INFORMATIQUE
Sciences
Université Paris Cité

Langage C –Cours 1

Lélia Blin

lelia.blin@irif.fr

2023 - 2024

Modalité du cours

Les Travaux pratiques (TP)

- Les Tps seront disponibles sur Moodle au fur et à mesure
- Il faut impérative s'inscrire sur dans son groupe sur moodle
- Il faudra rendre chaque semaine le code d'exercices de TP sur moodle

Validations de la matières

- TP: Rendu d'exercices de TP – tirage au sort d'un ou deux TP
  - Non rendu 0
- Partiel (durée 2h, QCM avec questions ouvertes)
- Examen (durée 2h, QCM avec questions ouvertes) `

Formules

$$EP = Max(Examen; (Examen + Partiel/2))$$

Note finale : $(3EP + TP)/4$

Caractéristiques du langage C

Langage C – programmation impérative

- Le langage C est inventé à la fin des années 70 pour programmer unix sur la machine PDP11.
- C est un langage de programmation impératif
  - le code est structuré autour de procédures et de fonctions qui effectuent des opérations sur des données.
  - Les données sont souvent organisées en structures de données simples telles que des tableaux et des structures.
  - Le C ne supporte pas les concepts clés de l'orienté objet, tels que l'encapsulation, l'héritage et le polymorphisme.

Langage C – langage typé

- C est un langage typé
  - Toute variable doit être déclarée avant utilisation avec un type et ne sera utilisé qu'avec des valeurs de ce type
  - Contrairement à Python par exemple.

Langage C – Manipulation des cases mémoire

- C permet de (et est fait pour) manipuler directement les cases mémoires de la machine.
- Il fait cela grâce aux pointeurs, C est un langage (haut-niveau) de bas niveau
- Contrairement à tous les autres langages haut-niveau par exemple.
    - Python, java, ruby, ...

Langage C – pas d'objet

- C ne propose en revanche aucune opération qui traite directement des objets de plus haut niveau
    - fichier informatique, liste, table de hachage doivent être construits à la main à partir des types de bases.

Langage C – gestion de la mémoire

- Il n'y a pas de gestion implicite de la mémoire
- il faut allouer et désallouer explicitement la mémoire que l'on veut utiliser pour le programme
- Il y a des allocations statiques (déclarations de variables) ou dynamiques (allocation lors de l'exécution).
- Il n'y a pas de garbage collecteur (ramasse miettes)
- Contrairement à Python, Java par exemple.

Langage C – très light

- Nécessite très peu de support à l'exécution, juste la libc qui est généralement installée sur toute les machines.

Avantages

- Ces caractéristiques en font un langage privilégié pour:
  - La programmation embarquée des micro-contrôleurs (quasiment exclusivement en C).
  - L'écriture de systèmes d'exploitation ou de modules noyaux (Les noyaux Linux et Windows sont en grande partie écrits en C)
  - Les programmes ou l'on souhaite maîtriser les ressources matérielles utilisées et les algorithmes implémentés (calcul intensifs, cryptographie,. . .)

Inconvénients

Il y a aussi un certain nombre de défauts:

- Peu de vérification du compilateur ⇒ de nombreux bugs d'inattention qui font perdre un temps incommensurable.
- Pas de gestion d'exception, modularité rudimentaire (pas d'espace de noms, pas de programmation objet).
- Source de nombreuses failles de sécurité (débordement de tampon).

Il est portant d'écrire un programme

... mais aussi de pouvoir le relire ou le faire lire par d'autre

Quelques règles d'écriture de programmes C

- Ne jamais placer plusieurs instructions sur une même ligne.
- Utiliser des identificateurs significatifs pour les noms de fonctions
- Indentez vos programmes!!!
- une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions
- Une accolade fermante est seule sur une ligne.
- Aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces

