FOUNDATIONAL CONCEPTS & TERMS

1. A program is a written set of instructions a computer (machine) can follow to solve a problem.

2. A program is written by a programmer using a language.

3. The building block of a machine is TRANSISTOR SWITCH.

4. A switch has only two states: ON (represented by 1) and OFF (represented by 0)

5. A machine is said to be binary considering that its building block has only two states (binary).

6. Programs have to be translated to MACHINE CODE/LANGUAGE (1's and 0's) for a machine to understand and execute.

7. Translators available are:
   a. **Assembler for Assembly language written with short codes called MNEMONICS**
   b. **Compiler** for compiled languages e.g. C, C++
   c. **Interpreter** for interpreted languages e.g. Python

8. A compiler translates the entire code before the machine starts executing it.

9. An interpreter does translation, immediately followed by machine execution, one line at a time.

10. A program written in C is saved with a file name and the extension .c i.e. programName.c

11. If program is written in a way in which writing rules are violated, it is said to contain **SYNTAX ERROR(S)**. This is similar to grammatical errors in written English.

12. If a program gets compiled successfully (no syntax error) the machine code is provided in a file with the same name as the program and extension .exe. For example, test.c will produce test.exe when compiled successfully. **This can only be observed in a PC, not in a phone.**

13. If a program written gives an output that is incorrect or behaves in an unexpected manner while been executed, it is said to have a **RUNTIME ERROR**.

The following are some important facts to remember when considering writing a program.

1. Every program written is intended to solve a problem or update/upgrade an existing program, because the new program solves the problem better.

2. It is the responsibility of the programmer to come up with the steps (ALGORITHM) that will be followed to solve problem at hand.

3. The C programming language (as well as many others) understood by the programmer is called a HIGH-LEVEL LANGUAGE, while the machine language is called a LOW-LEVEL LANGUAGE.

4. The instruction set (program) written by the programmer is called a source code; while the file containing it is called the source file.

5. Source files are created using text editors e.g. notepad.

6. The set of machine language instructions generated by a compiler is called the OBJECT CODE and the containing file is called the OBJECT FILE.

7. **The C programming language makes available to the programmer certain groups of predefined instructions called functions. They are meant for performing frequently needed tasks, e.g. displaying information, receiving data etc. These functions are called library functions.**

8. **Library functions are grouped to form files called HEADERS in which their complete description is given.**

9. **If a programmer is interested in using a given library function, then he/she must involve/include the header file in which it is defined in his/her program. If this is not done, using the function results in a syntax error.**

MAIN TYPES OF DATA

The main job done in any program is the manipulation data. The main types of data in C are outlined below:

Numeric Types

The main numeric types in C are: **char**, **int**, long, float (fractional), and **double** (fractional)

- **char**: single character with representation that is either symbolic (those that can be found on a keyboard) or numeric (all) i.e. ASCII code.
  Examples of its values include:
  - '#', '5', 'A', 'y' ⇔ Symbolic representation, when written in a program.
  - 65, 97, 55 ⇔ Numeric representation with 65 = 'A', 97 = 'a', and 55 = '7'

- **int**: Integer or whole number
  Examples
  - 4, 56, -45 ⇔ Decimal or Base 10 (Digits 0 – 9)
  - 0b11, 0b101 ⇔ Binary or Base 2 (Digits 0 & 1)
  - 0675, 0341 ⇔ Octal or Base 8 (Digits 0 – 7)
  - 0xa8f, 0XA8F ⇔ Hexadecimal or Base 16 (Digits 0 – 9, A – F or a – f)

- **long**: Similar to int except for the fact that its values, when written in a program must terminate in l or L.
  Examples
  - 4L, 56l, -45l ⇔ Decimal or Base 10
  - 0b11L, 0b101L ⇔ Binary or Base 2
  - 0675L, 0341L ⇔ Octal or Base 8
  - 0xa8fL, 0XA8FL ⇔ Hexadecimal or Base 16

- **float:** fractional number. Values of this data type are terminated with f or F when written in a program. E.g. -2.3f, 3.5F, 2.5e-5f, -1.2E-3F

- **double:** fractional number. E.g. -2.3, 3.5, 2.5e-5, -1.2E-3

Notes

- y x 10^z ⇔ yez or yEz which makes 2.5e-5 ⇔ 2.5 x 10^{-5} and -1.2E-3 ⇔ -1.2 x 10^{-3}
- Precision ⇔ Number of fractional numbers displayed after the (decimal) point.
- Double has twice the precision of float, hence is more accurate/suitable for mathematical computation.
- **An aggregation of 0 or more characters is called a string**
- **A string is normally bounded by double quotes, when written in a program e.g. "Hello"**

BASIC C PROGRAM STRUCTURE/TEMPLATE

Including a header:

```
#include <name_of_header.h>
```

C program template:

Programs like letters in written English have a structure followed. The basic structure of a C program is given below:

```
#include <stdio.h>
int main( ){

  //programming instructions

  return 0;
}
```

Note

A program can have as many headers as necessary.

DISPLAYING DATA

Any word that ends with brackets (empty or enclosing other expression(s)) is a function (call). **Every statement in C is terminated with a semi-colon (similar to the full stop in a written sentence in English).** Failing to terminate a statement with it (semi-colon) results in a syntax error during compilation.
To display data on a screen in C, the functions below can be used:

- **printf("format_string_without_placeholder");** ⇔ Displays the text between the double quotes on the screen

Example

```c
#include <stdio.h>

int main(){
  printf("Hello world!!!!");
  return 0;
}
```

Output

```
Hello world!!!!
```

- **printf("format_string_with_placeholder(s)", data, data, …, data);** ⇔ displays the control string with each data replacing a placeholder in the control string.
  - Placeholder ⇔ %alphabet
  - The alphabet used depends on the data type and format (some cases)

%d ⇔ Decimal integer
printf("%d", 0b11); ⇔ 3

%f ⇔ Floating point number **approximated to 6 decimal places**
printf("%f", 2.5e-3); ⇔ 0.002500

%c ⇔ Single character
printf("%c", 98); ⇔ b
printf("%c", '$'); ⇔ $

%s ⇔ String of characters
printf("Say %s", "hello"); ⇔ Say hello

%o ⇔ Octal integer
printf("9 in octal is %o", 9); ⇔ 9 in octal is 11

%x ⇔ Hexadecimal integer with alphabets in lowercase
printf("10 in base 16 is %x", 9); ⇔ 10 in base 16 is a

%X ⇔ Hexadecimal integer with alphabets in uppercase
printf("10 in base 16 is %X", 9); ⇔ 10 in base 16 is A

%e ⇔ Fractional number in scientific form with letter e, the sign (+ or -), and a 4-digit integer for the power/exponent
printf("%e", 0.025); ⇔ 2.500000e-002

%E ⇔ same as %e except for E replacing e
printf("%E", 0.025); ⇔ 2.500000E-002

%g ⇔ %e when power of the number in standard form is either less than -4 or greater than +5. It is %f otherwise. In each case trailing zeroes are removed.
printf("%g", 2.500e-4); ⇔ 0.00025
printf("%g", 2.5e-5); ⇔ 2.5e-0.005

%G ⇔ Similar to %g but chooses between %f and %E
printf("%G", 2.5e-4); ⇔ 0.00025
printf("%G", 2.5e-5); ⇔ 2.5E-0.005
printf("%g", 1.25100e5); ⇔ 125100
printf("%g", 1.25100e6); ⇔ 1.251e+006

IDENTIFIERS

An identifier is a name. A valid identifier in C is a combination of one or more characters drawn from the alphabets (a – z, A – Z), digits (0 – 9), underscore (_), and dollar ($) that doesn't start with a digit.

- _score, $cash, x, length ⇔ examples of valid identifiers
- #endsars, 4m, last name ⇔ examples of invalid identifiers

VARIABLES

System memory set aside for the storage of data in a program. **The size of this memory depends on the type of data and the nature of the system.** This size can be obtained by using the function **sizeof(data_type).**

VARIABLE DECLARATION/CREATION

- data_type name; ⇔ Single

```
double cgpa;
int age;
```

- data_type name, name, …, name; ⇔ Multiple of the same data type

```
double length, width, height;
```

VARIABLE INITIALIZATION

Assigning a value/**literal** to a declared variable.

- data_type name = value; ⇔ Single

```
double cgpa = 1.49;
```

- data_type name = value, name = value, …, name = value; ⇔ Multiple of the same data type

```
double length = 2.5, width = 4.5, height = 6;
```

NOTE

There specific terms used width some of the already mentioned data types. This may result in a change in the

1. memory size set aside for its variable
2. type of values it can be assigned i.e. whether to accept negative numbers or not.

These terms are presented below:

- **short:** used with int to reduce its size. Note that **short int** can simply be called **short.**
  - printf("%d", sizeof(int)); ⇔ 4
  - printf("%d", sizeof(short int)); ⇔ 2
  - printf("%d", sizeof(short)); ⇔ 2

- **long:** can be used with long and double to duplicate their size, consequently the range of values that they can represent.
  - printf("%d", sizeof(long)); ⇔ 4
  - printf("%d", sizeof(long long)); ⇔ 8
  - printf("%d", sizeof(double)); ⇔ 8
  - printf("%d", sizeof(long double)); ⇔ 12
- **const:** this is used to indicate that the value assigned to a variable is a constant i.e. won't change in the course of the program execution. Any attempt to change it will result in an error. **The name of a variable holding a constant is written in block, by convention.**
  - const double PI = 3.141592654;

Additional facts to know about variables include:

- It is not advisable to assign a value of a given type, to a variable that is associated with a smaller memory size and range. Some systems will raise a warning concerning possible loss in precision. For example, a double variable can be assigned a float value, but it isn't advisable for a float variable to be assigned a double value.
- If a variable contains a value, and gets assigned another afterwards, the new value replaces the previous value in the variable, i.e. a variable always contains the last value it was assigned.

RECEVING DATA FROM KEYBOARD

Function for this is **scanf("placeholder(s)", pointer(s));**

Vital facts to note:

- Placeholders have a format similar to those used in printf(…)
- A pointer refers to the memory address of a variable

- ❖ Writing the ampersand (&) character next to the name of a variable provides the pointer of the variable, i.e. &variable_name ⇔ variable pointer.
- ❖ Placeholders must match pointers in terms of number, data type, and position (where number > 1).

The major placeholders are given below:

❖ **%d** ⇔ stores value as a base 10 (decimal) integer

Example

```
printf("Enter an integer: ");
int val;
scanf("%d", &val);
printf("Value stored = %d", val);
```

Output

```
Enter an integer: 45
Value stored = 45
```

❖ **%lf** ⇔ stores value as a double

Example

```
printf("Enter a double: ");
double val;
scanf("%lf", &val);
printf("Value stored = %f", val);
```

Output

```
Enter a double: 2.3e-4
Value stored = 0.000230
```

❖ **%c** ⇔ stores value as a char. The value entered is the symbol (if available).

Example

```
printf("Enter a character: ");
char val;
scanf("%c", &val);
printf("Value stored = %c", val)
```

Outputs

```
Enter a character: c
Value stored = c
```

If the symbol is not available then, the code must be received as an integer, then interpreted internally as a char.

Example

```
printf("Enter a character: ");
char val;
scanf("%d", &val);
printf("Value stored = %c", val)
```

Output

```
Enter a character: 64
Value stored = @
```

Others are:

❖ **%o** ⇔ stores value as base 8 (octal) integer

Example

```
printf("Enter an octal integer: ");
int val;
scanf("%o", &val);
printf("Value stored = %d", val);
```

Output

```
Enter an octal integer: 10
Value stored = 8
```

❖ **%x** or **%X** ⇔ stores value as a hexadecimal integer

Example

```
printf("Enter a hexadecimal integer: ");
int val;
scanf("%x", &val);
printf("Value stored = %d", val);
```

Output

```
Enter a hexadecimal integer: a
Value stored = 10
```

❖ **%f** ⇔ stores value as a float

Example

```
printf("Enter a hexadecimal integer: ");
cinthar val;
scanf("%x", &val);
printf("Value stored = %d", val);
```

Output

```
Enter a fractional number: 2.4e-4
Value stored = 0.000240
```

ARITHMETIC OPERATORS

1. + (addition)

Example

```
5 + 5 = 10
```

2. – (subtraction)

Example

```
5 – 2 = 3
```

3. * (multiplication)

Example

```
5 * 2 = 10
```

4. / (division): result is fractional if at least one operand is fractional, and it is an integer (no fractional part, because it is truncated) if both operands are integer values.

Examples

```
7 / 2 = 3
7 / 2.0 = 3.5
7.0 / 2 = 3.5
7.0 / 2.0 = 3.5
```

5. % (modulus): remainder operator which returns the remainder of a division.

Example

```
5 % 2 = 1
25 % 5 = 0
3 % 5 = 3
```

OPERATOR PRECEDENCE

When an expression contains multiple arithmetic operators, in what order will they be used in the course of the evaluation? That's the answer operator precedence provides.

*, /, % ⇔ First

+, - ⇔ Last

Operators of the same level of precedence are evaluated one after the other, from left to right.

Example

```
6 + 2 * 4 / 3 % 5 – 1
6 + (2 * 4) / 3 % 5 – 1
=> 6 + (8 / 3) % 5 – 1
=> 6 + (2 % 5) – 1
=> (6 + 2) – 1
=> 8 – 1 = 7
```

The above result is confirmed by the statement below:

```
printf("%d", 6 + 2 * 4 / 3 % 5 - 1);
```

The output of the above statement is 7.

ASSIGNMENT OPERATORS

1. Assignment operator (=): assigns the value on the right to the variable on the left

2. Augmented assignment operators: assignment operator joined to an arithmetic operator
   a. += ⇔ (var = var + value) shortened to (var += value)
   b. -= ⇔ (var = var - value) shortened to (var -= value)
   c. *= ⇔ (var = var * value) shortened to (var *= value)
   d. /= ⇔ (var = var / value) shortened to (var /= value)
   e. %= ⇔ (var = var % value) shortened to (var %= value)

NOTE: The result of an assignment operator (augmented or not) is the value assigned to the variable.

Example

```c
#include <stdio.h>

int main(){
  int a = 0, b = 1, c = 0, d = 2, e = 12, f = 5;

  printf("%d\n", a = 1); // a = 1
  printf("%d\n", b += 2); // b = (b + 2) <=> b = 3
  printf("%d\n", c -= 7); // c = c - 7 <=> c = -7
  printf("%d\n", d *= 5); // d = d * 5 <=> d = 10
  printf("%d\n", e /= 2); // e = e / 2 <=> e = 6
  printf("%d", f %= 2); // f = f % 2 <=> f = 1

  return 0;
}
```

Output

```
1
3
-7
10
6
1
```

TYPE CASTING

This refers to the art of converting a data of a given type to a different type. The syntax for this is given below:

(type)data

For 5.0%2 will return an error, because % works with only integers, but (int)5.0%2 will work fine.

COMMENTS

Comments are explanatory notes added to a program to make it easier to understand. Comments are not executed, instead skipped. There are two types of comments, these are:

❖ **Single line comments**: Starts with two forward slashes i.e. //

Example

```c
// storage for amount invested
```

❖ **Multi-line comments**: Starts with the character combination, /* and ends with the combination, */

Example

```c
/* Program to calculate
the volume of a cone */
```

PROGRAMMING EXAMPLE

Write a program that computes and displays the volume of a cone, using values for diameter and height supplied by the user in response to a prompt. Formula for calculating the volume of a cone is $^1/_3 \pi r^2 h$, where the value of 3.141592654 is to be used for π.

Program

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

```c
int main(){
  // prompt for diameter and height
  printf("Enter values for diameter and height: ");

  // initialize variables for diameter, and height
  double d = 0, h = 0;

  // receive and save diameter and height
  scanf("%lf%lf", &d, &h);

  // compute and save the volume
  double r = d/2; // radius
  double vol = (3.141592654 * r * r * h) / 3;

  // display computed volume
  printf("Volume of the cone = %f", vol);

  return 0;
}
```

Output

```
Enter values for diameter and height: 6.8 5.2
Volume of the cone = 62,949139
```

EXERCISES

1. Given an airplane's acceleration a and take-off speed v, you can compute the minimum runway length needed for an airplane to take off using the following formula:

   $$length = v^2/(2a)$$

   Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), then, displays the minimum runway length. Here is a sample run:

   ```
   Enter speed and acceleration: 60 3.5
   The minimum runway length for this airplane is 514.286
   ```

2. Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

   Q = M * (finalTemperature – initialTemperature) * 4184

   where M is the weight of water in kilograms, initial and final temperatures are in degrees Celsius, and energy Q is measured in joules. Here is a sample run:

   ```
   Enter the amount of water in kilograms: 55.5
   Enter the initial temperature: 3.5
   Enter the final temperature: 10.5
   The energy needed is 1625484.0
   ```

3. Write a program that prompts the user to enter the minutes (e.g., 1 billion), and displays the number of years and remaining days for the minutes. For simplicity, assume that a year has 365 days. Here is a sample run:

   ```
   Enter the number of minutes: 1000000000
   1000000000 minutes is approximately 1902 years and 214 days
   ```

4. Write a program that reads an integer between 99 and 1000 and multiplies all the digits in the integer. For example, if an integer is 932, the multiplication of all its digits is 54.
   Hint: Use the % operator to extract digits, and use the / operator to remove the extracted digit. For instance, 932 % 10 = 2 and 932 / 10 = 93.
   Here is a sample run:

   ```
   Enter a number between 99 and 1000: 999
   The multiplication of all digits in 999 is 729
   ```

5. Write a program that reads a mile in a double value from the console, converts it to kilometers, and displays the result. The formula for the conversion is as follows:
   1 mile = 1.6 kilometers
   Here is a sample run:

```
Enter miles: 96
96 miles is 153.6 kilometers
```

MATH FUNCTIONS

Using these predefined functions requires including the header **math.h** in the program. Some of these important functions are given below with examples showing the results they return when used. Each function takes one or more arguments (values, variable, or expression between the brackets).

