Initiation au C
cours n°1

Antoine Miné

École normale supérieure

15 février 2007

Organisation du cours

Organisation des séances

2 heures :

- $\simeq$ 45mn de cours en salle R, suivi de
- $\simeq$ 1h15mn de TPs en salle S ou T,

(à adapter)

Supports de cours

Disponibles progressivement sur le site WEB :

http://www.di.ens.fr/~mine/enseignement/prog2006

- transparents,
- feuilles de TPs, corrections,
- fiches thématiques (syntaxe, compilation, débogage, etc.)

Organisation du cours

Cours libre : pas d'examen, pas d'ECTS

Le cours peut s'adapter aux besoins

Contact

Antoine Miné
mine@di.ens.fr
département d'informatique
bureau S14, saumon -1

Programme d'aujourd'hui

Aujourd'hui

- généralités culturelles
- premier programme, compilation
- variables, valeurs, types, expressions
- affichage sur l'écran
- conditionnelles simples
- boucles simples

Repères historiques

Repères historiques

Origine

Denis Ritchie et Ken Thomson (Bell Labs) cherchent un langage pour reprogrammer UNIX de façon portable.
Le C est dérivé du B (1969), BCPL (1966), CPL (1960), etc.

Historique

- 1969 UNIX par Ken Thomson (assembleur pour DEC PDP-7)
- 1972 invention du C par Denis Ritchie
- 1973 UNIX en C par Denis Ritchie et Ken Thomson (PDP-11)
- 1978 *The C Programming Language* : C K&R
- 1989 1ère normalisation : ANSI C (C89), ISO C90
- 1999 2ème normalisation : ISO C99

Le C aujourd'hui

Le C est toujours très utilisé :

- systèmes d'exploitations : Linux ≃6,8 millions de lignes
- bibliothèques : GNU libc ≃1 million de lignes
- compilateur : gcc (C, C++, ada, etc.) ≃1.4 million de lignes
- Internet : Apache (serveur WEB) ≃250 000 lignes
- applications : GIMP (retouche d'images) ≃700 000 lignes

Langages inspirés du C

- compatibles : C++, Objective-C
- de syntaxe similaire : Java, C#, etc.

Généralités sur le C

Un langage impératif

Programme = séquence de :

- déclarations (soit $X\dots$),
- instructions (actions à effectuer),

chacune terminée par un point-virgule ;

Paradigme impératif

Les instructions sont exécutées en séquence.

Autres paradigmes : langages logiques, orientés-objets, fonctionnels, multi-paradigmes, etc.

Un langage structuré

Bloc = suite d'instructions délimitée par `{` et `}`

Structures de contrôle

Contrôlent l'exécution d'un bloc :

- conditionnelles `if`, `else`,
- boucles `while`, `for`,

Il ne faut pas "sauter" d'une instruction à une autre (`goto`).

Les blocs et structures de contrôle peuvent s'imbriquer.

Un langage procédural

Fonction =

- bloc d'instructions,
- déclarée une fois, réutilisable de nombreuses fois,
- peut prendre des arguments et retourner une valeur,
- a un effet (modification de la mémoire, affichage à l'écran),
- peut être définie dans un autre module (bibliothèques).

Fonctions prédéfinies

- bibliothèque standard (affichage, fichiers, mémoire),
- bibliothèque mathématique, etc.

Il faut les importer par une directive spéciale `#include`.

Un langage déclaratif

Variable = morceau de mémoire où stocker une valeur

Attention

Toute variable doit être déclarée avant d'être utilisée !

Durée de vie d'une variable

- variable globale,
- variable locale à un bloc ou une fonction,
- mémoire dynamique.

Un langage typé

Typage

Les variables sont typées.
Le type est fixé lors de la déclaration.

Le type d'une variable détermine :

- l'ensemble de valeurs possibles (entiers, flottants, etc.),
- la quantité de mémoire à réserver (`sizeof`),
- le codage utilisé en mémoire,
- la sémantique des opérations (division /),
- permet de vérifier la cohérence du programme.

Un langage typé

Les types du C

- types entiers (`int`, `unsigned`, `char`, etc.),
- types flottant : (`double`, `float`),
- types composés : tableaux, enregistrements,
- types pointeurs : adresse des variables en mémoire,
- types de fonctions (prototypes),
- types définis par l'utilisateur.

