Introduction à la programmation impérative en langage C

Historique du langage C

- Né en 1972 dans les laboratoires Bell AT&T
  - Auteurs : Brian KERNIGHAN et Dennis RITCHIE :
- A partir de 1974 : le langage C est utilisé pour la mise en œuvre du système d'exploitation UNIX
- Langage de programmation
  - 1989 Normalisation ANSI : C89
  - 1999 Normalisation ISO : C99
  - 2011 Mise à jour : C11
- Le langage C aujourd'hui :
  - Son utilisation dépasse celle du système UNIX.
  - Un des langages les plus utilisés.

Caractéristiques

- Langage de type impératif : basé sur l'exécution d'instructions
  - Les instructions sont exécutées dans leur ordre d'apparition
  - Utilisation de structures de contrôle pour répéter des instructions (boucles)
- Modulaire: peut être découpé en modules qui peuvent être compilés séparément
- Typé: tout objet C doit être déclaré avant d’être utilisé
- Portable: sur n'importe quel système en possession d'un compilateur C
- Pas d'entrées/Sorties : on doit utiliser les fonctions d'entrée-sortie de la bibliothèque standard de C

Compilation (1)

- Les **programmes source** en C sont des **fichiers contenant du texte** et enregistrés avec l'**extension .c**

- Un programme source en C doit respecter exactement la syntaxe du langage C.

- Pour écrire un programme source, le programmeur utilise un **éditeur de texte.** (Gedit, Emacs, Vi, NotePad, …)

- Ce programme source doit ensuite être traduit en langage machine. Le programmeur procède alors à la **compilation de son programme source** afin d'obtenir un **programme (fichier) exécutable**.

- Compilation avec le compilateur gcc de Gnu C :
  - Par défaut, lorsqu'on demande à gcc de compiler un fichier, il produit en sortie un exécutable nommé **a.out**.
  - l'option -o de gcc permet de nommer l'exécutable.

    gcc -Wall prog.c -o prog
  - l'option -Wall (W pour Warning) permet d'afficher les avertissements qui ne sont pas pas automatiquement affichés sur la console.

Compilation (2)

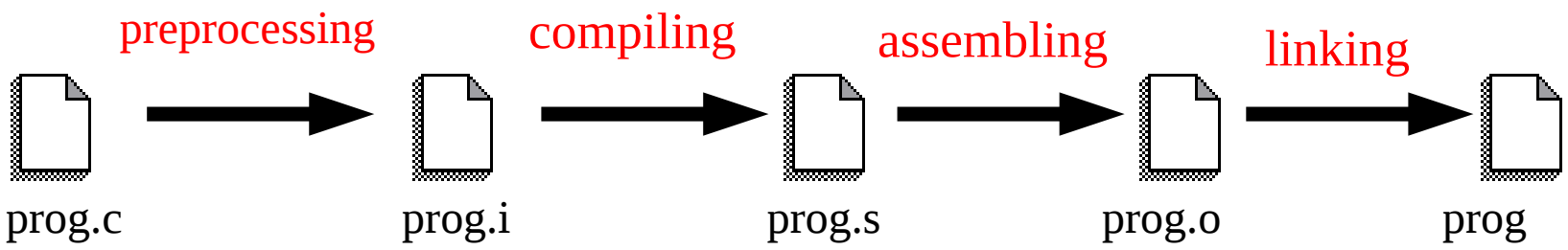
La phase de compilation procède en quatre temps :

(1) passage au pré-processeur (preprocessing)

(2) compilation en langage assembleur (compiling)

(3) conversion du langage assembleur en code machine (assembling)

(4) édition des liens (linking)



Les fichiers intermédiaires générés n'apparaissent pas dans le répertoire. Une fois que gcc a fini son exécution, les seuls fichiers qui apparaissent sont celui qui contient l'exécutable et le fichier source C.

