Le Langage C

(C) Achille Braquelaire

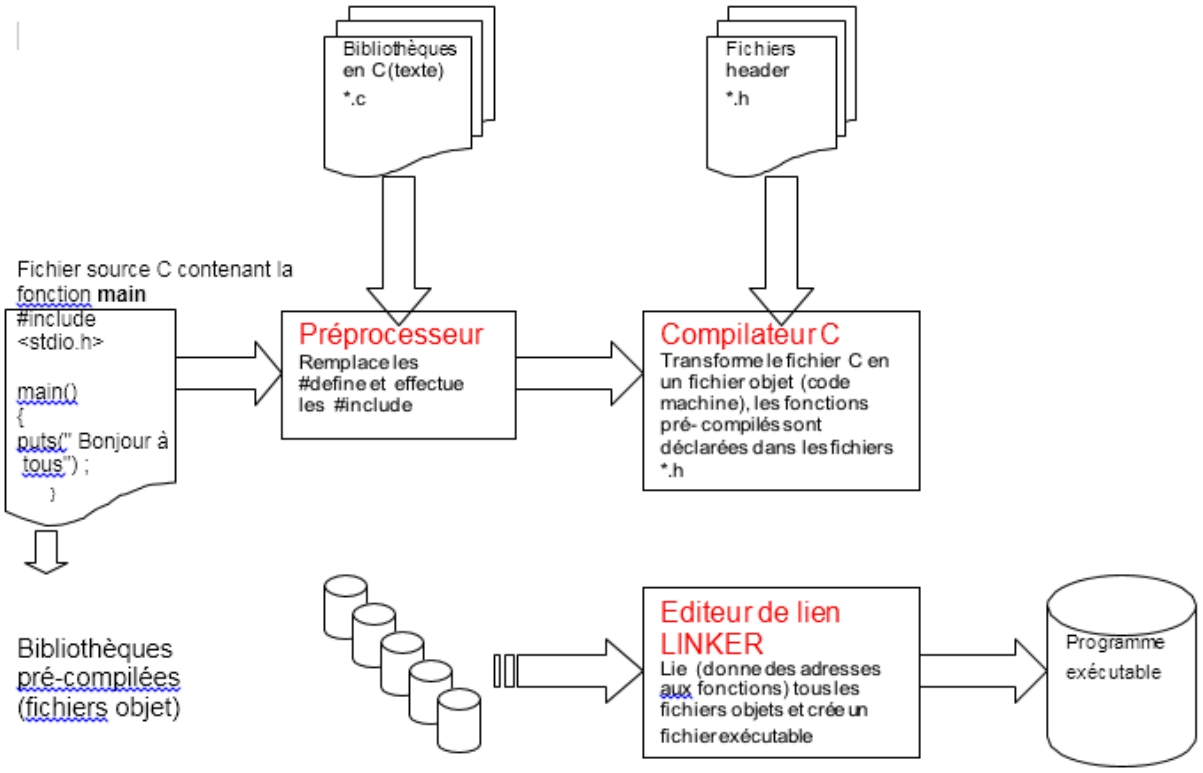
Lycée E. Branly **BTS SNEC** **Année 2016-2017**

SOMMAIRE

Vous pouvez tester le code en utilisant un compilateur en ligne :
http://www.tutorialspoint.com/compile_c_online.php

I. LES BASE DE LA PROGRAMMATION

1. Organisation générale d'un compilateur C



2. La syntaxe du C

```
#define TVA 20.0

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

float calc_ttc(float prix)
{
  float r;
  r=prix*(1.0+TVA/100.0);
  return(r);
}

void main()
{
  char s[20];
  float pht;

  do
  {
    puts("donnez le prix HT");
    pht=atof(gets(s));
    printf("Prix TTC en Euros : %f \n",calc_ttc(pht)); /
  }
  while (pht!=0.0); // Arrêt lorsque le prix entré est nul
}
```

Synonyme, TVA est remplacé par 20.0 dans tout le fichier par le pré processeur avant compilation

Librairies standards : les fichiers « header » *.h contiennent en général des équivalences ou les prototypes des fonctions précomplilées ici : stdio pour printf, gets, puts et stdlib pour atof

Fonction (ou sous programme) : en C il 'y a que des fonctions
Entrée de la fonction : prix de type réel
La fonction retourne un réel
r est une variable locale car déclarée dans la fonction, elle n'existe que lors de l'exécution de la fonction.

La fonction main est le point d'entrée de tous programmes en langage C qui est de type void car elle ne retourne aucune valeur (peut se trouver int main())

Variables visible dans toute la fonction main()

atof convertit une chaine en réel

Commentaire débutant par //

puts affiche uniquement une chaîne de caractère tandis que printf permet de formater le texte. \n permet d'aller à la ligne

3. Les commandes et commentair

En C, les lignes de commandes se terminent par un point-virgule (;), il y a des exceptions :

- #define
- Les fonctions : void **setup**() {
- Les boucles : while (compteur < 10)
- …

Pour commenter vos programmes, deux solutions :

- // commentaire sur une seule ligne
- /* commentaire sur
  plusieurs lignes
  */