1. sqrt(x) ⇔ $\sqrt{x}$ e.g. sqrt(25) = 5.0

2. cbrt(x) ⇔ $\sqrt[3]{x}$ e.g. cbrt(8) = 2.0

3. pow(x, y) ⇔ x^y e.g. pow(5, 3) = 125

4. exp(x) ⇔ e^x e.g. exp(0.5) = 1.64728

5. log(x) ⇔ ln x (Natural logarithm of x, i.e. logarithm of x to base e)

6. log10(x) ⇔ $\log_{10} x$ (Logarithm of x to base 10)

7. abs(x) ⇔ $|x|$ (absolute value of x returned as an integer)

8. fabs(x) ⇔ $|x|$ (absolute value of x returned as a floating-point number)

9. ceil(x) ⇔ $\lceil x \rceil$ (the nearest integer that is greater than or equal to x)
   ceil(1.2) = 2
   ceil(-2.3) = -2
   ceil(2.0) = 2

10. floor(x) ⇔ $\lfloor x \rfloor$ (nearest integer that is lesser than or equal to x)
    floor(1.2) = 1
    floor(-2.3) = -3
    floor(2.0) = 2

11. sin(x): returns sine of x radians

12. cos(x): returns cosine of x radians

13. tan(x): returns tangent of x radians

14. asin(y) ⇔ $\sin^{-1} x$: returns an angle in radians

15. acos(y) ⇔ $\cos^{-1} x$: returns an angle in radians

16. atan(y) ⇔ $\tan^{-1} x$: returns an angle in radians

17. sinh(x)

18. cosh(x)

19. tanh(x)

20. asinh(y) ⇔ $\sinh^{-1} y$

21. acosh(y) ⇔ $\cosh^{-1} y$

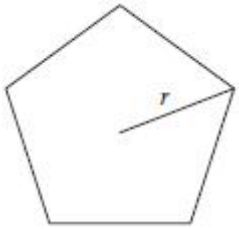
22. atanh(y) ⇔ $\tanh^{-1} y$

NOTE

- degress = $(radians \times 180)/\pi$
- radians = $(degress \times \pi)/180$

EXERCISES

1. Write a program that prompts the user to enter the length from the center of a pentagon to a vertex and computes the area of the pentagon, as shown in the following figure.



The formula for computing the area of a pentagon is Area = $\frac{5 \times s^2}{4 \times \tan\frac{\pi}{5}}$, where s is the length of a side. The side can be computed using the formula s = $2r \sin\frac{\pi}{5}$, where r is length from the center of a pentagon to a vertex. Here is a sample run:

```
Enter the length from the center to a vertex: 5.5
The area of the pentagon is 71.92
```

2. The great circle distance is the distance between two points on the surface of a sphere. Let (x1, y1) and (x2, y2) be the geographical latitude and longitude of two points. The great circle distance between the two points can be computed using the following formula:

$$d = radius * \arccos(\sin(x_1) * \sin(x_2) + \cos(x_1) * \cos(x_2) * \cos(y_1 - y_2))$$

Write a program that prompts the user to enter the latitude and longitude of two points on the earth in degrees and displays its great circle distance. The average radius of the earth is 6,371.01 km. Here is a sample run.

```
Enter point 1 (latitude and longitude) in degrees: 39.55 –116.25
Enter point 2 (latitude and longitude) in degrees: 41.5 87.37
The distance between the two points is 10691.79183231593 km
```

3. Write a program that receives a character and displays its ASCII code (an integer between 0 and 127). Here is a sample run:

```
Enter a character: E
The ASCII code for character E is 69
```

ESCAPE CHARACTERS

The escape character is \ (backslash). It changes the way certain characters are interpreted in a program. Below are some escape characters of interest.

1. \\ ⇔ \ (Backslash)
   printf(“5 \\ 2”); => 5 \ 2

2. \n ⇔ Line feed, new line, or ENTER key
   printf(“5\n2”); =>
   5
   3

3. \t ⇔ Horizontal tab
   printf(“\5\t2”); => 5 2

4. \a ⇔ Beep (noticeable only in PC’s)

5. \’ ⇔ ‘ (Single quote)
   printf(“Adam%cs apple”, ‘\’’); => Adam’s apple

6. \” ⇔ “ (Double quote)
   printf(“\”Hello\””); => “Hello”

FORMATTING PROGRAM OUTPUT

Besides using some of the escape characters to format a program output, provisions are also available via the placeholders through numbers written with them. Important facts to note include:

1. Field width ⇔ the amount of space (in terms of number of characters) set aside for a given output data. It is represented by the integer after % in the placeholder.
   Examples
   ```
   printf("Hi %d.", 5);
   ```
   Hi 5.

   ```
   printf("Hi %7d.", 5);
   ```
   Hi   5.

   Note: If the space provided is not enough for the data, the system ignores it and fits the data in the space that is just enough for it.

2. Precision ⇔ the number of fractional numbers to display after a point. It is mainly associated with fractional numbers. This number is represented by an integer separated from the field width by a point. **It can also be used without the field width.**
   ```
   printf("PI to 4 d.p. = %.4f.", 22/7.0);
   ```
   PI to 4 d.p. = 3.1429.

3. Alignment ⇔ If the field width is positive (not necessary to right the sign), the data is aligned to the right (default) of the field specified, while if it is negative, the alignment is to the left.
   ```
   printf("Hi %-7d.", 5);
   ```
   Hi 5   .

RELATIONAL AND EQUALITY OPERATORS

These operators when used in expressions, return true (1 or any number that is not 0) and false (0).

1. a == b ⇔ a is equal to b
   8 == 9 = 0
   2 == 2 = 1

2. a != b ⇔ a is not equal to b
   8 != 9 = 1
   8 != 8 = 0

3. a < b ⇔ a is less than b
   6 < 10 = 1
   6 < 6 = 0

4. a > b ⇔ a is greater than b
   5 > 4 = 1
   5 > 7 = 0

5. a <= b ⇔ a is less than or equal to b
   6 <= 3 = 0
   3 <= 3 = 1
   5 <= 9 = 1

6. a >= b ⇔ a is greater than or equal to b
   6 >= 3 = 1
   6 >= 6 = 1
   2 >= 7 = 0

LOGICAL OPERATORS

Operands are interpreted as Boolean data

1. ! ⇔ Not/Negation: converts true (1) to false (0) and false (0) to true (1).

2. || ⇔ Logical OR: result is true if at least one operand is true.

3. && ⇔ Logical AND: result is false if at least one operand is false.

4. ^ ⇔ Exclusive OR/XOR: result is true only if its two operands different otherwise it is false.
   Generally, the result is true if the number of true operands is odd and false if it is even.

NOTE

In C, provision is available to create variables of Boolean data type and replace the numeric forms of Boolean data with **true** or **false**. Important facts to note include

- header required ⇔ **stdbool.h**
- data type ⇔ **bool** or **_Bool**
- 0 ⇔ **false**
- 1 (any number that is not 0) ⇔ **true**

SHORT-CIRCUIT EVALUATION

When a system is evaluating an expression that is made up of several Boolean expressions, joined with the either || or &&, it stops further evaluation and returns the final result, when it encounters the

- first true for ||
- first false (0) for &&

The above form of evaluation is called short-circuit evaluation.

EXAMPLE 1

```c
#include <stdio.h>
#include <stdbool.h>
int main(){
  int a = -1, b = -1, c = -1;

  bool rzlt = (a = 0)||(b = 2)||(c = 0);

  // a gets assigned (0 = false), b gets assigned (2 = true),
  // but not c, because the 1st true has been encountered
  printf("Result is %d\n", rzlt);
  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d", c);

  return 0;
}
```

RESULT

```
Result is 1
a = 0
b = 2
c = -1
```

EXAMPLE 2

```c
#include <stdio.h>
#include <stdbool.h>
int main(){
  int a = -1, b = -1, c = -1;

  bool rzlt = (a = 0)&&(b = 2)&&(c = 0);

  // a gets assigned (0 = false), but not b or c
  // because the 1st false has been encountered
  printf("Result is %d\n", rzlt);
  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d", c);

  return 0;
}
```

RESULT

```
Result is 0
a = 0
b = -1
c = -1
```

BRANCHING/SELECTIONS

1. **Tenary Operator (?)**: used in an expression to decide on which of two possible results to return. The decision is based on the result of a preceding Boolean expression, a true returns the first, while a false return the second value. The entire expression (equivalent to the returned value) is either assigned to a variable or used in a function.

   Boolean_expression ? true_value : false_value

   Example

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  printf("Enter an integer: ");

  int n;
  scanf("%d", &n);

  printf("%d is %s.", n, (n%2 == 0) ? "even" : "odd");

  return 0;
}
```

   Results

```
Enter an integer: 7
7 is odd.

Enter an integer: 0
0 is even.
```

2. **if-Statement**: makes provision for just 1 possibility

```c
if(boolean_expression){
  // statements to execute if boolean_expression is true
}
```

   Example

```c
#include <stdio.h>
int main(){
  int n;
  printf("Enter an integer: ");
  scanf("%d", &n);

  if(n%2 == 0)
    printf("%d is even.", n);

  if(n%2 != 0)
    printf("%d is odd.", n);

  return 0;
}
```

   Results

```
Enter an integer: 4
4 is even.

Enter an integer: 7
7 is odd.
```

3. **if-else Statement**: makes provision for only 2 possibilities

```c
if(boolean_expression){
  // statements to execute if boolean_expression is true
}
else{
  // statements to execute if boolean_expression is false
}
```

Example

```c
#include <stdio.h>
int main(){
  int n;
  printf("Enter an integer: ");
  scanf("%d", &n);

  if(n%2 == 0)
    printf("%d is even.", n);
    else
      printf("%d is odd.", n);

  return 0;
}
```

Results

```
Enter an integer: 4
4 is even.

Enter an integer: 7
7 is odd.
```

4. **Nested if-else Statement:** makes provision for more than 2 possibilities

```
if(boolean_expression_1){
  // statements to execute if boolean_expression_1 is true
}
else if(boolean_expression_2){
  // statements to execute if boolean_expression_2 is true
}
:
else if(boolean_expression_n){
  // statements to execute if boolean_expression_n is true
}
else{
  // statements to execute if all boolean_expression_n is false
}
```

Example

```c
#include <stdio.h>
int main(){
  int score;
  printf("Enter a score: ");
  scanf("%d", &score);

  char grade;

  if(score < 45)
    grade = 'F';
    else if(score < 50)
      grade = 'D';
      else if(score < 60)
        grade = 'C';
        else if(score < 70)
          grade = 'B';
          else
            grade = 'A';

  // Display grade
  printf("Grade is %c", grade);

  return 0;
}
```

Results

```
Enter a score: 56
```

```
Grade I s C

Enter a score: 44
Grade is F
```

5. **switch-Statement:** uses the values stored in a variable to decide the course of action to take.

```
switch(variable){
  case value1:
    // execute statements if variable == value1
    break;
  case value2: case value3:
    // execute statements if variable == value2 || variable == value3
    break;
  :
  case valueN:
    // execute statements if variable == valueN
    break;
  default:
    // execute statements if variable != valueN
}
```

Example

```c
#include <stdio.h>
int main(){
  // Prompt for expression
  printf("Enter an arithmetic expression (no space) e.g. 5/2:\n");

  // Create storage for data
  int a, b;
  char operator;

  // Receive data (a, operator, and b)
  scanf("%d%c%d", &a, &operator, &b);

  // Check operator and perform the matching operation
  switch(operator){
    case '+':
      printf("Answer = %d", a + b);
      break; // discontinue checking
    case '-':
      printf("Answer = %d", a +- b);
      break;
    case '*':
      printf("Answer = %d", a * b);
      break;
    case '/':
      printf("Answer = %d", a / b);
      break;
    case '%':
      printf("Answer = %d", a % b);
      break;
    default:
      printf("Operator is invalid");
  }

  return 0;
}
```

Results

```
Enter an arithmetic expression (no space) e.g. 5/2:
5%2
Answer = 1

Enter an arithmetic expression (no space) e.g. 5/2:
25-56
```

```
Answer = -31

Enter an arithmetic expression (no space) e.g. 5/2:
5#2
Operator is invalid
```

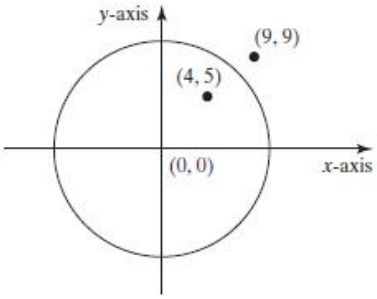
The program above, written with nested if-else statement is as presented below

```c
#include <stdio.h>

int main(){
  // Prompt for expression
  printf("Enter an arithmetic expression (no space) e.g. 5/2:\n");

  // Create storage for data
  int a, b;
  char operator;

  // Receive data (a, operator, and b)
  scanf("%d%c%d", &a, &operator, &b);

  // Check operator and perform the matching operation
  if(operator == '+')
    printf("Answer = %d", a + b);
    else if(operator == '-')
      printf("Answer = %d", a +- b);
      else if(operator == '*')
        printf("Answer = %d", a * b);
        else if(operator == '/')
          printf("Answer = %d", a / b);
          else if(operator == '%')
            printf("Answer = %d", a % b);
            else
              printf("Operator is invalid");

  return 0;
}
```

EXERCISES

1. Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of every pair of two edges is greater than the remaining edge.

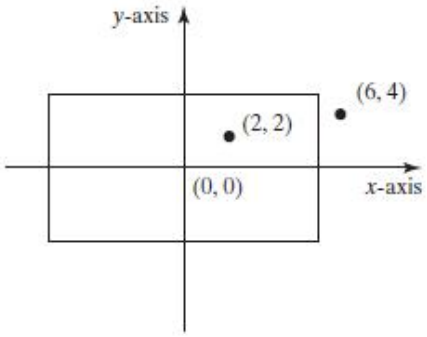
2. Write a program that prompts the user to enter a point (x, y) and checks whether the point is within the circle centered at (0, 0) with radius 10. For example, (4, 5) is inside the circle and (9, 9) is outside the circle, as shown in the figure below.



(Hint: A point is in the circle if its distance to (0, 0) is less than or equal to 10. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 + y_1)^2}$. Test your program to cover all cases.) Two sample runs are shown below:

```
Enter a point with two coordinates: 4 5
Point (4.0, 5.0) is in the circle

Enter a point with two coordinates: 9 9
Point (9.0, 9.0) is not in the circle
```

3. Write a program that prompts the user to enter a point (x, y) and checks whether the point is within the rectangle centered at (1, 1) with width 10 and height 5. For example, (2, 2) is inside the rectangle and (6, 4) is outside the rectangle, as shown in the figure below.

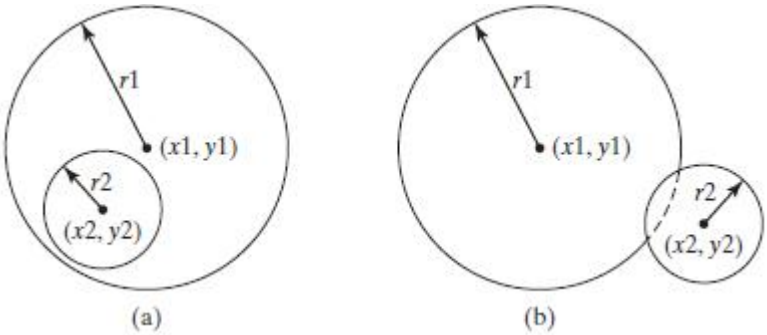


(Hint: A point is in the rectangle if its horizontal distance to (0, 0) is less than or equal to 10 / 2 and its vertical distance to (0, 0) is less than or equal to 5.0 / 2. Test your program to cover all cases.) Here are two sample runs.

