PART 7: Embedded Programming STM32

Basics C programming

Why C

- ✓ Need a higher-level language than assembly
- ✓ Close to Hardware
- ✓ Reusable
- ✓ Structured

Remarks About Programming

- ✓ A program is launched as soon as the power is up and never stops
- ✓ The program MUST have a while(1) loop
- ✓ The program must optimize power consumption, size and time execution (algorithm): a program by interrupt is often preferred

Compilation

Compilation is processed in various phases to create several files from a text file (the C file) to get the language machine file

- ✓ Preprocessing: the source file is analyzed by a program called preprocessor that performs purely textual transformations (deletion of comments, replacement of the #define…)
- ✓ Compilation / assembly: the file generated by the preprocessor is translated into assembler, then into machine code
- ✓ Link editing: a program is often separated into several source files. The linker allows you to link the different files (i.e. defining memory location)
- ✓ Boot Program: placed at the beginning of the program before the main to initialize the stack pointer and if necessary, the variables

Basic Program

In any embedded C program, any program must start this way

- ✓ main() function: this is starting point of the program
- ✓ Inside the main program, **an infinite loop** often written while(1) (sometimes for(;;))

```
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */

}
/* USER CODE END 3 */
```

```
98                  while (1)
08000190: 0x0000fee7   b.n     0x8000190 <main+20>
```

A compilation of the program translates the C program in assembly language

ISEN ALL IS DIGITAL! OUEST | yncréa

Assembly Program

The file xxx.ls is generated during the compilation and shows how the C program was 'translated' in assembly language

```
112    /* USER CODE END 2 */
113
114    /* Infinite loop */
115    /* USER CODE BEGIN WHILE */
116    while (1)
117  8000190:   e7fe        b.n 8000190 <main+0x14>
118      ...
```

Files in C program

Two types of files are included

- ✓ .c files : these files include the functions and the c programs
- ✓ .h files : called header files (or include files), it includes most of variables and definitions (see #define)
- ✓ Optionally assembly files (.s)

Size of Int

In many languages, it's common to declare an integer as an int

```
int entier = 5;
int tailleEntier;

main()
{
  tailleEntier = sizeof(entier);
  while (1);

}
```

Variable	Value	Type	Address
tailleEntier	2	int	0x0

tailleEntier	int	4

This small program allows to check the actual size of an integer

Size of an integer is equal to 2 bytes on the STM8

Size of an integer is equal to 4 bytes on the STM32

Size of Data Types in C

✓ On the STM8

Variable	Size in Bytes
char	1
int	2
Long	4

Variable	Size in Bytes
char	1
Short int	2
int	4
long	8

Type	C-Standard Type Size	Value Range (min)
signed char	At least 8-bits	$[-2^7, +2^7 - 1]$
unsigned char	At least 8-bits	$[0, +2^8 - 1]$
signed short int	At least 16-bits	$[-2^{15}, +2^{15} - 1]$
unsigned short int	At least 16-bits	$[0, +2^{16} - 1]$
signed long int	At least 32-bits	$[-2^{15}, +2^{15} - 1]$
unsigned long int	At least 32-bits	$[0, +2^{32} - 1]$

✓ Sizes of C data types are ambiguous and vary between architectures.

✓ C-standard specifies a minimum each variable can be

Problem with size of Int

- ✓ The size of int is dependent on platform
  - 2 bytes on the STM8
  - 4 bytes on a Cortex-M 32-bit microcontroller
- ✓ Portability of programs gets impossible (purpose of C-program is to give this flexibility vs assembly)
- ✓ Any microcontroller has many 8-bit operations (like reading a register in an STM8)

Do not Declare any variable as int in embedded C programming

Solving Int issue

- ✓ Create new types that impose the size of integer
  - int8_t : signed integer of exactly 8 bits
  - int16_t : signed integer of exactly 16 bits
  - int32_t : signed integer of exactly 32 bits