Premier programme

```
#include <stdio.h>

void main()

{

    int somme, i;

    somme = 0 ;

   for(i=1; i<=20;i++)
   {
      somme = somme + i;
   }

   printf ("%d\n", somme);

}
```

le nom de la fonction principale, aussi appelée point d'entrée du programme.

déclarations de variables de type entier (cases mémoire pouvant contenir un entier)

instruction d'affectation de valeur à la variable somme

affiche à l'écran la valeur de l'entier contenu dans somme

Les accolades { et } entourent les instructions constituant le corps de la fonction main.

Structure minimale d'un programme

- Tout programme qui utilise « printf » doit inclure « stdio.h »

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

- Tout programme source C doit contenir une fonction « main » qui sera appelée lors de l'exécution du programme

```
int main() {
    }
```

- Instructions :
  - La fonction « main » contient des *instructions* qui sont exécutées séquentiellement (une par une).
  - En M1102, la fonction « main » se termine toujours par l'instruction : return 0;

Convention d'écriture

- Une seule instruction par ligne
- Utiliser des identificateurs significatifs
- Indenter le code
- Toujours commenter mais efficacement

Séparateurs et commentaires

- Les caractères espace, tabulation et retour-chariot séparent les entités lexicales.
- Les commentaires sont délimités par /* et */
- Le point-virgule termine une instruction ou une déclaration.
- La virgule sépare les éléments d'une liste (dont les paramètres).
- Les accolades délimitent les blocs ou les listes d'initialisation.
- Les crochets servent aux tableaux
- Les parenthèses servent : à la priorité ; aux appels de fonction.

Identificateurs

- Séquence de lettres, chiffres et de soulignés (symbole _)
- Le premier caractère est une lettre ou un souligné
- Sensible à la case (majuscule/minuscule)
- Attention : les identificateurs trop long peuvent être coupés

Eléments de base

- **Structures de données** :
  - Notion de type : entiers, réels, caractères, ...
  - Notion de variable

- Instructions :
  - Affectations de variable
  - Instructions Conditionnelles : **if** et **switch**
  - Boucles : **while**, **do … while** et **fo**r