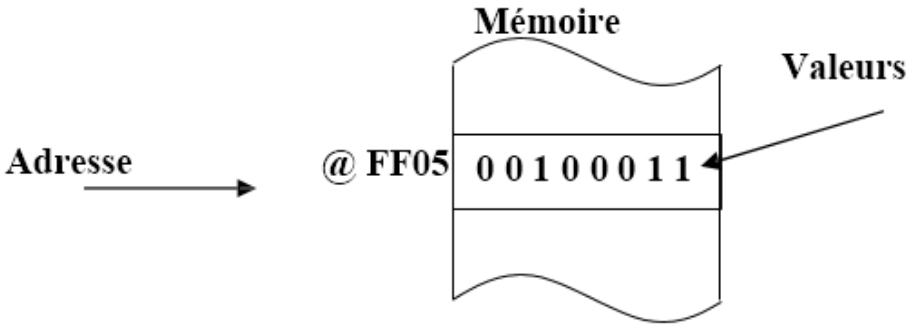
Ce dernier type de commentaire peut être utilisé pour ne pas exécuter temporairement une partie du code :

II. LES VARIABLES

1. Définition

- Une variable sert à stocker la valeur d'une donnée
- Une variable désigne un emplacement mémoire dont le contenu peut changer au cours d'un programme (d'où le nom)
- Chaque emplacement mémoire a un numéro qui permet d'y faire référence de façon unique : c'est l'adresse mémoire de cette cellule.

2. Représentation mémoire :



3. Les types de données :

Type	Nombre d'octets	Plage de valeurs	Utilisation
char	1	-128 (2^7) à 127 (2^7-1)	Représente un seul caractère (code ASCII) et peut être utilisé pour un « petit » entier compris entre -128 et 127.
unsigned char	1	0 à 255 (2^8-1)	Représente un seul caractère (code ASCII) ou un entier compris entre 0 et 255.
short	2	-32768 (-2^{15}) à 32767 (2^{15}-1)	Entier positif ou négatif.
unsigned short	2	0 à 65535 (2^{16}-1)	Uniquement les entiers positifs appelés « non signés »
int	2 (µp 16 bits) 4 (32 bits)	-32768 (-2^{15}) à 32767 (2^{15}-1)	Entier positif ou négatif.
unsigned int	idem	0 à 65535 (2^{16}-1)	Uniquement les entiers positifs appelés « non signés »

long	4	-2147483648 (-2^{31}) à 2147483647 ($2^{31}-1$)	Entier positif ou négatif de plus grande plage de valeurs
unsigned long	4	0 à 4294967295 (2^{32})	Idem que long, mais uniquement les entiers positifs, appelés « non signés »
float	4	3.4028235E+38 à -3.4028235E+38	Nombre réel
double	4/8	Idem float ou 3.4028235E+308 à -3.4028235E+308	Pour l'Arduino, double est synonyme de float.

4. Déclaration

Une variable doit être déclarée avant d'être utilisée. On peut lui donner une valeur initiale. Par défaut, la valeur est zéro.
On peut lister plusieurs variables de même type sans répéter le type et le ; final.

```
int vit= -4;
char i, j, toto ;
```

5. Cast : conversion de type de donnée

Une transformation peut être forcée par un **cast**

Conversion int en float	Conversion float en int

`float x ;`
`int a=5 ;`
`x=(float)a ; /* x vaudra 5.0 */`
	`float x=5.6 ;`
	`int a ;`
	`a=(int)x ; /* a vaudra 5*/`

6. Les constantes

a. Avec #define

```
#define pi 3.14
#define LED 13
```

Attention : ce ne sont pas des déclaration de variables mais le pré-processeur qui va remplacer les occurrences de « pi » et « LED » par 3.14 et 13

b. Avec const

```
const float pi=3.14;
const int LED=13;
```

Ce sont réellement des constantes qui sont dans la ROM (dans la RAM en lecture seule sur un PC) et ne sont donc pas modifiables.

7. Les tableaux

a. Exemples de déclarations

```
int tab1[5];                    // Déclaration d'un tableau de 5 entiers
char tab2[3] = {1,2,3}; // Déclaration d'un tableau de 3 types char
                        // et initialisation de ce tableau
char texte1[ ]="Bonjour";     // Déclaration d'un tableau de types char
                        // et initialisation de ce tableau avec les codes ASCII
                        // de la chaîne de caractères utilisée.
```

b. Remarques

- L'initialisation du tableau texte1 remplit la mémoire de la manière suivante :

'B'	'o'	'n'	'j'	'o'	'u'	'r'	'\0'

'B', 'o', 'n', 'j', 'o', 'u', 'r' représentant les codes ASCII respectifs des caractères du tableau.
Le caractère '\0' étant le caractère de code ASCII 0, indiquant la fin de la chaîne de caractère.
Ce tableau comprend donc non pas 7 caractères, mais 8 caractères.

- Comme les numéros des cases (index) commencent par 0, dans `char tab2[3] = {1,2,3}; ,` tab2[0]=1, tab2[1]=2, et tab2[3]=3

8. Les chaines de caractères

a. Déclaration

Une chaîne de caractères (string en anglais) est une suite de caractères stockée dans un tableau de char et terminée par le caractère '\0'.