  - uint8_t : unsigned integer of exactly 8 bits
  - uint16_t : unsigned integer of exactly 16 bits
  - uint32_t : unsigned integer of exactly 32 bits

- ✓ These types must be defined in a header file with the intrinsic types of the platform

Typedef Keyword

- ✓ Allows programmer to create his own type.
  - ✓ Can apply to standard types

```
typedef unsigned char BYTE;
```

  - After this type definition, the identifier BYTE can be used as an abbreviation for the type unsigned char for instance

```
BYTE  b1, b2;
```

  - ✓ Or derived types

```
typedef enum Color {
    COLOR_BLUE = 0,
    COLOR_RED = 1,
    COLOR_GREEN = 2,
} Color_t;
```

```
typedef struct Data {
    int32_t  temperature;
    unt32_t date;
    unt32_t time;
} Data_t;
```

Typedef: Integers

```
/* USER CODE BEGIN PV */
int8_t entier_8 = 5;
int16_t entier_16 = 5;
int32_t entier_32 = 5;
int64_t entier_64 = 5;

uint8_t tailleEntier_8, tailleEntier_16, tailleEntier_32, tailleEntier_64;
```

```
typedef signed char int8_t ;
typedef signed short int16_t;
typedef signed int int32_t;
typedef signed long long int64_t;
```

```
tailleEntier_8 = sizeof(entier_8);
tailleEntier_16 = sizeof(entier_16);
tailleEntier_32 = sizeof(entier_32);
tailleEntier_64 = sizeof(entier_64);
```

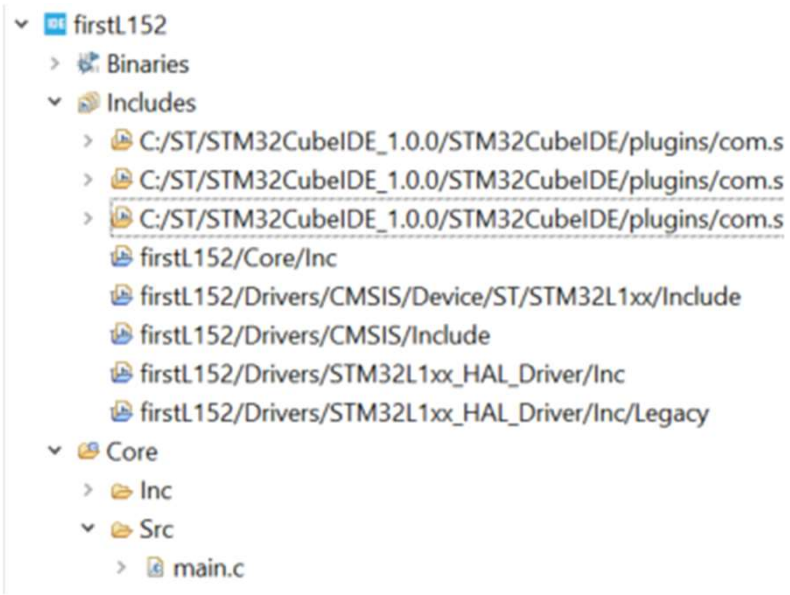
tailleEntier_8	uint8_t	1 '\001'
tailleEntier_16	uint8_t	2 '\002'
tailleEntier_32	uint8_t	4 '\004'
tailleEntier_64	uint8_t	8 '\b'