```
Enter a point with two coordinates: 2 2
Point (2.0, 2.0) is in the rectangle
```

```
Enter a point with two coordinates: 6 4
Point (6.0, 4.0) is not in the rectangle
```

4. Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in the figure below.



(Hint: circle2 is inside circle1 if the distance between the two centers <= r1 - r2 and circle2 overlaps circle1 if the distance between the two centers <= r1 + r2. Test your program to cover all cases.) Here are the sample runs:

```
Enter circle1's center x-, y-coordinates, and radius: 0.5 5.1 13
Enter circle2's center x-, y-coordinates, and radius: 1 1.7 4.5
circle2 is inside circle1
```

```
Enter circle1's center x-, y-coordinates, and radius: 3.4 5.7 5.5
Enter circle2's center x-, y-coordinates, and radius: 6.7 3.5 3
circle2 overlaps circle1
```

```
Enter circle1's center x-, y-coordinates, and radius: 3.4 5.5 1
Enter circle2's center x-, y-coordinates, and radius: 5.5 7.2 1
circle2 does not overlap circle1
```

5. Write a program that reads in three integers and prints "in order" if they are sorted inascending or descending order, or "not in order" otherwise. For example,
   1 2 5 in order
   1 5 2 not in order
   5 2 1 in order
   1 2 2 in order

INCREMENT & DECREMENT OPERATORS

Increment operator (++) ⇔ increases the value in a variable by 1.
Decrement operator (--) ⇔ decreases the value in a variable by 1.

Important facts to note:

- These operators only work with variables

- Written before or after the variable makes no difference, in the absence of any other operator or function i.e. **++variable** is same as **variable++**

Example

```c
#include <stdio.h>
int main(){
  double a = 2.5;
  printf("Original value = %f\n", a);

  a++;
  printf("Value after 1st increment = %f\n", a);

  ++a;
  printf("Value after 2nd increment = %f\n", a);

  a--;
  printf("Value after 1st decrement = %f\n", a);

  a--;
  printf("Value after 2nd decrement = %f\n", a);

  return 0;
}
```

Result

```
Original value = 2.500000
Value after 1st increment = 3.500000
Value after 2nd increment = 4.500000
Value after 1st decrement = 3.500000
Value after 2nd decrement = 2.500000
```

PRE/POST INCREMENT AND DECREMENT

Position of the operator with respect to it variable (the operand) when they are among other operators or in a function.

PRE ⇔ ++variable and --variable

They appear before the variable hence are considered before the next operator or function.

Examples

```c
#include <stdio.h>

int main(){
  int a = 6;
  int b = ++a;  //increase a by 1 and assign it to b

  // this will make a and b 7
  printf("a = %d\tb = %d", a, b);
  return 0;
}
```

Output

```
a = 7   b = 7
```

```c
#include <stdio.h>

int main(){
  int a = 6;

  // decrease a by 1 then print it
  printf("a = %d", --a);
  return 0;
}
```

Output

```
a = 5
```

POST ⇔ variable++ and variable--

They appear after the variable hence are considered after the next operator or function.

Examples

```c
#include <stdio.h>

int main(){
  int a = 6;
  int b = a++;  // assign a to b, then increase it by 1

  // this will make b 6 and a 7
  printf("a = %d\tb = %d", a, b);

  return 0;
}
```

Output
a = 7 b = 6

```c
#include <stdio.h>

int main(){
  int a = 6;

  // print a, then decrease it by 1
  printf("a = %d\n", a--);
  printf("a = %d", a);
  return 0;
}
```

Output
a = 6
a = 5

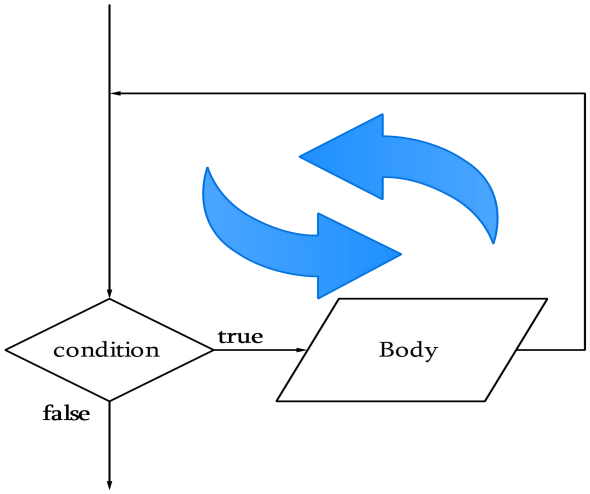
LOOPS/ITERATION

Loops are situations where an action or a set of actions, is to be performed a given number of times (**determinate loops**) or while a programming condition persists (**indeterminate loops**). Below are the looping techniques available:

while-Loop (Indeterminate)

```
while(condition){
  // execute statement(s) in body if condition is true
  // go back to condition
}
// Execute when condition becomes false
```

Flow of program control is depicted in the diagram (flow chart) below:



Example: Display the sum of all numbers from 1 to 40

```c
#include <stdio.h>
int main(){
  int sum = 0;

  int n = 1;
  while(n <= 40){
    sum = sum + n;
    n++; // get next number
  }
  // loop has ended, time to display sum
```

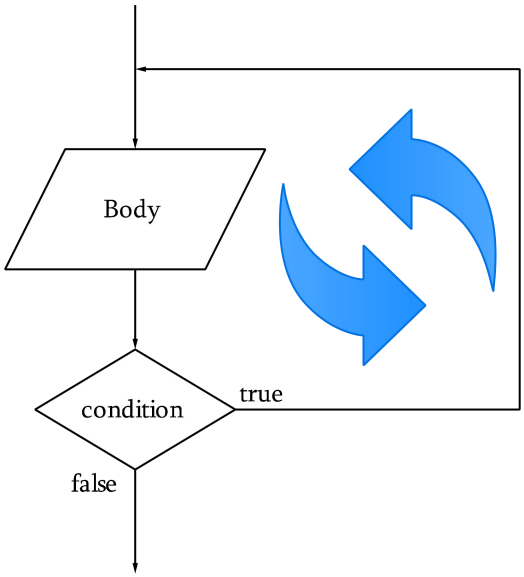
```
  printf("Sum from 1 through to 40 = %d", sum);

  return 0;
}
```

Result
Sum from 1 through to 40 = 820

do-while Loop (Indeterminate)

```
do{
  // Execute statements in body
  // Go to condition
}while(condition);
// Execute when condition becomes false
```

Flow of program control is given in the flow chart below:



Example: Display the sum of all numbers from 1 to 40

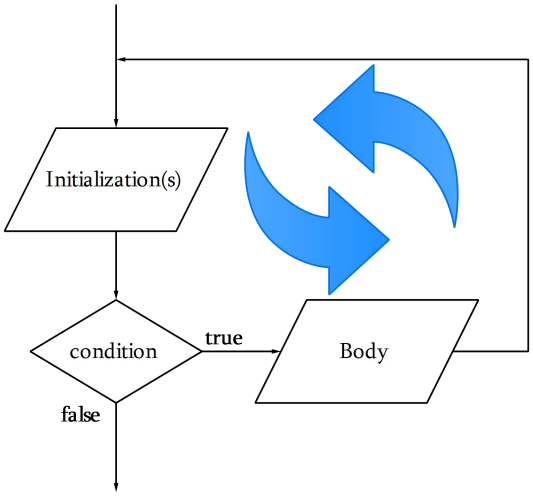
```
#include <stdio.h>
int main(){
  int sum = 0;

  int n = 1;
  do{
    sum = sum + n;
    n++;
  }while(n <= 40);
  // loop has ended, time to display sum
  printf("Sum from 1 through to 40 = %d", sum);

  return 0;
}
```

Result
Sum from 1 through to 40 = 820

for-Loop (Determinate)
This loop by structure has four sections, which are:

1. Initialization(s)
2. Condition
3. Body
4. Change value(s)

The flow diagram below illustrates how the for-loop operates:



Example: Display the sum of all numbers from 1 to 40

```c
#include <stdio.h>
int main(){
  int sum = 0;
  int n;

  for(n = 0; n <= 40; n++)
    sum = sum + n;

  // loop has ended, time to display sum
  printf("Sum from 1 through to 40 = %d", sum);

  return 0;
}
```

Result

```
Sum from 1 through to 40 = 820
```

Important facts to note include:

1. The initialization of a for-loop contains the first value of the target series
2. The final value of the series is tied to the condition of the for-loop
3. The body of a loop can contain
   - Another loop
   - Any of the branching statements
4. It is also possible to have a loop in the body of a branching statement.

BREAKING PROGRAM FLOW

Two keywords are used to interfere with the way the iterations of a loop are executed. These are:

1. **break**: it ends the loop containing it, when it is executed
2. **continue**: it ends the current iteration of the loop paving the way for the next one to start

```c
#include <stdio.h>
int main(){
  int n;

  for(n = 1; n <= 10; n++)
    printf("%d\n", n);

  return 0;
}
```

Output

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

```c
#include <stdio.h>
int main(){
  int n;

  for(n = 1; n <= 10; n++){
    if(n%3 == 0)
      break;
    printf("%d\n", n);
  }

  return 0;
}
```

Output

```
1
2
```

```c
#include <stdio.h>
int main(){
  int n;

  for(n = 1; n <= 10; n++){
    if(n%3 == 0)
      break;
    printf("%d\n", n);
  }

  return 0;
}
```

Output

```
1
2
4
5
7
8
10
```

NOTE

system(“cls”) defined in stdlib.h ⇔ clears the console when executed

Example: Program to illustrate what is possible with the above function.

```c
#include <stdio.h>
#include <stdlib.h>
int main(){
  // Create storage for data
  double bal = 0, amnt = 0;
  int tran = 0;
  char ans;

  do{
    // Clear screen
    system("cls");

    printf("MENU\n1: Deposit\n2: Withdraw\n3: Check Balance\n4: Quit\n\nChoose a transaction: ");

    // Receive receive choice
    scanf("%d", &tran);

    //
    if(tran == 1 || tran == 2){
      printf("Enter amount: $");
      scanf("%lf", &amnt);
      // update balance based on transaction type
      bal = (tran == 1) ? (bal + amnt) : (bal - amnt);
      printf("\nTransaction successful. Current balance is $%.2f\n", bal);
    }
    else if(tran == 3)
      printf("\nCurrent Wallet Balance = $%.2f\n", bal);
    else if(tran == 4){
      system("cls");
      break; // exit loop
    }
    else
      printf("\nChoice is invalid");
    printf("\nWish to perform another transaction (y/n)? ");
    scanf("%c%c", &ans, &ans); // 1st %c to remove the trailing \n from entering amount

  }while(ans == 'y' || ans == 'Y');

  printf("BYE!!!\n");
```

```
  return 0;
}
```

Output

```
MENU
1: Deposit
2: Withdraw
3: Check Balance
4: Quit

Choose a transaction: 1
Enter amount: $10500

Transaction successful. Current balance is $10500.00

Wish to perform another transaction (y/n)? y


MENU
1: Deposit
2: Withdraw
3: Check Balance
4: Quit

Choose a transaction: 2
Enter amount: $250

Transaction successful. Current balance is $10250.00

Wish to perform another transaction (y/n)? y


MENU
1: Deposit
2: Withdraw
3: Check Balance
4: Quit

Choose a transaction: 3

Current Wallet Balance = $10250.00

Wish to perform another transaction (y/n)? y


MENU
1: Deposit
2: Withdraw
3: Check Balance
4: Quit

Choose a transaction: 5

Choice is invalid
Wish to perform another transaction (y/n)? y


MENU
1: Deposit
2: Withdraw
3: Check Balance
4: Quit

Choose a transaction: 4
```

BYE!!!

EXERCISES

1. Write a program that displays all the numbers from 100 to 1,000, ten per line, that are divisible by 3 and 4. Numbers occupy a field width of 4 aligned to the left.
   Expected result:
```
108 120 132 144 156 168 180 192 204 216
228 240 252 264 276 288 300 312 324 336
348 360 372 384 396 408 420 432 444 456
468 480 492 504 516 528 540 552 564 576
588 600 612 624 636 648 660 672 684 696
708 720 732 744 756 768 780 792 804 816
828 840 852 864 876 888 900 912 924 936
948 960 972 984 996
```

2. Write a program that displays all the numbers from 100 to 200, ten per line, that are divisible by 3 or 4, but not both. Numbers occupy a field width of 4 aligned to the left.
   Expected result:
```
100 102 104 105 111 112 114 116 117 123
124 126 128 129 135 136 138 140 141 147
148 150 152 153 159 160 162 164 165 171
172 174 176 177 183 184 186 188 189 195
196 198 200
```

3. Use a while loop to find the smallest integer n such that n^3 is greater than 12,000.
   Expected result: n = 23

4. Use a while loop to find the largest integer n such that n^2 is less than 12,000.
   Expected result: n = 23

5. Write a program to compute the following summation:

$$\frac{1}{3}+\frac{3}{5}+\frac{5}{7}+\frac{7}{9}+\frac{9}{11}+\frac{11}{13}+\cdots+\frac{95}{97}+\frac{97}{99}$$

   Expected result: Sum = 45.124450

6. You can approximate π by using the following summation:

$$\pi = 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots+\frac{(-)^{i+1}}{2i-1}\right)$$

   Write a program that displays the π value for i = 10000, 20000, …, and 100000.
   Expected result:
```
i =  10000, pi = 3.141493
i =  20000, pi = 3.141543
i =  30000, pi = 3.141559
i =  40000, pi = 3.141568
i =  50000, pi = 3.141573
i =  60000, pi = 3.141576
i =  70000, pi = 3.141578
i =  80000, pi = 3.141580
i =  90000, pi = 3.141582
i = 100000, pi = 3.141583
```

7. Write a program that displays the following table (note that 1 inch is 2.54 centimeters):

```
Inches    Centimetres
1                2.54
2                5.08
...

9               22.86
10        25.40
```

   Expected result:

Inches	Centimetres
1	2.54
2	5.08
3	7.62
4	10.16
5	12.70
6	15.24
7	17.78
8	20.32
9	22.86
10	25.40

USER-DEFINED FUNCTIONS

A function is a block of code meant to perform a defined set of tasks (especially those repeated several times in a program).

FUNCTION STRUCTURE

A function has two parts and these are:

1. A head/prototype comprising: data type returned, name (valid identifier), brackets which may contain a comma-separated list of parameters (parameter ⇔ type name).
   Note: prototype without the return type is called the **FUNCTION SIGNATURE.**

2. A body/implementation comprising statement(s) enclosed in braces (immediately after the head).

FUNCTION SYNTAX

1. For a function with parameters
```
returnDataType functionName (type name, type name, …, type name)
{
  //statements making body
}
```

2. For a function that doesn't take a parameter
```
returnDataType functionName()
{
  // statements making body
}
```

NOTES

1. Return data type for a function that doesn't return any data is **void**
2. The keyword "return" is used to send back value from a function expected to return one. Format is **return value;** or **return variableName;**

Example

```
double getVolume(double diameter, double height)
{
  double r = diameter/2;
  double area = 3.141592654 * r * r;
  double volume = area * h;
  return volume;
}
```

CALLING A FUNCTION

This is actually using a function in another function.

Format

- functionName() ⇔ for a function that has no parameter
- functionName(argument, argument, …, argument) ⇔ for a function with parameters

NOTE

1. A function call can be assigned to a variable or made an argument in another function, if the called function returns a value. For this to work, data returned by the function must be of the right type.
2. The arguments must match parameters in terms of data type and position.

3. An argument can be a literal/value, variable, or a function (that returns data).

Examples 1
```
double v = getVolume(3.5, 7.8);
```

Example 2
```
printf("Volume = %.4f", getVolume(3.5, 7.8));
```

Example 3
```
double d = 3.5, h = 7.8, vol = 0;
vol = getVolume(d, h);
```

POSITIONING A CUSTOM FUNCTION WITH RESPECT TO MAIN

1. **Before main() ⇔ PREFERRED**
   The function is defined completely before writing the main function.

   Example
```c
#include <stdio.h>

double getVolume(double diameter, double height){
  double radius = diameter / 2;
  return 3.141592654 * radius * radius * height;
}

int main(){
  double d = 3.5, h = 7.8, vol = 0;
  printf("Volume of cone = %.4f", getVolume(d, h));

  return 0;
}
```

   Output
```
Volume of cone = 75.0448
```

2. **After main()**
   Function prototype, main function, and complete function definition.

   Example
```c
#include <stdio.h>

double getVolume(double diameter, double height);

int main(){
  double d = 3.5, h = 7.8, vol = 0;
  printf("Volume of cone = %.4f", getVolume(d, h));

  return 0;
}

double getVolume(double diameter, double height){
  double radius = diameter / 2;
  return 3.141592654 * radius * radius * height;
}
```

   Output
```
Volume of cone = 75.0448
```

BENEFITS OF USING FUNCTIONS

1. Smaller source files: Using a function provides a way of representing several lines of code with just a line, thus reducing the size of the source file.

2. Ease of maintenance: Making changes to how the task is performed becomes easier because it gets done in just one spot (the function implementation) instead of several.

RECURSION

When a function gets called with a base argument, it returns a single value. If it gets called with any other argument, it takes out the base argument and calls itself with what is left of the now modified argument. This process in which a function calls itself is termed recursion. It stops making further calls when what is left off the modified argument is the base argument.

Examples 1: Computing Factorial

0! = 1
1! = 1
2! = 2 x 1 = 2 x 1!
3! = 3 x 2 x 1 = 3 x 2!
4! = 4 x 3 x 2 x 1 = 4 x 3!
5! = 5 x 4 x 3 x 2 x 1 = 5 x 4!

From above, the general expression for the factorial of numbers from 2 and above is: **n! = n x (n – 1)!**
The concept of recursion is represented by (n – 1)!

A program with the recursive function for factorial is given below:

```c
#include <stdio.h>

// FUNCTION TO COMPUTE n!
long long factorial(int n){
  if(n == 0 || n == 1)
    return 1;
  else
    return n * factorial(n - 1); // WHERE RECURSION OCCURS
}

// USING THE FUNCTION TO DISPLAY FACTORIALS OF 0 - 6
int main(){
  for(int i = 0; i <= 6; i++)
    printf("%d! = %d\n", i, factorial(i));

  return 0;
}
```

Output

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
```

NOTE

(long long) was chosen as return type because its size of 8 bytes can represent a wider range of positive integers compared to the 4 bytes of int or long.