Le '\0' marque la fin de la chaîne quelle que soit la taille du tableau mais le nombre total de caractères dans la chaîne ne peut pas dépasser la taille du tableau.

Par exemple :

```
1.    char Str1[15];
2.    char Str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o'};
3.    char Str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};
4.    char Str4[ ] = "arduino";
5.    char Str5[8] = "arduino";
6.    char Str6[15] = "arduino"
```

1. Str1 : déclare un tableau de char sans initialisation.
2. Str2 : déclare un tableau de char (avec un élément supplémentaire) et le compilateur va rajouter le caractère null obligatoire
3. Str3 : ajout explicite du caractère null
4. Str4 : initialise avec une chaîne de caractère entre guillemets (double), le compilateur va dimensionner le tableau pour l'ajuster à la taille de la chaîne et va ajouter le caractère null
5. Str5 : initialise le tableau avec une taille explicite et une chaîne de caractères.
6. Str6 : initialise le tableau et laisse un espace supplémentaire pour une chaîne de caractères plus importante.

b. Fonctions de manipulations

Des fonctions, se trouvant dans <string.h> permettent de manipuler ces chaînes de caractères :

Nom de la fonction	Utilité	Exemple
strlen	Renvoie la taille	`strlen(nom))`
strcpy	Copie	`strcpy(nom2,nom);// Destination en premier, strncpy copie n caractères`
strncpy	Copie les n premiers caractères	`strncpy(nom2,nom,5);// Destination en premier, strncpy copie n caractères`
strcat	Ajoute	`strcat(nom,nom2); // Ajoute nom2 à la fin de nom`
strncat	Ajoute n caractères	`strncpy(nom,nom2,5); // Ajoute les 5 premierss caractères de nom2 à la fin de nom`
strcmp	Compare deux chaines	`if (strcmp(nom,"Nestor")==0) // strcmp = 0-> égaux`
strncmp	Compare les n premiers caractère de deux chaines	`if (strncmp(nom,"Nestor",2)==0) // strcmp = 0-> égaux, strncmp compare les deux premiers caractères`

Il existe également des fonctions de recherche :

Nom de la fonction	Utilité
strchr	Trouve un caractère dans la chaîne
strcspn	Trouve un ou plusieures caractères dans la chaîne
strpbrk	Recherche un ensemble de caractères dans une chaîne
strrchr	Trouve la dernière occurrence d'un caractère dans une chaîne

strspn	Trouve la longueur du segment initial d'une chaîne contenant tous les caractères d'un masque donné
strstr	Trouve la première occurrence dans une chaîne
strtok	Coupe une chaîne en segments

9. Les pointeurs

a. En mémoire

Un pointeur est une variable qui contient l'adresse d'une autre variable. Selon le type de donnée contenu à l'adresse en question, on aura un **pointeur d'entier, de float, double, char**, ou de tout autre type. En accédant à cette adresse, on peut accéder indirectement à la variable et donc la modifier.

Exemple : `short *pt ;` `//pt est un pointeur sur un entier.`

- *pt* vaut 0xF7B3 et *pt 372 (décimal)

F Variable *pt* de type
c pointeur sur un short
eur d'un pointeur il suffit de rajouter une étoile * devant l'identificateur
dentificateur de pointeur veut dire valeur pointée par.
*pt renvoit la valeur pointée par pt.

Exemple :

```
#include<stdio.h>
main()
{
  int *p;
  int i=1234;
  //la valeur du pointeur p prend l'@ de i
  p=&i;
  //affiche la valeur de la variable p, soit l'@ de i
  printf("%X\n",p);
  //affiche la valeur pointée par p
  printf("%d\n",*p);
}
```

b. Opérateurs * et &

Dans l'exemple `int i=1234;` i est le nom d'une variable. Pour avoir son pointeur il faut ajouter un **&** .

Ainsi :

- **&i** est l'adresse de la variable **i**.
- **&i** pointe vers **i**

En ce qui concerne `int *p;` c'est une définition d'un pointeur vers une variable de type `int`.

c. Pointeur et tableau

Dans l'exemple

```
int tab1[5];                    // Déclaration d'un tableau de 5 entiers
```

Le pointeur vers le tableau `tab1` n'est pas ~~`&tab1` ni `&tab1[0]`~~ mais son nom `tab1`.

Le pointeur de `int tab1[5];` est **tab1**

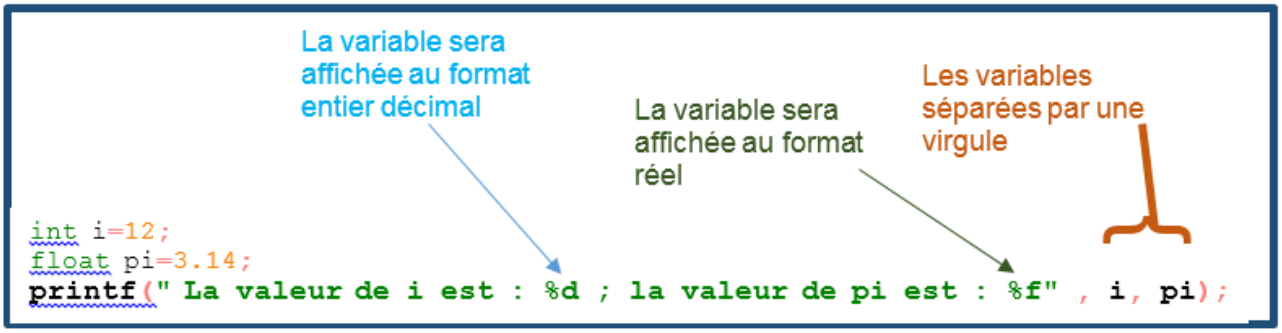
III. LES FONCTIONS D'AFFICHAGES ET DE SAISIE

Les fonctions décrites dans cette section se trouvent dans la bibiothèque standard :

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

1. Afficher avec printf

Elle permet de réaliser des sorties formatées de messages et/ou de valeurs des variables sous différents formats.



