Data Types

Data types of all variables must be known (specified at declaration) at compilation time. The main categories of data types in C includes,

Basic / Built-in:

- void – absence of type
- char – ASCII characters
- int – integer, signed
- float – floating point

Data types defined in **stdint.h** have guaranteed fixed data size.

Signed	Unsigned	Data Size (Bytes)	Min & Max Value
int8_t	int8_t	1	-2^7 to $(2^7 - 1)$ / 0 to $(2^8 - 1)$
int16_t	uint16_t	2	-2^{15} to $(2^{15} - 1)$ / 0 to $(2^{16} - 1)$
int32_t	uint32_t	4	-2^{31} to $(2^{31} - 1)$ / 0 to $(2^{32} - 1)$
int64_t	uint64_t	8	-2^{63} to $(2^{63} - 1)$ / 0 to $(2^{64} - 1)$

Derived

Array

A storage of multiple data items of a common basic data type (see above section). Can be visualised as a set of contiguous cells with each cell taking one item/element. There are 3 aspects of every memory cell – address, value and user-defined name (label, constant name, etc.).

Address	Memory Block	Remarks	
0x … 10	1	x[0]	← Base Address
0x … 14	2	x[1]	
0x … 18	3	x[2]	Offset
0x … XX	:	:	
0x … XX	:	:	
0x … 19C	100	x[99]	

- One-dimensional Array

Example:

```
int x[100] ;
```

The integer array is capable of storing 100 integer values with the first element called/referred to as **x[0]** and the last element as **x[99]**.

- Two-dimensional Array

Same definition as one-dimensional arrays with common data types. A two-dimensional array can be imagined as a matrix of **a** rows and **b** column subscripts.

Example:

```
int y[3][4] = { { 1 , 2 , 3 , 4 } , { 5 , 6 , 7 , 8 } , { 9 , 10 , 11 , 12 } } ;
```

Address in is HEXADECIMAL, and read as 'block' – byte number.

Eg: 1st block and 4 byes

Address	Memory Block	Remarks
0x … 10	1	y[0][0]
0x … 14	2	y[0][1]
0x … 18	3	y[0][2]
0x … 1C	4	y[0][3]
0x … 20	:	:
0x … 38	11	y[2][2]
0x … 3C	12	y[2][3]

[row] [column]

Structures

Object containing a set of members, who can be of different data types.

```
/* Creates a struct called 'grade_book' */
struct grade_book
{
        char name[30];
   int phone;
   float score;
};

/* No memory is allocated until its variable is created */

int main()
{
  /* Create variables of data type "struct grade_book" */
  struct grade_book st1;
  strcpy(st1.name, "Adam");
  st1.phone = 98761234;
  st1.score = 88.5;
}
```

struct member operator " . "

Structures can be simplified with the **typedef** keyword for a cleaner and more readable code.

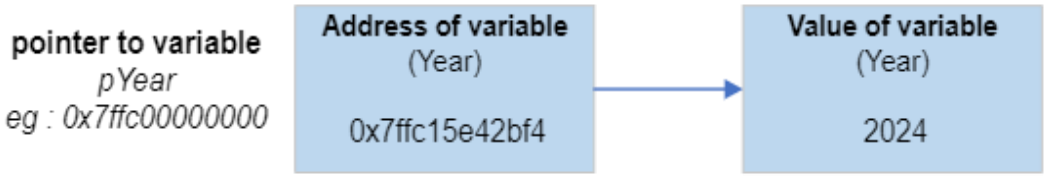
```
typedef struct class_2024 // 'class_2024' is optional
{
        char name[30];
   int phone;
   float score;
} grade_book; // struct still called by 'grade_book'

/* No memory is allocated until its variable is created */

int main()
{
  /* Create variables of data type "struct grade_book" */
  grade_book st1 = {"Adam", 98761234, 88.5};
```

Pointers

A pointer is a variable that stores the **memory address** of another variable as its value. Advantages of using pointers includes, improved code readability and efficient handling of arrays and structures, and passing of data to functions. It provides a way to perform dynamic memory allocation and deallocation, and direct access to the memory. Lastly, it enables the creation of complex data structures such as linked lists, stacks, queues, trees, and graphs.



- Address Operator (**&**)

The unary operator **&** reads as 'address of' is called the address operator (or reference operator) that gets the **memory address** of its operand. If we assign a variable to a pointer, it is called a **pointer to a variable** that points to the memory location where variable is stored.

```
int main() {
  int Year = 2024;
  printf("%d\n", Year);     // returns "2024"
  printf("%p\n", &Year);     // returns "0x7ffc15e42bf4", the memory address

}
```

- Indirection Operator (*)

The unary operator * reads as “value pointed by” is called the indirection operator (or dereference operator) that evaluates the **value** of its pointer variable operand, it access the value of the variable instead. The operator operates only on a pointer variable.

```
int main() {
  int Year = 2024;
  int* pYear = &Year; //Pointer declaration
  printf("%p\n", pYear);     // returns “0x7ffc15e42bf4”, the memory address
  printf("%d\n", *pYear);     // returns “2024”, value of Year through the Pointer

}
```

User-defined

- structure, union, enumeration, etc.