Example 2: Fibonacci Series

The series starts with 0, as first value, and 1 (base value) as second value. Every other value in the series is obtained by summing the two values in front. This fact is applicable to all terms of the series, starting from the third. The series up to the sixth term is as given below:

F0 = 0
F1 = 1
F2 = 1 + 0 = 1
F3 = 1 + 1 = 2
F4 = 1 + 2 = 3
F5 = 2 + 3 = 5

The generalized expression for all terms from the third is given as: Fn = F(n-2) + F(n-1)
Where F(n – 2) and F(n – 1) represent recursive calls.

A program with the recursive function for Fibonacci series is presented below:

```c
#include <stdio.h>

// FUNCTION TO COMPUTE THE nth TERM OF THE FIBONACCI SERIES
int fibo(int n){
  if(n == 0 || n == 1)
    return n;
  else
    return fibo(n - 2) + fibo(n - 1); // 2 recursive calls running separately
}

// USING THE FUNCTION TO DISPLAY FACTORIALS OF 0 - 6
int main(){
  for(int i = 0; i <= 6; i++)
    printf("F(%d) = %d\n", i, fibo(i));

  return 0;
}
```

Output

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
```

USING A CUSTOM HEADER

To create one, save function definitions in a file saved with a valid identifier of your choice as name, and the extension h i.e. (.h). In order to use this header file

1. Ensure the header file is the same directory as the source file(s) of the program(s)
2. Use the syntax below to make it a part of the program
   #include "name.h"

An illustrative example is given below

Custom Header file (myHeader.h)

```c
double getAreaOfCircle(double d){
  return 3.141592654 * d * d / 4;
}

double getAreaOfTriangle(double b, double h){
  return 0.5*b*h;
}

double getAreaOfSquare(double s){
  return s * s;
}
```

Program using myHeader.h

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

int main(){
  double diameter = 4.8, base = 10.4, height = 5.6, side = 7.2;
  printf("Area of circle = %f\n", getAreaOfCircle(diameter));
  printf("Area of rectangle = %f\n", getAreaOfTriangle(base, height));
  printf("Area of square = %f\n", getAreaOfSquare(side));

  return 0;
```

Output

```
Area of circle = 18.095574
Area of rectangle = 29.120000
Area of square = 51.840000
```

NOTE: Leaving out the customer header will result in syntax error during compilation of the program.

EXERCISES

3. Define a function called hypotenuse that calculates the length of the hypotenuse of a right triangle when the other two sides are given. The function should take two arguments of type double and return the hypotenuse as a double. Test your program with the side values specified below:

Triangle	Side 1	Side 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

2. Write a function that displays a solid rectangle of asterisks whose sides are specified in the integer parameters side1 and side2. For example, if the sides are 4 and 5, the function displays the following:

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

3. Modify the function created in Exercise 3 to form the rectangle out of whatever character is contained in character parameter fillCharacter. Thus, if the sides are 5 and 4, and fillCharacter is "@", then the function should print the following:

```
@@@@
@@@@
@@@@
@@@@
@@@@
```

4. Write and test a recursive function that returns the sum of all numbers from 1 to the integer provided as parameter.

RANDOM NUMBER GENERATION

Header: **stdlib.h**

Function

rand(): generates a random integer that is constant between multiple program runs.

Example

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

```c
int main(){
  for(int i = 0; i < 5; i++)
    printf("%d\n", rand());
  return 0;
}
```

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
41	41	41	41	41	41
18467	18467	18467	18467	18467	18467
6334	6334	6334	6334	6334	6334
26500	26500	26500	26500	26500	26500
19169	19169	19169	19169	19169	19169

Note: Random integer generated is between 0 and, at least 32767 (defined as RAND_MAX in stdlib.h).

AVOIDING REPETITION

This requires providing an integer called a seed as argument to the function **srand()** also defined in stdlib.h. The sequence of random numbers generated are dependent on the seed.

Example

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

int main(){
  int seed;
  printf("Enter seed: ");
  scanf("%d", &seed);

  // Use the entered integer as seed
  srand(seed);

  // Generate 5 random numbers
  for(int i = 0; i < 5; i++)
    printf("%d\n", rand());
  return 0;
}
```

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
Enter seed: 1	Enter seed: 2	Enter seed: 3	Enter seed: 1	Enter seed: 2	Enter seed: 3
41	45	48	41	45	48
18467	29216	7196	18467	29216	7196
6334	24198	9294	6334	24198	9294
26500	17795	9091	26500	17795	9091
19169	29484	7031	19169	29484	7031

Using the system time (time(NULL) defined in time.h) as seed, will help us rid the program of the above repetition observed with the seeds provided. I.e. **srand(time(NULL));**

Example

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

int main(){
  // Use system time as seed
  srand(time(NULL));

  // Generate 5 random numbers
```

```
  for(int i = 0; i < 5; i++)
    printf("%d\n", rand());
  return 0;
}
```

Run 1	Run 2	Run 3	Run 4	Run 5	Run 6
32251	32323	32421	32479	32564	32639
5552	12641	7413	4277	21591	6661
17317	17113	28750	22625	28342	13234
12919	18023	19027	26183	29237	25637
13345	10890	25415	14470	20505	28364

ADJUSTING RANGE

To generate random numbers from min to max, the formula to use is:

- ⇨ **min + rand() % (1 + max – min)**
- ⇨ **min + rand() % (range + 1)**, where **range = max – min**

For example in the throwing of a die where any number from 1 to 6 is expected:

- ⇨ range = 6 – 1 = 5
- ⇨ result = 1 + random() % (5 + 1)
- ⇨ result = 1 + random() % 6

Example

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

int main(){
  // Use system time as seed
  srand(time(NULL));

  int a = 1, max = 6;

  // Generate the sequence for rolling a die 50 times
  for(int i = 0; i < 50; i++)
    printf("%d ", min + rand()%max);
  return 0;
}
```

Run 1

```
2 1 1 1 4 4 2 4 1 4 1 1 1 4 5 2 4 2 4 1 3 2 3 4 3 3 1 6 4 2 4 2 5 2 6 3 2 3 3 1 4 3 2 3 1 6 2 3 1 2
```

Run 2

```
5 4 4 6 4 6 6 4 4 4 2 6 5 1 1 1 1 1 3 5 4 1 5 3 6 5 3 4 5 3 2 3 4 3 4 1 3 4 2 4 4 2 3 2 2 5 1 1 1 1
```

Run 3

```
1 4 4 1 5 3 1 3 2 4 4 4 3 3 4 4 5 2 3 5 2 6 4 6 6 2 4 3 2 2 6 1 6 3 2 5 3 4 5 1 6 6 4 5 6 5 1 2 3 1
```

ARRAYS

It is a finite (definite size which can't be changed afterwards) cluster of variables of the same data type with a shared name. Some important points to note include:

i. Each variable in the cluster is called an ELEMENT.
ii. The number of variables in the cluster is called the SIZE, LENGTH, or DIMENSION of the array.
iii. The name of an element is made up of the shared name and a unique integer called its INDEX.
iv. Index values start from 0 (for the first element) to (SIZE – 1) for the last element.

ONE-DIMENSIONAL/SINGLY SUBSCRIPTED ARRAYS

Visually, this is similar to a column of cells.

CREATING

dataType arrayName[SIZE]; ⇔ size is provided explicitly

Examples

```
double cgpas[60];
char objAns[50];
int scores[150];
```

The size doesn't always have to be numbers/literals. They are better represented by a constant value defined using one of the formats below:

1. **#define NAME value** ⇔ among the pre-processor directives (preferred) i.e. outside a function (in this case the main-function and th)

2. **const type NAME = value;** ⇔ within a function (in this case the main-function)

REFERENCE/NAME FOR ELEMENT

This is needed to either put value in an element (write to) or get data from the element (read from).

Syntax

arrayName[indexValue]

Examples

1. scanf("%d", &score[0]); ⇔ will write an integer from the keyboard into the first element
2. printf("%d", cgpa[1]); ⇔ will read the value of the second element and display it on screen

;

INITIALIZING AN ARRAY

Must be done during its definition. Cannot be done after.

Syntax

❖ dataType arrayName[SIZE] = {data, data, …, data};

Facts to note include:

1. listed data can be less than the size of the array
2. in initializing, the first data goes to the first element, second to the second element, and so on
3. if listed data is less than size, elements for which there are no data to assign get assigned 0 or its equivalent depending on the data type

❖ dataType arrayName[] = {data, data, …, data}; ⇔ **size = number of data listed.**

BATCH PROCESSING

This requires one loop that provides the index values of the array, thus providing the opportunity to process each element (per iteration) in the body of the loop.

Example

1. Write a program that reads 10 values into an array, and afterwards displays read values.

Program

```c
#include <stdio.h>
#define SIZE 6

int main(){
  printf("Enter %d numbers\n", SIZE);

  // Create array for 10 values
  double numbers[SIZE];

  // Write read data into the array
  for(int index = 0; index < SIZE; index++)
    scanf("%lf", &numbers[index]);

  // Display numbers read
  printf("\nNumbers read are: ");
  for(int index = 0; index < SIZE; index++)
```

```
    printf("%f ", numbers[index]);

  return 0;
}
```

Output

```
Enter 6 numbers
1.3
-0.345
1.4e-7
4
2.56
7.452

Numbers read are: 1.300000 -0.345000 0.000000 4.000000 2.560000 7.452000
```

2. Write a program that reads 10 integers, and then displays the separately the even and odd integers read.

Program

```
#include <stdio.h>
#define SIZE 10 //Size of the array

int main(){
  // Define array
  int numbers[SIZE] = {0};

  // Prompt for numbers
  printf("Enter %d integers\n", SIZE);

  // Read the numbers into the array, one after the other
  for(int index = 0; index < SIZE; index++){
    scanf("%d", &numbers[index]); // Read and save number in current element
  }

  // Display even numbers read
  printf("\nEven numbers read:\n");
  for(int index = 0; index < SIZE; index++){
    if(numbers[index]%2 == 0){
      printf("%d\t", numbers[index]);
    }
  }

  // Display odd numbers read
  printf("\n\nOdd numbers read:\n");
  for(int index = 0; index < SIZE; index++){
    if(numbers[index]%2 != 0){
      printf("%d\t", numbers[index]);
    }
  }

  return 0;
}
```

Sample Output

```
Enter 10 integers
1 2 3 4 11 0 -3 -4 7 6
```

```
Even numbers read:
2       4       0       -4      6

Odd numbers read:
1       3       11      -3      7
```

MULTIDIMENSIONAL ARRAY

Arrays treated before now have elements that contain value/literals. Multidimensional arrays are array that have arrays embedded in their elements. Depending on the extent to which this is done, multidimensional arrays can be 2, 3, …, n dimensions.

Syntax for defining

- ⇨ type name[SIZE_1][SIZE_2]; // 2 dimensions
- ⇨ type name[SIZE_1][SIZE_2][SIZE_3]; // 3 dimensions

Referencing an element requires providing next to the name, index values matching the dimension of the array

- ⇨ name[index_1][index_2] // element in a 2-dimensional array
- ⇨ name[index_1][index_2][index_3] // element in a 3-dimensional array

Batch-processing of elements requires using nested loops. The number of loops nested matches the dimension of the array i.e. 2 loops for an array of 2 dimensions and 3 for an array of 3 dimensions.

Example 1

The program below populates the defined 2-dimensional array with random integer values from 10 to 99, and displays the content in a properly formatted manner.

Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define CLASS_SIZE 9
#define COURSE_COUNT 3

int main(){
  // Define 2-dimensional array
  int scores[CLASS_SIZE][COURSE_COUNT] = {0};

  // Populate array with random scores between 10 and 99
  srand(time(NULL));
  for(int i = 0; i < CLASS_SIZE; i++){
    for(int j = 0; j < COURSE_COUNT; j++){
      scores[i][j] = 10 + rand() % (1 + 99 - 10);
    }
  }
  // Display content array in tabular form
  printf("%-10s%-10s%-10s\n", "Course 1", "Course 2", "Course 3");
  for(int i = 0; i < CLASS_SIZE; i++){
    for(int j = 0; j < COURSE_COUNT; j++){
      printf("%-10d", scores[i][j]);
    }
    printf("\n");
  }

  return 0;
}
```

Sample Output 1

Course 1	Course 2	Course 3
35	88	60
54	13	65
86	58	96
75	10	60
18	35	60
93	42	76
16	98	34
18	95	11
55	28	42

Sample Output 2

Course 1	Course 2	Course 3
23	34	55
62	75	64
33	69	99
40	33	30
45	74	59
26	92	17
63	49	15
66	49	86
60	13	99

Example 2

The program below repeats the process above but, this time with a 3-dimensional array.

Program

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define DEPT_COUNT 3
#define CLASS_SIZE 8
#define COURSE_COUNT 6

int main(){
  // Define array of
  int scores[DEPT_COUNT][CLASS_SIZE][COURSE_COUNT] = {0};

  // Populate array with random scores between 10 and 99
  srand(time(NULL));
  for(int i = 0; i < DEPT_COUNT; i++){
    for(int j = 0; j < CLASS_SIZE; j++){
      for(int k = 0; k < COURSE_COUNT; k++){
        scores[i][j][k] = 10 + rand() % (1 + 99 - 10);
      }
    }
  }
  // Display content of the array in multiple tabular forms
  for(int i = 0; i < DEPT_COUNT; i++){
    printf("Department %d\n", i + 1);

    // Print table head
    for(int c = 0; c < COURSE_COUNT; c++){
      printf("Course %-3d", c + 1);
    }
    printf("\n");
    //printf("%-10s%-10s%-10s%-10s\n", "Course 1", "Course 2", "Course 3", "Course 4");


    for(int j = 0; j < CLASS_SIZE; j++){
      for(int k = 0; k < COURSE_COUNT; k++){
        printf("%-10d", scores[i][j][k]);
      }
      printf("\n");
    }
    printf("\n");
```

```
  }

  return 0;
}
```

Sample output 1

Department 1

Course 1	Course 2	Course 3	Course 4
95	53	87	87
53	53	28	72
46	40	66	51
72	36	63	49
58	45	44	36
99	74	98	46
85	12	22	17
48	73	32	80

Department 2

Course 1	Course 2	Course 3	Course 4
10	85	21	12
75	52	72	58
86	36	21	69
91	69	86	34
85	12	90	81
66	99	95	89
57	89	19	75
60	13	14	34

Department 3

Course 1	Course 2	Course 3	Course 4
43	22	54	94
59	43	80	91
69	61	40	22
48	45	62	11
40	50	93	64
31	86	99	84
41	11	87	74
34	17	58	79

Sample output 2

Department 1

Course 1	Course 2	Course 3	Course 4
92	16	19	80
90	42	35	32
68	18	88	63
72	19	22	58
78	67	93	67
28	19	88	89
61	46	89	65
91	30	84	99

Department 2

Course 1	Course 2	Course 3	Course 4
65	20	28	73
36	59	65	20
47	21	80	46
13	28	42	53
89	80	47	77
59	94	65	17
53	87	62	72
80	13	29	16

Department 3

Course 1	Course 2	Course 3	Course 4
23	30	84	52
85	60	20	73
99	18	30	92
16	17	77	33
90	67	97	73
63	24	81	82
11	51	30	30
75	17	70	60

ARRAYS IN FUNCTIONS

In order to use an array as a parameter in a function, the array and its size in bytes are to be present. Other parameters can be added, based on their relevance in the function.

For varying array sizes and truly generic functions, the heads below will do:

- ⇨ One-dimensional array: returnType functionName(…int r, type name[]…) where r represents the number of rows if the array is seen as a single column
- ⇨ Two-dimensional array: returnType functionName(…int r, int c, type name[][c]…) where r is row count and c is column count
- ⇨ Three-dimensional array: returnType functionName(…int x, int y, int z, type name[][y][z]…)

Important points to note before delving into implementation include:

1. sizeof(…) returns the size of its argument. The argument in this case can be an array, an embedded array, or an element.

2. The values to use as arguments for r, c, x, y, and z can be calculated from the calling function using the generic formula: **sizeof(array)/sizeof(firstElement)**
   This implies that from the calling function/environment:
   ⇨ sizeof(array)/sizeof(array[0]) gives the value of r or x (as used in the function heads above)

   ⇨ sizeof(array[0])/sizeof(array[0][0]) gives the value of c or y

   ⇨ sizeof(array[0][0])/sizeof(array[0][0][0]) will give the value of z

NOTE

   ⇨ **sizeof(arr[0]) = sizeof(dataType) for an array of 1 dimension**

   ⇨ **sizeof(array[0])/sizeof(array[0][0]) for an array of 2 dimensions**

   ⇨ **sizeof(array[0][0])/sizeof(array[0][0][0]) for an array of 3 dimensions**

The examples below show the use of a single function to display arrays (1 to 3 dimensions) of varying sizes.

Example 1

```c
#include <stdio.h>

void display(int r, int arr[]){
  for(int i = 0; i < r; i++)
    printf("%-4d", arr[i]);
  printf("\n");
}

int main(){
  printf("Content of array 1\n");
  int array1[8] = {9};
  display(sizeof(array1)/sizeof(array1[0]), array1);

  printf("\nContent of array 2\n");
  int array2[10] = {3,4,6};
  display(sizeof(array2)/sizeof(array1[0]), array2);

  return 0;
}
```

Output

```
Content of array 1
9   0   0   0   0   0   0   0

Content of array 2
3   4   6   0   0   0   0   0   0   0
```

Example 2

```c
#include <stdio.h>

void show(int r, int c, int arr[][c]){
  for(int i = 0; i < r; i++){
    //Show columns in current row
    for(int j = 0; j < c; j++){
      printf("%-4d", arr[i][j]);
    }
    printf("\n");
  }
}
```

