Pratique du C
Premiers pas

Licence Informatique — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Semestre 5 — 2011-2012

Details pratiques

Équipe pédagogique :

Francesco	De Comité	G 1 (info)
Alexandre	Sedoglavic	G 2 (info)
Mikaël	Salson	G 3 (info)
Adrien	Poteaux	G 4 (info)
Samy	Meftali	G 1 (miage)
Jean-François	Roos	G 2 (miage)

Toutes les informations (emploi du temps, semainier, documents, etc.) sont disponibles à l'url :
`http://www.fil.univ-lille1.fr/portail`
Licence → S5 info → PDC.

Conçu aux laboratoires Bell par D. Ritchie pour développer le système d'exploitation UNIX (des langages A et B ont existé mais ne sont plus utilisés) :

- C n'est lié à aucune architecture particulière ;
- C est un langage typé qui fournit toutes les instructions nécessaires à la programmation structurée ;
- C est un langage compilé.

En 1983, l'ANSI décida de normaliser ce langage et définit la norme ANSI C en 1989. Elle fut reprise intégralement en 1990 par l'ISO.

Les principes historiques de C sont :

1. Trust the programmer.
2. Make it fast, even if it is not guaranteed to be portable.
3. Keep the language small and simple.
4. Don't prevent the programmer from doing what needs to be done.
5. Provide (preferably) only one (obvious) way to do an operation.

Trust the programmer

Le langage C n'a pas été conçu pour faciliter sa lecture (contrairement à Ada par exemple).
Un concours annuel (International Obfuscated C Code Contest — www.ioccc.org) récompense d'ailleurs le programme le plus illisible.

Par exemple, le cru 2001 présentait le programme suivant :

```
m(char*s,char*t) {
return *t-42?*s?63==*t|*s==*t&&m(s+1,t+1):
          !*t:m(s,t+1)||*s&&m(s+1,t);
}
main(int c,char **v) { return!m(v[1],v[2]); }
```

Trust the programmer

Le langage C n'a pas été conçu pour faciliter sa lecture (contrairement à Ada par exemple).
Un concours annuel (International Obfuscated C Code Contest — www.ioccc.org) récompense d'ailleurs le programme le plus illisible.

Un autre exemple de 1999 :

```
#include <stdio.h>
int O,o,i;char*I="";
main(l){O&=l&1?*I:~*I,*I++||(l=2*getchar(),i+=O>8
?o:O?O:o+1,o=O>9,O=-1,I="t8B~pq‘",l>0)?main(l/2):
printf("%d\n",--i);}
```

Trust the programmer

Le langage C n'a pas été conçu pour faciliter sa lecture (contrairement à Ada par exemple).
Un concours annuel (International Obfuscated C Code Contest — www.ioccc.org) récompense d'ailleurs le programme le plus illisible.

Vous ne validerez pas cet enseignement si vous suivez ces exemples.

Par contre, vous l'aurez réussi si vous les comprenez sans problèmes.

Make it fast, even if it is not guaranteed to be portable.

Les compilateurs traîtent les commandes C en fonction des spécificités de l'architecture (implantation des types au plus efficace).
De plus, on peut faire appel à l'assembleur pour des tâches critiques. Par exemple, dans le code du noyau Linux :

```
static inline int flag_is_changeable_p(u32 flag){ u32 f1,f2;
        asm("pushfl\n\t"
            "pushfl\n\t"
            "popl %0\n\t"
            "movl %0,%1\n\t"
            "xorl %2,%0\n\t"
            "pushl %0\n\t"
            "popfl\n\t"
            "pushfl\n\t"
            "popl %0\n\t"
            "popfl\n\t"
            : "=&r" (f1), "=&r" (f2)
            : "ir" (flag)); return ((f1^f2) & flag) != 0;
}
```

Keep the language small and simple

Les 32 mots-clefs de l'ANSI C

- les spécificateurs de type :
  char **double** **enum** **float** **int** **long**
  short **signed** **struct** **union** **unsigned** **void**
- les qualificateurs de type : **const** **volatile**
- les instructions de contrôle :
  break **case** **continue** **default** **do** **else**
  for **goto** **if** **switch** **while**
- spécificateurs de stockage :
  auto **register** **static** **extern** **typedef**
- autres : **return** **sizeof**

Keep the language small and simple

Les 40 opérateurs de l'ANSI C

- les opérateurs

( )	[ ]	->	!	~
++	−−	-	(type)	*()
&()	sizeof	,	*	/
%	+	-	>>	<<
>	<	<=	>=	==
<<=	&	^	|	&&
||	? :	+=	-=	*=
/=	%=	^=	!=	>>=

Et c'est tout ! ! !

