Unit 2

→ All programming languages are built from 4 basic elements.
① Expressions
② Statements
③ Statement blocks
④ function blocks.

* Expression: ① Created by combining operators & operands.
eg → a+b + → operator
a & b → operands

* Statements: ① Complete instruction for a program the computer.
② All statements end with a semi colon (;)
eg → j = 5 + k * 2;
→ Variable → a location in the memory that has been assigned a name.

→ Operator Precedence → Refers to the order in which complex expressions are resolved. *, / & % operators are resolved before + & −.

* Statement blocks: ① One or more statements grouped together.
② Viewed by the compiler as if they are a single statement
③ Starts with an opening brace character { and ends with closing brace character }.

→ Arduino C data Types

boolean	1 Byte	
char	1 Byte	−128 to +127
unsigned char	1 Byte	0 to 255
byte	1 Byte	0 to 255
int	2 Bytes	−32768 to 32767

unsigned int	2 bytes	0 to 65535
word	2 bytes	0 to 65535
void	0	
long	4 bytes	
unsigned long	4 bytes	
float	4 bytes	
double	4 bytes.	
string		
String		
array		

void → used in function declarations. Indicates that the function is expected to return no, information.

Boolean → Holds one of two values, true or false.

Char → Stores a character value. Character literals are written in single quotes. Eg: 'A'
Stored as numbers. Each character has an ASCII code assigned to it.
A → 65, a → 97

Unsigned Char → Unsigned data type, encodes numbers from 0 to 255.
eg: Unsigned char ch = 121;

Byte: Stores an 8-bit unsigned number from 0 to 255.
eg: byte m = 25;

Int: Primary data types for number storage. Stores a 16 bits (2 byte) value. Range → -32768 to [redacted]. +32767.

Unsigned int: Same as int. But stores only the values. No -ve values.
Eg: Unsigned int c = 60;

Word: On the UNO and other ATMega based boards, a word stores a 16-bit unsigned number. On Due & Zero, it stores 32 bit Unsigned No.
eg: word w=1000;

long: long variables are extended size variables for number storage. (4 bytes). Eg: long velocity = 102346;

Unsigned long: Do not store -ve numbers. Range → $(2^{32}-1)$.

short: 16 bit data type. Stores a 16 bit (2 Byte) value.
Eg: short val = 13;

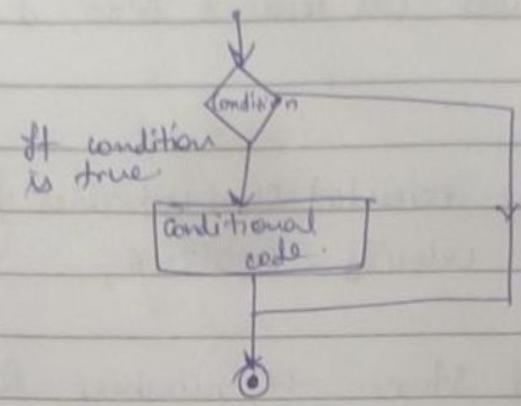
float: Used to store decimal values.

* Keywords in C: Words that have a special meaning to the compiler. Reserved words (for compiler's use).
We cannot use them for our own variable/function names. If we do, then the compiler will flag it as an error.

* Variables: Naming and storing a value for later use by the program.
Valid variable names may contain:
→ Characters a-z & A-Z
→ Underscore (_)
→ Numbers from 0 to 9 (These cannot be the 1st character in the name).

* Decision making: ① Requires that the programmer must specify one or more conditions to be evaluated/tested by the program.
② A set of statements should be executed if the condition is true otherwise another set of statements must be executed.
③ In Arduino programming language, we make decisions with if statement.
④ The if statement will check if a condition is true, then it

will execute the block of code within the curly brackets.