```c
int main(){
  printf("Content of array 1\n");
  int arr1[3][9] = {9};
  show(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), arr1);

  printf("\nContent of array 2\n");
  int arr2[5][7] = {3,4,6};
  show(sizeof(arr2)/sizeof(arr2[0]), sizeof(arr2[0])/sizeof(arr2[0][0]), arr2);

  return 0;
}
```

Output

```
Content of array 1
9   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0

Content of array 2
3   4   6   0   0   0   0
0   0   0   0   0   0   0
0   0   0   0   0   0   0
0   0   0   0   0   0   0
0   0   0   0   0   0   0
```

Example 3

```c
#include <stdio.h>

void show(int x, int y, int z, int arr[][y][z]){
  for(int i = 0; i < x; i++){
    printf("x = %d\n", i);
    // Show table for current value of x
    for(int j = 0; j < y; j++){
      for(int k = 0; k < z; k++){
        printf("%-4d", arr[i][j][k]);
      }
      printf("\n");
    }
    printf("\n");
  }
}

int main(){
  printf("Content of array 1\n");
  int arr1[3][3][6] = {9};
  show(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), sizeof(arr1[0][0])/sizeof(int),
arr1);

  printf("\nContent of array 2\n");
  int arr2[4][2][5] = {3,4,6};
  show(sizeof(arr2)/sizeof(arr2[0]), sizeof(arr2[0])/sizeof(arr2[0][0]), sizeof(arr2[0][0])/sizeof(int),
arr2);

  return 0;
}
```

Output

```
tent of array 1
x = 0
9   0   0   0   0   0
0   0   0   0   0   0
0   0   0   0   0   0

x = 1
0   0   0   0   0   0
0   0   0   0   0   0
0   0   0   0   0   0

x = 2
0   0   0   0   0   0
0   0   0   0   0   0
0   0   0   0   0   0


Content of array 2
x = 0
3   4   6   0   0
0   0   0   0   0

x = 1
0   0   0   0   0
0   0   0   0   0

x = 2
0   0   0   0   0
0   0   0   0   0

x = 3
0   0   0   0   0
0   0   0   0   0
```

ARRAY OPERATIONS (ONE DIMENSION)

Algorithms will be developed to show some of the operations that can be performed with arrays. The term **UPPER BOUND (UB)** will refer to the element with the highest index value, while the term **LOWER BOUND (LB)** will refer to the element with the lowest index value.

INSERTION

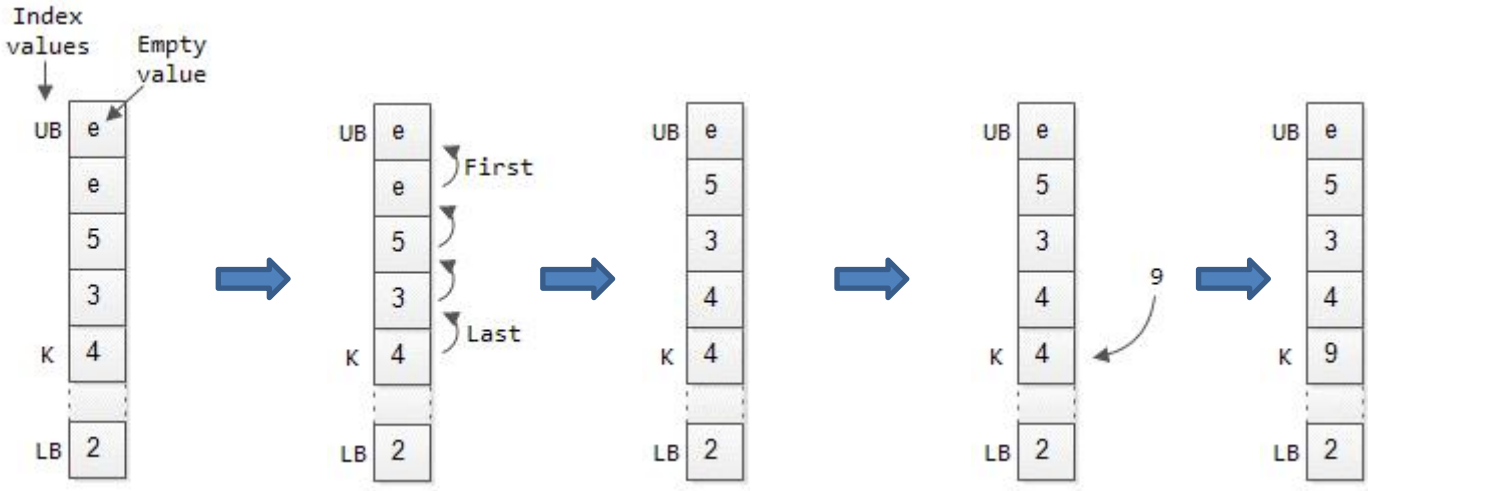
The main purpose for this operation is to have data d, inserted into index k of the array arr. For the insertion to happen, there must be room in arr for d. Questions to note include:

1. How can an empty element be identified?
2. How do we know that the array is not full (i.e. there is room for d), because loss of data is not permitted?

The following assumptions are made to answer the questions above and set the stage for development of the algorithm.

1. An element will be considered empty if it contains a predefined value (the empty value)
2. It will be considered that there is room for d, if the upper bound is empty.
3. If the array contains non-empty element(s), these elements in arrangement, will be contiguous starting from the lower bound, while the empty elements will be the same, but from the upper bound.

A diagrammatical presentation of an insertion taking place is shown below:



Initial State | Shifting | After shifting | Insertion | After insertion

OBSERVATIONS

During shifting

- ⇨ Value in an element gets copied into the one immediately above it
- ⇨ Two series of index values can be obtained
  1. From elements receiving copied data i.e. UB down to (K+1)
  2. From elements providing copied data i.e. K up to (UB-1)
- ⇨ A loop is needed to provide the values in the series, with the copying taking place in the body of the loop

During insertion

- ⇨ Assignment of the incoming data to index K completes it

ALGORITHM

```
If upper bound != empty value
  Return 0 to indicate insertion failure
Else
  Set i to UB
  While i >= (K+1)
    Assign element (i – 1) to the element i
    Decrement i (to get next index in the series)
  Assign data (9) to element k
  Return 1 to indicate insertion success
```

IMPLEMENTATION AND TESTING

```c
#include <stdio.h>
#define EMPTY -1

// Function to insert d into index k of arr
int inserted(int r, int arr[], int k, int d){
  // Get value of ub from r (length or dimension)
  int ub = r - 1;
  if(arr[ub] != EMPTY)
    return 0;
  else{
    // Commence shifting
    for(int i = ub; i >= k+1; i--)
      arr[i] = arr[i - 1];
    // Insert data
    arr[k] = d;
    return 1;
  }
```

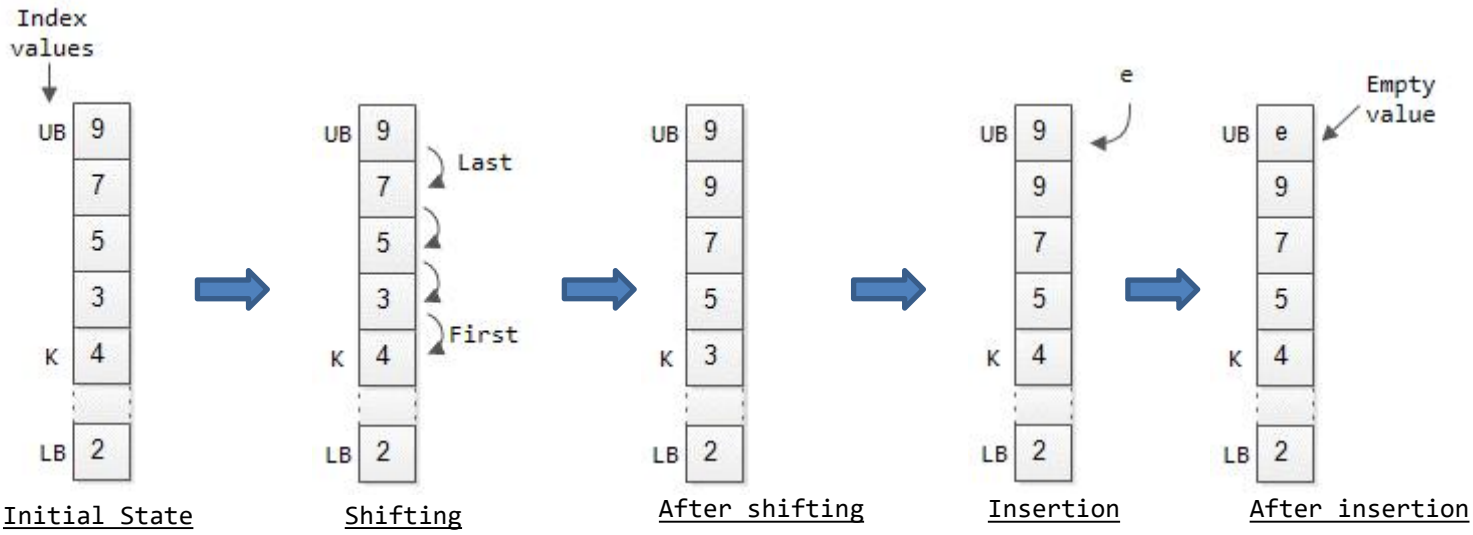
```c
}

// Function to display content of array from UB to 0
void show(int r, int arr[]){
  for(int i = r - 1; i >= 0; i--)
    printf("Index %d = %2d\n", i, arr[i]);
  printf("\n");
}

// Function to display status of operation
void status(int flag){
  printf("Insertion %s\n", flag?"successful":"failed");
}

int main(){
  int arr[5] = {5,6,4,7,-1};
  int dim = sizeof(arr)/sizeof(int); // dimension of array

  printf("Initially\n");
  show(dim, arr);

  // Insert 9 into index 1
  status(inserted(dim, arr, 1, 9));
  show(dim, arr);

  // Insert 15 into index 2
  status(inserted(dim, arr, 2, 15));
  show(dim, arr);

  return 0;
}
```

RESULT/OUTPUT

```
Initially
Index 4 = -1
Index 3 =  7
Index 2 =  4
Index 1 =  6
Index 0 =  5

Insertion successful
Index 4 =  7
Index 3 =  4
Index 2 =  6
Index 1 =  9
Index 0 =  5

Insertion failed
Index 4 =  7
Index 3 =  4
Index 2 =  6
Index 1 =  9
Index 0 =  5
```

DELETION

This is about deleting the data in specified index (k) of an array (arr). The steps are are given below:



Facts to note:

- ⇨ The shifting like before will be done in a loop. The series of index values expected from the loop are either:
  - ➢ K up to (UB – 1) when based on elements receiving data in the shift operation, or
  - ➢ (K + 1) up to UB when based on elements releasing data in the shift operation.
- ⇨ Insertion as usual is an assignment operation

ALGORITHM

```
Set i = K
While i <= (UB – 1)
  Set Array[i] = Array[i + 1]
  Increment i
Set Array[UB] =  EmptyValue
```

Note: The algorithm can be adjusted to display the content of the array after each delete operation.

IMPLEMENTATION AND TESTING

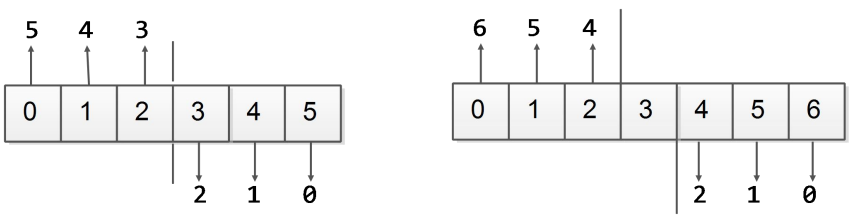
```c
#include <stdio.h>
#define EMPTY -1

// Function to display content of array from UB to 0
void show(int r, int arr[]){
  printf("\n");
  for(int i = r - 1; i >= 0; i--)
    printf("Index %d = %2d\n", i, arr[i]);
  printf("\n");
}

// Function to delete data from index k of arr
void delete(int r, int arr[], int k){
  // Calculate ub
  int ub = r - 1;

  // Shifting operation
  for(int i = k; i <= ub - 1; i++)
    arr[i] = arr[i + 1];
```

```
  // Insertion
  arr[ub] = EMPTY;

  printf("After deleting data in index %d", k);
  show(r, arr);
}

int main(){
  int arr[6] = {5,6,4,7,11,9};
  int dim = sizeof(arr)/sizeof(int);

  printf("Initial State");
  show(dim, arr);

  // Delete 6 at index 1
  delete(dim, arr, 1);

  // Delete 5 at index 0
  delete(dim, arr, 0);

  return 0;
}
```

RESULT/OUTPUT

```
Initial State
Index 5 =  9
Index 4 = 11
Index 3 =  7
Index 2 =  4
Index 1 =  6
Index 0 =  5

After deleting data in index 1
Index 5 = -1
Index 4 =  9
Index 3 = 11
Index 2 =  7
Index 1 =  4
Index 0 =  5

After deleting data in index 0
Index 5 = -1
Index 4 = -1
Index 3 =  9
Index 2 = 11
Index 1 =  7
Index 0 =  4
```

REVERSAL



Observation

⇨ If index L is to be swapped index U, then L + U = UB (Upper bound). Applicable arrays or even and odd length.
⇨ Index of the lower elements run from 0 to (dimension/2) – 1.
⇨ Index of the upper elements run from (dimension/2 + 1) to ub.
⇨ A loop is needed to get index values from either the upper or lower halve of the array, with the swap taking place in the body.

Algorithm

```
Set i = 0
While i <= dim/2 – 1
  Set temp = array[i]
  Set array[i] = array[ub – i]
  Set array[ub – i] = temp
  Increment i

OR

Set i = dim/2 + 1
While i <= ub
  Set temp = array[i]
  Set array[i] = array[ub – i]
  Set array[ub – i] = temp
  Increment i
```

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int arr[]){
  for(int i = 0; i < r; i++)
    printf("%-3d", arr[i]);
  printf("\n");
}

void reverse(int r, int arr[]){
  int ub = r - 1; // to determine upper bound
  for(int i = 0; i <= r/2; i++){
    int temp = arr[i];
    arr[i] = arr[ub - i];
    arr[ub - i] = temp;
  }
}

int main(){
  int arr[9] = {1,2,3,4,5,6,7,8,9};
  int dim = sizeof(arr)/sizeof(int);
  printf("BEFORE REVERSAL\n");
  show(dim, arr);

  reverse(dim, arr);

  printf("\nAFTER REVERSAL\n");
  show(dim, arr);

  return 0;
}
```

RESULT

```
BEFORE REVERSAL
1  2  3  4  5  6  7  8  9

AFTER REVERSAL
9  8  7  6  5  4  3  2  1
```

SEARCH (LINEAR)

Search for d in arr, starting from the lower bound. Return the index of the first occurrence of d or -1 if d is not in arr.