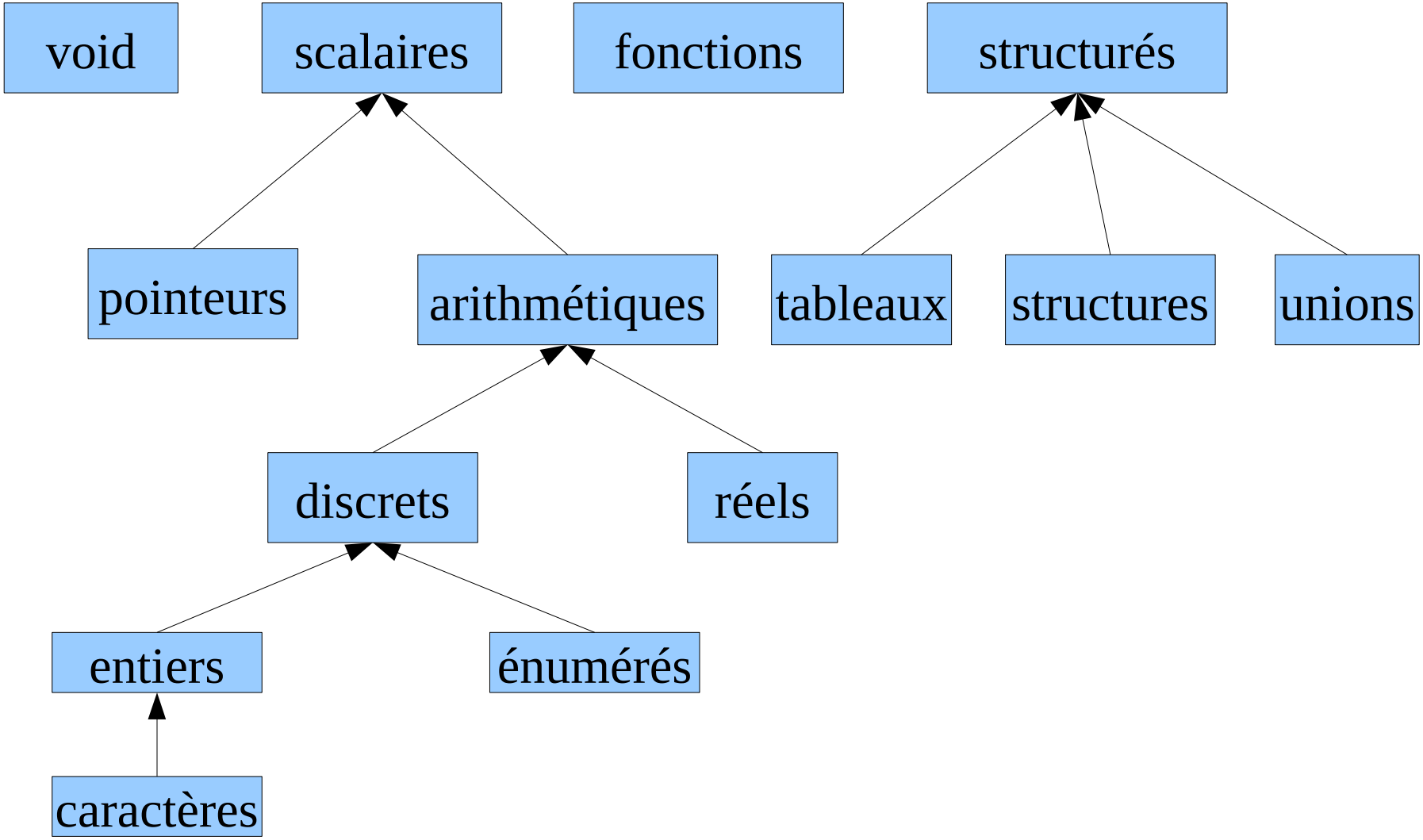
Notion de type

- Les données manipulées en C sont typées

- Pourquoi typer les données ?
  - Connaitre l'occupation mémoire de chaque donnée
  - Connaître l'ensemble d'opérations applicables sur la donnée

- Différents types :
  - **Types simples** : entiers, réels, caractères, énumérés, …
    - Entiers : int
    - Décimaux : float
    - Booléens (false ou true) : int (0 : faux, autre : vrai)
    - Caractère : char
  - **Types composites** : tableaux, structures, unions
  - **Pointeurs**

Types de C


void
scalaires
fonctions
structurés
pointeurs
arithmétiques
tableaux
structures
unions
discrets
réels
entiers
énumérés
caractères

Diverses représentations

- int x = 123
  - Hexadécimal (base 16) : Ox7B
  - Décimal (base 10) : 123
  - Octal (base 8) : 0173
  - Binaire (base 2) : 0111 1011

- Chiffres base 16 : 0 1 2 3 4 5 6 7 8 9 A B C D E F
- Chiffres base 8 : 0 1 2 3 4 5 6 7
- Chiffres base 2 : 0 1

- Dans un nombre, le chiffre le moins significatif est appelé de poids faible, le chiffre le plus significatif est appelé de poids fort.
- En pratique, l'ensemble des outils informatiques manipulent les données sous leur représentation binaire (addition, soustraction, etc)