Don't prevent the programmer from doing what needs to be done : C est un langage de bas niveau

Il n'est pas rare d'entendre dire que C est un assembleur de haut niveau i.e. un assembleur typé qui offre des structures de contrôle élaborées et qui est — relativement — portable (et porté) sur l'ensemble des architectures.

Ce langage est pensé comme un *assembleur portable* : son pouvoir d'expression est une *projection* des fonctions élémentaires d'un microprocesseur idéalisé et suffisament simple pour être une abstraction des architectures réelles.

Don't prevent the programmer from doing what needs to be done : C est un langage de bas niveau

C est un langage de bas niveau, il

- permet de manipuler des données au niveau du processeur (sans recourir à l'assembleur) ;
- ne gère pas la mémoire (ramassage de miettes) ;
- ne prévoit pas d'instruction traitant des objets composés comme des chaînes de caractères, des structures, etc. (pour comparer deux chaînes, il faut utiliser une fonction) ;
- ne fournit pas d'opération d'entrée-sortie dans le langage.

C utilise des bibliothèques pour ces tâches.

Pratique du C dans le cursus de formation

L'ambition du cours est de se comprendre à plusieurs niveaux (dans l'ordre chronologique) :

1. C comme langage de programmation.
2. Relations entre C et architecture.
3. Relations entre C et OS.

sans pour autant faire les cours se trouvant dans la même filière informatique du LMD à Lille :

Architecture élémentaire	(info 202)
Pratique du C	(info 301)
Pratique des systèmes	(info 305)

Le premier programme et sa compilation

En fin de cours, les détails du code suivant seront limpides :

```
/* ceci est un commentaire */
#include <stdio.h>
int
main
(int argc, char **argv)
{
 printf("Salut le monde \n") ;
 return 0 ; /* valeur de retour (0 i.e. EXIT_SUCCESS) */
}
```

- `include` est une directive au préprocesseur pour incorporer ce qui permet l'usage de la fonction `printf` de la bibliothèque standard ;
- les instructions se terminent par un point-virgule ;

Le premier programme et sa compilation

En fin de cours, les détails du code suivant seront limpides :

```
/* ceci est un commentaire */
#include <stdio.h>
int
main
(int argc, char **argv)
{
 printf("Salut le monde \n") ;
 return 0 ; /* valeur de retour (0 i.e. EXIT_SUCCESS) */
}
```

- la fonction `main` est imposée pour produire un exécutable (qui commence par exécuter `main`). Elle est définie par l'en-tête de la fonction : type de retour, nom, argument ; les accolades contiennent les instructions composant la fonction. L'instruction `return` est une instruction de retour à la fonction appelante ;
- appel de la fonction `printf` — déclarée dans `stdio.h` — avec une chaîne en paramètre.

Principe de la compilation élémentaire

1. **Édition du fichier source :** fichier texte contenant le programme — nécessite un éditeur de texte (emacs, vi).
2. **Traitement par le préprocesseur :** le fichier source est traité par un *préprocesseur* qui fait des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc).
3. **La compilation :** le fichier engendré par le préprocesseur est traduit en *assembleur* i.e. en une suite d'instructions associées aux fonctionnalités du microprocesseur (faire une addition, etc).
4. **L'assemblage :** transforme le code assembleur en un fichier *objet* i.e. compréhensible par le processeur
5. **L'édition de liens :** afin d'utiliser des librairies de fonctions déjà écrites, un programme est séparé en plusieurs fichiers source. Une fois le code source assemblé, il faut *lier* entre eux les fichiers objets. L'édition de liens produit un fichier *exécutable*.

Programmation sans filet $\Rightarrow$ maîtrise indispensable du langage

En C, le programmeur est censé maîtriser parfaitement le langage et son fonctionnement.

Les tests d'erreurs et de typages ne sont fait qu'au moment de la compilation i.e. rien n'est testé lors de l'exécution (les convertions de types, l'utilisation d'indices de tableau supérieurs à sa taille, etc).

De plus, le compilateur est `laxiste` : il vous laisse faire tout ce qui a un sens (même ténu) pour lui.

Un programme peut donc marcher dans un contexte (certaines données) et provoquer des erreurs dans un autre.

Outils utilisés en TP

Vous êtes libres d'utiliser vos outils préférés. . .

. . . à condition que ceux-ci soient :

- `emacs` ou `vi` pour l'édition de vos fichiers sources ;
- `gcc` pour la compilation. Il s'agit du gnu C compiler et on peut y adjoindre certains drapeaux. Par exemple,

  ```
  % gcc -Wall -ansi -pedantic foo.c
  ```

  indique que vous désirez voir s'afficher tous les avertissements du compilateur (très recommandé) ;
- `gdb` pour l'exécution pas à pas et l'examen de la mémoire (`ddd` est sa surcouche graphique).

Une séance sera consacrée à la compilation séparée et à certains outils de développement logiciels.