ALGORITHM

```
Set i = 0 (LB)
While i < dimension
  If arr[i] == d
    Return i
  Increment i
Return -1
```

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int arr[]){
  for(int i = 0; i < r; i++)
    printf("%-3d", arr[i]);
  printf("\n");
}

int search(int r, int arr[], int d){
  for(int i = 0; i < r; i++){
    if(arr[i] == d)
      return i;
  }
  return -1; // d is not in arr
}

int main(){
  int arr[] = {1,2,7,3,9,0,4,5,6,7,8,9};
  int dim = sizeof(arr)/sizeof(int);

  int v = 7;
  printf("Index of %d in\n ", v);
  show(dim, arr);
  printf("is %d\n", search(dim, arr, v));

  v = 10;
  printf("\nIndex of %d in\n ", v);
  show(dim, arr);
  printf("is %d\n", search(dim, arr, v));
  return 0;
}
```

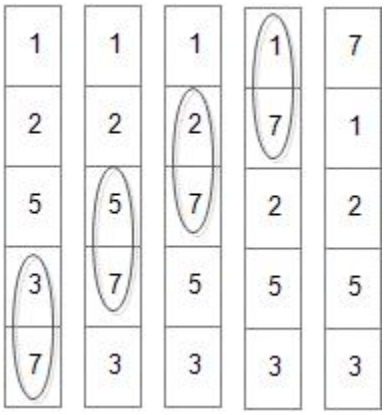
RESULT/OUTPUT

```
Index of 7 in
 1  2  7  3  9  0  4  5  6  7  8  9
is 2
```

Index of 10 in
1 2 7 3 9 0 4 5 6 7 8 9
is -1

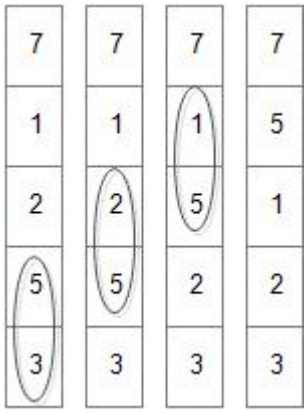
SORTING

Rearranging the values in an array, either from smallest to the largest (ascending order), or from the largest to the smallest (descending order).



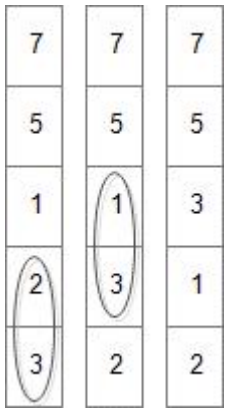
CYCLE 1
Compared index:
0 to 3 (Bottom-top)
1 to 4 (Top-bottom)

Total compared: 5



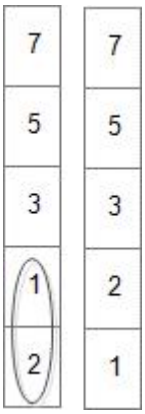
CYCLE 2
Compared index:
0 to 2 (Bottom-top)
1 to 3 (Top-bottom)

Total compared: 4



CYCLE 2
Compared index:
0 to 1 (Bottom-top)
1 to 2 (Top-bottom)

Total compared: 3



CYCLE 2
Compared index:
0 (Bottom-top)
1 (Top-bottom)

Total compared: 2

OBSERVATIONS

1. Number of compared elements, n starts from the whole array (dimension) down to 2
2. In each cycle, the lower index values picked for comparison run from 0 up to (n-2), while the upper index values run from 1 up to (n – 1)
3. One loop will be needed for the series in (1), while an inner loop will be needed for either of the series in (2)

ALGORITHM (ASCENDING ORDER)

```
Set n = dimension
While n >= 2
  Set i = 0
  While i <= (n-2)
    If array[i] > array[i+1] // lower greater than upper
      Set temp = array[i]
      Set array[i] = array[i+1]
      Set array[i+1] = temp
    Increment i
  Decrement n
```

Or

```
Set n = dimension
While n >= 2
  Set i = 1
  While i <= (n-1)
    If array[i] < array[i-1] // upper lesser than lower
      Set temp = array[i]
      Set array[i] = array[i-1]
      Set array[i-1] = temp
    Increment i
```

Decrement n

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int arr[]){
  for(int i = 0; i < r; i++)
    printf("%-3d", arr[i]);
  printf("\n");
}

void sort(int r, int arr[]){
  for(int n = r; n >= 2; n--){
    for(int i = 0; i < (n-2); i++){
      if(arr[i] > arr[i+1]){
        int temp = arr[i];
        arr[i] = arr[i+1];
        arr[i+1] = temp;
      }
    }
  }
}

int main(){
  int arr[] = {1,2,7,3,9,0,4,5,6,7,8,9};
  int dim = sizeof(arr)/sizeof(int);
  printf("BEFORE SORTING\n");
  show(dim, arr);

  sort(dim, arr);

  printf("\nAFTER SORTING\n");
  show(dim, arr);

  return 0;
}
```

OUTPUT/RESULT

```
BEFORE SORTING
1  2  7  3  9  0  4  5  6  7  8  9

AFTER SORTING
0  1  2  3  4  5  6  7  7  8  9  9
```

ARRAY OPERATIONS (TWO DIMENSIONS)

These operations are primarily matrix operations, where a matrix is represented by a two-dimensional array with a specific row and column count.

ADDITION & SUBTRACTION

To perform addition/subtraction with matrix A and B, they must have the same number of rows and columns. The resulting matrix C, will have the same dimension as well.

ALGORITHM

```
For each value of i from 0 up to (r – 1)
  For each value of j from 0 up to (c – 1)
    Set c[i][j] = a[i][j] ± b[i][j]
```

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int c, int arr[][c]){
  for(int i = 0; i < r; i++){
    for(int j = 0; j < c; j++)
      printf("%-4d", arr[i][j]);
    printf("\n\n");
  }
}

void add(int r, int c, int matA[][c], int matB[][c], int matC[][c]){
  for(int i = 0; i < r; i++){
    for(int j = 0; j < c; j++){
      matC[i][j] = matA[i][j] + matB[i][j];
    }
  }
}

int main(){
  int arr1[2][4] = {{1,2,7,3},{9,0,4,5}};

  int arr2[2][4] = {{1,3,1,5},{7,3,9,2}};

  int arr3[2][4]= {0};

  printf("Array 1\n");
  show(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), arr1);

  printf("\nArray 2\n");
  show(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), arr2);

  add(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), arr1, arr2, arr3);

  printf("\nSummation Result\n");
  show(sizeof(arr1)/sizeof(arr1[0]), sizeof(arr1[0])/sizeof(arr1[0][0]), arr3);

  return 0;
}
```

RESULT/OUTPUT

```
Array 1
1   2   7   3

9   0   4   5


Array 2
1   3   1   5

7   3   9   2


Summation Result
2   5   8   8

16  3   13  7
```

TRANSPOSITION

The rows of a matrix become columns. This is as shown below

For example, if $A = \begin{matrix} 4 & 2 & 6 \\ 1 & 8 & 7 \end{matrix}$

Its transpose $A^T = \begin{matrix} 4 & 1 \\ 2 & 8 \\ 6 & 7 \end{matrix}$

Major thing to note is the exchange of row and column count. This means that, the of index values must be switched to match the elements of the transpose with the correct elements in the original array.

ALGORITHM

```
For each value of i from 0 to (matrixRows – 1)
  For each value of j from 0 to (matrixCols – 1)
    Set transpose[j][i] = matrix[i][j]
```

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int c, int arr[][c]){
  for(int i = 0; i < r; i++){
    for(int j = 0; j < c; j++)
      printf("%-4d", arr[i][j]);
    printf("\n\n");
  }
}

void transpose(int r, int c, int mat[][c], int tra[][r]){
  for(int i = 0; i < r; i++){
    for(int j = 0; j < c; j++){
      tra[j][i] = mat[i][j];
    }
  }
}

int main(){
  int arr[2][4] = {{1,2,7,3},{9,0,4,5}};

  int arrT[4][2] = {1};

  printf("Matrix\n");
  show(sizeof(arr)/sizeof(arr[0]), sizeof(arr[0])/sizeof(arr[0][0]), arr);

  transpose(sizeof(arr)/sizeof(arr[0]), sizeof(arr[0])/sizeof(arr[0][0]), arr, arrT);

  printf("\nTranspose\n");
  show(sizeof(arrT)/sizeof(arrT[0]), sizeof(arrT[0])/sizeof(arrT[0][0]), arrT);

  return 0;
}
```

RESULT/OUPUT

```
Matrix
1   2   7   3
9   0   4   5
```

Transpose
1 9
2 0
7 4
3 5

MULTIPLICATION

Matrix A can only multiply matrix B if columns in A equal rows in B. The result, matrix C will have row count matching A and column count matching B. That is:
[m by p] x [p by n] = [m by n]

During the multiplication
1. Values from a row in A (A[i][k]), multiply corresponding values in each column of B (B[k][j])
2. Sum of the products gives the value in C with row index coming from A, and column index coming from B i.e. C[i][j]

The example below illustrates it all:

A = $\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$ and B = $\begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{bmatrix}$

A x B = $\begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} \end{bmatrix}$

ALGORITHM

```
For values of i from 0 to (m - 1)
  For values of j from 0 to (n - 1)
    Set C[i][j] = 0
    For values of k from 0 to (p - 1)
      Set C[i][j] = C[i][j] + (A[i][k] * B[k][j])
```

IMPLEMENTATION & TESTING

```c
#include <stdio.h>

void show(int r, int c, int arr[][c]){
  for(int i = 0; i < r; i++){
    for(int j = 0; j < c; j++)
      printf("%-5d", arr[i][j]);
    printf("\n");
  }
}

void multiply(int m, int n, int p, int matA[][p], int matB[][n], int matC[][n]){
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      matC[i][j] = 0;
      for(int k = 0; k < p; k++){
        matC[i][j] += matA[i][k] * matB[k][j];
      }
    }
  }
}

int main(){
  int matA[3][3] = {{2, 1, 3},
                    {3, 3, 1},
                    {2, 0, 4}};
```

```
  int matB[3][2] = {{1, 2},
                    {3, 4},
                    {2, 1}};

  int matC[3][2] = {0};

  printf("Matrix A\n");
  show(sizeof(matA)/sizeof(matA[0]), sizeof(matA[0])/sizeof(matA[0][0]), matA);
  printf("\nMatrix B\n");
  show(sizeof(matB)/sizeof(matB[0]), sizeof(matB[0])/sizeof(matB[0][0]), matB);

  multiply(
           sizeof(matC)/sizeof(matC[0]), // rows in C
           sizeof(matC[0])/sizeof(matC[0][0]), // cols in C
           sizeof(matB)/sizeof(matB[0]), // row count for B same as column count for C
           matA,
           matB,
           matC);

  printf("\nA x B\n");
  show(sizeof(matC)/sizeof(matC[0]), sizeof(matC[0])/sizeof(matC[0][0]), matC);

  return 0;
}
```

RESULT/OUTPUT

```
Matrix A
2    1    3
3    3    1
2    0    4

Matrix B
1    2
3    4
2    1

A x B
11   11
14   19
10   8
```

POINTERS

Data is stored in the memory of a computer. Each storage location in a computer's memory has a unique address. Address values are numeric and they run from 0 to a maximum value determined by the size of the memory (number of storage locations). Since address of storage locations in memory are numeric, it is possible to store and treat them like any other data.
Each memory location is a byte. A variable occupies several memory locations (bytes) depending on the type of data it is to hold.

A pointer is a variable which stores the memory address of another variable. This is normally the address of the first byte occupied by the variable. Since a variable contains a specific value while a pointer holds the address of a variable.

DECLARING A POINTER

Format for declaration:

type* name;

The advisable name to use is formed by combining the name of the variable and Ptr. This is to make it obvious that, the name refers to a pointer.

Example

```
int numb; //variable
int* numbPtr; // pointer to variable numb
```

INITIALIZING A POINTER

This is done by assigning the address of a variable to the pointer.

&variableName => variable address

Example

```
int numb;
int* numbPtr = &numb;
```

DISPLAYING POINTER VALUE

The main placeholder for displaying a pointer is **%p**, which displays the pointer values in hexadecimal format; but because a pointer is an integer value hence can be displayed with any of the placeholders for displaying whole numbers (%o, %d, %x, and %X). It is also important that the memory locations used for storing variables depend on the system (computer of phone) running the program, hence pointer values displayed may not be the same as those given in the example below:

Code

```c
#include <stdio.h>

int main(){
  int numb; //variable
  int* numbPtr = &numb; //pointer

  printf("Using %%p: %p\n", numbPtr);
  printf("Using %%x: %x\n", numbPtr);
  printf("Using %%X: %X\n", numbPtr);
  printf("Using %%o: %o\n", numbPtr);
  printf("Using %%d: %d\n", numbPtr);

  return 0;
}
```

Output

```
Using %p: 0029FF08
Using %x: 29ff08
Using %X: 29FF08
Using %o: 12377410
Using %d: 2752264
```

ACCESSING VARIABLE VALUE THROUGH ITS POINTER

This can be done by using the indirection operator (*) with the pointer. A variable refers to the value it contains.

- **&variable => pointer**
- ***pointer => variable**

Code

```c
#include <stdio.h>

int main(){
  double numb = 2.56; //variable
  int* numbPtr = &numb; //pointer

  printf("numb = %f\n", numb);
  printf("*numbPtr = %f\n", *numbPtr);

  return 0;
}
```

Output

```
numb = 2.560000
*numbPtr = 2.560000
```

The operators & and * are complement operators i.e. each nullifies the effect of the other.

For this reason

$$variable = *\&variable$$

In the same vein

$$variablePtr = \&*variablePtr = *\&variablePtr$$

The example below illustrates all of these.

Code

```c
#include <stdio.h>

int main(){
  double numb = 2.56; //variable
  int* numbPtr = &numb; //pointer

  printf("VARIABLES\n");
  printf("numb = %f\n", numb);
  printf("*&numb = %f\n", *&numb);

  printf("\nPOINTERS\n");
  printf("numbPtr = %p\n", numbPtr);
  printf("&*numbPtr = %p\n", &*numbPtr);
  printf("*&numbPtr = %p\n", *&numbPtr);

  return 0;
}
```

Output

```
VARIABLES
numb = 2.560000
*&numb = 2.560000

POINTERS
numbPtr = 0029FF08
&*numbPtr = 0029FF08
*&numbPtr = 0029FF08
```

The operator * cannot precede an ordinary variable, on account of which **&*variable** is invalid.

POINTER ASSIGNMENT & CASTING

The term **void** refers to a type, as we can tell when using it in functions; when there is no value to return from the function. It is possible to declare a pointer of type void.
A pointer of one type cannot be assigned to a pointer of another type. It normally results in a warning and in some cases error in program output. This implies that

```
int a = 10;
double *pointer = &a;
```

Line 2 is abnormal since we are assigning the address of an int variable to a double pointer.

The only exception to this rule, is a void pointer. It can be assigned to a pointer of any type, and in the same vein a pointer of any type can also be assigned to it. This implies that:

```
int a = 10;
void *pointer = &a;
```

In order to use a void pointer to indirectly access the value it is pointing to, its type (void) has to be treated like the data type of the value. This is done through type casting which is in the format **(type*)pointer**. An attempt to do this directly is illegal. **This implies that**

- **printf("%d", *aPtr); is illegal**
- **printf("%p", *(int*)aPtr);** will output 10.

Furthermore casted or not, the address contained is the same.

- **printf("%p", aPtr); == printf("%p", (int*)aPtr); = 0028FF1**

POINTERS AND ARRAYS

The following facts about pointers and arrays are important:

1. The address of an element in an array can be referenced by using &name[index]
2. The name of an array without the square brackets and an index value is a pointer to the address of the first element in the array. This implies that given the declaration **int array[10];** **array = &array[0]**
3. The size of an element in bytes is equal to size of the data type of the array, and can be obtained from the difference of the pointers of two successive elements.
   Size of element = &array[n] - &array[n-1] e.g. &array[1] - &array[0]

The program below is for illustration.

```c
#include <stdio.h>

int main()
{
  int i[6], n; float f[6]; double d[6];

  printf("Size of an integer: %dbytes\n", sizeof(int));
  printf("Size of a float:    %dbytes\n", sizeof(float));
  printf("Size of a double:   %dbytes\n\n", sizeof(double));

  /* Print the table heading */
  printf("\t\tInteger Array\t\tFloat Array\t\tDouble Array");
  printf("\n============================================================");

  /* Print the addresses of each array element. */
  for(n = 0; n < 10; n++)
    printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", n, &i[n], &f[n], &d[n]);

  printf("\n============================================================\n\n");

  printf("i = %ld\t&i[0] = %ld\n", i, &i[0]);
  printf("f = %ld\t&f[0] = %ld\n", f, &f[0]);
  printf("d = %ld\t&d[0] = %ld\n", d, &d[0]);

  return 0;
}
```

Output

```
Size of an integer: 4bytes
Size of a float:    4bytes
Size of a double:   8bytes

                Integer Array      Float Array      Double Array
============================================================
Element 0:      2686724         2686700         2686648
Element 1:      2686728         2686704         2686656
Element 2:      2686732         2686708         2686664
Element 3:      2686736         2686712         2686672
Element 4:      2686740         2686716         2686680
Element 5:      2686744         2686720         2686688
============================================================
```

```
i = 2686724     &i[0] = 2686724
f = 2686700     &f[0] = 2686700
d = 2686648     &d[0] = 2686648
```

Notice that the difference between successive addresses corresponds to size of the various data types given by **sizeof()**.
The portion of the output under the table confirms that the name of the array and the address of its first element are the same.

POINTER ARITHMETIC

If a pointer is incremented or decremented by a value, it is done by the size of the data type.

```
int i;
int *iPtr = &i;
printf("iPtr: %d\n", iPtr);
```

Output -> iPtr: 2686696

```
iPtr++; // increase by 4 bytes
printf("iPtr++: %ld\n", iPtr);
```

Output -> iPtr++: 2686700

```
iPtr += 2; // increase by 8 bytes (2 x 4)
printf("iPtr += 2: %ld\n", iPtr);
```

Output -> iPtr += 2: 2686708

```
iPtr -= 3; // decrease by 12 bytes
printf("iPtr -= 3: %ld\n", iPtr);
```

Output -> iPtr -= 3: 2686696

When **one pointer is subtracted from another pointer, the result is equivalent to the actual difference divided by their data type.**

Given that iPtr and jPtr are pointers of type int and iPtr = 2686692, while jPtr = 2686688
jPtr – iPtr = -1, which is (2686688 – 2686692)/4
iPtr – jPtr = 1, which is (2686692 – 2686688)/4

- Pointers can only be subtracted if they are of the same type
- **Two pointers cannot be added.**

PASSING ARGUMENTS BY REFERENCE

There are two ways to pass arguments to a function.

1. Passing by value
2. Passing by reference

Passing by value refers to a scenario where a copy of the argument is copied into the parameter of the function and it is used in the computation in the body. This is what has been done thus far with the function calls made thus far.
In passing by reference, the address of the argument is what is passed. This approach is considered useful because:

1. Passing a large data object e.g. an array, to avoid the overhead (memory-wise) involved in passing it by value. Now it has been confirmed that arrays are passed to functions by reference and not by value.
2. It provides a means of making a function manipulate the data of several variables in the environment (function) from which it is called. This is a way to mimic return multiple values from a function. As you will remember, all the functions that have been written thus far have the capability of returning just a single value or nothing (void) at all.

How then is a call by reference done? It is done by simply declaring the pointer as a parameter, and when the time comes to call the function, the value of the variable is passed indirectly through the address and using the indirection operator (*). In passing by value, it is the variable (direct reference to the value) itself that is passed.

The programs below are presented to illustrate the difference between passing arguments by value and by reference.