Editeur

- Utilisez un éditeur pour programmer:
  - Vim, emacs, Atom, sublimeText, vscode

Mon premier programme

```c
/* Un premier programme simple*/
#include <stdio.h>
#include <stdlib.h>

/*Programme principal*/
int main(void){
  int x=12;
  for(unsigned i=0;i<10;++i){
      printf("i vaut %u et x vaut %d```n",i,x);
          x+=3;
        }
  return EXIT_SUCCESS
}
```

premier-prog.c

Compiler

- C est un langage **compilé**
- Pour compiler le programme précédent :
  - `gcc -Wall -o premier-prog premier-prog.c`
  - `gcc` : compilateur (GNU Compiler Collection)
  - `-Wall` : tout ce qui est anormal est signalé
  - `-o premier-prog` : met la sortie du compilateur dans `premier-prog`
  - `premier-prog.c` : nom du fichier source

Compiler

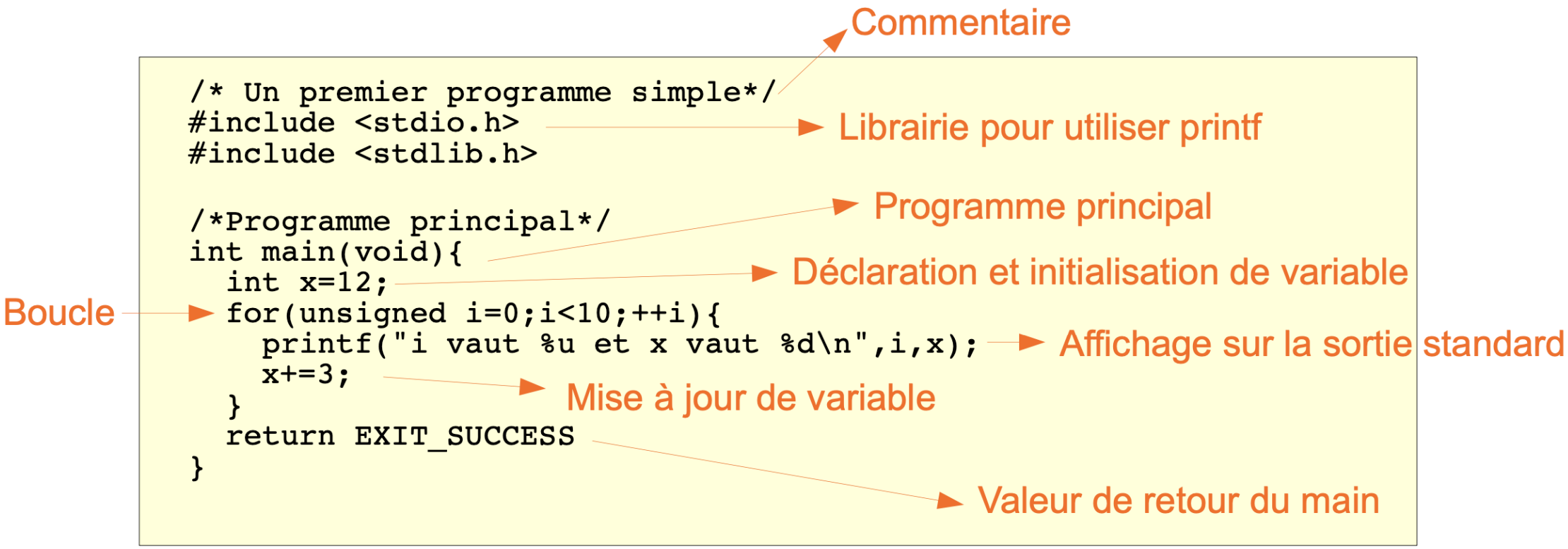
- Un programme C correct peut être compilé et exécuté sur différentes plate-formes - **il est capital d'utiliser l'option -Wall**
- Il existe différents compilateurs : clang,c99,icc,...
- Si tout se passe bien, le compilateur produit le fichier `premier-prog`
- Si il y a un problème, le compilateur signale les erreurs de compilation et les warning (avertissement)
- Si il n'y a que des warning le fichier exécutable est produit mais son fonctionnement n'est pas garanti !!!