There are two forms of data type conversion,

Implicit (automatic)

For expressions with mixed data types, compiler automatically promotes lower type to higher type to avoid loss of data. Note that information lose is possible for implicit conversions (eg. int to unsigned). Conversion is based on the following Data Type Orde (lowest to highest),

bool > char > short > unsigned short > int > unsigned > long > unsigned long > long long > unsigned long long > float > double …

Explicit (type casting)

Explicit is user-defined to make a convert to a specific data type. Results can become unpredictable as per the following example. Use the following C syntax,

(**type**) expression

Example,

```
float x = 1.9999;
int y = (int)x; //Explicit conversion from float to int
```

Variables and Constants

Have meaningful naming conventions which indicates its purpose concisely using only alphanumeric characters and underscore, not forgetting all variables are case sensitive.

Do not begin with a number and some keywords / reserved words are not allowed. Note that names begin with '_' are for reserved identifiers while a trailing '_t' are reserved for standard types.

Variables

Usually in lowercase

Constants

Usually in uppercase. Declaration of a **constant** is similar to a **variable** by prepending ***const*** keyword and the value specified, eg.

```
const int FYP = 2024
```

Alternatively, declare a constant using preprocessor directive ***#define***. It creates a macro which is the association of an identifier or parameterised identifier with a token string, eg.

```
#define FYP 2024
```

Operators

Group	Operator(s)	Remarks
Arithmetic	+ - * / % ++ --	Addition, subtraction, multiplication, division, modulo, increment, decrement
Comparison / Relational	< <= > >= == !=	Less than, equal or less than, more than, equal or more than, equals to, not equals to
Logical	! && ||	NOT, AND, OR To combine 2 or more Boolean expressions

Bitwise	~ << >> & ^ |	NOT, left shift, right shift, AND, XOR, OR Manipulate data, perform bit-level operations on operands
Assignment	= += -= *= /= %= ^= |= <<= >>=	Assign/Set value to a named variable
Miscellaneous	() [] . -> (type) * & sizeof ?: ,	

sizeof operator

A **compile-time** (when source code is converted to binary code, as compared to run-time) unary operator (operator that perform operations on a single operand to produce a new value) to compute size of its operand. Result of sizeof is unsigned integral type usually denoted by size_t, can be applied to any data type (integer, floating point, etc.)

Syntax:

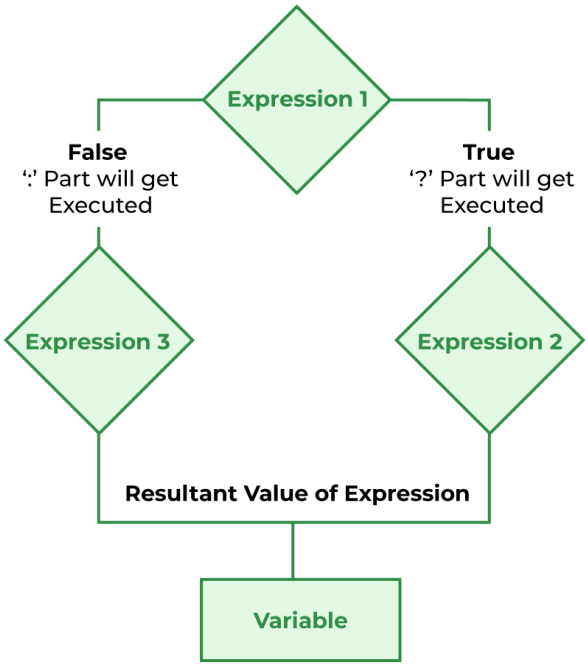
```
sizeof(expression);
```

?: (conditional / ternary) operator

Conditional operator similar to if-else statements that takes less space.

Syntax:

```
variable = expression_1 ? expression_2 : expression_3;
OR
variable = (condition) ? expression_2 : expression_3;
OR
(condition) ? (variable = expression_2) : (variable = expression_3);
```



Selection

Selections or conditional are used to determine a different set of steps to execute based on Boolean expression(s).

if , else if , else

- Use **if** to specify a block of code only to be executed if a specified condition is true (condition_1)
- [Optional] Use **else if** to specify a new / additional condition (condition_2) to test after condition_1 is false.
- [Optional] Use **else** to specify a block of code to be executed if both conditions are false.