Les formats disponibles sont à Formats utilisés par printf et scanf

2. Caractère pouvant être affiché par printf

Code de	Signification
\n	Nouvelle ligne
\a	Bip code ASCII 7
\r	Retour chariot
\t	Tabulation
\f	Saut de page
\\	Antislash
\ "	Guillemet
\'	Apostrophe

3. Puts et putchar

puts	putchar
Affiche une chaîne de caractères	Affiche un seul caractère

`puts("Hello, World!");`
`// identique à printf mais dans \n`
`printf("Hello, World!\n");`
	`putchar('A');`

4. Lire avec scanf

Elle permet de saisir des valeurs de variables formatées à partir du clavier. Comme printf elle est composée d'un format et des identificateurs de variables à saisir. A la différence de printf, le format ne peut contenir de texte, il est juste composé du format des valeurs à saisir.

scanf (format, liste d'adresses) : permet de saisir les données au clavier	
Exemple 1	**Exemple 2**

`scanf("%d",&a);`
`// Attend la saisie d'un entier`
	`scanf("%d%d%f",&a,&b,&c);`
	`/* Attend la saisie de deux entier puis d'un float`
	`Tous seront séparés par des espaces */`

Remarque:

- Le symbole & est obligatoire devant les identificateurs car *scanf* attend des adresses et non des valeurs.
- Les formats disponibles sont à Formats utilisés par printf et scanf

5. Gets et getchar

gets	getchar
Lit une chaîne de caractères	**Lit un seul caractère**

`char nom[20];`
`printf("Quel est votre nom ? : ");`
`gets(nom);`
`printf("Bonjour %s \n",nom);`
	`char car;`
	`car=getchar();`
	`putchar(car);`

6. Formats utilisés par printf et scanf

Ecriture	*Format d'affichage*
%d	*Entier Décimal*
%o	*Entier Octal*
%x	*Entier Hexadécimal*
%u	*Entier Non Signé*
%c	*Caractère*
%s	*Chaîne de caractères*
%f	*Flottant*

IV. LES OPERATEURS

1. Définition

Fonctions	O	Description	Exemples
Identificateurs	()	Appel de fonction	
	[]	Indice de tableau	tableau[3]=5;
opérateurs unaires	!	Négation logique (NOT)	b=!a; (si a>0 => b=0, si a=0 =>b=1)
	~	Complément binaire bit à bit	b=~a
	-	Moins unaire	b=-a;
	+	Plus unaire	b=+a;
	++	Préincrément ou postincrément	b=a++; (b=a puis a=a+1)
	--	Prédécrément ou postdécrément	b=a--; (b=a puis a=a-1)
	&	Adresse de	b=&a; (b égale l'adresse de a)
	*	Indirection (adressage indexé)	b=*a; (b=contenu de l'adresse de a)
opérateurs binaires	*	Multiplication	c=a*b;

	/	Division	c=a/b;
	+	Plus binaire	c=a+b;
	-	Moins binaire	c=a-b;
	<<	Décalage à gauche	c=a<<b; (a est décalé b fois à gauche)
	>>	Décalage à droite	c=a>>b; (a est décalé b fois à droite)
	&	ET entre bits	c= a & b; (ET logique bit à bit)
	^	OU exclusif entre bits	c= a ^b;
	|	OU entre bits	c= a|b;
Tests	<	Strictement inférieur	if a < b
	<=	Inférieur ou égal	if a >= b
	>	Strictement supérieur	if a > b
	>=	Supérieur ou égal	if a >= b
	==	Egal	if a ==b (si a est égale à b)
	!=	Différent	if a != b
	&&	ET logique	if ((a=5) && (b=2))
	||	OU logique	if ((a=5) ||(b=2))
	?:	Condition	z=(a>b)?a:b (Si a>b a z=a sinon z=b)
Affectation	=	Affectation simple	a=b; (a prend la valeur de b)
Auto-affectations	*=	Affectation produit	a*=2 (a=a*2)
	/=	Affectation quotient	a/=2 (a= a/2)
	%=	Affectation reste	a%=2 (a= le reste de a/2)
	+=	Affectation somme	a+=2 (a=a+2)
	-=	Affectation différence	a-=2 (a=a-2)
	&=	Affectation ET entre bits	a&=5 (a=a&5)
	^=	Affectation OU EX entre bits	a^=5 (a=a^5)
	|=	Affectation OU entre bits	a|==5 (a=a|=5)
	<<=	Affectation décalage gauche	a<<=5 (a=a<<5)
	>>=	Affectation décalage droite	a>>=5 (a=a>>5)