Les constantes numériques

Les entiers machines : on peut utiliser 3 types de notations :

- la notation décimale usuelle (66, -2) ;
- la notation octale (commençant par un 0 (en C, la constante 010 est différente de 10)) ;
- la notation hexadécimale (commençant par un $0x$ (en C, la constante $0x10$ est égale à 16)) ;

Les réels machines : ne sont pas en précision infinie et sont notés par :

- *mantisse* -273.15, 3.14 et
- *exposant* indiqué par la lettre e : $1.4e10$, $10e-15$.

Les constantes (non) numériques

Les caractères se répartissent en deux types :

- *imprimable* spécifié par `'`. Ainsi, `'n'` correspond à l'entier 110 (et par le biais d'une table au caractère n) ;
- *non imprimable* qui sont précédés par \. Ainsi, `'\n'` correspond à un saut de ligne.
  On peut aussi utiliser pour ces caractères la notation `'\`ξ`'` avec ξ le code ASCII octal associé (cf. `% man ascii`).

Identificateur : un nom associé à de l'espace mémoire

Les identificateurs servent à manipuler les objets du langage i.e. manipuler de l'espace mémoire auquel on donne un nom.
Ils désignent de la mémoire contenant des données (des variables, etc.) ou de la mémoire contenant du code à exécuter (des fonctions).
Ils ne peuvent pas commencer par un entier (mais peuvent les contenir). C distingue les majuscules des minuscules : (x et X sont deux identificateurs différents).
Certaines règles de bon usage sont à respecter ;

- les caractères non ASCII — i.e. non portables — ne devraient pas être utilisés (pas d'accent) ;
- les identificateurs débutant par un *blanc_souligné* sont propres au système d'exploitation et ne devraient pas être utilisés par un programmeur en dehors de l'OS ;
- il vaut mieux choisir des identificateurs parlant.

Les entiers machines

Il y a 8 types associés aux entiers machines :

- `int` est le type de base des entiers signés et correspond au mot machine (historiquement 16 bits ou actuellement 32 bits, bientôt 64). Les entiers représentables sont donc dans l'intervalle $[-2^{31}, 2^{31}[$. Ce type est modifiable par un attribut `unsigned`
- `unsigned int` est le type de base des entiers non signés codés sur le même nombre d'octets (donc compris entre $[0, 2^{32} - 1]$).

Les entiers machines

Ces types sont modifiables par les attributs `short` et `long` :

- `short int` est un type plus *court* (codé sur 16 bits) et représentant les entiers dans $[-2^7, 2^7[$.
- `long int` est un type plus *long* (codé sur 32 bits — pour des raisons historiques) et représentant les entiers dans $[-2^{31}, 2^{31}[$.
- `long long int` est un type encore plus *long* (codé sur 64 bits) et représentant les entiers dans $[-2^{63}, 2^{63}[$.

La taille dépend de l'architecture de la machine et peut varier.

Les entiers machines

On indique comment typer une constante en utilisant les suffixes :

u ou U	unsigned (int ou long)	550u
l ou L	long	123456789L
ul ou UL	unsigned long	12092UL

On peut manipuler en C des entiers plus grand en employant des représentations non spécifiées par le langage (tableaux, listes chaînées — voir la librairie gnu multiprecision (GMP) par exemple).

Les entiers machines

Pour connaître le nombre d'octets associés à un type, on utilise le mot clef du langage `sizeof`. Par exemple,

```
int
main
(void)
{
  return sizeof(long long int);
}
```

est un programme qui retourne le nombre d'octets codant le type `long long int`.

```
% # sur les machines de tp
% gcc foo.c ; ./a.out
% echo $?
8
%
```

Le type `char`

Le langage C associe aux caractères le type `char` généralement codé sur 1 octet. Le type `char` est généralement signé de -128 à 127.

Historiquement, le code ASCII nécessitait 7 bits.
Le reste du monde utilisant des accents, l'ISO définit une foultitude de codage sur 1 octet : le code ASCII de base jusqu'à 127 et le reste à partir de 128.
Le type `char` est modifiable par l'attribut `unsigned` pour coder les entiers de 0 à 255.

Il faut bien garder à l'esprit que le type `char` représente des entiers dont la correspondance avec des lettres de l'alphabet est faîte en dehors du langage par une table (voir `% man ascii`).

Le type flottant (float) et les "booléens"

Il existe deux types pour le codage des réels en C :

- `float` pour les flottants simple précision 32 bits généralement ;
- `double` pour les flottants double précision 64 bits généralement.

On ne peut pas les modifier (`unsigned`, `short`) comme les autres types si ce n'est pour :

- `long double` qui correspond à un codage sur 12 octets.

Les booléens sont — sémantiquement — représentés par les entiers :

- valeur logique fausse : valeur nulle `0` ;
- valeur logique vraie : tout entier $\neq$ `0`.