Compiler et exécuter

```
gcc -Wall -o premier-prog premier-prog.c
./premier-prog
i vaut 0 et x vaut 12
i vaut 1 et x vaut 15
i vaut 2 et x vaut 18
i vaut 3 et x vaut 21
i vaut 4 et x vaut 24
i vaut 5 et x vaut 27
i vaut 6 et x vaut 30
i vaut 7 et x vaut 33
i vaut 8 et x vaut 36
i vaut 9 et x vaut 39
```

Terminal

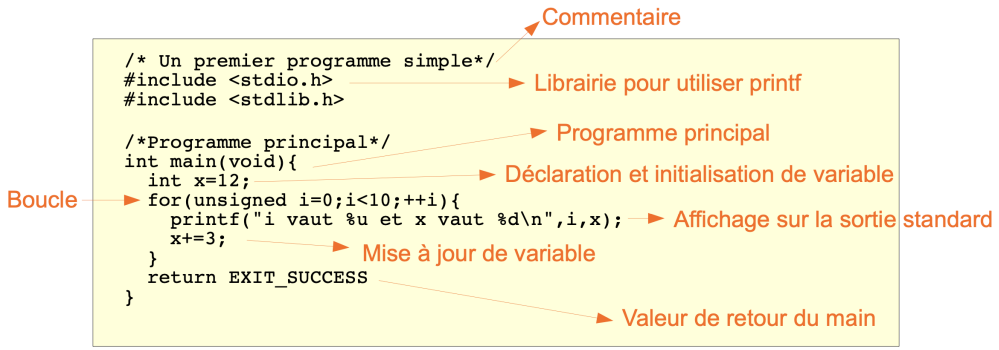
Mon premier programme



premier-prog.c

Lélia Blin

Mon premier programme



premier-prog.c

```
printf("i vaut %u et x vaut %d\n",i,x)
```

- affiche **i vaut** suivi de la valeur de i qui est au format unsigned (**%u**) suivi de **x vaut** suivi de la valeur de x qui est au format int (**%d**)
- Pour chaque valeur passée en argument de `printf` on a dans la chaîne une séquence %format qui indique qu'on prend en compte un argument et le format auquel on souhaite l'afficher

Valeur de retour du main

- Le main est de type **int main(void)** (on verra qu'il peut prendre d'autres arguments pour lui passer des chaînes de caractères)
- Il retourne donc une valeur entière pouvant être utilisée par l'environnement d'exécution
- La plupart du temps 0 veut dire une exécution correcte et un chiffre différent de 0 une exécution qui s'est mal passé

stdlib.h

- Pour être sûr que l'on respecte les règles du systèmes, deux valeurs stockées dans stdlib.h
- **EXIT_SUCCESS**: tout s'est bien passée
- **EXIT_FAILURE** : il y a eu un problème

Grammaire du C en bref

- **Mots-clefs** : `#include` , `int` , `unsigned` , `void` , `for` , `return` , ...
- **Ponctuations** : {...}, (...), /.../ , , , ; ,...
- **Littéraux **: 0, 10, 3...
- **Idenditifiants** : i, x, EXITSUCCESS,... (ils sont composés de lettres, de chiffres et du caractère et ne peuvent pas commencer par un chiffre)
- **Identifiants de fonction** : main, printf,....
- **Opérateurs **: = , < , ++,...

Grammaire du C en bref

- **Toutes les instructions finissent par un point-virgule !!**
- **Tous les identifiants doivent être déclarés (avant leur utilisation)**
- soit dans le programme : **int x**, **unsigned i**, **int main(void)**
- soit dans d'autres fichiers :
  - printf dans **stdio.h**
  - EXIT_SUCCESS dans **stdlib.h**

Types

- Un programme manipule des valeurs et chaque valeur a un **type** qui est déterminé de façon statique (à la compilation)
- Il est important de savoir quel type est utilisé et à quel instant
- Le type d'une valeur peut avoir un effet sur les opérations réalisées sur cette valeur
- Il y a des types pour des **valeurs signés et d'autres non-signés**

Types

- Un même type n'a pas forcément la même taille selon les environnements d'exécution (sur certains systèmes une valeur de type int est codé sur 2 octets et sur d'autres sur 4 octets)
- Pour connaître la taille occupée par la valeur d'un type, on a l'opérateur sizeof :
  - `printf(''Taille int :%lu\n'', sizeof(int))` nous affichera la taille d'un int