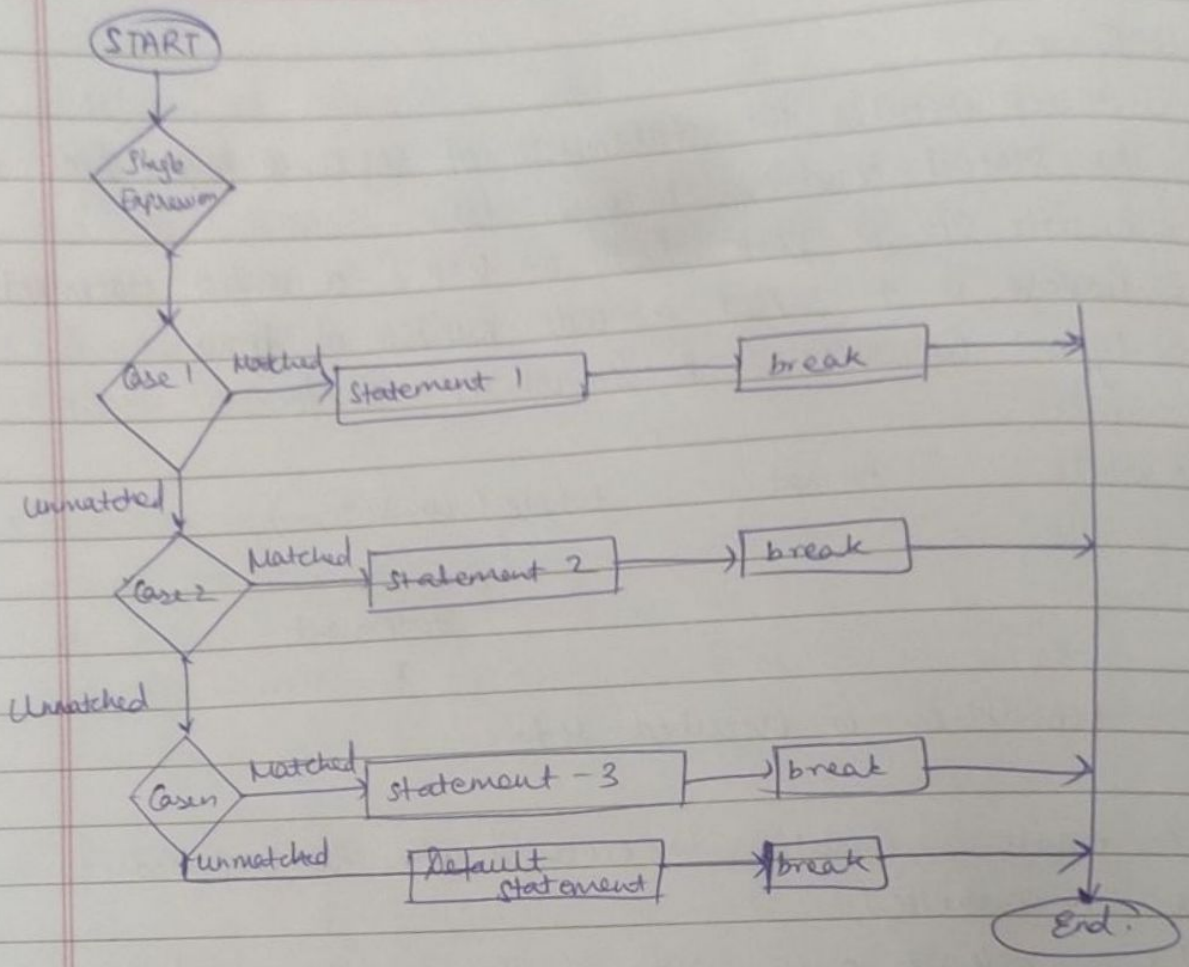


eg: if (condition)
{
// code to execute if condition is true
}

else
{
// code to execute if condition is false.
}.

* SWITCH CASE STATEMENT:

① Switch case tests the value of a variable and compares it with multiple cases.
② Once the case match is found, then the block of statements associated with it is executed.
③ Each case has a different name/number (Identifier)
④ If case match is NOT found, then the default statement is executed. Control goes out of switch block.