Code

```
/*COMPUTING CUBE OF A VARIABLE USING CALL-BY-VALUE*/

#include <stdio.h>

int cubeByValue(int n)
{
  return n * n * n; // cube computed using local value
}

int main( void )
{
  int number = 5; /* initialize number */

  printf( "The original value of number is %d", number );

  /* pass number by value to cubeByValue */
  number = cubeByValue( number );

  printf( "\nThe new value of number is %d\n", number );

  return 0; /* indicates successful termination */
} /* end main */
```

Output

```
The original value of number is 5
The new value of number is 125
```

Code

```
/*COMPUTING CUBE OF A VARIABLE USING CALL-BY-REFERENCE*/
#include <stdio.h>

void cubeByReference(int *nPtr)
{
  *nPtr = *nPtr * *nPtr * *nPtr; // cube computed using local value
}

int main( void )
{
  int number = 5; /* initialize number */

  printf( "The original value of number is %d", number );

  /* pass address of number to cubeByReference */
  cubeByReference( &number );

  printf( "\nThe new value of number is %d\n", number );

  return 0; /* indicates successful termination */
} /* end main */
```

Output

```
The original value of number is 5
The new value of number is 125
```

Functions that have been written thus far, return only one value or nothing at all. We have not encountered a function that can return multiple values. The concept of passing by reference is used to make a function compute several values. The idea is simply passing all the variables to hold the computed values, by reference to the function. For example, if we have to write and use a function that computes both the area and circumference of a circle using the radius passed to it, the variable for the area and the circumference will have to be passed by reference to the function. The calculated values will be assigned

to them in the body of the function through their address which was passed as argument (passing by reference). This is as show below:

Code

```
#include <stdio.h>

void circle(double *areaPtr, double *circPtr, double r)
{
  *areaPtr = 22 * r * r / 7;
  *circPtr = 2 * 22 * r / 7;
}

int main()
{

  double radius = 6.5;

  double area = 0, circ = 0;

  // calculate the area and circumference using the provided radius
  circle(&area, &circ, radius);

  printf("Radius        = %lf\n", radius);
  printf("Area          = %lf\n", area);
  printf("Circumference = %lf\n", circ);

  return 0;
}
```

Output

```
Radius        = 6.500000
Area          = 132.785714
Circumference = 40.857143
```

NOTE

- If sizeof() function is given a pointer of a given data type, it returns the size of the data type.
- Pointers of one type can't be assigned to pointers of other types
- A pointer of type void however can however be assigned pointers of other types, and can also be assigned to pointers of other types

STRINGS

A string is a value made up of a group of characters (digits, alphabets etc.) considered a single unit. A string value is normally written in-between double quotes e.g. "EEE 234" is a string value. When entering a string from the keyboard, it is not important to include the double quotes i.e. **"EEE 234"** will have to be entered as **EEE 234**.

DECLARING AND INITIALIZING STRINGS

1. **char name[size] = value;** e.g. char fullname[40] = "Tony Miebi";
   It is advised that the size used be very big so as to create room to store various strings of varying lengths all of which are lesser than the size of the char array used. The array can be declared without initializing, since a size has been specified.
2. **char name[] = value;** e.g. char surname[] = "Miebi";
   The assigning of value must be done at the point of declaration, and not after. It is compulsory to initialize the array when using this method. This is because, the size of the array is determined from the string value provided.
3. **char *name = value;** e.g. char *fullname = "Tony Miebi";
   Using this approach makes it possible, to separate declaration and initialization.

```
char code[10];
name = "EEE234"; // doesn't work

char *code;
code = "EEE234"; // works perfectly
```

Besides, it is also possible to assign different string values to the pointer after initialization. The length of the new string value doesn't matter.

```
char *code = "EEE234";
code = "FCE 246"; // works perfectly
```

It is not possible to change any of the characters of a string assigned to a pointer, because the value assigned is a constant. This is however possible with a character array.

```
char *texts = "Hello";
texts[1] = 'u'; // doesn't work

char texts[] = "Hello";
texts[1] = 'u'; // works perfectly
```

DISPLAYING STRING

Displaying strings can be done using the function **printf()** or **puts()**.

- When using **printf()**, the format specifier (placeholder) used is **%s**. The reference to the string used is its name.

  Given that

  ```
  char *s1 =  "Value 1";
  char s2[] = "Value 2";
  printf("%s\n", s1); outputs Value 1
  printf("%s\n", s2); outputs Value 2
  ```

  Or its value

  ```
  printf("%s", "Value 3"); outputs Value 3
  ```

- When using puts(), the reference to the string or its value is passed as argument. This function always displays a new line after the string.

  ```
  char *s1 = "Hello";
  puts(s1); puts("Everyone");
  ```

  Outputs

  ```
  Hello
  Everyone
  ```

It is possible to display specific characters in a string by using the name of the string and the index of the character to be displayed.

For example

```
char greetings[6] = "Hello";

printf("%c", greetings[0]); outputs H
```

Since **greetings** is the pointer to the first character

```
printf("%c", *greetings); also outputs H
```

INPUT OPERATIONS WITH STRINGS

There are two functions used for entering data into a string from the input stream. These are **scanf()** and **gets()**.

- **scanf()**: The format specifier required is %s and the Ampere's AND (&) is not required when specifying the variable, like is done with variables, i.e. scanf("%s", name); is acceptable while scanf("%s", &name); is unacceptable.

  The string literal read from the input stream doesn't a white character in it (space or tab). The reading of characters forming the string is stopped the moment a white character is encountered.

  The code segment below illustrates all of these.

  ```
  printf("Enter course code: ");
  char code[10];
  scanf("%s", code);
  printf("Code read: %s\n", code);
  ```

  The output is

  ```
  Enter course code: EEE 234
  Code read: EEE
  ```

  What about if two strings are needed and they are both entered after the prompt for the first one.

  ```
  printf("What is your forename?  ");
  char forename[10];
  ```

```
scanf("%s", forename);

printf("What is your surname?  ");
char surname[10];
scanf("%s", surname);

printf("Forename: %s\n", forename);
printf("Surname: %s\n", forename);
```

The output is

```
What is your forename?  James Apkan
What is your surname?

Forename: James
Surname: Apkan
```

Notice that an opportunity was not presented for the entering of surname. This is because after removing "James" alone (thanks to the space after it), the input stream still had "Akpan" which was then automatically accepted as the surname.

The scanf() function works smoothly with strings declared as character arrays. This is not the case for strings declared as character pointer, since they are for string constants.

The number of characters in a string literal to be read can be placed between % and s in the format specifier. This is important to prevent the scanf() function from reading data into memory locations outside the string. This can make the program crash.
The code segment below illustrates this:

```
printf("Enter a fruit: ");
char fruit[10];
scanf("%4s", fruit);
printf("Fruit read: %s\n", fruit);
```

The output is

```
Enter a fruit: Coconut
Fruit read: Coco
```

It is important to note that the truncating of data read from the input stream is not limited to strings alone. It is also operational when dealing with other data types.

- **gets()**: The variable to store the string read is provided as argument to the function. This function receives characters from the stream until it encounters the new line character, which implies that it reads white characters.

```
printf("Enter fullname: ");
char fullname[50];
gets(fullname);
printf("Fullname read: %s\n", fullname);
```

Output

```
Enter fullname: James Akpan
Fullname read: James Akpan
```

This function returns a pointer of type char which points to the variable containing the string read (its argument).

```
printf("Enter fullname: ");
char fullname[20];
printf("\ngets(fullname) = %p\n", gets(fullname));
printf("&fullname = %p\n", &fullname);
printf("fullname = %p\n", fullname);
```

Output

```
Enter fullname: James Akpan

gets(fullname) = 0xbee209c4
```

```
&fullname = 0xbee209c4
fullname = 0xbee209c4
```

Notice that the pointer values printed are the same.

CHARACTER-HANDLING LIBRARY

The library referred to here is **ctype.h**. It contains functions that are used to perform tests and manipulations on characters (in symbolic or numeric form). Some of these functions include:

1. **int isdigit(int c)**: returns true (1) if c is a digit and false if not.
   ```
   isdigit('g') = 0
   isdigit('2') = 1
   ```

2. **int isalpha(int c)**: returns true if c is an alphabet and false otherwise.
   ```
   isalpha('d') = 1
   isalpha('2') = 0
   ```

2. **int isalnum(int c)**: returns true if c is an alphabet or digit and false otherwise.
   ```
   isalnum('2') = 1
   isalnum('d') = 1
   isalnum('&') = 0
   ```

3. **int isxdigit(int c)**: returns true if c is an hexadecimal digit (0 - 9, a - f, A - F) and false otherwise.
   ```
   isxdigit('a') = 1
   isxdigit('F') = 1
   isxdigit('8') = 1
   isxdigit('z') = 0
   ```

4. **int islower(int c)**: returns true if c is a lower case letter and false otherwise.
   ```
   islower('f') = 1
   islower('A') = 0
   ```

5. **int isupper(int c)**: returns true if c is an upper case letter and false otherwise.
   ```
   isupper('f') = 0
   isupper('F') = 1
   ```
6. **int isspace(int c)**: returns true if c is a white space character (‘\n’, ‘ ’, ‘\f’ – form feed, ‘\r’ – carriage return, ‘\t’, or ‘\v’ – vertical tab) and false otherwise.
   ```
   isspace('\n') = 1
   isspace('&') = 0
   ```

7. **int iscntrl(int c)**: returns true if c is an escape character excluding those for backslash, single quote, and double quote.
   ```
   iscntrl('\\') = 0
   iscntrl('\n') = 1
   ```

8. **int isprint(int c)**: returns true if c is a printable character including space.
   ```
   isprint(' ') = 1
   isprint('\a') = 0
   ```

9. **int isgraph(int c)**: returns true if c is a printable character that is visible (spaces aren't included)
   ```
   isgraph(' ') = 0
   isgraph('a') = 1
   ```

10. **int ispunct(int c)**: returns true if c is not a digit, space, or alphabet.
    ```
    ispunct('#') = 1
    ispunct('a') = 0
    ```

11. **int tolower(int c)**: converts c to lowercase.
    ```
    tolower('a') = a
    tolower('A') = A
    ```

12. **int toupper(int c)**: converts c to uppercase.

```
toupper('A') = A
toupper('a') = A
```

STRING MANIPULATION FUNCTIONS

These functions are in the header **string.h.**

1. **char *strcpy(char *s1, const char *s2):** Copies s2 into s1. The new value of s2 is returned by the function after this.

```
char s1[] = “Hello”;
char s2[] = “Hi”;
printf(“strcpy(s1, s2) = %s\n”, strcpy(s1, s2));
printf(“s1 after %s\ns2 after %s\n”, s1, s2);
```

Output

```
strcpy(s1, s2) = Hi
s1 after Hi
s2 after Hi
```

2. **char *strncpy(char *s1, const *char s2, size_t n):** copies at most the first n characters of s2 to replace the first n characters of s1 after. The new value of s1 (pointer to s1 after copying) is returned. The term at most is used because null characters are exempted.

```
char s1[8] = "Standing";
char s2[5] = “Break”;
printf(“strncpy(s1, s2, 2) = %s\n”, strncpy(s1, s2, 2));
printf(“s1 after %s\ns2 after %s\n”, s1, s2);
```

Output

```
strncpy(s1, s2, 2) = Branding
s1 after Branding
s2 after Break
```

3. **char *strcat(char *s1, const char *s2):** appends the content of s2 to s1. The first character of s2 overwrites the terminating null character of s1. After appending, the new value of s1 is returned by the function.

```
char s1[9] = "sand";
char s2[9] = “bag”;
printf(“strcat(s1, s2) = %s\n”, strcat(s1, s2));
printf(“s1 after %s\ns2 after %s\n”, s1, s2);
```

Output

```
strcat(s1, s2) = sandbag
s1 after sandbag
s2 after bag
```

4. **char *strncat(char *s1, const char *s2, size_t n):** appends at most the first n characters of s2 to s1. The new value of s1 is returned. A null character is automatically added to the new string.

```
char s1[9] = "band";
char s2[9] = “ages”;
printf(“strncat(s1, s2, 3) = %s\n”, strncat(s1, s2, 3));
printf(“s1 after %s\ns2 after %s\n”, s1, s2);
```

Output

```
strncat(s1, s2, 3) = bandage
s1 after bandage
s2 after ages
```

5. **int strcmp(const char *s1, const char *s2):** the function compares s1 and s2, and returns 0 if they are equal, a value less than 0 if s1 is greater than s2, or a value greater than 0 if s1 is greater than s2. Determining which string is greater is done by comparing the numeric codes of corresponding characters in both strings. Based on this

```
strcmp("bad", "bed") = -1
strcmp("bad", "Bad") = 1
strcmp("ebb", "ebb ") = -1
strcmp("egg", "egg") = 0
```

6. **int strncmp(const char *s1, const char *s2):** does the same thing as strcmp() except that the first n characters of both s1 and s2 are compared in this case.

```
strncmp("damaged", "damages", 6) = 0
strncmp("damaged", "damages", 7) = -1
```

7. **char *strchr(const char *s, int c):** Locates the first occurrence of c in s and returns a pointer to c in s. Otherwise a NULL pointer is returned, signifying the character c is not in the string.

```
char s[] = "Madam";

printf("strchr(s, \'e\')   = %p\n", strchr(s, 'e'));
printf("strchr(s, \'a\')   = %p\n", strchr(s, 'a'));
printf("Address of 1st a = %p\n", &s[1]);
printf("Address of 2nd a = %p\n", &s[3]);
```

Output

```
strchr(s, 'e')   = 00000000
strchr(s, 'a')   = 0028FF1B
Address of 1st a = 0028FF1B
Address of 2nd a = 0028FF1D
```

Notice that pointer values from line 2 and 3 of the output are equal.

Displaying the pointer to c as a string, will display all characters in the string, starting from c to the end.

```
char s[] = "Madam";
printf("%s\n", strchr(s, 'a'));
```

Output

```
adam
```

8. **char *strrchr(const char *s, int c):** Locates the last occurrence of c in string s. If c is found, a pointer to c in string s is returned. Otherwise, a NULL pointer is returned.

```
char s[] = "Madam";

printf("strrchr(s, \'e\')  = %p\n", strrchr(s, 'e'));
printf("strrchr(s, \'a\')  = %p\n", strrchr(s, 'a'));
printf("Address of 1st a = %p\n", &s[1]);
printf("Address of 2nd a = %p\n", &s[3]);
```

Output

```
strrchr(s, 'e')  = 00000000
strrchr(s, 'a')  = 0028FF1D
Address of 1st a = 0028FF1B
Address of 2nd a = 0028FF1D
```

Notice that pointer values from line 2 and 4 of the output are equal.

9. **size_t strcspn(const char *s1, const char *s2):** Determines and returns the length of the initial segment of string s1 consisting only of characters not in string s2. The count is terminated the moment a character in s2 is encountered.

```
strcspn("madam", "add") = 1
strcspn("madam", "men") = 0
strcspn("madam", "end") = 2
```

10. **size_t strspn(const char *s1, const char *s2):** Determines and returns the length of the initial segment of string s1 consisting only of characters contained in string s2. The count of the characters in this region is terminated the moment a character not in s2 is encountered.

    strspn("madam", "add") = 0
    strspn("madam", "men") = 1
    strspn("mallam adam", "a lamb") = 8

11. **char *strpbrk(const char *s1, const char *s2):** Locates the first occurrence any character in s2 in s1. If a character from string s2 is found, a pointer to the character in string s1 is returned. Otherwise, a NULL is returned. The returned pointer can be displayed as a numeral or a string which is actually a substring starting from where the character occurred.

    s1 = “turntable”
    strpbrk(s1, "tan") = turntable (string) = 0028FF06 (&s1[0])
    strpbrk(s1, "am") = able (string) = 0028FF0B (&s1[5])
    strpbrk(s1, "odd") = (null) = 00000000

12. **char *strstr(const char *s1, const char *s2):** Locates the first occurrence of s2 in string s1. If the string is found, a pointer to the string in s1 is returned. Otherwise, a NULL is returned.

    strstr("banana", "na") = nana (displayed as a string)
    strstr("banana", "na") = anana
    strstr("banana", "ant") = (null)

13. **char *strtok(char *s1, const char *s2);:** The function is used to break the string s1 into parts delimited by string s2. Each part is called a token. Tokenization is the term used for the process of completely breaking the string into all its tokens. The pointer to the first token is obtained by using the string to be tokenized (s1) as first argument. **Further calls to strtok() to get the pointers of more tokens requires using NULL as first argument**. If there is no token available again, a NULL is returned.

Code 1

```c
#include <stdio.h>
#include <string.h>

int main( void )
{
  char string[] = "Computer Architecture";
  printf("String to tokenize: %s\n\n", string);

  printf("First token : %s\n", strtok(string, " "));
  printf("Second token: %s\n", strtok(NULL, " "));

  return 0;
}
```

Output 1

```
String to tokenize: Computer Architecture

First token : Computer
Second token: Architecture
```

If the string to be tokenized is very lengthy, it is possible to use a while loop to process the string to be tokenized. The code program below illustrates how this can be done.

Code 2

```c
#include <stdio.h>
#include <string.h>

int main( void )
{
  char string[] = "The course code for Computer Architecture is EEE-234";
```

```
    printf("String to be tokenized: %s\n\n", string);

    char *tokenPtr; // pointer to the current token

    // get first token
    tokenPtr = strtok(string, " ");

    // test for a token
    while(tokenPtr != NULL)
    {
      printf("%s\n", tokenPtr); // display current token

      tokenPtr = strtok(NULL, " "); // get the next token for testing
    }

    return 0;
}
```

Output 2

```
String to be tokenized: The course code for Computer Architecture is EEE-234

The
course
code
for
Computer
Architecture
is
EEE-234
```

14. **size_t strlen(const char *s1):** returns the length of s1 i.e. the number of characters appearing before the first NULL character.

```
strlen("EEE 234") = 7
strlen("Men\0Women") = 3
```

STRUCTURES

A structure is a collection of related variables irrespective of type under the same name. A structure is also called an **aggregate**. Each content of a structure is called a **member**.