Types basiques entiers

Classe	Nom systématique	Autres noms
Entiers non signés	`_Bool`	`bool`
Entiers non signés	`unsigned char`	
Entiers non signés	`unsigned short`	
Entiers non signés	`unsigned int`	`unsigned`
Entiers non signés	`unsigned long`	
Entiers non signés	`unsigned long long`	
Entiers signés	`char`	
Entiers signés	`signed char`	
Entiers signés	`signed short`	`short`
Entiers signés	`signed int`	`int` (ou `signed`)
Entiers signés	`signed long`	`long`
Entiers signés	`signed long long`	`long long`

Types basiques entiers

- Comme dit précédemment : pas de garantie sur le nombre d'octets occupé par une valeur d'un certain type
- Les valeurs de type (unsigned) char occupent un octet
- Il n'y a pas de garantie sur les valeurs possibles pour les types et sur comment elles se comparent entre elles

Types basiques entiers

- On sait que :
  - `char` $\leq$ `short` $\leq$ `int` $\leq$ `long` $\leq$ `long long`
  - `bool` $\leq$ `unsigned char` $\leq$ `unsigned short` $\leq$ `unsigned` $\leq$ `unsigned long` $\leq$ `unsigned long long`

$\leq$ voulant dire ici le rang des valeur du type de gauche est inclus dans le rang des valeurs du type de droite

Pour utiliser le type `bool`, il faut inclure la librairie `stdbool.h`

Types basiques flottants (‘Réels’)

Classe	Nom systématique
Flottant signé	`float`
Flottant signé	`double`
Flottant signé	`long double`

Types utilisés dans ce cours

- Dans ce cours, on utilisera principalement :
  - `bool` pour les booléens
  - `char` pour les caractères
  - `int` pour les entiers signés
  - `unsigned` pour les entiers non-signés
  - `double` pour les flottants (plus précis que float)

Remarques :

> - En fait toutes ces valeurs sont des nombres
> - On peut faire des opérations arithmétiques sur les valeurs de type `bool` et `char`, mais il faut être attentif (éviter sur le type `bool`)

Constantes

- Comment écrire des constantes pour ces types
  - **Constantes booléennes** :
    - true et false (il faut inclure `stdbool.h`)
  - **Constantes entières décimales** :
    - 123, 15, 28
  - **Constantes entières hexadécimales** :
    - 0x suivi par des symboles parmi 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (A signifiant 10, B signifiant 11, etc).
    - Par exemple en non signé : 0x1A représente l'entier 26 et 0xA1 l'entier 161. On peut aussi utiliser les lettres minuscules a,b,c,d,e,f.

Constantes

- **Constantes flottante décimales** :
  - 10.2 ou encore 10.2e3 (ou 10.2E3). Ici le e est utilisé pour indique 10 à la puissance. Par exemple 3e2 vaut 3 * 100=300.
- **Constantes entières de caractères** :
  - ‘a’, ‘Z’, ‘0’, ‘1’, ‘\n’, ‘\0’
- **Constantes chaînes de caractères** :
  - "Hello !"

Exemple

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

int main(){
  unsigned x=0xA1;
  printf("Valeur decimale de x : %u et hexadecimale : 0x%x\n",x,x);
  double y=10.6;
  printf("valeur de y : %lf\n",y);
  double z=32e2;
  printf("valeur de z : %lf et en notation exponentielle %le\n ",z,z);
  return EXIT_SUCCESS;
}
```

test-format.c

Valeurs limites

- Les valeurs entières et flottantes ont des limites -> pas de précision infinie
- Elles dépendent de l'environnement d'exécution
- Pour les entiers, elles sont définies dans la librairie limits.h

Valeurs limites:

- INT_MIN et INT_MAX donnent les valeurs minimales et maximales pour les int
- UNINT_MAX donne la valeur maximale pour les unsigned
- LONG_MIN et LONG_MAX donnent les valeurs minimales et maximales pour les long
- Pour les flottants, elles sont définies dans la librairie float.h
- les double négatifs sont compris entre -DBL_MAX et -DBL_MIN et les double positifs entre DBL_MIN et DBL_MAX

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main() {
   printf("Valeur minimale pour INT %d\n", INT_MIN);
   printf("Valeur maximale pour INT %d\n", INT_MAX);
   printf("Valeur maximale pour UNSIGNED %u\n", UINT_MAX);
   printf("Valeur minimale pour LONG %ld\n", LONG_MIN);
   printf("Valeur maximale pour LONG %ld\n", LONG_MAX);
   printf("Valeur maximale pour DOUBLE positif %le\n", DBL_MAX);
   printf("Valeur minimale pour DOUBLE positif %le\n", DBL_MIN);
   return EXIT_SUCCESS;
}
```

val-limit.c

Quelques remarques (1)

