int *ptr;
int main(void)
{
        int i;
        ptr = &my_array[0]; /* point our pointer to the first element of the array */
        printf("\n\n");
        for (i = 0; i < 6; i++)
        {
                printf("method 1: my_array[%d] = %d\n",i,my_array[i]); /*<-- A */
                printf("method 2: my_array[%d] = %d\n",i, *(ptr + i)); /*<-- B */
        }
        return 0;
}
```

Pointers and arrays

- **Compile and run the previous program**
  - Note lines A and B and that the program prints out the same values in either case.
  - Also observe how we dereferenced our pointer in line B.
    - We first added i to it and then dereferenced the new pointer.
  - Change line B to read:
    - printf(“my_array[%d] = %d\n”,i, *ptr++);
  - Run it again... then change it to:
    - printf(“my_array[%d] = %d\n”,i, *(++ptr));

Pointers and arrays

- In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name,
  - Thus in our code where we wrote: ptr = &my_array[0];
  - We can write: ptr = my_array; to achieve the same result.

Pointers and arrays

- **The name of an array is a pointer.**
  - The name of the array is the **address of first element** in the array.
  - For example, while we can write
    - ptr = my_array;
  - we cannot write
    - my_array = ptr;

Pointers and arrays

- The reason is that while **ptr is a variable**, **my_array is a constant**.
  - That is, the location at which the first element of my_array will be stored and cannot be changed once my_array[] has been declared.

Pointers and arrays

- An object is a **named region of storage**;
  - An **lvalue** is an expression referring to an object.
  - Since my_array is a named region of storage, why is my_array in the previous assignment statement not an **lvalue**?
  - To resolve this problem, some refer to my_array as an “**unmodifiable lvalue**”.

Pointers and arrays

- Modify the example program by changing
  - ptr = &my_array[0]; to ptr = my_array;
- Run it again to verify the results are identical

Pointers and arrays

- In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name,

```
#include <stdio.h>
int my_array[] = {1,23,17,4,-5,100};
int *ptr1, *ptr2;
int main(void)
{
int i;
ptr1=my_array;
ptr2=&my_array[0];
printf("%d \n", ptr1);
printf("%d", ptr2);
return 0;
}
```

Pointers and arrays

- An array's name is also referred to as a **constant pointer**.
  - What does that mean?
    - When we declare a variable we set aside a spot in memory to hold the value of the appropriate type.
    - Once that is done, the name of the variable can be interpreted in one of two ways.

Pointers and arrays

- An array's name is also referred to as a **constant pointer**.
  - When used on the left side of the assignment operator, the compiler interprets it as **the memory location** to which to move that value resulting from **evaluation of the right side of the assignment operator**.
  - But, when used on the right side of the assignment operator, the name of a variable is interpreted to mean **the contents stored at that memory address** set aside to hold the value of that variable.

Pointers and arrays

- Since my_array is a constant,
  - Once the compiler establishes where the array itself is to be stored, it **“knows” the address of my_array[0]** and on seeing:
    - ptr = my_array;
  - It simply uses this address as a constant in the code segment and **there is no referencing of the data segment** beyond that.

Pointers and arrays

- This might be a good place explain further the use of the (void *) expression.
- As we have seen, we can have pointers of various types.

Pointers and arrays

- **(void *) expression**
  - Pointers can be of various types.
  - On different systems, the size of a pointer can vary.
    - It is also possible that the size of a pointer can vary depending on the **data type of the object to which it points**.
      - You can run into **trouble attempting to assign the values of pointers of various types to pointer variables of other types**.

Pointers and arrays

- To minimize this problem, C provides for a pointer of type void.
  - It can be declared as follows:
    - void *vptr;

Pointers and arrays

- A void pointer is sort of a generic pointer.
  - For example, while C **will not permit the comparison of a pointer to type integer with a pointer to type character**,
    - Either of these can be compared to a void pointer.

Pointers and arrays

- **Exercises**
  - Write a program in C to store n elements in an array and print the elements using pointer.
  - Write a program in C to sort an array using Pointer.
  - Write a program in C to compute the sum of all elements in an array using pointers.