DEFINING STRUCTURES

The syntax below illustrates the defining of a structure:

```
struct tag{
  type name;
  type name;
  :
  type name;
};
```

For example, a structure for a course (tag is equivalent to the name of the structure) can be defined as shown below:

```
struct course{
  char *code;
  int  units;
};
```

The definition of a structure comes before the main method, after all pre-processor directives.

USING A STRUCTURE AS A DATA TYPE

A structure is a derived data type, as a result variables, arrays and pointers can be defined using it as the data type.

- **struct tag ≡ type**

```
struct course electives;
struct course yr1Courses[12];
struct course *coursePtr;
```

A structure can have variables, arrays, and pointers of native data types, as well as other structures as its members. It can also contain its own pointer as member, but not its own variable.

```
struct course{
  char *code;
  char *title;
  int  units;
  struct course *nxtCourse; // acceptable as a member
  sturct course prevCourse; // unacceptable as a member
};
```

A structure that contains its own pointer (meant to point to another instance of its) is called a **self-referential structure**. Self-referential structures play a key role in the creating different types of data structures.

TYPEDEF

struct tag as type can be considered lengthy. Using typedef makes it possible to use a shorter identifier as replacement. The format for its usage is:

typedef struct tag Alias;

e.g. **typedef struct course Course;**

This is done immediately after defining the structure and after it, declarations done earlier will now be done as shown below:

```
Course electives;
Course yr1Courses[12];
Course *coursePtr;
```

INITIALIZING STRUCTURES

This is done when declaring the structure variable. The initializing is done by providing a comma-separated list of values in a braces. The values must match the members in order and type. It is not compulsory to provide values for all members.

Acceptable

- `struct course c1 = {“EEE 234”, 4};`
- `struct course c1 = {“EEE 234”};`
- `struct course *cPtr = &c1;`

Unacceptable

- ```
  struct course c1;
  c1 = {“EEE 234”, 4};
  ```

OPERATIIONS WITH STRUCTURES

The operations that can be carried out with structures include:

1. Assigning one structure variable to another structure variable

   ```
   Course c1 = {“FCE246”, 2};
   Course c2;
   c2 = c1
   ```

2. Declaring a pointer and assigning to it the address of a variable

   ```
   Course *c1Ptr;
   ```
   ```

```
c1Ptr = &c1;
```

3. Accessing the members of a structure.
   This is done through:
   i. Using the dot operator (.) with a regular variable. The format is **variable.member** e.g. c1.unit, c1.code
   ii. Using the **structure pointer operator (->)** with pointers. The format is pointer->member e.g. c1Ptr->code, c1Ptr->unit

   It is also possible to use the indirection operator (*) with the pointer in which case the dot operator is used e.g. (*c1Ptr).code, (*c1Ptr).unit.

4. Determining the size of a structure using the sizeof() function. For example sizeof(struct course) or sizeof(Course)

The program below illustrates all of these.

Code

```
#include <stdio.h>

struct course{
  char code[8];
  int unit;
};

typedef struct course Course;

int main()
{
  Course c = {"EEE 234", 4};
  Course *cPtr = &c;

  printf("c.code = %s\n", c.code);
  printf("c.unit = %d\n", c.unit);

  printf("\ncPtr->code = %s\n", cPtr->code);
  printf("cPtr->unit = %d\n", cPtr->unit);

  printf("\n(*cPtr).code = %s\n", (*cPtr).code);
  printf("(*cPtr).unit = %d\n", (*cPtr).unit);

  return 0;
}
```

Output

```
c.code = EEE 234
c.unit = 4

cPtr->code = EEE 234
cPtr->unit = 4

(*cPtr).code = EEE 234
(*cPtr).unit = 4
```

Input operations involving structure members is not very different from what input operations have been thus far. The main thing to note is how write the structure member to receive the data with a variable or pointer. It is important to remember that, a string can only be used in input operations, if it is declared as an array. The program below illustrates this:

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

struct course{
```

```
  char code[8];
  int unit;
};

typedef struct course Course;

int main( ){

  Course c;

  printf("Enter code: ");
  gets(c.code);

  printf("Enter units: ");
  scanf("%d", &c.unit);

  printf("\nCode entered: %s\nUnits entered: %d\n", c.code, c.unit);

  return 0;
}
```

Output

```
Enter code: EEE 234
Enter units: 4

Code entered: EEE 234
Units entered: 4
```

USING STRUCTURES WITH ARRAYS

The most important thing to note is that, the type has to be the alias chosen for the structure. Furthermore, every element in the array contains several values, not just one.

Code

```
#include <stdio.h>

struct course{
  char code[8];
  int unit;
};

typedef struct course Course;

int main(){
  const int SIZE = 6;
  Course one1stSem[SIZE];

  int i;
  for(i = 0; i < SIZE; i++){
    printf("\nEnter course code %d: ", i + 1);
    scanf("%s", &one1stSem[i].code);

    printf("Enter unit %d: ", i + 1);
    scanf("%d", &one1stSem[i].unit);
  }

  printf("\n\nCOURSE CODES\tUNITS\n**********\t*****\n");
  for(i = 0; i < SIZE; i++)
    printf("%-13s\t%d\n", one1stSem[i].code, one1stSem[i].unit);

  return 0;
}
```

Output

```
Enter course code 1: MTH105
Enter unit 1: 4

Enter course code 2: PHY105
Enter unit 2: 4

Enter course code 3: CHM101
Enter unit 3: 4

Enter course code 4: GST101
Enter unit 4: 3

Enter course code 5: GST100
Enter unit 5: 2

Enter course code 6: FCE131
Enter unit 6: 2


COURSE CODES    UNITS
************    *****
MTH105          4
PHY105          4
CHM101          4
GST101          3
GST100          2
FCE131          2
```

It is also possible to have an array as a member of a structure

USING STRUCTURES WITH FUNCTIONS

It is possible to make an array a member of a structure, as well as have an array of a structure.

Structures are passed to a function by value and not by reference. As a result of this array that exist in a structure is passed by copy with passed through a structure unlike an array standing on its own. If it becomes necessary a pointer of structure is passed as argument.

In the program below, there are two structures: one for a course, the other for a student. In each student structure exists an array of courses and finally a student variable is created in the program. The program prompts for the details to initialize with which to initialize the student variable created. Finally, all received details are displayed in a properly formatted manner. Details such as grade are computed with the aid of a function which receives a score and returns the appropriate grade. The program below illustrates the use of arrays with structure as type.

Code

```c
#include <stdio.h>

struct course{
  char code[8];
  int score;
  char grade;
};

typedef struct course Course;
```

```c
struct student{
  char matNo[11];
  Course firstSem[3]; // an array of 3 courses
};

typedef struct student Student;

// function to return grade of score received
char getGrade(int score){
  if(score < 45)
    return 'F';
  else if(score < 50)
    return 'D';
  else if(score < 60)
    return 'C';
  else if(score < 70)
    return 'B';
  else
    return 'A';
}

int main( void )
{
  Student student1;

  // get matric number details
  printf("Enter matriculation number: ");
  gets(student1.matNo);

  // get the code and scores for the courses
  int i;
  for(i = 0; i < 3; i++){
    printf("\nEnter course code for course %d: ", i + 1);
    gets(student1.firstSem[i].code);

    char dummy; // to receive the trailing ENTER character after taking score

    printf("Enter score for course %d: ", i + 1);
    scanf("%d", &student1.firstSem[i].score); // leaves behind ENTER key
    scanf("%c", &dummy); // take the ENTER key away
  }

  // get grades for each course score
  for(i = 0; i < 3; i++)
    student1.firstSem[i].grade = getGrade(student1.firstSem[i].score);

  // print all details
  puts("\n\n");
  printf("MATRIC. NUMBER: %s\n", student1.matNo);
  printf("**************************\n\n");

  printf("%-11s  %-5s  %-5s\n", "COURSE CODE", "SCORE", "GRADE");
  for(i = 0; i < 3; i++)
    printf("%-11s  %-5d  %-5c\n",
           student1.firstSem[i].code,
           student1.firstSem[i].score,
           student1.firstSem[i].grade);


  return 0;
}
```

Output

```
Enter matriculation number: UG\03\0552
```

```
Enter course code for course 1: EEE 234
Enter score for course 1: 56

Enter course code for course 2: EEE 232
Enter score for course 2: 23

Enter course code for course 3: FCE 246
Enter score for course 3: 67



MATRIC. NUMBER: UG\03\0552
*************************

COURSE CODE  SCORE  GRADE
EEE 234      56     C
EEE 232      23     F
FCE 246      67     B
```

The size of a structure can be obtained by passing the structure type to the function. The type can be either a combination of the struct-keyword and the tag or the alias defined for it using typedef. It is important to state that the size of a structure is not necessarily the sum of the size of its members. This is because the memory locations set for its members do not belong to a contiguous block, like we have in arrays. It is also possible to provide a declared structure variable as argument to this function. The code segment below illustrates this.

Code

```c
#include <stdio.h>

struct samp{
  int a;
  double b;
};

typedef struct samp Samp;

int main( void ){

  Samp s;

  printf("sizeof(Samp): %d\n", sizeof(Samp));
  printf("sizeof(struct samp): %d\n", sizeof(struct samp));
  printf("sizeof(s): %d\n", sizeof(s));

  printf("sizeof(s.a) + sizeof(s.b): %d\n", sizeof(s.a) + sizeof(s.b));

  printf("\n\nAddress of s.a: %d\n", &s.a);
  printf("Address of s.b: %d\n", &s.b);

  return 0;
}
```

Output

```
sizeof(Samp): 16
sizeof(struct samp): 16
sizeof(s): 16
sizeof(s.a) + sizeof(s.b): 12


Address of s.a: 2686736
```

Address of s.b: 2686744

ENUMERATION CONSTANTS

Sometimes a variable of a specific data type is intended to hold a specific set of values, even though there is an infinite amount of data that can be held. For example, a variable for grades in a 5-point system are fixed, because grades are restricted to A, B, C, D, E, and F. Another example is a variable to hold data representing days of the week.
Enumeration constants are used to model these types of data. An enumeration constant is a list of integer values represented by integer values represented by identifiers. Each integer value represents a specific data, which is described very clearly by its identifier.
Enumeration constants like structures are also user-defined data type and are defined using the enum keyword.

```
enum name{
  ID, ID, ....., ID
};
```

For example

```
enum days{
  SUN, MON, TUE, WED, THU, FRI, SAT
};
```

The integer values of the enumeration constants start from 0 by default for the first constant and an increment by 1 gives the value of the next enumeration constant. It is also possible to set the value of the first constant in the list, or any other constant in the list. In doing this, it is completely alright to tie the same value to more than one constant.
For example

```
enum grade{
  F, E = 0, D = 2, C, B, A
};
```

From the above example F = 0, C = 3, B = 4, and A = 5.

A variable of an enumerated type can be declared with the name of the enumerated type - preceded by the enum keyword - standing as type. It is possible to replace this combination, with an alias introduced by typedef. The variable can only be assigned any one of the constant identifiers. An integer value will not work even though internally, each constant identifier represents an integer. When the value assigned to such a variable is displayed, it is in the form of the value it is assigned. It is also possible to use this variable in relational and logical operations and also create pointers of these variables.

Code

```c
#include <stdio.h>

enum grade{
  F, E = 0, D = 2, C, B, A
};

typedef enum grade Grade;

int main( void ){

  Grade g1;
  g1 = F;

  enum grade g2 = C;

  Grade g3;
  g3 = g2; // assign one enum type to another
```

```
  Grade *g1Ptr = &g1;

  printf("g1 = %d\n", g1);
  printf("g2 = %d\n", g2);
  printf("g3 = %d\n", g3);
  printf("g1Ptr = %d\n", g1Ptr);

  return 0;
}
```

Output

```
g1 = 0
g2 = 3
g3 = 3
g1Ptr = 2686736
```

BITWISE OPERATORS

These are logical operators which operate on their operands at the binary level on a bit by bit basis. All but one are binary in nature (take two operands). The result of an operation involving bitwise operator is best presented in binary form.

1. **Bitwise AND, &**: Operates like the logical AND, && in the sense that it is false (0) if at least one operand is false.

   Code

   ```
   #include <stdio.h>

   int main( void ){
     /*
      *     6 = 0110
      *     8 = 1000
      * 6 & 8 = 0000 = 0
      */
     printf("6 & 8 = %d\n", 6 & 8);

     return 0;
   }
   ```

   Output

   ```
   6 & 8 = 0
   ```

2. **Bitwise inclusive OR, |**: this operator gives a 1 if at least one of its operands is 1.

   Code

   ```
   #include <stdio.h>

   int main( void ){
     /*
      *     6 = 0110
      *     8 = 1000
      * 6 | 8 = 1110 = 14
      */
     printf("6 | 8 = %d\n", 6 | 8);

     return 0;
   }
   ```

   Output

   ```
   6 | 8 = 14
   ```

3. **Bitwise exclusive OR, ^**: this operator gives a 1 only if both its operands are different i.e. 0 and 1, or 1 and 0.

   Code

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

```c
int main( void ){
  /*
   *      6 = 0110
   *     12 = 1100
   * 6 ^ 12 = 1010 = 10
   */
  printf("6 ^ 12 = %d\n", 6 ^ 12);

  return 0;
}
```

Output

```
6 ^ 12 = 10
```

4. **Left shift, <<**: shifts the bits of the first operand left by the number of places specified by the second operand. This is equivalent to adding the number of 0's specified by the second operand to the right of the first operand. You will notice that each shift to the left is equivalent to a multiplication by 2.

Code

```c
#include <stdio.h>

int main( void ){
  /*
   *      6 = 0110
   *  6 << 1 = 01100 = 12
   *  6 << 3 = 0110000 = 48
   */
  printf("6 << 1 = %d\n", 6 << 1);
  printf("6 << 3 = %d\n", 6 << 3);

  return 0;
}
```

Output

```
6 << 1 = 12
6 << 3 = 48
```

5. **Shift right, >>**: shifts the bits of the first operant right by the number of places specified by the second operand. It is equivalent to striking off the number of bits specified by the second operand from the right side of the left operand. Each shift right is an integer division by two.

Code

```c
#include <stdio.h>

int main( void ){
  /*
   *        7 = 0111
   *   7 >> 1 = 011 = 3
   *       48 = 0110000
   *  48 >> 3 = 0110 = 6
   */
  printf("7 >> 1 = %d\n", 7 >> 1);
  printf("48 >> 3 = %d\n", 48 >> 3);

  return 0;
}
```

Output

```
7 >> 1 = 3
48 >> 3 = 6
```

6. **One's complement, ~**: It is a unary operator. It complements all the bits of its operand. Figuring out the result requires understanding that an integer is 32 bits. In signed numbers, the most significant bit represents + if it is 0, and – if it is 1. **Negative numbers are presented in signed 2's complement form. If one wished to know the true identity of a negative number in binary form,**

its 2's complement has to be taken to get its positive form (this is done by complementing all its bits and adding 1 to the result)

Code

```
#include <stdio.h>

int main( void ){
  /*
   *   3 = 00000000000000000000000000000011
   *  ~3 = 11111111111111111111111111111100
   *  +ve equivalent = 00000000000000000000000000000011
   *                  +                               1
   *                 = 00000000000000000000000000000100 = -4
   *  ~3 = -4
   */
  printf("~3 = %d\n", ~3);

  return 0;
}
```

Output

```
~3 = -4
```

QUESTIONS FOR REVIEW

1. If the declarations below are in the same program (function), state why it is not acceptable to declare variables this way.

```
int a;
double a;
```

2. How is uniqueness among the references of an array achieved?

3. Explain why the two code segments below will display exactly the same character.

```
(I)   printf("%c", name[0]);
(II)  printf("%c", *name);
```

4. What will be the output of the code segment below:

```
char *code = "EEE 234";
int i;
for(i = 0; i < 7; i++)
  printf("%c\n", code[i]);
```

5. Using the above string (in 4), what will be the output of `printf("%c\n", *(code + 4));`

6. Given the code segment below

```
void main(){
  display(8);
}

void display(int a){
  printf("%d\n", a);
}
```

Rewrite the above code segment, in such a way as to prevent a syntax error from occurring (feel free to make additions to the code).

7. If a structure is defined as shown below:

```
struct course{
  char code[8];
  int unit;
};
```

What will be the output of the code segment below, if "EEE 234" was submitted by pressing the ENTER-key, in response to the first prompt (prompt for course code)?

```
struct course c;
```

```
printf(“Enter course code: “);
scanf(“%s”, c.course);
printf(“Enter unit: “);
scanf(“%d”, &c.unit);
printf(“%s\n%d\n”, c.code, c.unit);
```

8. If a structure is defined as shown below:

```
struct course{
  char code[8];
  int unit;
};
```

What will be the output of the code segment below, if “EEE 234” was submitted by pressing the ENTER-key, in response to the first prompt (prompt for course code)?

```
struct course c;
printf(“Enter units: “);
scanf(“%d”, c.unit);
printf(“Enter course code: “);
gets(c.code)
printf(“%s\n%d\n”, c.code, c.unit);
```