* Conditional operator:

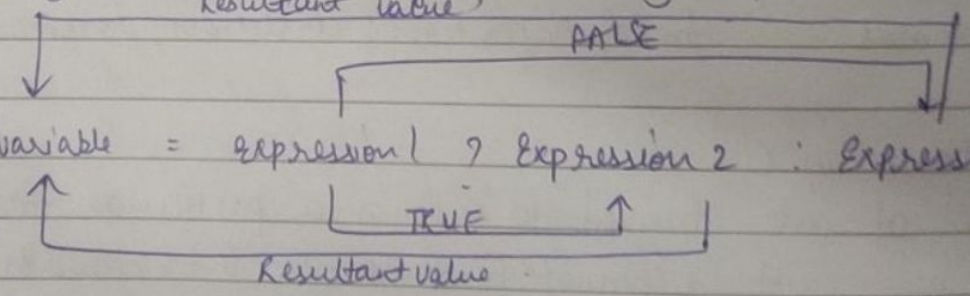
① Also known as ternary operator.

② Represented by two symbols, '?' and ':'.

③ Works on 3 operands.

④ Behaviour is similar to if-else statement.

eg → (age >= 18) ? (printf("Eligible for voting")) : printf("not eligible");



* LOOPS in C:

① A loop executes the statements of sequence many times until the stated condition becomes false.

② Consists of 2 parts: body of loop & a control statement.

③ Purpose is to repeat a code number of times.

④ Types: ⓐ while ⓑ Do-while ⓒ for

ⓐ while: format:

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

→ Condition is executed before.

ⓑ Do-while: → Condition is executed after the loop body.
→ Exit controlled.
→ Runs atleast once even if the condition is false.

```
do
{
    Statements
} while (Expression);
```

ⓒ for loop: ① More efficient.

② format: for (Initialization; condition; Increment/decrement)

* Functions in C:

① Body of code designed to perform a particular task.

eg: int Rectangle (int length, int breadth)
{
// body function body.

Type Specifier. function name. function arguments.

* Function Type Specifier → Determines the return type of func.
→ Can be int, char, long, double, byte, etc).
→ If no value is returned, then void keyword is used.

* Function Name → → Name of the function.
→ Same naming rules as that of variables.

* Function Arguments → Can be 0 or more.
→ Used to pass data to the func. that it may need to perform its task.
→ Multiple func. arguments: Commas b/w arguments.
The function rectangle has 2 arguments → length & breadth.

* Function Body: → All the statements b/w opening bracket '{' and closing bracket '}' comprise of the function body.
→ If the func. type specifier is anything other than void, then it must contain the keyword return.

* Function Signature: → Tells the name of the function and the data that it expects to be passed as arguments.
→ eg. for the func. Volume of Cube(), the func. signature is
Volume of Cube(int width, int length, int height)
→ Func. signatures are imp. b/c you can have more than one functions with the same name.

* Overloaded functions: → Anytime a function has 2 or more different signatures, it is called an overloaded function.
→ Arduino C permits overloaded functions.
→ Overloaded functions add a degree of consistency in a programming situation.

* What makes a "Good" function?
→ The function name must reflect what the func. does
→ function should be cohesive (Designed to accomplish a single task)
→ Avoid Coupling (the need for one func. to depend on the results of other function)

* POINTERS: eg: int *myPointer;
(Type) (Name of pointer) (asterisk)

3 Basic components to a pointer: ① Type
(marks the variable as pointer instead of a normal/regular variable) ② Asterisk
③ Name.

→ Pointer variables have the same naming rules as all other variables in C.
→ Common convention is to begin with ptr. eg: ptrSisters, ptrName etc.

(*) Asterisk: ① Used in pointer definition to tell the compiler that it is a pointer and not a regular variable.

→ Pointer Type specifiers: Dictate the type of data to be used by that pointer.

→ Pointer Scalars: The scalar value is equal to the no. of bytes required to store that data type in memory.

→ A pointer should only hold a memory address or null.
→ Pointer variables don't hold values.
→ Using a pointer → with address of operator (&)
→ A pointer never points to anything useful until it is initialized.

* Why are pointers useful?
① Functions cannot change the value of an argument passed to it because func. arguments are pass by value data items.
② Pointers are used in this case to change the value of the argument.