Typedef creates new types with the right size for the integers

ISEN ALL IS DIGITAL! OUEST | yncréa

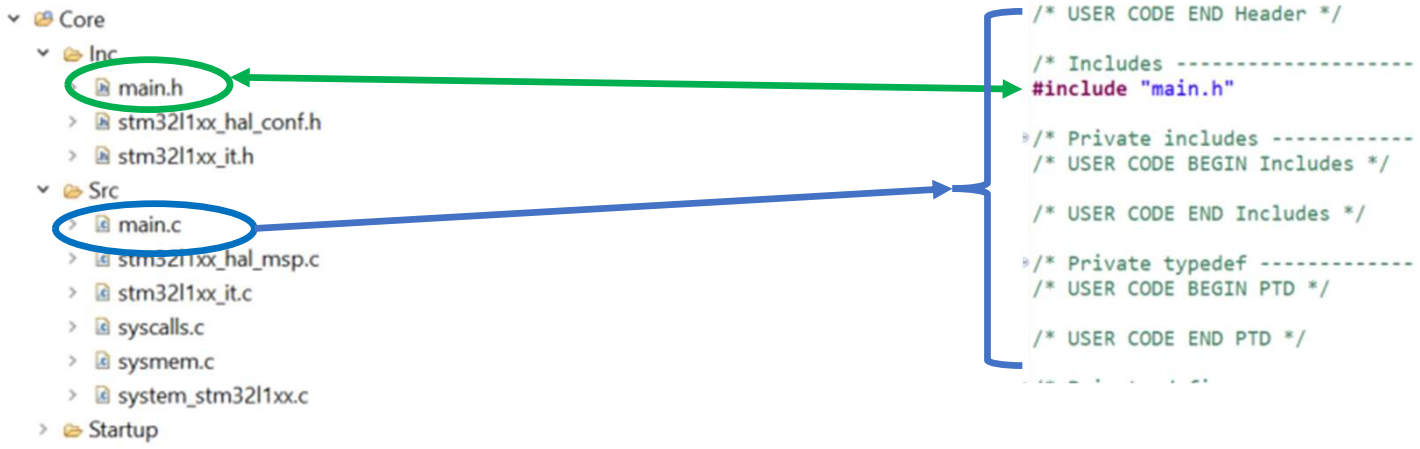
Types defined by Silicon Vendor

- ✓ The tedious task of renaming is done by the silicon vendor who provides a header file with all the necessary typedef
- ✓ When creating a project with STM32CubeIDE, a bunch of include files are integrated



include Keyword

- ✓ In an IDE (Integrated Development Environment) tool, the files are organized with 2 folders: One for C programs and one for .h files
- ✓ The main.c includes the main() function. '#include' at the top of the main main.c means it will use variables and functions defined in the corresponding file
- ✓ The grammar is #include ''name of header file '' (Be careful, no semicolon ';')

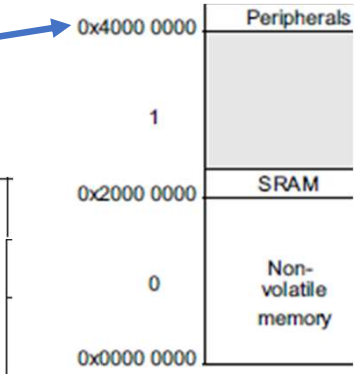


stm32l152xe.h File

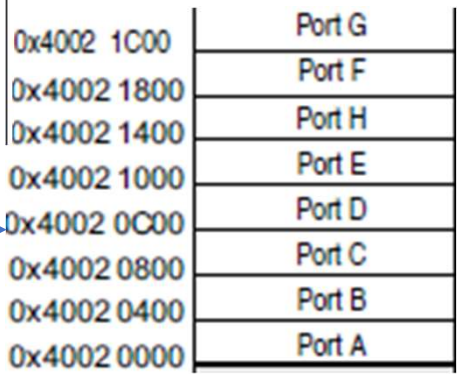
```
#define PERIPH_BASE           (0x40000000UL)

#define AHBPERIPH_BASE        (PERIPH_BASE + 0x00020000UL)
```