Schématiquement, une variable correspond à un emplacement en mémoire. Dans le code source, ce dernier est manipulé par l'entremise de l'identificateur de la variable. Avant utilisation, toutes les variables doivent être :

- soit *définies localement* ce qui correspond à l'allocation d'une zone mémoire (segment de pile) ;
- soit *définies globalement* ce qui correspond :
  - à la création d'une entrée dans la table des symboles ;
  - à l'allocation d'une zone mémoire (segment de données) ;
  - au stockage de cette adresse dans la table ;
- soit *déclarées* `extern` ce qui correspond à :
  - une variables définies dans un autre fichier source ;
  - la création d'une entrée dans la table des symboles ;
  - *mais pas à son allocation* : l'adresse est inconnue à l'assemblage ;
  - (l'adresse est résolue à l'édition de liens).

Pour déclarer une variable, il faut faire suivre le nom du type par un identificateur. Par exemple :

```
int i;          /* pas tr\'es */
int j, k;       /* explicite  */
short int s;    /* ces identificateurs :-( */
float f;
double d1,d2;
```

Bien qu'il soit vivement conseillé de découpler déclaration et initialisation, on peut affecter des variables à la déclaration :

- Caractère : **char** `nom =' A ';`
- Chaîne de caractères : **char** `*foo =" bar ";`
- Entier machine : **int** `nom = 666;`
- Flottant machine : **float** `nom = 3.14;`

Implicitement, nous venons de nous servir de 2 opérateurs :

- la virgule permet de définir une suite ;
- l'opérateur d'affectation =.

Qualificatif précisant le type

On peut modifier les types en précisant le codage machine à l'aide des mots-clefs **signed**, **unsigned**, **short**, **long**.

short int	$[-2^7, 2^7[$	**int**	$[-2^{31}, 2^{31}[$
unsigned int	$[0, 2^{32}[$	**long int**	$[-2^{63}, 2^{63}[$
unsigned short int	$[0, 2^8[$	**unsigned long int**	$[0, 2^{64}[$

On peut aussi modifier les flottants par les mots-clefs **double** et **long double**.

Une expression correspond à la composition d'identificateurs et d'opérateurs. Elle se termine par un point virgule.

L'action d'un opérateur sur un identificateur peut avoir 2 types de conséquences :

- retourner la valeur de l'expression ;
- un effet latéral portant sur l'identificateur.

Par exemple, l'affectation `foo = 2` provoque :

- l'effet latéral : l'entier 2 est affecté à la variable `foo` ;
- et retourne la valeur qui vient d'être affectée.

On peut donc avoir une expression du type :

```
bar = foo = 2 ;
```

L'opérateur ++ provoque :

- l'effet latéral : incrémente l'expression ;
- et retourne la valeur qui vient d'être obtenue.

```
foo = ++bar ;
```

Comment déterminer la sémantique d'une expression ?

Il faut maîtriser l'action des opérateurs :

- opérateurs arithmétiques classiques :
  $+$ addition $-$ soustraction $*$ multiplication
  $/$ division % reste de la division
- les opérateurs relationnels $>$, $<$, $<=$, $>=$, $==$
- les opérateurs logiques booléens && et, || ou, ! non
- les opérateurs logique bit à bit & et, | ou inclusif, ~ ou exclusif
- les opérateurs d'affectation composée $+=$, $-=$, $/=$, $*=$, $\%=$, etc.
- les opérateurs d'incrémentation et de décrémentation
- l'opérateur conditionnel ternaire foo = x>=0 ? x :-x.
- conversion de type char foo = (char) 48.14 ;

Priorité et ordre d'évaluation des opérateurs

L'instruction `X *= Y + 1;` n'est pas équivalente à `X = X * Y + 1;`

16	`()` `[]` `->` `.`	G
15	`++` `--` (postfixé)	D
14	`!` `~` `++` `--` (préfixé) `-` (unaire)	D
	`*` (indirection) `&` (adresse) `sizeof`	D
13	`*` (multiplication) `/` `%`	G
12	`+` `-`	G
11	`<<` `>>`	G
10	`<` `<=` `>` `>=`	G
9	`==` `!=`	G
8	`&` (et bit à bit)	G
7	`^`	G
6	`|`	G
5	`&&`	G
4	`||`	G
3	`?:`	D
2	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	D
1	`,`	G

Pour l'opérateur $\circ$, la priorité G indique que l'expression $\text{exp}_1 \circ \text{exp}_2 \circ \text{exp}_3$ est évaluée de gauche à droite.