→ Relational tests on pointers are acceptable only when both the operands are pointers.

if(ptr1 > ptr2) → acceptable
if(ptr1 > 10) → unacceptable.

→ You should not perform relational operations on pointers if they do not point to the same data object.

* Two Dimensional Arrays: ① Often used to represent tabular data.
② Row-column format
③ A one-dimensional array resolves to a pointer to char, so the name of the array is the l-value for the array
In 2 dimensional array, you have a pointer to an array, rather than a pointer to a pointer.

* STRUCTURE:
① User defined data type
② Allows us to combine data items of different kinds.
③ Used to represent a record.
④ defined using the 'struct' keyword.
format: struct [structure name]
{
member definition;
.
.
.
};

* UNION : ① Special Data type in C that allows storing different data types in the same memory location.
② You can define a union with many members but only one member can contain a value at a given time.
③ They provide an efficient way of using the same memory location for different purposes.
④ Defined using the 'union' statement.
format:

```
union [union name]
{
member definition
;
;
};
```

* Similarities b/w Structure & Union:
① Both are user defined data types used to store data of different types as a single unit.
② Their members can be objects of any type, including other structures or unions or Arrays.
③ A member can also consist of a bit field.
④ Both structure and union support only assignment '=' and sizeof operators.
⑤ They can be passed by value to functions and can be returned by value by functions.
⑥ '.' operator is used for accessing members.

* Difference b/w Structure & Union.

Structure	Union
→ Keyword struct is used to define.	Keyword Union is used to define.
→ The size of each structure is greater than or equal to the sum of size of its members.	Size of union is equal to the size of the largest member.
→ Each member is assigned unique storage area of location	Memory allocated is shared by individual members of union.
→ Altering the value of a member will not affect other members.	Altering the value of a member will alter other member values.
→ Individual members can be accessed at a time	Only one member can be accessed at a time.
→ Several members of a structure can initialize at once.	Only the 1st member of a union can be initialized.

POINTERS

Pointers are variables that contain the address or location of a variable, constant, function, or data object.

```
Char *p; // p is a pointer to a character
int *fp; // fp is a pointer to an integer
```

```
Char *p ; // p is a pointer to a character
Char a,b ; // a and b are characters
p = &a ; // p is now pointing to a
```

In this example p is assigned the address of a, so p is "pointing to" a.

ARRAYS

An array is another system of indirection. An array is a data set of a declared type, arranged in order. An array declared like any other variable or constant, except the number of required elements:

```
int digits [10]; // this declares an array of 10 int
Char Str [20] ; // this declares an array of 20 char
#include <stdio.h>
const char S[15] = {"This is a test"}
char i;
char *p;
```



```
Void main (void)
{
for (i=0 ; i<15 ; i++) // Print each character of the array by using i as an index
Putchar (S[i]);
p=S;                   // point to string as a whole
for (i=0 ; i<15 ; i++) // Print each character of the array and move to the
Putchar (*p++) ;       // next element by incrementing the pointer p.
while(1) ;
}
```

The library function putchar() to send one character at a time to the Standard output device, usually like Serial port.

Libraries

Libraries provide extra functionality for use in sketches, e.g. working with hardware or manipulating data. To use a library in a sketch, select it from **Sketch > Import Library**.

Standard Libraries

- **EEPROM** - reading and writing to "permanent" storage
- **Ethernet** - for connecting to the internet using the Arduino Ethernet Shield
- **Firmata** - for communicating with applications on the computer using a standard serial protocol.
- **LiquidCrystal** - for controlling liquid crystal displays (LCDs)
- **SD** - for reading and writing SD cards
- **Servo** - for controlling servo motors
- **SPI** - for communicating with devices using the Serial Peripheral Interface (SPI) Bus
- **SoftwareSerial** - for serial communication on any digital pins
- **Stepper** - for controlling stepper motors
- **Wire** - Two Wire Interface (TWI/I2C) for sending and receiving data over a net of devices or sensors.

The Matrix and Sprite libraries are no longer part of the core distribution.

Leonardo Only Libraries

- **Keyboard** - Send keystrokes to an attached computer.
- **Mouse** - Control cursor movement on a connected computer.