Complément à 2

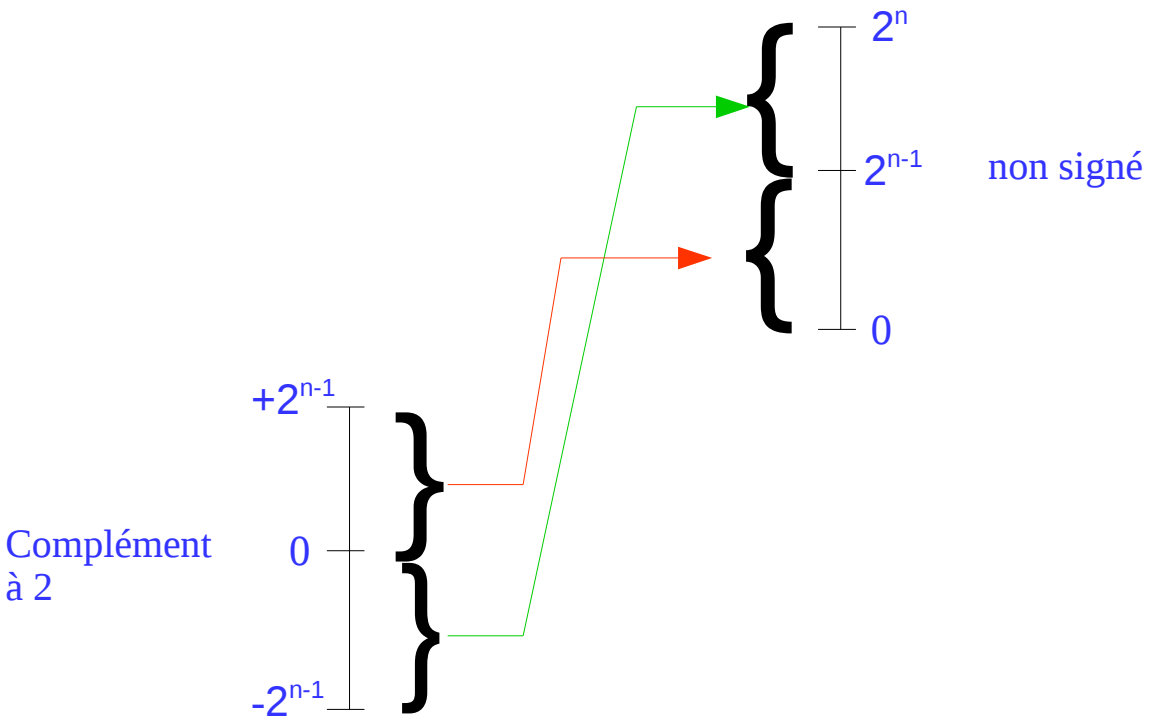
- Le complément à 2 est utilisé pour représenter les nombres négatifs. Il s'agit d'une transformation sur la représentation binaire d'un nombre.
  - Le complément à deux d'un nombre est égal à l'inverse du nombre plus 1.
- Exemple : -15 (sur un octet)
  - Représentation binaire de 15 : 00001111
  - Complément à 1 (= inversion) : 11110000
  - Ajouter 1 : 1111 0001

  Sur un octet, $-15_{10} = 1111\ 0001_2$

- Avec n bits on peut représenter des entiers entre: $-(2^{n-1})$ et $+(2^{n-1}-1)$

- Dans ce système,
  - le bit de poids fort d'un nombre négatif est égal à 1
  - Avantages : les opérations basiques (addition, soustraction) fonctionnent sans aucune transformation. On peut également déterminer le signe facilement en testant le bit de poids fort.

Complément à 2



Complément à 2

- Addition : somme de -17 et 12
  - 11101111 = -17
  - 00001100 = 12

  - 11111011
- Complément à 2 :
  - 11111011
  - 00000100
  - 00000101_2 = 5

    -17 + 12 = -5
- Dépassement de capacité : Que se passe-t-il lorsque l'on excède les possibilité d'un octet ? On parle de **dépassement de capacité**.
  - Cas de deux entiers de signe positif. On a un dépassement de capacité quand la retenue est distincte du dernier bit de report (i.e. celui sur le bit de signe).
  - Exemples : sur 8 bits, nombres de -128 à +127

    + 103 : 0110 0111
    +65 : 0100 0001
    report 0 ≠ 1 0 0 0 1 1 1

    +168 ≠ 1010 1000 (on obtient un nombre qui représente -88 en complément à deux) !!

La mémoire

- Toutes les informations se situent en mémoire, notamment :
  - Les variables
  - Le contexte des appels de fonctions

- La mémoire vive est analogue à un « ruban » de cases pouvant chacune contenir un octet. Chaque case est identifiée par son adresse, c'est à dire son numéro.