Syntax:

```
if (condition_1)
{
// code block to be executed if condition_1 is true
}
else if (condition_2)
{
// code block to be executed if condition_1 is false and condition_2 is true
}
else
{
// code block to be executed if both conditions are false
}
```

switch , case

Use **switch-case** to specify many alternative blocks of code to be executed instead of using multiple **if-else** statements.

Syntax:

```
switch (expression)
{
   case a:
   //block to be executed if a is true
   break; //ends this case & passes to next statement after the switch-case construct
.
. // multiple case blocks to be tested
.
   case z:
   //block to be executed if z is true
   break;
   default:
   //block to be executed if no case matches
}
```

Iteration

Iteration or loops are used to repeatedly execute a block of code based on either a count or condition.

You can 'exit' any loop with the **break** keyword or by using **if-else** and continue immediately with the program after the loop.

Also, using the **continue** keyword within any loop to jump to the end of the code block, skipping statements between the keyword and end of the loop code block, the loop then continues normally

for

To repeatedly execute a block of code based on **count**

Syntax:

```
for ( initial condition ; test ; update )
{
//code block to be executed as long as test result is true
}
```

Initial condition: Expression to initialise a local loop variable (eg. *int count = 1*)

Test: Checks that the loop variable meets the test condition to repeatedly execute the block of code, otherwise it ends the loop (eg. *count* <= *10*)

Update: Loop variable automatically increases / decreases based on this expression (eg. *i*++)

while

A simpler **for**-loop with only a single test

Syntax:

```
while( test )
{
//code block to be executed while test result is true
//usually 'update' within the code block
}
```

do-while

An 'inverted' **while** loop that execute the code block once first before test.

Syntax & Example:

```
int x = 5;
do
{
   printf("%d \n", x);
   x = x – 2;
} while( x >= 0 );
```

Result will be,

```
5 // code block executed once first, 'update' is done
3 // code block executed as test (x >= 0) is true and x = 3
1 // code block executed as test (x >= 0) is true and x = 1
```

Function

A function is a block of code (**definition**) which only runs when called upon, executing the defined actions/tasks.

Syntax:

```
return_type function_name (parameter list)
{
/* function's definition here (the code block) */
}
```

- **return_type** : indicates the data type of the value the function returns. If function does not return any value, it must be declared as a ***void*** - refer to subsection on 'Data Types'.
- **function_name** : the name of the function, following naming convention of variables and constants. Execution of the function when the function_name is called upon.
- **parameter_list** : indicates the type, order/position and number of parameters of the function declared (inputs to the function).If no input parameters, use ***void***.

Program control is transferred to the called function and all defined actions within the function are executed before transferring the program control back to the calling program when its return statement is executed. A function (like variables) should be declared and defined before it is being called. The function being called can be defined in the same code file "main.c" or in another source file.

If function called is defined in another source file (eg. a header file, "main.h"), the called function must be included at the beginning of the file. Make sure to have all the source files saved in the same folder/directory.

Example:

```
#include "DAVE.h"

int main(void
{
/* main function's code block */
}
```

Function Call

- Pass by Value:

  Actual value of parameters passed into the function are copied onto the local variables such that they can be used but cannot be changed, within the called function.

  ```
  int my_function(int a, char b)
  ```

- Pass by Reference:

Address of the parameters passed into the function are copied onto the local variables such that they can be used and changed, within the called function.

```
int my_function(int *a, char *b)
```

Macro-like Function

A simple preprocessor macro only substitutes a value,

```
#define PI 3.14 //defines the variable 'PI as '3.14' value
```

A macro can be defined to work the same way as a function, eg. to perform a repetitive task.

Syntax:

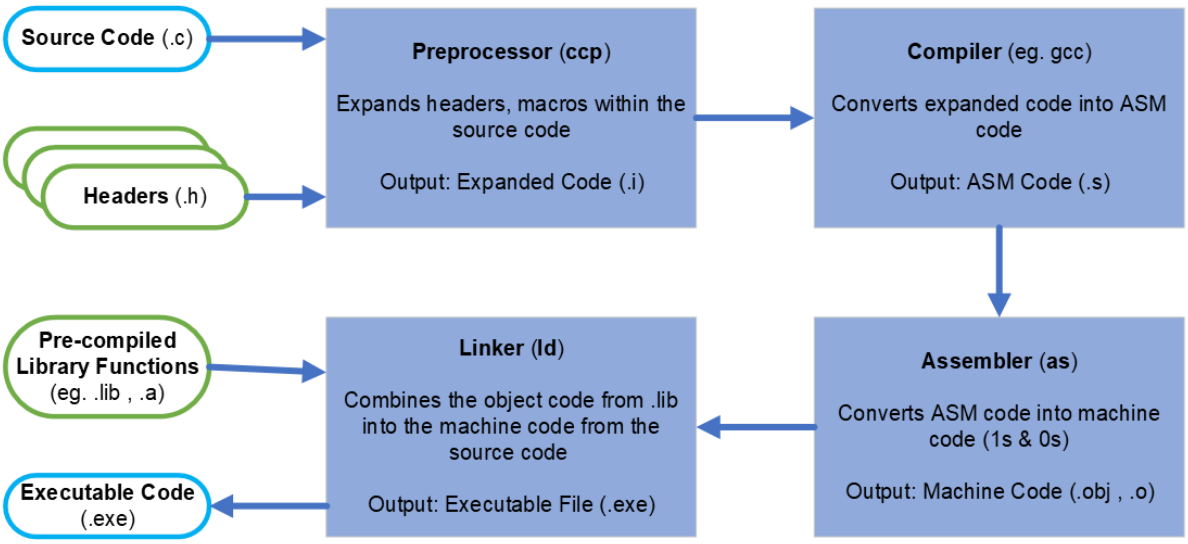
```
#define marcro_name(arg) (macro_definition)
```

- **marco_name** : the name of the macro (function), following naming convention of variables and constants. Execution of the marco when the macro_name is called upon.
- **arg** : indicates the argument (variable)
- **marco_definition** : define the macro to be executed, eg. formula to find circumference of a circle

Compilation Process

Overview

The C program source code can be written in a text editor and necessary files to be compiled and linked in order to produce the desired executable program is selected (saved within the same folder). The compiler converts this code into machine code that the computer can understand, combining it with pre-compiled library files to generate an executable file that cab be loaded and run by the CPU.



Preprocessor (.c , .h → .i)

Preprocessing is the first step of the process, inserting the content stored in the ".h" header files (function declarations, macro declarations) into the ".c" source code file, in the process called substitution or expansion.

The preprocessor directives (eg. *#define* statements) are used to indicate what substitutions are required. Commonly used directives: *#define , #include, #undef , #ifdef , #ifndef , #if , #else , # elif , #endif , #error , #pragma*. Aside from *#define* used as an example earlier on, there is also *#include* (eg. *#include <stdint.h>* to include the integer types header file).

In the case of DAVE IDE, see 'DAVE.h' file under Dave > Generated, containing all the necessary DAVE APP's Header Files after you have added the APPs and 'Generate Code'. In the 'main.c' source code, the first line calls to *#include "DAVE.h"*.

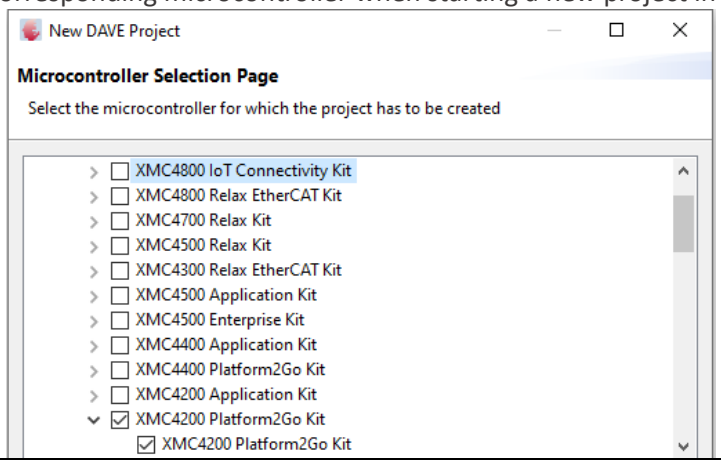
Compiler (.i → .s)

The second step, compilation converts the expanded code into assembly code. Since the assembly language is specific to the target processor [Infineon XMC4200 Platform2Go], even if the same source code and compiler were used, a different assembly code would be generated. The ".s" assembly code is then passed to the assembler.

Assembler (.s → .obj / .o)

The third step, assembly, converts the assembly code into binary or machine code (aka object code) within the .obj (Windows) or .o (Unix) extension. The code generated here is the actual machine level instructions understood only by the specific processor. The code is then passed to the linker.

Hence, select the corresponding microcontroller when starting a new project in DAVE IDE.



Linker (.obj / .o , .lib / .a → .exe)

Linking is the last step. C programs usually use the standard library functions (eg. *printf()* etc.). These static library functions are pre-compiled and their codes are saved in files within the .lib (Windows) or .a (Unix) extension. The linker includes these pre-compiled library object codes directly to the source object code generated in the previous step before outputting an executable file.

References

- C Programming content from EE2028 lecture slides by Dr Henry Tan