Table 11-1. Arduino C Preprocessor Directives

Directive	Action
`#define NAME value`	Ascribes the identifier `NAME` to the constant `value`.
`#undef NAME`	Removes `NAME` from the list of defined constants
`#line lineNumberValue "filename.ino"`	Allows the compiler to refer to any line numbers in the file named `filename.ino` to be referenced as line `lineNumberValue` from this point on by the compiler. Normally used in debugging. This is not in the Arduino C reference material, but the compiler recognizes it.

`#if definedConstant expression operand`	Conditional compilation. Example:
	`#if LED == 12`
	`  #define VOLTS 5`
	`#endif`
	This is not in the Arduino C reference material, but the compiler recognizes it.

`#if defined NAME`
`  // statement(s)`
`#endif`
	Allows for conditional compilation of statements if `NAME` is defined. The statement block ends with `#endif`. This is not in the Arduino C reference material, but the compiler recognizes it.

`#if !defined NAME`
`  // statement(s)`
`#endif`
	Same as `#if defined`, but processes statement block only if `NAME` is not defined. This is not in the Arduino C reference material, but the compiler recognizes it.

`#ifdef`	Same as `#if defined`. This is not in the Arduino C reference material, but the compiler recognizes it.
`#ifdef`	Same as `#if defined`. This is not in the Arduino C reference material, but the compiler recognizes it.
`#ifndef`	Same as `#if !defined`. This is not in the Arduino C reference material, but the compiler recognizes it.

`#else`	Can be used with `#if` like an `if-else` statement but to control compiled statements. Example:
	`#if defined ATMEGA2560`
	`  #define BUFFER 64`
	`#else`
	`  #define BUFFER 32`
	`#endif`
	This is not in the Arduino C reference material, but the

`#include "filename.xxx"`	Opens the file named `filename.xxx` and reads the contents of the file into the program source code. Usually, if double quotes surround the file name, then the search for the file is in the currently active directory. If angle brackets are used (`<filename.xxx>`), then the search begins in some implementation-defined manner. This is not in the Arduino C reference material, but the compiler recognizes it.

```
#define FIRESENSOR 145
```

#undef

The `#undef` is used to turn off a previously-defined `#define` preprocessor directive. For example, suppose you have a source file with something like the following code in it:

```
#ifdef DEBUG
   Serial.print("The counter value is: ");
   Serial.println(myCounter);
#endif
```

#line

The `#line` directive is used most often while debugging a program. The syntax is:

```
#line lineNumberValue "filename.ino"
```

where `lineNumberValue` is the line number you want to the compiler to use from that point on in the source code file name `filename.ino`. Therefore:

```
#line 100 "C:\Temp\myCode.ino"
```

```
#define DEBUG 1
//
// A whole bunch of program lines
// that still need to be debugged
//
#undef DEBUG
```

```
#if definedConstant expression operand
// Statement(s)
#endif
```

might be written as:

```
#if BOARD == ATMEGA168
        #define MAXEEPROM    1024
#endif
```

```
#if !defined expression
```

is the negative of the previous directive. That is:

```
#if !defined BOARD
       #define MAXEEPROM   1024
#endif
```

This says that if `BOARD` has not been `#defined` in the program, then `MAXEEPROM` gets set to 1024. This directive can also be written using the `#ifndef` in the same manner:

```
#ifndef BOARD
       #define MAXEEPROM   1024
#endif
```

The result is exactly as before: If `BOARD` has not been `#defined` in the program, then `MAXEEPROM` is set to 1024.

is the negative of the previous directive. That is:

```
#if !defined BOARD
        #define MAXEEPROM   1024
#endif
```

This says that if BOARD has not been #defined in the program, then MAXEEPROM gets set to 1024. This directive can also be written using the #ifndef in the same manner:

```
#ifndef BOARD
        #define MAXEEPROM   1024
#endif
```

The result is exactly as before: If BOARD has not been #defined in the program, then MAXEEPROM is set to 1024.