- Question : comment stocker des valeurs plus longues qu'un octet en mémoire ?
  -

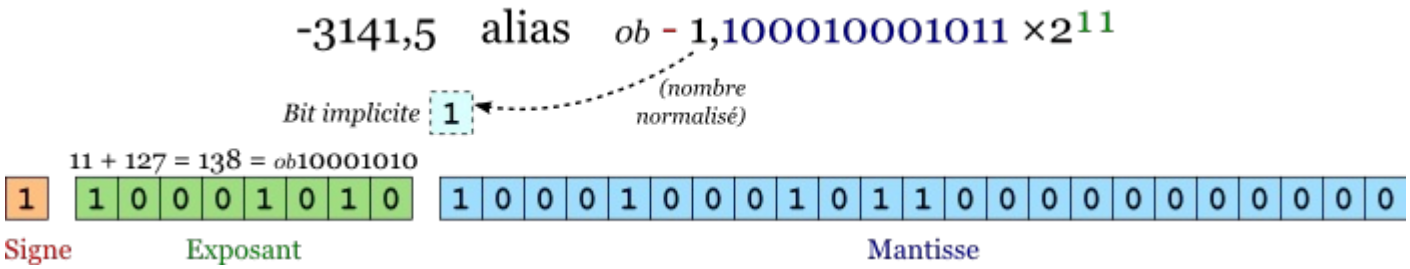
Les INT

- Taille : variable (8,16,32, 64 bits ou 1,2,4,8 octets)
- Représentent les nombres entiers positifs/négatifs
  - Pour les entiers signés, le complément à 2 est utilisé

Type	Taille (octet)	Min	Max
Short	2	-32 768	+32 767
unsigned short	2	0	65 535
int	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 -2 147 483 648	+32 767 +2 147 483 647
unsigned int	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 0	65 535 4 294 967 295
long int	4	-2 147 483 648	+2 147 483 647
unsigned long int	4	0	4 294 967 295

Les FLOAT/DOUBLE

- Les **float/double** représentent les nombres dit « à virgule flottante ».
- Leur précision se mesure en nombre de **chiffres significatifs**.
- Leur représentation en mémoire est découpée en trois parties :
  - **Signe**, codé sur un bit : ce bit vaut 0 si le nombre est positif, 1 s'il est négatif.
  - **Mantisse,** correspond à la partie décimale de la partie significative (avec un seul chiffre avant la virgule). On ne garde que la partie décimale car, du fait de la notation binaire, la partie entière est forcément égale à 1 → 1+mantisse représente la partie significative.
    - Elle se code (attention on est à droite de la virgule) :
    - pour le premier bit avec le poids 2^{-1} soit 0,5 ; puis 2^{-2} donc 0,25 ; 2^{-3} soit 0,125, etc
    - Par exemple : ,010 donnera ,25.
  - **Exposant :** puissance à laquelle il faut élever 2 (et non 10 !).
  - **Notation scientifique :** signe x 1 + mantisse x $2^{(\text{exposant - decalage})}$
  - Exemple : codage décimal -3141,5 en utilisant la norme IEEE 754 :
    - Écrire le nombre (sans le signe) en binaire : 3141,5 = 110001000101,1
    - Ensuite, décaler la virgule vers la gauche, de façon à ne laisser qu'un 1 sur sa gauche, il faut 11 décalages ; 110001000101,1 (bin) = 1,100010001011 x 2^{11}
    - Représentation binaire de l'exposant : 8 bits d'exposant (donc un décalage de 127) + 11



Les FLOAT/DOUBLE

- Float : 8 chiffres de précision pour la partie mantisse et un exposant qui va jusqu'à $\pm 10^{38}$

- double : environ 10 chiffres de précision pour la partie mantisse et un exposant qui va jusqu'à $\pm 10^{308}$

- Attention : il doit toujours y avoir au moins un point ou un exposant d'écrit, pour signifier au compilateur qu'on veut un nombre à virgule

- Les notations suivantes sont équivalentes : 0.0e0 ; 0.0 ; 0.e0 ; 0. ; .0e0 ; .0 ; 0e0 ;

Type	Taille (octet)	Min	Max
float	4	$-3.4*10^{-38}$	$3.4*10^{38}$
double	8	$-1.7*10^{-308}$	$1.7*10^{308}$
long double	10	$-3.4*10^{-4932}$	$3.4*10^{4932}$