Un langage (assez) bas niveau

Le C permet :

- des opérations mal définies ou invalides,
- l'accès direct à la mémoire (pointeurs),
- le contournement du système de types.

Inconvénients

- dangereux
- peu d'abstraction

Avantages

- rapidité
- contrôle total sur la mémoire

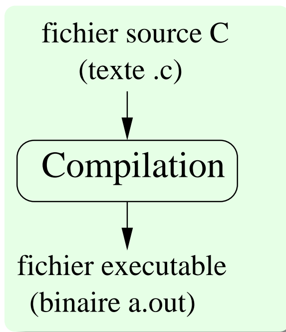
Le système est là pour protéger les autres programmes !
(`Segmentation Fault`)

Un langage compilé

Source = fichier `.c` contenant le texte du programme en C.

Langage binaire : seul compréhensible par l'ordinateur.

La **compilation** transforme le source en binaire.



La compilation

Opération complexe :

- analyse syntaxique et typage,
- gestion des ressources très bas niveau (registres, adresses),
- découpage en instructions très élémentaires.

Un langage compilé

Compilation et exécution :

- compiler une fois, exécuter une ou plusieurs fois,
- recompiler si le source change,
- recompiler pour un autre système / microprocesseur.

Avantages de la compilation

- rapidité (exécution en langage machine),
- vérification complète du programme (syntaxe, typage),
- optimisation globale,
- liens avec d'autres langages,
- binaire autonome.

Différent des langages interprétés (shell, Perl, BASIC, etc.) !

Un langage normalisé

Normalisation imposée par le succès du langage C :

- disponible sur des systèmes très différents,
- chaque constructeur fournit son compilateur.

Attention : tout n'est pas normalisé !

- extensions non standard (dépendent du compilateur),
- bibliothèques non standard (dépendent du système),
- comportements indéfinis dans la norme.

Avantage de la normalisation

Il est possible d'écrire des programmes portables avec un peu de soins !

Premier programme en C

Premier programme

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

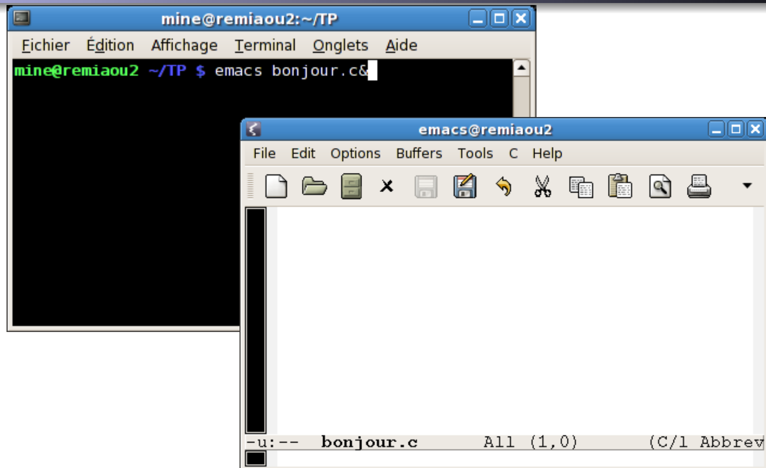
Effet :

- affiche `Bonjour tout le monde !`,
- retourne le code 0 (tout s'est bien passé).

Compilation et exécution

Lancement de l'éditeur en tâche de fond (&).

Compilation et exécution



Lancement de l'éditeur en tâche de fond (&).

Compilation et exécution

mine@remiaou2:~/TP

Fichier Édition Affichage Terminal Onglets Aide

```
mine@remiaou2 ~/TP $ emacs bonjour.c&
```

emacs@remiaou2

File Edit Options Buffers Tools C Help

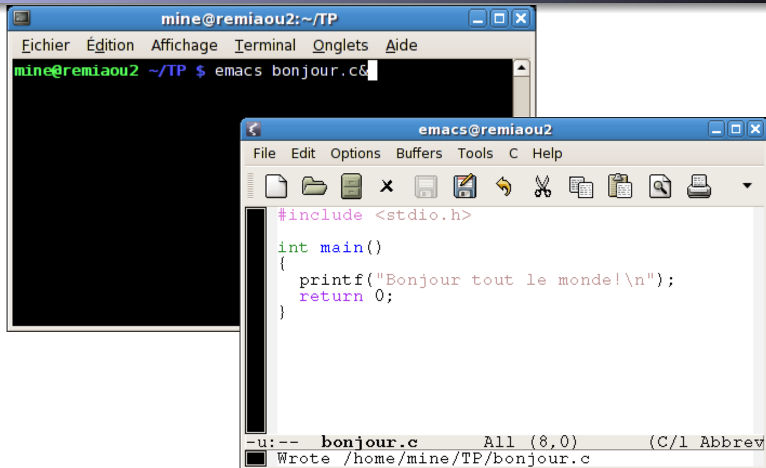
```c
#include <stdio.h>

int main()
{
  printf("Bonjour tout le monde!\n");
  return 0;
}
```