#else, #endif

All of the conditional preprocessor directives must end with a #endif directive. However, you can have an if-else type of directive by using #else:

```
#ifdef BOARD
        #define MAXEEPROM   1024
#else
        #define MAXEEPROM   512
#endif
```

In this case, if BOARD is defined, then MAXEEPROM is set to 1024, otherwise it is set to 512. This gives you a little more flexibility for setting MAXEEPROM.

Finally, you can also use #elif to form a cascading if statement, as in:

```
#if BOARD == ATMEGA168
        #define MAXEEPROM   512
#elif BOARD == ATMEGA2560
        #define MAXEEPROM   4096
#else
        #define MAXEEPROM   1024
#endif
```

The #elif simplifies the code from what it would be if the directive was not used.

Table 11-2. Standard C Header Files

Header file name	Description
stdio.h	Standard I/O header file with macro for file redirection and most file I/O
stdlib.h	Memory allocation functions, string conversions, value-to-ASCII conversions
string.h	A host of memory and string processing declarations
math.h	Math declarations, symbolic constants (e.g., pi), transcendental function declarations.
ctype.h	Character processing declarations (e.g., isalpha())

Memory Types :-

The architecture of a microprocessor may require that variables and constants be stored in different types of memory. Data that will not change should be stored in one type of memory, while data that must be read from and written to repetitively in a program should be stored in another type of memory. A third type of memory can be used to store variable data that must be retained even when power is removed from the system. When special memory types such as pointers and register variable are accessed, additional factors must be considered.

Constant and Variables

The AVR microcontroller was designed using Harvard architecture, with separate address space for data (SRAM), program (FLASH), and EEPROM memory. The CodeVision AVR® and other compilers implement three types of memory descriptors to allow easy access to these very different types of memory.

The default or automatic allocation of variables, where no memory descriptor keyword is used, is in SRAM.
Constants can be placed in FLASH memory (Program Space) with the flash or const keywords.
For variables to be placed in EEPROM, the eeprom keyword is used.
When declarations are made, the position of the flash and eeprom keywords become part of the meaning.
If const, flash, or eeprom appear first, this states to the compiler that the actual allocation of storage or the location of data is in that memory area.
If the type is declared followed by the flash or eeprom keyword, this indicates that it is a variable that references FLASH or EEPROM, but the variable itself physically located in SRAM. This scenario is used when declaring pointers into FLASH or EEPROM.

The following declarations will place physical data directly in program memory (FLASH) These data values are all constant and cannot be changed in any way by program execution.

```
flash integer_constant = 12345;
flash char char_constant = 'a';
flash long long_int_constant1 = 99L;
```

EEPROM space is a non volatile yet variable area of memory. Variables can be placed in EEPROM memory simply by declaration

```
eeprom int cycle_count; // allocates an integer space in EEPROM
eeprom char ee_string[20]; // allocates a 20 byte area in EEPROM
eeprom struct {
    char a;
    int b;
    char c[15];
} se; // allocates 18 byte structure 'se' in EEPROM
```

The permanent (FLASH) and semipermanent (EEPROM) memory areas have many system specific uses in the embedded world. FLASH space is an excellent area for non changing data. The program code itself resides in this region. Declaring items such as text string and arithmetic lookup tables in this region directly frees up valuable SRAM space.

If a string is declared with an initializer like

```
Char mystring[30] = ("This String is Placed in SRAM");
```

30 byte of SRAM will lee allocated, and the "This String is placed in SRAM" is physically placed in FLASH memory with the program. On startup, this FLASH resident data is copied to SRAM and the program works from SRAM whenever accessing mystring. This is a waste of 30 bytes of SRAM unless the String is intended for alteration by the program during run time. To prevent this loss of SRAM space, the String could lee Stored in FLASH memory directly by the declaration:

flash char mystring[30] = "This String is placed in SRAM".

The EEPROM area is called non volatile, meaning that when power is removed from the microprocessor the data will remain intact, lut it is semipermanent in that the program can alter the data located in this region. EEPROM also has a life - it has a maximum number of write cycles that can lee performed lefore it will electrically fail. This memory area will have a rating of 10,000 write operations. max. There are no limitation on the no of