Considérons la situation suivante :

```
int foo = 2 ;
unsigned char bar = 3 ;
float var = foo +  bar ;
```

Les opérateurs ne pouvant agir que sur des données de types homogènes, il y a 2 conversions de type dans cet exemple :

- l'opérateur + provoque — si nécessaire — la conversion d'une des opérandes après son évaluation ;
- l'opérateur = provoque — si nécessaire — la conversion de l'opérande de droite dans le type de l'identificateur de gauche après son évaluation et avant son affectation à cet identificateur.

En cas de doute, il faut utiliser la conversion de type explicite :

```
int foo = 2;
unsigned char bar = 3 ;
float var = foo / bar ; var = ((float) foo / (float) bar) ;
```

Opérateurs arithmétiques usuels

Addition

- Syntaxe : $\Rightarrow$ $expression_1$ + $expression_2$
- Sémantique :
  - évaluation des expressions et calcul de l'addition ;
  - retourne la valeur de l'addition ;
  - ordre indéterminé d'évaluation des expressions ;
  - conversion éventuelle d'une des opérandes après évaluation.

Soustraction

- Syntaxe : l'opérateur peut être utilisé de manière unaire ou binaire : $\Rightarrow$ - *expression*
  $\Rightarrow$ $expression_1$ - $expression_2$

Multiplication

- Syntaxe : $\Rightarrow$ $expression_1$ * $expression_2$
- Sémantique : voir addition.

Division

- Syntaxe : $\Rightarrow$ *expression*$_1$ / *expression*$_2$
- Sémantique : comme l'addition
  - pas de distinction entre division entière ou réelle ;
  - division entière $\Leftrightarrow$ *expression*$_1$ et *expression*$_2$ entières ;
  - cas de la division entière :
    - opérandes positives : arrondi vers 0 $(13/2 = 6)$;
    - une opérande négative : dépend de l'implantation ;
      $13/-2 = -6$ ou -7.

Modulo

- Syntaxe : $\Rightarrow$ *expression*$_1$ % *expression*$_2$
- Sémantique :
  - *expression*$_1$ et *expression*$_2$ entières ;
  - reste de la division entière ;
  - si un opérande négatif : signe du dividende en général ;
  - Assurer que `b * (a / b) + a % b` soit égal à `a`.

Opérateurs de comparaison

- Syntaxe :

  $\Rightarrow$ *expression*$_1$ *opérateur* *expression*$_2$

  où *opérateur* est l'un des symboles :

opérateur	sémantique
>	strictement supérieur
<	strictement inférieur
>=	supérieur ou égal
<=	inférieur ou égal
==	égal
!=	différent

- Sémantique :
  - évaluation des expressions puis comparaison ;
  - valeur rendue de type `int` (pas de type booléen) ;
  - vaut 1 si la condition est vraie ;
  - vaut 0 si la condition est fausse ;
  - Ne pas confondre : test d'égalité (==) et affectation (=).

Opérateurs logiques

Et logique

- Syntaxe :

  $\Rightarrow$ *expression*$_1$ `&&` *expression*$_2$

- Sémantique : *expression*$_1$ est évaluée et :
  1. si valeur nulle : l'expression `&&` rend `0` ;
  2. si valeur non nulle : *expression*$_2$ est évaluée et
     - **2.1** si valeur nulle : l'expression `&&` rend `0` ;
     - **2.2** si valeur non nulle : l'expression `&&` rend `1`.
- Remarque :

  expression$_2$ non évaluée si *expression*$_1$ fausse

  Utile : `(n != 0) && (N / n == 2)`

  Désagréable : `(0) && (j = j - 1)`.

Ou logique

- Syntaxe : $\Rightarrow$ *expression*$_1$ `||` *expression*$_2$
- Sémantique : *expression*$_1$ est évaluée et :
  1. si valeur non nulle : l'expression `||` rend `1` ;
  2. si valeur nulle : *expression*$_2$ est évaluée et
     - 2.1 si valeur nulle : l'expression `||` rend `0` ;
     - 2.2 si valeur non nulle : l'expression `||` rend `1`.
- Remarque : *expression*$_2$ non évaluée si *expression*$_1$ vraie

Non logique

- Syntaxe : $\Rightarrow$ `!` *expression*
- Sémantique : *expression* est évaluée et :
  1. valeur nulle : l'opérateur `!` délivre `1` ;
  2. valeur non nulle : l'opérateur `!` délivre `0`.

Opérateurs de traitement des bits

Non bit à bit

- Syntaxe : ⇒ `~` *expression*
- Sémantique :
  - évaluation de *expression* ⇒ type entier ;
  - calcul du non bit à bit sur cette valeur ;
  - rend une valeur entière.

Et bit à bit

- Syntaxe : ⇒ $expression_1$ `&` $expression_2$
- Sémantique : évaluation de $expression_1$ et $expression_2$ qui doivent être de valeur entière.