-u:** bonjour.c All (8,0) (C/l Abbrev

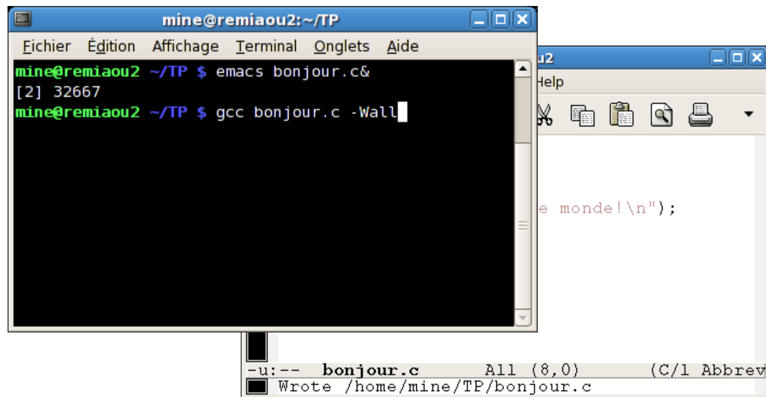
On tape le texte du programme.

Compilation et exécution



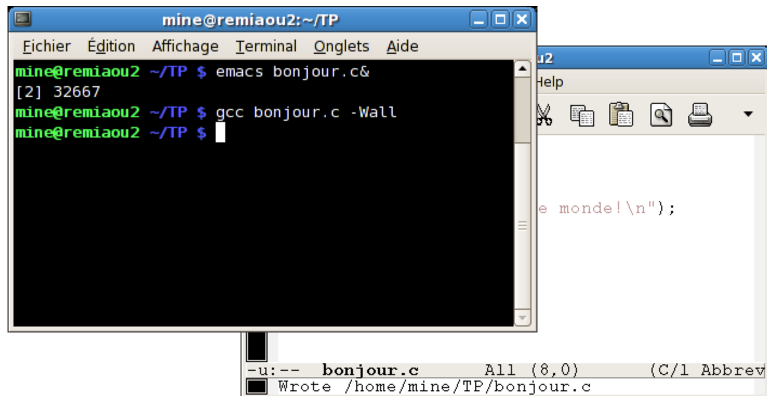
Il ne faut pas oublier de sauvegarder.

Compilation et exécution



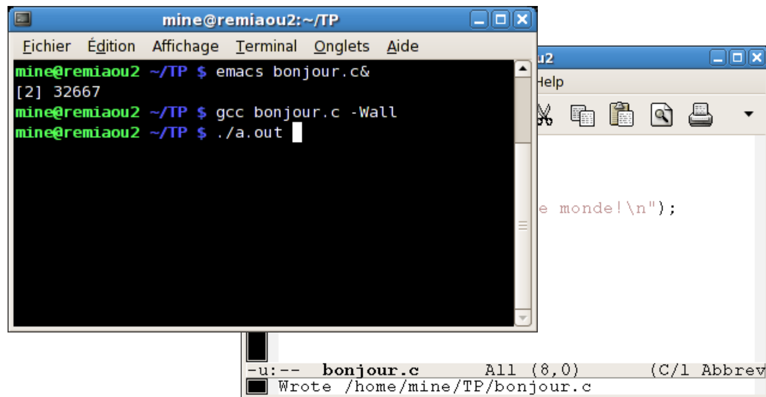
Lancement de la compilation avec gcc.