2. Exemple

```
#include <stdio.h>
char a, b, c;
int i1, i2, i3, i4; long l1, l2; float f1, f2, f3; double d1;

main()
{
i1 = i2 = i3 = 11;      /* voila des exemples d'affectations multiples */
a = b = c = 40;
f1 = f2 = f3 = 12.987;

/* Opérations arithmétiques de base */
i3 = -i2;   /* donne i3 = -11 */
i3 = i1 + i2;      /* donne i3 = 22 */
i3 = i1 - i2;      /* donne i3 = 0 */
i3 = i1 * i2;      /* donne i3 = 121 */
i3 = i1 / i2;      /* donne i3 = 1 */
i3 = 15+(i1 = 4*i2);          /* donne i1= 44 puis i3 = 59 */
i4 = i3 / i2;      /* donne i4 = 5 */
i4 = i3 % i2;      /* % = modulo donc i4 = 8 */
i4 = i3 >> 2;      /* i4 = i3 apr,s 2 décaléges … droite (*4) */
i4 = i3 << 3;      /* i4 = i3 apr,s 3 décaléges … gauche (/8) */

/* Opérations arithmétiques de niveau 2 : automodifications */
a = a + 1;  /* incrémentation de a */
a += 1;     /* idem */
a++;  /* idem APRES utilisation de a */
++a;  /* idem AVANT utilisation de a */
a = a - 1 ; /* décrémentation de a */
```

```
a -= 1;      /* idem */
a = 40;
b = a--;     /* b = 40 puis a = 39 */
b = --a;     /* a = 38 puis b = 38 */
a += 10;     /* a+10, a = 48 */
b -= 5;      /* b-5, b = 33 */
f1 *= 1.5;   /* f1 * 1.5, f1 = 8.658 */
f2 /= 2.2;   /* f2 / 2.2, f2 = 5.9032 */
a <<= 1;     /* a décalé 1 fois à gauche a = 24 */
b >>= 2;     /* b décalé 2 fois à droite b = 132 */

/* Opérateurs Logiques de niveau 1 */
b = 15; c = 12;
a = b & c;   /* a = b ET c donc a = 10 */
a = b | c;   /* a = b OU c donc a = 17 */
a = b ^ c;   /* a = b OU EXCLUSIF c donc a = 7 */
a = ~b;      /* a = NOT b donc a = $EF */

/* Opérations de niveau 3 : Affectation conditionnelle */
/* syntaxe générale : (Test) ? action_si_oui : action_si_non; */
i1 = (i2 > 3) ? 2 : 10;  /* Cette instruction doit se lire:
si i2 est supérieur à 3 alors i1 devient 1 sinon, i1 devient 10 */
c = (a > b) ? a : b;    /* c est le maxi d'entre a et b */
c = (a > b) ? b : a;    /* c est le mini d'entre a et b */

c = (a >= 0) ? a : -a;  /*  c est la valeur absolue de a */

/* Opérateurs divers */
a = sizeof (f1);  /* sizeof retourne la taille en octet de */
b = sizeof (l1);  /* l'expression paramètre */
f2 = int(15) / int(4);  /* conversion de type : CAST. En absence des */
f2 = int (15/4);  /* commandes de cast, f2 vaudrait 3.75, ici 3 */

/* Comparaisons logiques de base */
if (i1 == i2) i= -13;   /* puisque i1 = i2, i3 deviendra -13 */
if (i1 > i3) a = 'A';         /* donne 'A' (65) … la variable a */
if (!(i1 > i3)) a = 'B';      /* puisque i1 >i3, a ne change pas */
if (b <= c) f1 = 0.0;   /* puisque b = c alors f1 devient 0 */
if (f1 != f2) f3 = i3/2;      /* comme f1 <> f2, f3 devient 6.5 */

/* Comparaisons logiques de niveau 2 */
/* bas,es sur le fait qu'une valeur vraie est diff,rente de 0 et qu'une valeur
fausse est nulle */
if (i1 = (f1 != f2)) i3 = 111;      /* comme f1 <> f2, le résultat du test
est vrai, le résultat logique du test vrai est affecté à i1 et i3 change */
if (i1) i3 = 222;       /* i1 est vrai donc i3 change */
if (i1 != 0) i3 = 333;  /* i1 vrai donc non nul donc i3=333 */
if (i1 = i2) i3 = 444;  /* i2 recoit i1, cette valeur non nulle
donc vraie  i3 = 222 */

/* Comparaisons logiques niveau 3 : Tests multiples */
i1 = i2 = i3 = 77;
if ((i1 == i2) && (i1 == 77)) i3 = 33;    /* && = ET logique: i3 change */
if ((i1 > i2) || (i3 > 12)) i3 = 22;      /* || = OU logique: i3 change */
if (i1 && i2 && i3) i3 = 11;  /* tous non nuls donc i3 change */
if ((i1=1)&&(i2=2)&&(i3=3)) i3 = 44;      /* Attention, affectations des
variables donc i3 change */
if ((i1==2)&&(i2=3)) i3 = 55; /* i1 <> 2 donc i3 ne change pas */
}
```