Ou bit à bit

- Syntaxe : ⇒ $expression_1$ `|` $expression_2$

Ou exclusif bit à bit

- Syntaxe : ⇒ $expression_1$ `^` $expression_2$
- Sémantique : voir et bit à bit.

Décalage à gauche

- Syntaxe : $\Rightarrow$ $expression_1$ << $expression_2$
- Sémantique :
  - évaluation de $expression_1$ et $expression_2$;
  - doivent être de valeur entière, positive pour $expression_2$;
  - $expression_1$ décalée à gauche de $expression_2$ bits en remplissant les bits libres avec des zéros.

Décalage à droite

- Syntaxe : $\Rightarrow$ $expression_1$ >> $expression_2$
- Sémantique : voir décalage à gauche :
  - si $expression_1$ `unsigned` : décalage logique
    les bits rentrants à droite sont des zéros ;
  - si $expression_1$ signée : décalage arithmétique
    les bits rentrants à droite sont égaux au bit de signe.

Instruction composée (du bon usage des accolades).

- Syntaxe : *instruction-composée* :

  $\Rightarrow$ { *liste-de-déclarations*$_{option}$ *liste-d'instructions*$_{option}$ } *liste-de-déclarations* :

  $\Rightarrow$ *déclaration*

  $\Rightarrow$ *liste-de-déclarations déclaration*

  liste-d'instructions :

  $\Rightarrow$ *instruction*

  $\Rightarrow$ *liste-d'instructions instruction*

  Une expression isolée n'a pas de sens.

Attention à l'usage des instructions composées et des variables :

```
#include <stdio.h>
char foo = 'c' ;
int main(void){
  printf(" %c ",foo) ;
  char foo = 'a' ; \*on n'utiliser qu'un nom de variable*\
  printf(" %c ",foo) ;
  {
  char foo = 'b' ; \*mais c'est une tr\'es mauvaise id\'ee*\
  printf(" %c ",foo) ;
  }
  printf(" %c \n",foo) ;
  return 0 ;
}
```

```
% gcc InstructionsComposees.c ; a.out
 c a b a
```

De toutes façons :

```
%gcc -Wall -ansi -pedantic InstructionsComposees.c
InstructionsComposees.c: In function 'main':
InstructionsComposees.c:9: warning:
ISO C89 forbids mixed declarations and code
```

- Sémantique des instructions composées : 2 objectifs
  1. Grouper un ensemble d'instructions en une seule instruction ;
  2. Déclarer des variables accessibles seulement à l'intérieur de *instruction-composée*

  $\Rightarrow$ *Structure classique de blocs*

- Remarques
  - pas de séparateur dans *liste-d'instructions* :

    le ; est un terminateur pour les expressions
  - accolades (`{}`) correspondant au `begin end` de Pascal.

- Syntaxe : *instruction-conditionnelle* :

  $\Rightarrow$ `if` `(` *expression* `)` *instruction*$_1$

  $\Rightarrow$ `if` `(` *expression* `)` *instruction*$_1$ `else` *instruction*$_2$

- Remarques sur la syntaxe :
  - *expression* doit être parenthésée ;
  - pas de mot clé `then` ;
  - ambiguïté du `else` :

    ```
    if (a > b) if (c < d) u = v; else i = j;
    ```

    Règle : relier le `else` au premier `if` de même niveau d'imbrication n'ayant pas de `else` associé

    ```
    if (a > b) { if (c < d) u = v; } else i = j;
    ```

- Sémantique :
  - évaluation de *expression* ;
  - si valeur non nulle : exécution de $instruction_1$;
  - si valeur nulle : exécution de $instruction_2$ si elle existe.
- Remarques sur la sémantique :
  - `if` : teste égalité à zéro de *expression* ;
  - *expression* : pas forcément une comparaison ;
  - *expression* : comparable à zéro ;

```
if (a != 0) { ... }
if (a) { ... }
```

Instruction à choix multiples

Syntaxe :

```
switch ( expression )
{
    case   expr-cste1   :   liste-d'instructions1 option
                                                  breakoption ;
    case   expr-cste2   :   liste-d'instructions2 option
                                                  breakoption ;

                            ...

    case   expr-csten   :   liste-d'instructionsn option
                                                  breakoption ;
    default   :   liste-d'instructionsoption
}
```

1. Évaluation de *expression* ;
2. Résultat comparé avec *expr-cste*$_1$, *expr-cste*$_2$, etc. ;
3. Première *expr-cste*$_i$ égale à *expression* : exécution de *liste-d'instructions* correspondante ;
4. Instruction `break` : termine l'exécution du `switch` ;
5. Si aucune *expr-cste*$_i$ égale à *expression* : exécution de *liste-d'instructions* de l'alternative `default` si celle-ci existe, sinon on ne fait rien.