Compilation et exécution



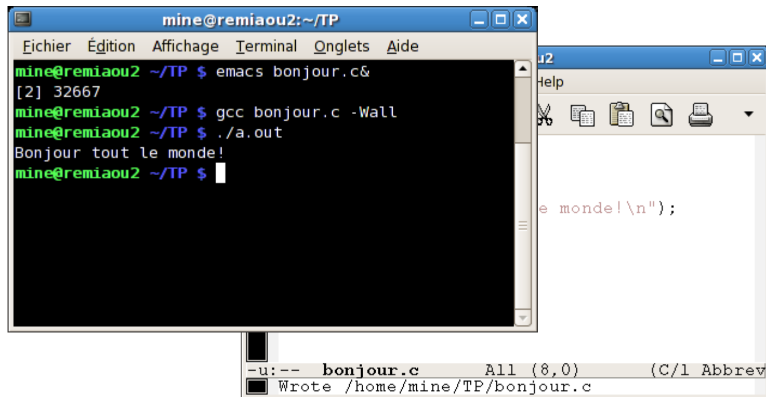
Si le compilateur ne dit rien, tout s'est bien passé.
Un fichier `a.out` a été créé.

Compilation et exécution



Lancement de l'exécutable.

Compilation et exécution



Le programme s'exécute et rend la main.

Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

Tout programme C doit contenir une fonction appelée `main`.
L'exécution commence au début de `main`.

Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

Par convention, la fonction `main` renvoie un code de retour :

- il est de type `int` (entier),
- la convention est de retourner 0 si tout se passe bien,
- les parenthèse de `return` sont facultatives,
- le code de retour est exploitable depuis le shell.

Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

La fonction printf permet d'écrire sur l'écran.

- elle fait partie de la bibliothèque C standard,
- elle doit être importée depuis l'en-tête stdio.h.

Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

`printf` prend en argument une chaîne de caractères :

- tapée entre guillemets `"`,
- `\` sert à entrer des caractères spéciaux :
  `\n` signifie “retour à la ligne”.

Exemple d'erreur

bonjour.c faux

```
1  #include <stdio.h>
2
3  int main()
4  {
5     printf("Bonjour tout le monde!\n")
6     return(0);
7  }
```

Exemple d'erreur

bonjour.c faux

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("Bonjour tout le monde!\n")
6    return(0);
7 }
```

Résultat

```
$ gcc bonjour.c -Wall
bonjour.c: In function 'main':
bonjour.c:6: error: expected ';' before 'return'
bonjour.c:7: warning : control reaches end of ...
```

Il manque un `;`. Aucun `a.out` n'est généré.

Exemple d'avertissement

bonjour.c faux

```
1 int main()
2 {
3    printf("Bonjour tout le monde!\n");
4    return(0);
5 }
```

Résultat

```
$ gcc bonjour.c -Wall
bonjour.c: In function 'main':
bonjour.c:3: warning: implicit declaration of function
'printf'
```

Il manque un `#include <stdio.h>`.

C'est un avertissement non fatal généré par `-Wall`.

Les options `-Wall` et `-Wextra`

`-Wall` attire l'attention, entres autres, sur :

- les oublis d'imports `#include`,
- les ambiguïtés syntaxiques courantes,
- les incohérences de types.

La norme est très laxiste ne considère pas ces points comme des erreurs !

`-Wextra` ajoute des avertissements supplémentaires.

Toujours compiler avec `-Wall` et `-Wextra` !

Espacement

L'espacement et les sauts de lignes sont libres.

Exemple correct mais illisible

```
     # include <stdio.h>
int main                           (
      ){
printf
  ("toto\n"
 );return(0)   ;}
```

Exceptions

- `#include <stdio.h>` doit être sur une seule ligne,
- les sauts de ligne comptent dans les chaînes de caractères.

Commentaires

Commentaires : tout ce qui est entre /* et */ est ignoré.

Exemple commenté

```
#include <stdio.h> /* pour avoir printf */

/* la fonction principale
*/
int main(/* rien ici */)
{
  printf("toto\n");
  return(0); /* OK */
}
```

Conseils :
- indentez votre code (tabluation sous Emacs),
- commentez votre code.

Variables et expressions

Déclaration de variables

Les variables doivent être déclarées avant d'être utilisées.

Syntaxe

```
type nom;
```

Exemple

```
int main()
{
  int x;
  double y;
  x = 12;
  y = x/3.0;
  return 0;
}
```

Durée de vie et visibilité

Variables locales :

- déclarées en début de bloc (sauf C99),
- créées quand le programme “entre” dans le bloc,
- détruites en fin de bloc,
- masquent les autres variables de même nom.

Exemple (fragment)