V. LES STRUCTURES DE CONTROLE

1. La boucle pour

Cette boucle est utilisée lorsqu'on connait le nombre d'itérations avant même de l'appeler c'est-à-dire :

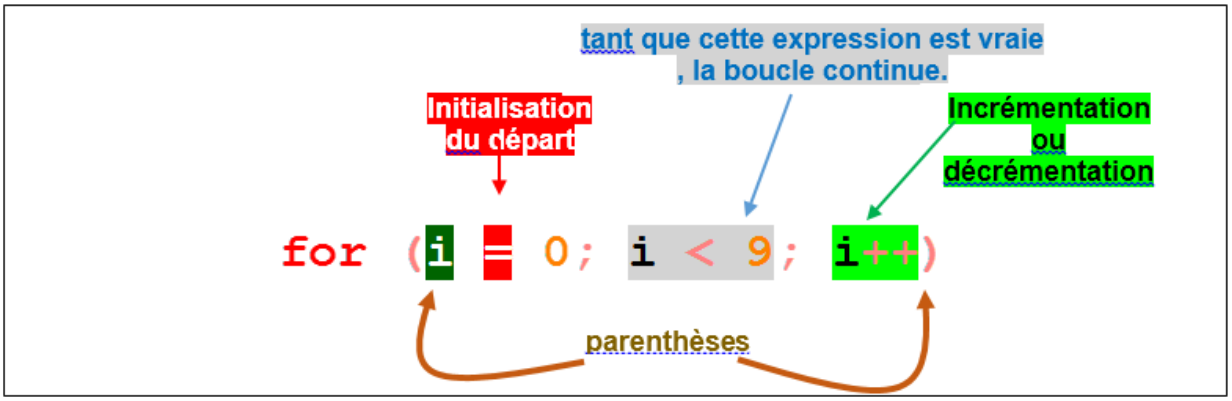
La boucle pour est utilisée lorsque l'on connait le nombre de fois que la boucle va être parcourue

a. Exemple

Algo	Langage C

Pour i variant de 0 à 9
afficher("coucou)
FinPour
	`int i ;`
	`for (i = 0; i < 9; i++)`
	`{`
	`printf("coucou\n") ;`
	`}`

Explication :



b. Autres exemples

Exemple 1	Exemple 2

`char car;`
`for(car='A';car!='Z'+1;car++)`
`printf("%c",car);`
	`char i;`
	`for (i = 0 ; i < 100 ; i++)`
	`{`
	`printf("i = %d\n", i); /*affiche les i de 0 à 99*/`
	`}`

c. Remarques

- Généralement, en informatique, on commence à compter à partir de zéro :
- `for (int i = 0; i < 9; i++)` est équivalent à `for (int i = 1; i <=10 ; i++)`

2. Tant que

Cette boucle est utilisée quand on ne connait pas le nombre de fois que la boucle doit être itérée.

a. Exemple

Algo	Langage C

Somme<-0
Tant que somme<100
faire somme=2*somme+1
	`Somme=0 ;`
	`while (somme < 100)`
	`{`
	`somme = 2 * somme + 1;`
	`}`

b. Remarques

- Attention : La condition se trouvant au début de la boucle, il faut s'assurer qu'elle soit bien vérifiée. Dans l'exemple ci-dessus, Somme est mis à zéro
- Il n'y a pas de point-virgule à la fin de la ligne contenant le `while`
- `while (1) {…}` est une boucle infinie.

3. La boucle faire … tant que

La condition n'est vérifiée qu'à la fin de la boucle : celle-ci sera exécutée au moins une fois.

a. Exemple

Algo	Langage C

Reponse : caractère
répéter
Afficher ("Voulez-vous sortir (O/N) ?")
lire reponse
tant que reponse ≠ 'O'
	`char caractere ;`
	`do`
	`{`
	`printf("Voulez-vous sortir (O/N) ?");`
	`scanf("%c", &caractere);`
	`}`
	`while (caractere != 'O');`

b. Remarque

- Attention aux accolades et au point-virgule à la fin de while.

VI. LES BRANCHEMENTS CONDITIONNELS

1. Si ... alors…(sinon)

a. Syntaxe

Condition du test : ==, <,>,<=,>=, !=,&&,|| …

Pour faire des choix : l'instruction if

Algo	Langage C

Si a > b
Alors
Afficher "A>B"
Sinon
Afficher "A>B"
FinSi
	`if (a > b)`
	`{`
	`printf("A>B \n");`
	`}`
	`else`
	`{`
	`printf("A<B \n");`
	`}`

b. Remarques

- il ne faut pas confondre en C, l'opérateur d'égalité **==** et celui de l'affection **=**
- *sinon* est optionnel,
- il est possible d'imbriquer des if avec else if

2. Selon … cas …

Cette structure remplace une série de if consécutifs avec l'avantage d'une meilleure lisibilité.

a. Syntaxe

Algo	Langage C

Saisir jour
Selon jour :
Cas 1 : afficher "lundi"
Cas 2 : afficher "mardi"
…
Cas 7 : afficher "dimanche"
	`int jour ;`
	`scanf("%d", &jour);`
	