Remarques :

- *expr-cste*$_i$: valeur connue à la compilation (constante) ;
- *expr-cste*$_i$: pas deux fois la même valeur ;
- s'il n'y a pas de `break` à la fin d'un `case` : exécution des *liste-d'instruction* des `case` suivants ;
- l'alternative `default` est optionnelle.

Exemple utilisant le break :

```
int nb = 1 ;
switch(nb){
   case 1 : printf("un"); break;
   case 2 : printf("dos"); break;
   case 3 : printf("tres"); break;
   default : printf("erreur: pas dans la chanson\n");
}
```

Exemple n'utilisant pas le break :

```
switch(c){
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9': nb_chiffres++; break;
   default: nb_non_chiffres++;
}
```

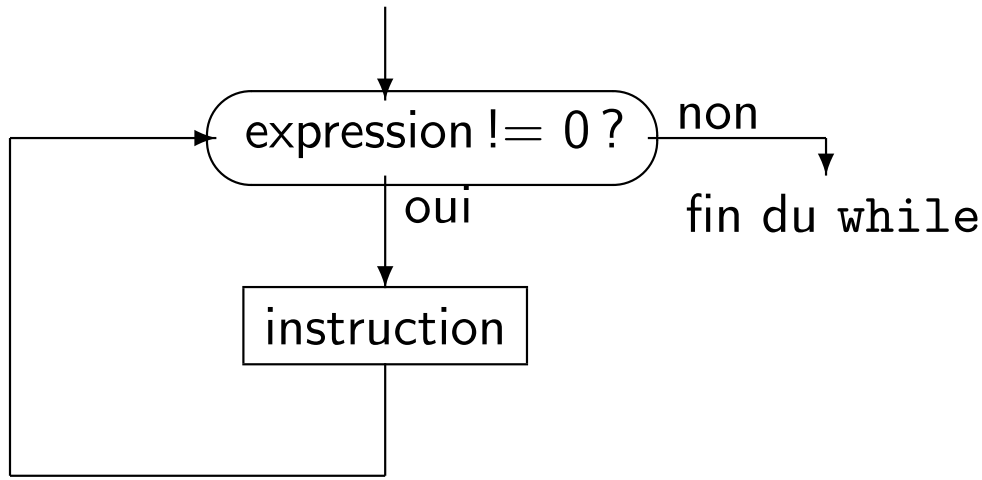
Instructions itératives

Trois instructions d'itération : *instruction-itérative* :

- ⇒ *instruction-while*
- ⇒ *instruction-do*
- ⇒ *instruction-for*

Instruction while

- Syntaxe : ⇒ `while` (*expression*) *instruction*
- Sémantique : boucle avec test en début d'itération

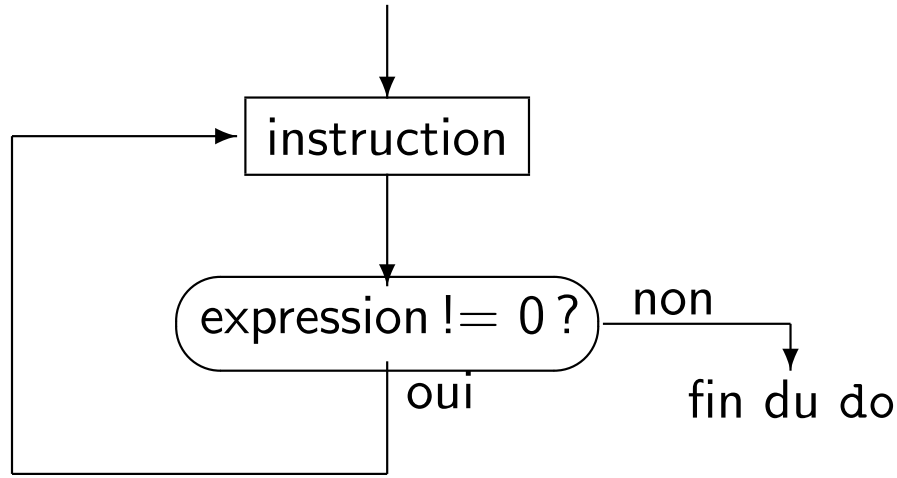


- Exemple : somme des $n = 10$ premiers entiers

```
int n = 10; int i = 1; int somme = 0;
while (i <= n) { somme = somme+i; i = i+1; }
```

Instruction do ... while

- Syntaxe : ⇒ `do` *instruction* `while ( expression ) ;`
- Sémantique : boucle avec test en fin d'itération



- Exemple : somme des $n = 10$ premiers entiers

```
int n = 10; int i = 1; int somme = 0;
if (i <= n) do {
    somme = somme + i;
    i = i + 1;
  } while (i <= n);
```

Instruction for

Syntaxe :