```
{ int x;
  { int y;
    int x;
    y=x+1; /* il s'agit du dernier x */
  }
  y=x+1; /* erreur: y inconnu */
  int z; /* erreur: pas en début de bloc */
}
```

Durée de vie et visibilité

Variables globales :

- déclarées en dehors de tout bloc, fonction,
- existent toujours,
- surtout utiles quand on a plusieurs fonctions (cours suivant).

Exemple

```
#include <stdio.h>
int x;
int main()
{
  x=12;
  return x;
}
```

Identificateurs

Identificateur = nom de variable

- suite de lettres, chiffres ou soulignés : `a-z A-Z 0-9 _`
- commence par une lettre ou souligné,
- pas d'accent, d'espace, de ponctuation,
- sensible à la casse.

Exemples

```
int i;
int I;    /* différent de i */
int Mon42emeEntier;
double nombre_flottant2;
```

Noms réservés

Certains noms sont réservés par le langage.
Une variable ne doit pas porter un nom réservé.

Noms réservés

`auto`	`break`	`case`	`char`	`const`	`continue`
`default`	`do`	`double`	`else`	`enum`	`extern`
`float`	`for`	`goto`	`if`	`inline`	`int`
`long`	`register`	`restrict`	`return`	`short`	`signed`
`sizeof`	`static`	`struct`	`switch`	`typedef`	`union`
`unsigned`	`void`	`volatile`	`while`		

Tout nom commençant par _ suivi d'une majuscule ou de _

Par contre, on peut utiliser : `Auto`, `break2`, `_case`.

Types de base

Types courants

`int`	entiers machine
`char`	caractères, également entiers
`double`	nombres flottants double précision

Autres types entiers et flottants :

`unsigned`		entiers positifs
`short`	`unsigned short`	petits entiers
`long`	`unsigned long`	gros entiers
`long long`	`unsigned long long`	très gros entiers
`signed char`	`unsigned char`	très petits entiers
`float`		flottants simple précision

Types de base

Pourquoi tous ces types ?

- compromis entre capacité et occupation mémoire,
- interprétation dépendante de la machine !

Exemples courants

`int`	32-bits : [-2147483648 ;2147483647]
`unsigned`	32-bits : [0 ;4294967295]
`char`	8-bits : [-128 ;127] ou [0 ;255]
`double`	64-bits : magnitude 10^{-308} à 10^{308}, 16 chiffres significatifs

Affectations et expressions

Affectation = modification de la valeur d'une variable

Syntaxe

```
variable = expression ;
```

Expression =

- constantes entières : 2, -45, `'a'` $(=141)$,
- constantes flottantes : `3.45`, `-4.5e-12` $(=4.5 \times 10^{-12})$,
- variables,
- opérateurs,
- parenthèses : `(`, `)`.

Opérateurs arithmétiques

Opérateurs courants

+	addition
-	soustraction (binaire) ou négation (unaire)
*	multiplication
/	division (entière ou flottante)
%	reste de la division (entière)

Priorités

Comme en mathématiques :

- *, /, % prioritaires sur + et −,
- si même priorité, on évalue de gauche à droite,
- en cas de doute, mettre des parenthèses.

Sémantique de l'affectation

Deux étapes :

- évaluation de l'expression en un entier ou flottant, utilise la valeur courante des variables,
- stockage du résultat dans la variable destination.

Exemple

```
x = 2+3*4;    /* ici, x vaut 14 */
y = 2*(x+1); /* ici, y vaut 30 */
x = x - 1;    /* ici, x vaut 13 */
```

Initialisation des variables

Attention !

Le contenu d'une variable est indéfini avant la première affectation.

Exemple

```
int x;
int y;
/* x et y sont aléatoires */
y = 100 / x; /* opération dangereuse */
```

Raccourcis : déclaration et initialisation combinée

```
int x = 12;
int y = 2+3*x;
```

Affichage sur écran

La fonction printf

printf permet d'afficher :

- du texte,
- la valeur d'expressions entières ou flottantes.

Arguments de printf

printf prend un nombre arbitraire d'arguments :

- 1er argument : texte à afficher,
- arguments suivants : expressions à évaluer.

Les arguments sont séparés par une virgule ,

Effet

Dans le texte, chaque %x est remplacé par la valeur d'un argument.

Utilisation de printf

Exemple

```
#include <stdio.h>
int main()
{
  int x = 12;
  printf("x = %i\n",x);
  return 0;
}
```

Résultat :