Les CHAR

- Taille : 1 octet, traité comme un entier signé :
  - Non signé : 0 → 255
  - Signé : -128 → 127
- Pour spécifier un caractère, on l'entoure du symbole ' :
  - 'c' → le caractère c
- En C, les caractères sont considérés comme des entiers :
  - L'entier correspondant est sa valeur en code ASCII
  - Tout caractère à une valeur entière dans l'intervalle [0, 127]
  - Exemple : 'a' → 97, 'c' → 99, 'A' → 65

- Les opérateurs arithmétiques peuvent être appliqués sur les caractères :
  - 'a' + 1 → 'b' , 'F' – 2 → 'D', etc.

Type énuméré

- Les énumérations permettent de définir un type par la liste des valeurs qu'il peut prendre.

- Un objet de type énumération est défini par le mot-clef **enum** et un identificateur de modèle, suivis de la liste des valeurs que peut prendre cet objet : enum modele {constante-1, constante-2,...,constante-n};

- En réalité, les objets de type **enum** sont représentés comme des int. Les valeurs possibles constante-1, constante-2,...,constante-n sont odées par des entiers de 0 à n-1.
- Exemple :

```
main() {
        enum booleen {faux, vrai};
        enum booleen b ;
        b = vrai;
        printf("b = %d\n",b);
      }
```

Opérateurs sur le type entier

- **Opérateurs arithmétiques:**

  + - * / %

  – ***/*** division entière: 6 ***/*** 9 vaut 0

  – **%** modulo (reste de la division entière) 6 **%** 9 vaut 6

- **Opérateurs relationnels**:

  > >= < <= == !=

  – **==** → teste l'égalité

  – **!=** → teste la différence

- Si les opérandes sont des entiers, l'opérateur / réalisera la division entière, sinon il réalisera la division réelle : 7 / 2 vaut 3 et 7.0 / 2 vaut 3.5

- Les opérandes doivent êtres des entiers pour l'opérateur % (modulo)

Opérateurs sur le type réel

- **Opérateurs arithmétiques:**

  + - * /

- **Opérateurs relationnels**:

  > >= < <= == !=

Opérateurs booléens

opérateur	opération
&&	ET Logique
||	OU Logique
!	Négation

- **1 && 2** **! 3** **0 || ! (2 > 0)**
  vrai **faux** **faux**

- **0 && 0** **0 || 0** **! 0** **non nul || 0** **non nul || non nul**
  faux **faux** **vrai** **vrai** **vrai**

Règles de priorité sur les opérateurs

opérateur	priorité
()	→
!, ++, --, &, - (unaire), (type), sizeof	←
*** / %**	→
+ - (binaire)	→
< <= > >=	→
== !=	→
&& ||	→
= += *= /= %=	←

- **Les opérateurs && et || sont toujours évalués séquentiellement de gauche vers la droite.**
- **Pour la plupart des autres opérateurs, l'ordre d'évaluation des arguments n'est pas précisé.**

Données

→ Les informations traitées par un programme

→ Ces informations sont contenues dans :

- des variables
- des constantes

- **Variables** :
  - doivent être déclarées
  - leur contenu peut être lu et/ou modifié pendant l'exécution du programme

- **Constantes** :
  - doivent aussi être déclarées
  - initialisées par une valeur
  - leur contenu ne peut être modifié
  - elles peuvent être lues

Notion de variable (1)

- Correspond à un emplacement dans la mémoire de l'ordinateur, représenté par un **identificateur** (le *nom* de la variable).

- Toute déclaration contient :
  - Un identifieur : nom qui servira à désigner la case mémoire
  - Un type : permet de déterminer la taille de l'emplacement mémoire nécessaire pour stocker la valeur.