AHBPERIPH_BASE = 0x4002 0000

Boundary address	Peripheral	Bus
0x4002 3000 - 0x4002 33FF	CRC	AHB
0x4002 1C00 - 0x4002 1FFF	GPIOG	
0x4002 1800 - 0x4002 1BFF	GPIOF	
0x4002 1400 - 0x4002 17FF	GPIOH	
0x4002 1000 - 0x4002 13FF	GPIOE	
0x4002 0C00 - 0x4002 0FFF	GPIOD	
0x4002 0800 - 0x4002 0BFF	GPIOC	
0x4002 0400 - 0x4002 07FF	GPIOB	
0x4002 0000 - 0x4002 03FF	GPIOA	




```
/*!< AHB peripherals */
#define GPIOA_BASE            (AHBPERIPH_BASE + 0x00000000UL)
#define GPIOB_BASE            (AHBPERIPH_BASE + 0x00000400UL)
#define GPIOC_BASE            (AHBPERIPH_BASE + 0x00000800UL)
#define GPIOD_BASE            (AHBPERIPH_BASE + 0x00000C00UL)
#define GPIOE_BASE            (AHBPERIPH_BASE + 0x00001000UL)
#define GPIOH_BASE            (AHBPERIPH_BASE + 0x00001400UL)
#define GPIOF_BASE            (AHBPERIPH_BASE + 0x00001800UL)
```



GPIOC_BASE = 0x4002 0C00

Volatile

C's volatile keyword is a qualifier telling the compiler that the value of the variable may change at any time without any action being taken by the code the compiler finds nearby (in other words, whenever a variable value could change unexpectedly)

In practice, only three types of variable could change

- ✓ Memory-mapped peripherals registers
- ✓ Global variables modified by an interrupt service routine
- ✓ Global variables accessed by multiple tasks within a multi-threaded application.

Variable Location

```
/* USER CODE BEGIN PV */
volatile uint32_t myvarOutMain;
```

Memory Monitor: &(myvarOutMain) : 0x20000028

myvarOutMain saved in the data section of the RAM

```
int main(void)
{
  /* USER CODE BEGIN 1 */
  volatile uint32_t myvarInMain = 0xDEADBEEF;
  myvarOutMain = 0x12345678;
```

Memory Monitor: &(myvarInMain) : 0x20013FF4

myvarInMain saved in the stack

The location to store a variable depends on where it is declared in the program and how it is declared

Pointer

- ✓ For a C programmer, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address
- ✓ This way, each cell can be easily located in the memory by means of its unique address (the STM32L152 is viewed as a 4GB (=4,096,000,000,000 bytes) memory)
- ✓ When a variable is declared, the memory needed to store its value is assigned a specific location in memory

Pointer: * Symbol

```
/* USER CODE BEGIN 1 */
  uint32_t myvar = 0xDEADBEEF;
  uint32_t * pointer_myvar;
  pointer_myvar = &myvar;
  * pointer_myvar = 0x12345678;
```

- ✓ myvar is declared as an unsigned 32-bit integer (4 bytes on stm32)
- ✓ pointer_myvar is declared as a pointer pointing to an uint32_t
- ✓ point_myvar is loaded with the address of myvar
- ✓ This location can be pointed to change the value at this specific location

Struct

- ✓ Structure is a data type that allows to combine items of the same or different kinds
- ✓ The struct tag is optional, and each member is a normal variable definition.
- ✓ At the end of the structure definition, before the final semicolon, you can specify one or more structure variables, but it is optional.