	`switch (jour) {`
	`case 1 : printf("lundi\n"); break;`
	`case 2 : printf("mardi\n"); break;`
	`…`
	`case 7 : printf("dimanche\n");`
	`break;`
	`default : printf ("Erreur : le jour doit être compris entre 1 et 7\n");`
	`}`

b. Remarques sur la syntaxe :

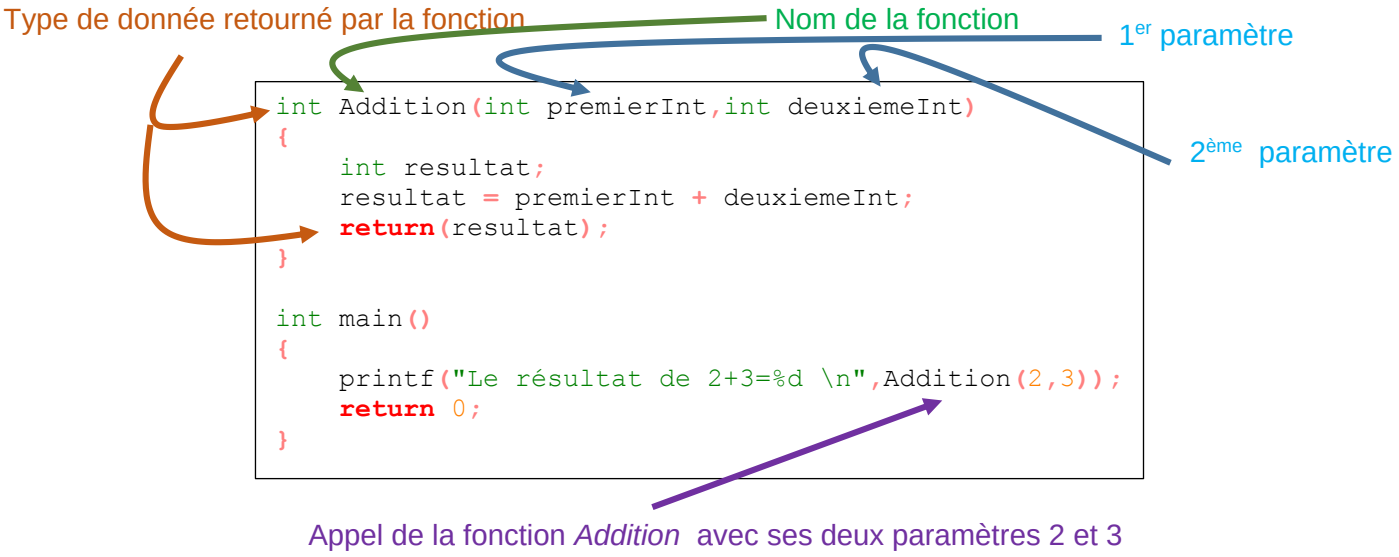
- Tous les cas sont dans une même accolade {…}
- Chaque ligne commence par **case x :** (les deux points sans le point-virgule)
- **x** doit être de type entier
- Mais comme 'A' est le code ascii de la lettre A, il est possible d'écrire **case 'A' :**

- Faire attention de bien mettre l'instruction **break**; qui fait sortir du **switch – case**, sinon on continue jusque **default** !!
- Si aucune valeur n'existe, les instructions sous **default** sont exécutées

VII. LES FONCTIONS

1. But - Principe

Une fonction est une instruction isolée du reste du programme, qui possède un nom, et qui peut être appelée par ce nom à n'importe quel endroit du programme et autant de fois que l'on veut.



2. Exemples

```
#include <stdio.h>

void Affiche10Fois() //Fonction sans paramètre ne retournant pas de valeur
{
    int i;
    for( i = 0; i < 9; i++) printf("coucou\n") ;
}

int Aleatoire() // Fonction sans paramètre, retournant une valeur
{
  srand(time(NULL)); // indispensable pour obtenir un nombre différent
                     //à chaque fois
  return (rand() % 20 );  // % 20 représente le reste de la division entière par
20
}

/* Fonction utilisant un ou plusieurs paramètres
    ne retournant pas de valeur */
void AfficherAddition(int i, int j)
{
    printf("L'addition entre %d et %d est %d \n",i,j,i+j);
}

//Fonction utilisant un ou plusieurs paramètres, retournant une valeur
float Discriminant(float b, float a, float c)
{
    float delta = b*b - 4*a*c;
    return(delta);
}
void main()
{
```

```
    int nombre;
    Affiche10Fois();
    nombre=Aleatoire();
    printf("un nombre aléatoire entre 0 et 20 : %d\n",nombre);
    AfficherAddition(2,3); // Affiche L'addition entre 2 et 3 est 5
    nombre = Discriminant(6,3,2);
    printf("Le Discriminant est %d \n",nombre); // Affiche L'addition entre 2 et 3
est 5

}
```

3. Utilisation de fonctions

a. Cas des bibliothèques standards

Pour pouvoir utiliser `printf(…)` il faut mettre au début du code `#include <stdio.h>` pour que le compilateur sache que cette fonction est définie « ailleurs ».

b. Programmation modulaire

Dans un projet plus conséquent et pour une meilleure visibilité mais aussi pour simplifier la maintenance du code, vous serez amené à créer plusieurs fichiers : un *main.c* dans lequel on trouvera `void main()`, *gestionPorts*.c où vous mettrez les fonctions permettant de gérer les ports d'un microcontrôleurs, un *LDC*.c pour l'écran LCD,*reseaux.c* …

Un prototype de fonction doit être déclaré avant l'utilisation de la fonction, il faut donc regrouper tous les prototypes d'un module (fichier xxx.c) dans un en-tête (<xxx.h>). Ce dernier n'a plus alors qu'à être inclus dans le code qui utilise ces fonctions.