- Syntaxe :

  nom_type nom_variable ;

- Exemple :

  int a ; → déclare une variable de nom a de type entier

  float b ; → déclare une variable de nom b de type flottant

Notion de variable (2)

- Déclarations multiples de variables de même type :

  nom_type nom_variable_1, nom_variable_2, ..., nom_variable_N ;

- Exemples :

  int a, b ; → OK

  float b, int c ; → Faux

  float b ; int c ; → OK

- Attention : la déclaration d'une variable ne stocke aucune valeur dans l'emplacement mémoire
  - Il faut donc affecter une valeur à la variable (initialiser la valeur de la variable)

- Affectation d'une valeur à une variable :

  nom_variable = expression ;

  int a ; → déclaration

  a = 21; → affectation

Notion de variable (3)

- int a, b, c ;
  - `a = 8/4 ; a → 2`
  - `b = a ;  b → 2`
  - `c = a + 1; c → 3`

- Les types numériques sont convertis sans erreur de compilation :
  - `int a ;`
  - `a = 12.7 ;  → a aura pour valeur 12`
  - `a = a + 1 ; (a joue ici double rôle)`

  - `float f ;`
  - `f = 12 ;  →  f aura pour valeur 12.0`

  - `float f ;`
  - `f = 12.4 ;`
  - `int a ;`
  - `a = f ;  →  a aura pour valeur 12`

Conversion de type en C

- **Conversions implicite**: conversions faites par la machine

```
float c ; int a,b;
a = 3 ; b = 5 ; c = a+b ;
```

  - `c` est de type `float` ; `a+b` de type `int`.
  - Il y a une conversion de type implicite qui transforme le type du résultat `a+b` en flottant.

- **Conversion implicite non désirée** :

```
float c ; int a,b;
a = 1 ; c = 1.5 ; b = a+c ; (ici b vaudra 2 et non 2.5)
```

- **Conversion explicite** : c'est l'utilisateur qui force la conversion

```
float a, c ; int b;
a = 3 ; c = 1.5 ;
b = (int) a * c ; (ici b vaudra 4 et non 4.5)
```

Notion de variable (4)

- Erreurs classiques :

```
int main(){
  a = 21;
  /* … affichage d'un résultat … */
  return 0;
  }
```

→ Erreur à la compilation : « erreur : 'a' undeclared »

```
#include <stdio.h>
  int main(){
  int a;
  a = a+2;
  /* … */
  return 0;
  }
```

→ Avertissement à la compilation :«attention : 'a' is used uninitialized in this function

Opérateurs d'auto-incrémentation

- **Pré-incrémentation:**

```
int a,b;
```

`b = a++ ;` est équivalent à :

» `b = a ;`

» `a = a + 1;`

`b = a-- ;` est équivalent à :

» `b = a ;`

» `a = a - 1;`

- **Post-incrémentation :**

```
int a,b;
```

`b = ++a ;` est équivalent à :

» `a = a + 1 ;`

» `b = a ;`

»

`b = --a ;` est équivalent à :

» `a = a - 1 ;`

» `b = a;`

Constantes

- 4 types de base pour une constante :
  - entiers
  - réels
  - caractères
  - chaînes de caractères
- Syntaxe : `#define nom_constante valeur_constante`
- NB : on ne met pas de ; à la fin de la déclaration de constante
- Exemple :

```
#define TVA 20.6
#define PI 3.14
#define CHAINE "langage C"
```

Fonctions d'entrées-sorties

Les fonctions printf et scanf font partie de la bibliothèque standard d’entrées/sorties (stdio).

- **Printf :** écriture formatée sur la sortie standard *stdout* (l’écran, par défaut)
  - Syntaxe : printf(format,liste d'expression)
    - format : chaine entre " " + directives de formatage débutant par %

- **Scanf** : lecture formatée sur l’entrée standard *stdin* (le clavier, par défaut)
  - Syntaxe : scanf(format,&variable)
  - Permet de lire des infos depuis le clavier, de les interpréter suivant le format et de stocker les résultats dans les variables dont **les adresses sont données en argument**.