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

ISEN ALL IS DIGITAL! OUEST | yncréa

Struct for GPIO

- ✓ All the ports have a common structure with the same 11 registers.
- ✓ It makes sense to combine these 11 registers in one structure and apply this structure to all the ports

```
/* USER CODE BEGIN 0 */
typedef struct
{
  uint32_t MODER;        /*!< GPIO port mode register,                  Address offset: 0x00      */
  uint32_t OTYPER;       /*!< GPIO port output type register,           Address offset: 0x04      */
  uint32_t OSPEEDR;      /*!< GPIO port output speed register,          Address offset: 0x08      */
  uint32_t PUPDR;        /*!< GPIO port pull-up/pull-down register,     Address offset: 0x0C      */
  uint32_t IDR;          /*!< GPIO port input data register,            Address offset: 0x10      */
  uint32_t ODR;          /*!< GPIO port output data register,           Address offset: 0x14      */
  uint32_t BSRR;         /*!< GPIO port bit set/reset registerBSRR,     Address offset: 0x18      */
  uint32_t LCKR;         /*!< GPIO port configuration lock register,    Address offset: 0x1C      */
  uint32_t AFR[2];       /*!< GPIO alternate function register,         Address offset: 0x20-0x24 */
  uint32_t BRR;          /*!< GPIO bit reset register,                  Address offset: 0x28      */
} GPIO_struct;
```

This structure matches the hardware of a GPIO on the STM32L152

Access to GPIO W/ struct

- ✓ GPIO_portB is declared as a pointer to a structure GPIO_struct
- ✓ In the STM32L152RE8, the address of port B is at address 0x40020400
- ✓ Pointing to the GPIO_portB, it is possible to select MODER element and changing the value of this register

```
GPIO_struct * GPIO_portB;
GPIO_portB = (GPIO_struct *) 0x40020400;
(* GPIO_portB).MODER = 0xFFFFFFFF;
```

Improvment (1/2)

The annotaton (*typeStructure).structureElement is tedious to write and usually replaced by typeStructure→structureElement

```
GPIO_struct * GPIO_portB;
GPIO_portB = (GPIO_struct *) 0x40020400;
(* GPIO_portB).MODER = 0xFFFFFFFF;
```

```
GPIO_struct * GPIO_portB;
GPIO_portB = (GPIO_struct *) 0x40020400;
GPIO_portB->MODER = 0xFFFFFFFF;
```

ISEN ALL IS DIGITAL! OUEST | yncréa

Improvement (2/2)

The GPIO_portB can be initialized in one line

```
GPIO_struct * GPIO_portB;
GPIO_portB = (GPIO_struct *) 0x40020400;
(* GPIO_portB).MODER = 0xFFFFFFFF;
```

```
GPIO_struct * GPIO_portB = (GPIO_struct *) 0x40020400;
GPIO_portB->MODER = 0xFFFFFFFF;
```

C-Program Compiled

```
/* USER CODE BEGIN 0 */
typedef struct
{
  uint32_t MODER;
  uint32_t OTYPER;
  uint32_t OSPEEDR;
  uint32_t PUPDR;
  uint32_t IDR;
  uint32_t ODR;
  uint32_t BSRR;
  uint32_t LCKR;
  uint32_t AFR[2];
  uint32_t BRR;
} GPIO_struct;
```

```
GPIO_struct * GPIO_portB = (GPIO_struct *) 0x40020400;
GPIO_portB->MODER = 0xFFFFFFFF;
```

```
118                          GPIO_struct * GPIO_portB = (GPIO_struct *) 0x40020400;
08000192: 0x0000054b   ldr     r3, [pc, #20]    ; (0x80001a8 <main+44>)
08000194: 0x00007b60   str     r3, [r7, #4]
119                          GPIO_portB->MODER = 0xFFFFFFFF;
08000196: 0x00007b68   ldr     r3, [r7, #4]
08000198: 0x4ff0ff32   mov.w   r2, #4294967295
0800019c: 0x00001a60   str     r2, [r3, #0]
```

✓ The compilation of the code shows the usage of ldr, str and mov instructions

How to Write Shift

Preferred version

```
// initilisation PC2
PC_DDR |= 1<<2; // pin2 in output mode
PC_CR1 |= 1<<2; // pin2 in push-pull
```

Instead of

```
PC_DDR |= 0x04;
PC_CR1 |= 0x02
```

Masking in C (1/2)

Masking bits to 1 :

```
PA_ODR=PA_ODR | 0b00011000; // set bits 3 and 4 of PA_ODR
```

or more readable

```
PA_ODR |= 1<<4| 1<<3;
```

Masking bits to 0 :

```
PA_ODR=PA_ODR & 0b11100111;// reset bits 3 and 4 of PA_ODR
```