- Les constantes entiers décimales sont signées par défaut **MAIS** :
  - On peut forcer à ce qu'elles soient non signés en mettant U à la fin, par exemple 10U
  - Quand on les affecte, l'affectation convertit dans le type du terme de gauche, par exemple dans unsigned x=10 10 est converti en unsigned
  - Il faut faire attention quand on diminue la taille du type -> en fait il faut éviter de faire cela

Quelques remarques (2)

- Il faut éviter d'utiliser la représentation hexadécimal pour exprimer des valeurs négatives
- On peut faire des choses comme unsigned a=-2, dans ce cas l'opérateur unaire - est réalisé pour le type unsigned et cela revient à faire UINT_MAX+1-2 -> il faut là aussi éviter cela
- Ce qui se passe dans le cas de conversion avec diminution de taille de type dépend de l'environnement d'exécution

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

int main(){
  unsigned a=-2;
  unsigned b=UINT_MAX+1-2;
  unsigned c=0x80000001; //sur ma machine 4 octets pour unsigned
  int d=0x80000001;  // et 4 octets pour int
  unsigned e=UINT_MAX;
  int f=UINT_MAX; // cette valeur ne sera pas UNINT_MAX
  printf("a vaut %u\n",a);
  printf("b vaut %u\n",b);
  printf("c vaut %u\n",c);
  printf("d vaut %d\n",d);
  printf("e vaut %u\n",e);
  printf("f vaut %d\n",f);
  return EXIT_SUCCESS;
}
```

conversion.c

Structure générale d'un programme

```
[directives au préprocesseur]
[déclaration de variables externes]
[fonctions secondaires]

int main(void) (ou int main(int argc, char**argv)*/){
  déclaration de variabbles et instructions
}

type fonction(arguments){
  déclaration de variabbles et instructions
}

type fonction(arguments){
  déclaration de variabbles et instructions
}

type fonction(arguments){
  déclaration de variabbles et instructions
}
```

Initialisation (1)

- **Toutes les variables doivent être initialisées**
  - Sauf quelques rares exceptions
- Dans tous les cas, bonne pratique de programmation
- On les initialise au moment de leur déclaration

```c
int x=2;
double y=12.3;
char a='d';
```

Initialisation (2)

- On peut écrire des choses comme :
  - `int x=1 ,y=2, z=3;` équivalent de `int x=1; int y=2; int z=3;`
  - pas vraiment nécessaire (ce n'est pas parce que votre code est le plus succinct qu'il est mieux ou plus correct...juste il peut être moins lisible)
- **Rappel** : lors d'une affectation la valeur est convertie implicitement dans le type de gauche. Ainsi dans `unsigned x=10;` 10 est converti en unsigned

Affectation

- Elle a la forme : **variable = expression**
- Le terme de gauche peut être une variable ou un élément de tableau ou un pointeur mais pas une constante
- L'affectation évalue expression est donne sa valeur à variable
- L'affectation est elle-même **une expression qui possède une valeur**, ainsi x=12 a pour valeur 12