Les formats d'E/S

- LA SORTIE AVEC FORMAT: ***printf***
  - %d : l’argument est un entier et l’affichage se fera en décimal
  - %c : l’argument est un caractère
  - %s : l’argument est une chaîne de caractères
  - %f : l’argument est un *float*
- L’ENTREE AVEC FORMAT: **scanf**
  - %d : la donnée est un entier saisi sous forme décimale
  - %c : la donnée est un caractère
  - %s : la donnée est une chaîne de caractères
  - %f : la donnée est un réel (float)
- **Attention pour le type s** :
  - chaîne ne contenant ni espace, ni tabulation, ni retour à la ligne. Le caractère '\o' est ajouté à la fin.
  - Pas besoin d'utiliser l'opérateur & dans le cas d'une chaîne.

Les formats d'E/S

- Exemple :

```
 int a = 12 ;
float b = 3.4 ;
char car = 'a' ;
printf("%d \n", a) ;            → Affiche 12 puis un retour à la ligne
printf(" %d %f ",a ,b) ;   → Affiche : 12 3.4
printf("Voici un caractère : %c", car) ; → Affiche : Voici un caractère : a
```

Exemple complet

```
#include <stdio.h>
int main(void) {
int i;
float r;
char c;
printf("saisir un entier : ");
scanf("%d", &i);
printf("saisir un réel : ");
scanf("%f", &r);
printf("saisir un caractère : ");
Scanf(" %c", &i);
Return 0;
}
```

Instructions

- Une instruction peut être :
  - Une **expression**, elle se termine alors par un point virgule.
    - Par exemple : une déclaration de variable, une affectation, un appel de fonction.
  - Une **structure de contrôle** : boucle ou alternative.
    - Par exemple :

```
If(...){
      ...
}
else{
      ...
}
```

```
for(...) {
      ...
}
```

```
while(...) {
            ...
}
```

  - Un **bloc d'instructions**.

Bloc d'instructions

- Regroupe plusieurs instructions entre accolades

```
{
    instruction_1;
    instruction_2;
     …
    instruction_n;
}
```

Instruction conditionnelle

- Exécution séquentielle des instructions

  → soumettre l'exécution d'un bloc d'instructions à une condition

- Exemple:
  - Demander à l'utilisateur son numéro d'étudiant
  - Si ce numéro est plus grand que 2000000
    - → calculer son année d'inscription
    - → afficher l'année d'inscription
  - Demander à l'utilisateur son age

Instruction conditionnelle simple

- Permet de conditionner l'exécution d'une suite d'instructions

```
if (condition) {
    instruction
}

if (condition) {
    instruction_1;
    instruction_2;
     …
    instruction_n;
}
```

Instruction conditionnelle avec alternative

- Permet de définir des instructions à exécuter si la condition est fausse

```
if (condition) {
    instruction_1;
    instruction_2;
     …
    instruction_n;
    }
else {
        instruction_1;
    instruction_2;
     …
    instruction_n;
    }
```

Instruction conditionnelle imbriquée

```
if (condition) {
  instruction_1;
 instruction_2;
     …
 instruction_n;
} else if (condition) {
     instruction_1;
     instruction_2;
        …
     instruction_n;
} ...
```

Instruction If

- « Else » qui pendent
  - Un else qui pend est attaché au premier if sans else

```
if (condition)
if (autre condition) instruction ;
else
autre instruction ;
```

Instruction switch/case

SYNTAXE :

```
switch (expression) {
case v1 : instruction_1
                    break ;
case v2 : instruction_2
                    break ;
default : instruction_default
}
```

- L'évaluation de <expression> doit retourner une valeur de type int.
- si la valeur de <expression> vaut v1 ou v2... .alors l'instruction correspondante sera exécutée sinon instruction_default sera exécutée.
- Break permet de sortir du switch.

switch/case (exemple 1)

```
mod = nombre % 2;
switch (mod)
{
case 0: printf(" nombre pair \n ");
break;
case 1: printf(" nombre impair \n ");
break;
}
```

switch/case (exemple 2)

```
switch (car)
{
case 'A':
case 'E':
case 'I':
case 'O':
case 'U': voyelle++;
break; ← si car est une voyelle et break est absent, alors espace sera incrémenté
case ' ': espace++;
break;
default : consonne++;
}
```