Or more readable

```
PA_ODR &= ~(1<<4 | 1<<3)
```

```
#define SetBit(var,place)      (var|=(1<<place))
#define ClrBit(var,place)      ….
```

Masking in C (2/2)

Querying Status bit:

```
var = (PA_IDR & (1<<12)) >> 12; // Query value of bit 12 of value called PA_IDR
```

Toggling a bit :

```
PA_ODR=PA_ODR ^ 0b00011000; // toggle bits 3 and 4 of PA_ODR
```

or more readable

```
PA_ODR ^= (1<<4| 1<<3);
```

ISEN ALL IS DIGITAL! OUEST | yncréa

#define Keyword

In C programming language, the #define directive allows the definitions of macros within the code. These macros definitions allow constant values to be declared for use throughout the code

Macro definitions are not variables and cannot be changed by the program like variables. It is used when creating constants that represent numbers, strings or expressions

Syntax

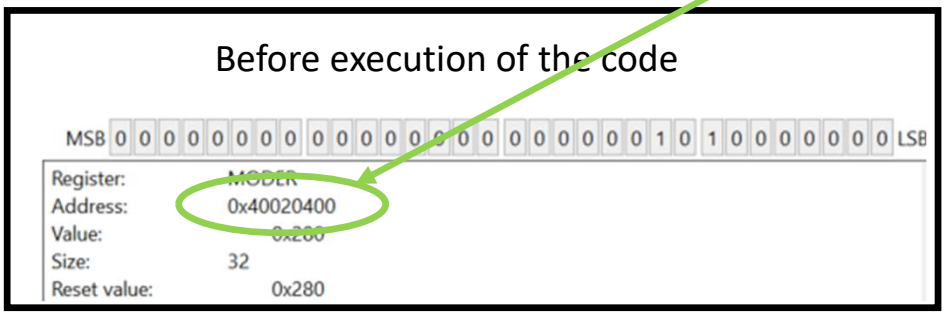
	#define	CNAME	value	
Or	#define	CNAME	(expression)	(called Macros)

How to Write Shift Improvement

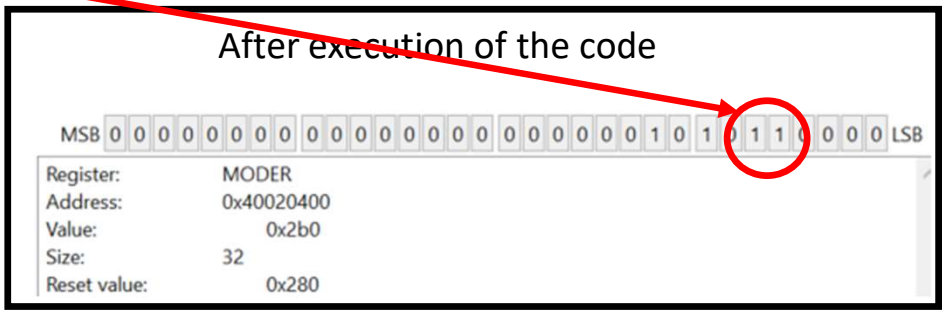
```
#define GPIO_Pin2 (uint32_t) 1<<4| 1<<5
#define PortBModeR            0x40020400

GPIO_struct * GPIO_portB = (GPIO_struct *) PortBModeR;
GPIO_portB->MODER |= GPIO_Pin2;
```

GPIO_struct is a pointer to a GPIO_Port at the address 0x500A (PortC)

Before execution of the code



After execution of the code



#define are normally written in an header file. The C code is then easily reusable for a different microcontroller. Only the header file would be different with updated data.