```
x = 12
```

Utilisation de printf

Exemple

```
#include <stdio.h>
int main()
{
  int x = 12;
  printf("1/3 vaut %f mais 3x vaut %i\n\n", 1.0/3.0, 3*x);
  return 0;
}
```

Résultat :

```
1/3 vaut 0.333333 mais 3x vaut 36
```

Caractères magiques dans printf

Caractères magiques \ et %

`\n`	passe à la ligne suivante
`\"`	affiche "
`\\`	affiche \
`%i`	affiche un entier passé en argument
`%c`	affiche un caractère passé en argument
`%f`	affiche un flottant passé en argument
`%%`	affiche %

Attention :

- il faut autant d'arguments supplémentaires que de %*x*,
- l'argument doit être entier pour `%i` et `%c`, flottant pour `%f`,
- l'option `-Wall` vérifie cela pour vous !

Conditionnelles

La construction if

Syntaxe

if (*expression*) { *instructions* }

Effet :

- l'expression est évaluée,
- le bloc suivant n'est exécuté que si l'expression est vraie.

Raccourcis : si il n'y a qu'une instruction :
if (*expression*) *instruction* ;

Les parenthèses sont par contre obligatoires !

Expressions booléennes

On peut comparer la valeur de deux expressions arithmétiques :

Opérateurs de comparaison

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

Exemple

```
if (x>2*y) { x=2*y ; y=0 ; }
```

La priorité des opérateurs de comparaison est plus faible que celle des opérateurs arithmétiques.

Opérateurs booléens

On peut combiner la valeur de vérité d'expressions booléennes :

Opérateurs booléens

`||`	ou logique	(binaire)
`&&`	et logique	(binaire)
`!`	négation logique	(unaire)

Exemple

```
if ((x>0 && y>0) || (x<0 && y<0)) signe_xy=1 ;
```

Conseil : utilisez des parenthèses pour ne pas vous tromper dans les priorités.

La construction if else

Syntaxe

if (*expression*) { *instructions1* }
else { *instructions2* }

Effet :

- l'expression est évaluée,
- le premier bloc est exécuté si l'expression est vraie,
- le deuxième bloc est exécuté si l'expression est fausse.

Exemple de test

Exemple

```
if (age>99) printf("vous êtes trop vieux!\n");
else {
  if (age<18) printf("interdit aux mineurs!\n");
  else printf("vous avez %i ans\n",age);
}
```

Opérateurs à ne pas confondre

Ne pas confondre

l'opérateur d'affectation = et l'opérateur de comparaison ==

Programme C "correct"

```
1  int main()
2  {
3    int x;
4    x==0;
5    if (x=0) {}
6    return 0;
7  }
```

Conseil : compiler avec `-Wall -Wextra`

```
y.c: In function 'main':
y.c:4: warning: statement with no effect
y.c:5: warning: suggest parentheses around assignment used as tr
```

Boucles `while`

La construction while

Syntaxe

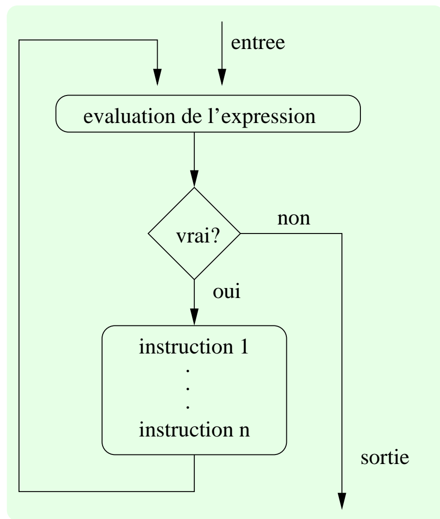
while (*expression*) { *instructions* }

Effet : tant que *expression* est vraie, le bloc est exécuté.

Notes :

- si la condition est initialement fausse, le bloc n'est jamais exécuté,
- la condition est re-testée après chaque "tour" de boucle,
- les { } sont facultatives. mais les ( ) obligatoires.

Déroulement d'une boucle



Exemple de boucle

Exemple

```
#include <stdio.h>
int main()
{
  int pommes = 10;
  while (pommes > 0) {
    printf("j'ai %i pommes dans ma pochette\n",
              pommes);
    pommes = pommes-1;
  }
  printf("je n'ai plus de pommes!\n");
  return 0;
}
```