⇒ `for (` *expression*$_{1\ option}$ `;`

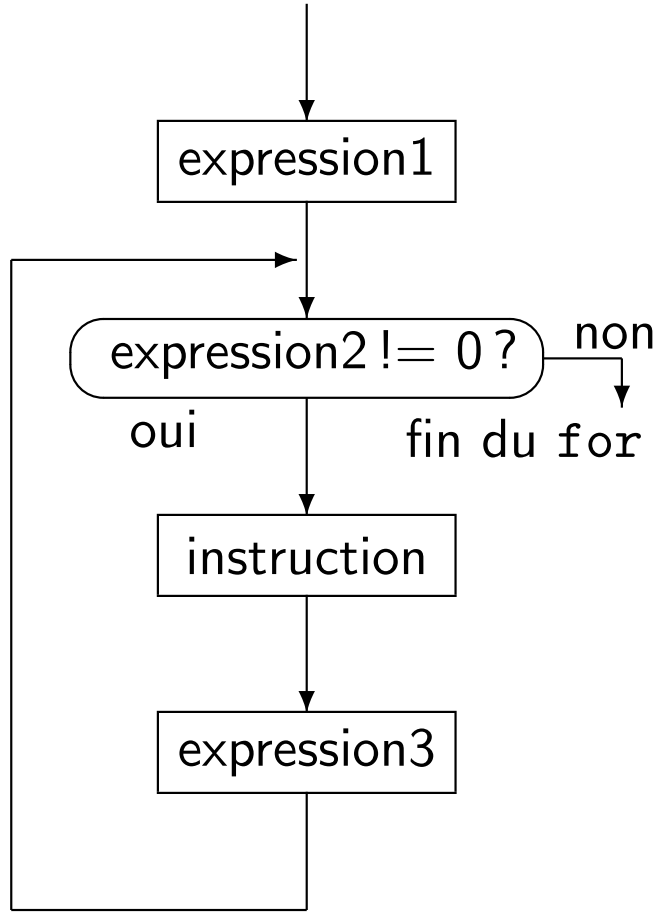
expression$_{2\ option}$ `;`

expression$_{3\ option}$ `)`

instruction

Sémantique :

itération bornée



Instruction for (suite)

- Remarques
  - $expression_1$ et $expression_3$: valeurs inutilisées
    $\Rightarrow$ effet latéral : expressions d'affectation
  - $expression_1$: instruction d'initialisation ;
  - $expression_3$: instruction d'itération ;
  - $expression_2$: expression de test (arrêt quand nulle) ;
  - *instruction* : corps de la boucle.
- Exemple : somme des $n = 10$ premiers entiers

```
int i ; int n = 10; int somme = 0;
for (i = 0; i <= n; i = i + 1)
    somme = somme + i;
```

- Même exemple avec un corps de boucle vide

```
int i ; int n ; int somme ;
for (i=0,n=10,somme=0; i<n; somme=somme+(i=i+1)) ;
```

Attention à la confusion avec d'autres langages :

```
#include <stdio.h>

int main(void){

  int foo ;
  for( foo=0 ; foo<10 ; foo++)
          printf("%d\n",foo) ;

  for( int bar=0 ; bar<10 ; bar++)
          printf("%d\n",bar) ;

  return 0 ;
}
```

La compilation donne :

```
% gcc for.c
for.c: In function ‘main’:
for.c:9: ‘for’ loop initial declaration used outside C99 mode
```

Instruction `goto` (possible mais à proscrire) :

- Syntaxe : ⇒ `goto` *identificateur* ;
- Sémantique :
  - toute instruction est étiquetable ;
  - si on la fait précéder d'un identificateur suivi du signe :
  - et d'un identificateur : *étiquette* ;
  - `goto` : transfère le contrôle d'exécution à l'instruction étiquetée par *identificateur*.
- Remarques :
  - étiquettes visibles que dans la fonction où elles sont définies ;
  - s'utilise pour sortir de plusieurs niveaux d'imbrication ;
  - permet d'éviter des tests répétitifs ;

```
for (...) {     for (...) {
    ...
    if (catastrophe) goto erreur;
}
erreur: printf("c'est la cata...");
```

Rupture de contrôle

Instruction `break` :

- Syntaxe : *instruction* : $\Rightarrow$ `break ;`
- Sémantique :
  - provoque l'arrêt de la première instruction `for`, `while`, `do` ou `switch` englobante ;
  - reprend l'exécution à l'instruction suivant l'instruction terminée.

Instruction `continue` :

- Syntaxe : ⇒ `continue ;`
- Sémantique :
  - uniquement dans une instruction `for`, `while` ou `do` ;
  - provoque l'arrêt de l'itération courante ;
  - passage au début de l'itération suivante ;
  - équivalent `goto suite` avec :

```
while(...) {          do {              for(...) {
  ...                   ...               ...
suite: ;              suite: ;          suite: ;
}                     } while(...);     }
```