Reprenons le programme précédent avec la compilation modulaire et trois fichiers main.c, MesFonctions.c et MesFonctions.h :

MesFonctions.c

```
#include <stdio.h>
void Affiche10Fois()
{
    int i;
    for( i = 0; i < 9; i++)
printf("coucou\n") ;
}

int Aleatoire()
{
  srand(time(NULL));
  return (rand() % 20 );
}

void AfficherAddition(int i, int
j)
{
    printf("L'addition entre %d
et %d est %d \n",i,j,i+j);
}

float Discriminant(float b, float
a, float c)
{
    float delta = b*b - 4*a*c;
    return(delta);
}
```

MesFonctions.h

```
void Affiche10Fois();
int Aleatoire();
void AfficherAddition(int,int);
float Discriminant(float,float,float);
```

main.c

```
#include <stdio.h>
#include "MesFonctions.h"

void main()
{
   int nombre;
   Affiche10Fois();
   nombre=Aleatoire();
   printf("un nombre aléatoire
entre 0 et 20 : %d\n",nombre);
   AfficherAddition(2,3);
   nombre = Discriminant(6,3,2);
   printf("Le Discriminant est %d
\n",nombre);
}
```

Remarque :

Observer la différence entre la déclaration des fonctions et le prototype : les noms des variables a été supprimé et un « ; » a été ajouté à la fin de la ligne :

MesFonctions.c	MesFonctions.h
`void Affiche10Fois()`	`void Affiche10Fois() ;`
`int Aleatoire()`	`int Aleatoire() ;`
`void AfficherAddition(int i, int j)`	`void AfficherAddition(int, int) ;`
`float Discriminant(float b, float a, float c)`	`float Discriminant(float, float, float) ;`

VIII. STRUCT ET ENUM TYPEDEF

1. *struct*

struct est utilisé pour créer des types de variables avancées

a. Syntaxe

```
struct ma_structure {
    type1 champ1;
    type2 champ2;
...
    typeN champN;
} var1, var2, ..., varM;
```

b. Accès aux champs

```
struct complexe {
    int reel;
    int imaginaire;
} c;

int main(void) {
      c.reel = 1;
      c.imaginaire = 2;
      return 0;
}
```

2. *Les déclarations de type synonyme: typedef*

On a vu les types de variables utilisés par le langage C: char, int, float.
Le programmeur a la possibilité de créer ses propres types, il suffit de les déclarer en début de programme.

a. Synonyme de int

```
#include <stdio.h>
typedef int entier;     /* on définit un nouveau type "entier" synonyme de "int"
*/
entier a,b,c ;    /* On déclare 3 variables de type entier*/

/*Programme principal*/
main()
{
entier d=3 ;
}
```

b. Créer une équivalence de struct

L'exemple avec les nombres complexes peut s'écrire :

```
#include <stdio.h>
typedef struct {
      int reel;
      int imaginaire;
} NombreComplexe;

int main(void) {
      NombreComplexe c;
```

```
        c.reel = 1;
        c.imaginaire = 2;
        return 0;
}
```

3. Les énumérations avec enum

a. Utilité

Le langage C nous permet aussi de créer ce que l'on appel des énumérations. En fait il s'agit simplement de créer des constantes qui prendront des valeurs définies et pourront être utilisé dans le programme.

La construction enum permet de donner à ces cas numérotés des noms qui clarifient la documentation.

b. Equivalence avec #define

```
enum {toto,titi,tata} var=titi;
```

donne aux trois noms les valeurs 0,1,2 et initialise la variable var à la valeur titi (cela ne nous intéresse plus de savoir que c'est 1).

Avec enum	Avec #define

`enum {toto,titi,tata} var=titi ;`	`#define toto 1`
	`#define tata 2`
	`#define titi 3`
	`int var = titi;`

c. Avec typedef pour créer un synonyme

Un autre exemple :

```
enum {Stop,Avance,Recule,Tourne} etatRobot=Avance;
```

Un programme, par exemple, pilotera les déplacements d'un robot, on utilisera en général un **switch case** pour décider des mouvements des moteurs.

```
#include <stdio.h>
enum EnumEtat {Stop, Avance, Recule, Tourne};

typedef enum EnumEtat EtatRobot;

int main()
{
  EtatRobot Robot = Stop;

  switch (Robot)
  {
    case Stop : printf("Robot vaut Stop\n"); break;
    case Avance : printf("Robot vaut Avance\n"); break;
    case Recule : printf("Robot vaut Recule\n"); break;
    case Tourne : printf("Robot vaut Tourne\n"); break;
    default : printf("Erreur !\n"); break;
  }
  return 0;
}
```