```
int x=2;
int y=x=2*x;
```

> À **ÉVITER**, sauf si voulez rendre votre code illisible

Opérateurs arithmétiques

- `+` addition
- `-` soustraction
- `*` multiplication
- `/` division (entières si les deux membres sont entiers, flottant sinon)
- `%` reste de la division entière (ne s'applique qu'à des entiers)

Rappel :

- attention aux types que vous utilisez et éviter de mélanger `unsigned` et `int`
- Pour `/` et `%` si vous avez des entiers signés, on peut avoir des cas bizarres
  - → Non défini en C89 (i.e. dépend de l'environnement d'exécution)
- En C99, normalement si `a/b` vaut d et `a%b` vaut r, on a $|r| < |b|$ et a est égal à $b * d + r$
- On ne peut pas faire `/0` (division par 0) ou `%0` (modulo 0) → Erreur à l'exécution

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

int main(){
  int x1=(-17)/(-5); // vaut 3
  int y1=(-17)%(-5); // vaut -2
  int x2=(-17)/5; // vaut -3
  int y2=(-17)%5; // vaut -2
  int x3=17/(-5); // vaut -3
  int y3=17%(-5); // vaut 2
  int x4=17/5; // vaut 3
  int y4=17%5; // vaut 2
  printf("%d %d\n",x1,y1);
  printf("%d %d\n",x2,y2);
  printf("%d %d\n",x3,y3);
  printf("%d %d\n",x4,y4);
  return EXIT_SUCCESS;
}
```

modulo.c

Opérateurs relationnels

- `>` strictement supérieur
- `>=` supérieur ou égal
- `<` strictement inférieur
- `<=` inférieur ou égal
- == égal
- `!=` différent

> **Utilisation** : expression1 op expression2 (où op est l'opérateur)

Remarque

- en C, l'entier 0 correspond à false et un entier différent de 0 est interprété comme true
- Ces expressions renvoient donc 0 ou 1, mais avec `stdbool.h` on peut parler de false et true
- **Attention** :
  - Ne pas confondre `x=5` (affectation) avec `x==5` (test d'égalité)
  - Ne pas mélanger les entiers signés et non signés dans des comparaisons

Exemple

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

int main(){
  int a=-1;
  unsigned b=1;
  if(a<b){
    puts("a < b");
  }else{
    puts("a >= b"); // affiche cela sur ma machine
  }
}
```

test-egalite.c

Opérateurs logiques booléens

- et : `&&` ← vrai uniquement si les deux termes sont vrais, faux sinon
- ou : `||` ← faux uniquement si les deux termes sont faux, vrai sinon
- non : `!` ← faux si le terme associé est vrai, vrai sinon
- Là encore 0 est false et toute valeur différente de 0 est true
- L'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé
  - Ainsi si l'on fait `((1<2) || (x==5))` alors on ne testera pas si x vaut 5
  - De même si l'on fait `((2<1)&& (x==5))` alors on ne testera pas si x vaut 5

Opérateurs d'affectation spéciaux

- `+=` , `-=` , `*=` , `/=` , `%=`
  - exp1 op = exp2 est équivalent à exp1= exp1 op exp2
  - Ainsi `x*=5` est la même chose que `x = x*5`
- `x++` , `++x` , `--x` , `x--`
  - Pour `x++` et `++x` , dans les deux cas la valeur de x est augmenté de 1, mais dans le premier cas la valeur retournée est celle de x avant l'incrémentation et dans l'autre celle après
  - Donc `++x` est équivalent à `x = x+1` et `--x` est équivalent à `x=x-1`
  - Évitez d'utiliser `x++`

Opérateur conditionnel ternaire

- condition ? expression1 : expression2
- Cette expression est égale à
  - expression1 si condition est évaluée à vrai
  - expression2 sinon
- <u>Par exemple </u>:
  - `int y = x>=0 ? x : -x;` met dans y la valeur absolue de x
  - `int m= (a>b)? a : b;` met dans m le maximum de a et b

Opérateur de conversion de type

- (type) objet
  - force la conversion (appelée cast) de objet dans le type type
  - À utiliser avec précaution (mais on sera parfois obligé)

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

int main(){
  int a=1;
  int b=2;
  double c= a/b;
  double d= (double)a/b;
  printf("%lf\n",c); //Affiche 0.000000
  printf("%lf\n",d); //Affiche 0.500000
  return EXIT_SUCCESS;
}
```

Gare aux effets de bord

On peut écrire des choses comme :

```
int a=1,b=2,c=3,d=4;
a=b=c += ++d ;
a=(b=(c=c+(d=d+1)));//idem que ligne du dessus
```

- Il faut éviter d'écrire des choses comme cela sauf si vous voulez faire de l'obscurcissement de code.
- Instructions qui modifient plusieurs variables en une seule instruction → effet de bord
- Pire si on fait `i=i++ ?` Warning du compilateur !