Interrupts

The vector table is declared in the table vector in the start-up file.

```
182   .word USART3_IRQHandler
183   .word EXTI15_10_IRQHandler
184   .word RTC_Alarm_IRQHandler
185   .word USB_FS_WKUP_IRQHandler
```

```
/**
  * @brief This function handles EXTI line[15:10] interrupts.
  */
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

The interrupt handler must have the name written in the vector table

This name is standardized by ARM for compatibility between platforms and vendors

Embedded Programming Guidelines

- ✓ Space memory is limited
- ✓ Be rigourous
- ✓ Limited size for variables
- ✓ Multiplication and division by power of 2 replaced by offset/shift
- ✓ Make pre-calculated tables of values to replace trigo calculations
- ✓ Avoid floating if possible.

C requirement

Opérateurs arithmétiques		
Opération	Signe	Exemple
addition	+	s = a + b
soustraction	-	d = a - b
multiplication	*	p = a * b
division	/	q = a / b
Modulo (reste de la division)	%	r = a % b

C requirement

Opérateurs bit à bit *(bitwise)*		
Opération	Signe	Exemples
et *(and)*	&	a = 0 1 0 1 b = 0 1 1 0 a&b = 0 1 0 0
ou inclusif *(or)*	|	a = 0 1 0 1 b = 0 1 1 0 a|b = 0 1 1 1
ou exclusif *(xor)*	^	a = 0 1 0 1 b = 0 1 1 0 a^b = 0 0 1 1
complément (inversion des bits)	~	a = 0 1 0 1 ~a = 1 0 1 0
décalage à gauche de n bits	<<	b = a << n
décalage à droite de n bits	>>	b = a >> n

C requirement

Opérateurs d'affectation		
Opération	**Signe**	**Exemples et équivalences**
affectation simple	=	lvalue = expr;
multiplication et affectation	*=	x *= 3; x = x * 3;
division et affectation	/=	x /= 3; x = x / 3;
modulo et affectation	%=	x % = 3; x = x % 3;
addition et affectation	+=	x += 3; x = x + 3;
soustraction et affectation	-=	x -= 3; x = x - 3;
décalage gche et affectation	<<=	x <<= 3; x = x << 3;
décalage dte et affectation	>>=	x >>= 3; x = x >> 3;
et bit à bit et affectation	&=	x &= 3; x = x & 3;
ou bit à bit et affectation	|=	x |= 3; x = x | 3;
xor bit à bit et affectation	^=	x ^= 3; x = x ^= 3;

C requirement

Post et pré incrémentations et décrémentations

Incrémenter une variable signifie lui ajouter un. **Décrémenter** : lui soustraire un.
Pré incrémenter une variable signifie que lorsque l'ordinateur la récupère en mémoire, il l'incrémente d'abord avant de s'en servir.
Post incrémenter signifie que l'ordinateur récupère la variable en mémoire, l'utilise telle quelle dans l'opération demandé, puis, l' incrémente après l'opération avant de la restituer, incrémentée, en mémoire.

Dans les exemples : on suppose que la valeur de la variable "**a**" est, au départ, **3**.

Nom	Signe	Exemples
Pré incrémentation	++ a	(5 * ++a) égale 20; ensuite a égale 4;
Post incrémentation	a ++	(5 * a++) égale 15; ensuite a égale 4;
Pré décrémentation	-- a	(5 * --a) égale 10; ensuite a égale 2;
Post décrémentation	a --	(5 * a--) égale 15; ensuite a égale 2;

C requirement

Opérateurs booléens d'assertion		
Opérateurs	Assertion	Opérandes a et b
a == b	a égale b	numériques caractères
a != b	a différent de b	numériques caractères
a < b	a inférieur à b	numériques
a <= b	a inférieur ou égal à b	numériques
a > b	a supérieur à b	numériques
a >= b	a supérieur ou égal à b	numériques

Opérateurs	Opération	Opérandes a et b
a && b	a **et** b	booléens
a || b	a **ou** b	booléens
!a	**non** a	booléen