Branchement conditionnel (1)

```
if(expression-1){
  liste-instructions-1
}else if(expression-2){
  liste-instructions-2
}
...
else if(expression-n){
  liste-instrusctions-n
}else{
  liste-instructions
}
```

Branchement conditionnel (2)

- Le nombre de `else if` est arbitraire
- Le dernier `else` est facultatif
- On peut ne pas mettre d'accolade si l'on n'a qu'une seule instruction, mais pas recommandé
- Rappel : 0 est considéré comme faux et toute valeur différente de 0 est considérée comme vrai

Branchement multiple (1)

```
switch(expression){
  case constante-1 :
      liste-instructions-1
      break;
  case liste- constante-2 :
      instructions-2
      break;
  ...
  case constante-n :
      liste-instructions-3
      break;
  default :
      instructions

}
```

Branchement multiple (2)

- L'expression testée est un entier (pour rappel les char sont des entiers)
- Le switch saute à la première valeur qui est égale à l'expression
- Si on rencontre break on sort du switch, sinon on exécute toutes les autres instructions suivantes

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

int main(){
  int x=3;
  switch(x){
  default : puts("+++++");
  case 4 : puts("++++");
  case 3 : puts("+++");
  case 2 : puts("++");
  case 1 : puts("+");
  case 0: ;
  }
  return EXIT_SUCCESS;
}
```

test-switch.c

Boucle while et do...while

```
while(condition){
  liste-instructions
}

do{
  liste instructions
} while (expression)
```

- La différence entre les deux est que dans `do... while` , la liste d'instructions est exécutée au moins une fois
- Une boucle `while(true){....}` ne termine jamais (sauf si l'on utilise l'instruction break à un moment

Boucle for

```
for(expr1 ; expr2 ; expr3){
  liste-instructions
}
EQUIVALENT À
expr1
while(expr2){
 liste-instructions
 expr3
}
```

- expr 1 est une affectation ou une définition de variables
- expr2 est une condition teste si l'on doit continuer l'itération
- expr3 met à jour la variable utilisée dans expr1

Déclaration de fonction

```
type ma_fonction(arguments);

DEFINITION DE FONCTION

type ma_fonction (arguments){
  liste d'instruction ou de déclarations de variables

}
```

Fonction (1)

- Avant d'être utilisée, une fonction doit être déclarée ou définie
- Une fonction a un type de retour et peut prendre 0 ou plusieurs arguments
- Le `return` d'une fonction doit être cohérent avec son type de retour
- Si une fonction ne renvoie rien, son type de retour est `void`
- Si on finit le bloc d'instructions de la fonction sans `return`, c'est équivalent à faire `return ;`

Fonction (2)

- Si une fonction ne prend pas d'arguments, on indiquera `void` dans ces arguments
- main est spécial, si on ne met pas de return, cela revient à faire return `EXIT_SUCCESS`
- Quand on déclare la fonction, les noms des arguments sont optionnels
- Le passage des arguments est par valeur

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

unsigned puissance2(unsigned);
int main(void){
  unsigned x=puissance2(4);
  printf("%u \n",x);
}
unsigned puissance2(unsigned n){
  unsigned r=1;
  while(n>0){
    r*=2;
    --n;
  }
  return r;
}
```

test-fonction.c

printf et puts

- Servent à écrire sur la sortie standard
- Syntaxe :
  - `puts(chaine)`
  - `printf(chaine,expression-1,expression-2,...,expression-n)`
- `puts` rajoute un saut à la ligne à l'affichage et ne permet pas d'afficher des expressions
- La chaine donnée à `printf` contient des spécifications de format pour chaque expression

Formats pour `printf` :

- int :%d (décimale signé)
- unsigned :%u (décimale non signé),%x (hexadécimale non signé)
- long : %ld (décimale signé)
- unsigned long :%lu (décimale non signé)
- double :%lf (décimale virgule fixe),%le (décimale notation exponentielle)
- char : %c