Éléments d'informatique – Cours 5. Tableaux, while, expressions booléennes.

Pierre Boudes

13 octobre 2010

This work is licensed under the *Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License*.

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction (autant de paramètres supplémentaires que de %). Le caractère suivant le symbole % détaille la conversion à utiliser. La conversion %d met une valeur au format entier décimal.

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction (autant de paramètres supplémentaires que de %). Le caractère suivant le symbole % détaille la conversion à utiliser. La conversion %d met une valeur au format entier décimal.
- Exemples :
  - `printf("Bonjour\n")` affiche `Bonjour` et un saut de ligne

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction (autant de paramètres supplémentaires que de %). Le caractère suivant le symbole % détaille la conversion à utiliser. La conversion %d met une valeur au format entier décimal.
- Exemples :
  - `printf("Bonjour\n")` affiche `Bonjour` et un saut de ligne
  - `printf("i vaut %d\n", i)` affiche `i vaut` suivi de la valeur décimale de i (et d'un saut de ligne)

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction (autant de paramètres supplémentaires que de %). Le caractère suivant le symbole % détaille la conversion à utiliser. La conversion %d met une valeur au format entier décimal.
- Exemples :
  - `printf("Bonjour\n")` affiche `Bonjour` et un saut de ligne
  - `printf("i vaut %d\n", i)` affiche `i vaut` suivi de la valeur décimale de i (et d'un saut de ligne)
  - `printf("(%d, %d)\n", 31, -4)` affiche `(31, -4)` et un saut de ligne. Remarquez qu'il y a deux paramètres en plus

printf/scanf (1)

- Pour afficher un texte à l'écran, nous utilisons la fonction printf (*print formatted*).
- Chaque % dans le texte à afficher est substitué par la valeur formatée d'un paramètre supplémentaire de la fonction (autant de paramètres supplémentaires que de %). Le caractère suivant le symbole % détaille la conversion à utiliser. La conversion %d met une valeur au format entier décimal.
- Exemples :
  - `printf("Bonjour\n")` affiche `Bonjour` et un saut de ligne
  - `printf("i vaut %d\n", i)` affiche `i vaut` suivi de la valeur décimale de i (et d'un saut de ligne)
  - `printf("(%d, %d)\n", 31, -4)` affiche `(31, -4)` et un saut de ligne. Remarquez qu'il y a deux paramètres en plus
- Réciproquement pour faire entrer dans le programme une donnée saisie par l'utilisateur, nous utiliserons `scanf`.
- Exemple : `scanf("%d", &x)`

La mémoire, les octets

- La mémoire vive, ou mémoire de travail est un dispositif électronique dans lequel sont stockées les données en cours de traitement. Les données y sont codées en binaire (comme dans le reste de l'ordinateur), à l'aide de bits (0 ou 1) regroupés en octets (groupes de 8 bits).

La mémoire, les octets

- La mémoire vive, ou mémoire de travail est un dispositif électronique dans lequel sont stockées les données en cours de traitement. Les données y sont codées en binaire (comme dans le reste de l'ordinateur), à l'aide de bits (0 ou 1) regroupés en octets (groupes de 8 bits).
- Du point de vue logiciel la mémoire se présente comme une succession d'octets, numérotés par les entiers à partir de 0. La mémoire est ainsi un grand tableau, dont chaque case (ou cellule) renferme un octets. Les numéros sont les adresses des cases.

La mémoire, les octets

- La mémoire vive, ou mémoire de travail est un dispositif électronique dans lequel sont stockées les données en cours de traitement. Les données y sont codées en binaire (comme dans le reste de l'ordinateur), à l'aide de bits (0 ou 1) regroupés en octets (groupes de 8 bits).
- Du point de vue logiciel la mémoire se présente comme une succession d'octets, numérotés par les entiers à partir de 0. La mémoire est ainsi un grand tableau, dont chaque case (ou cellule) renferme un octets. Les numéros sont les adresses des cases.

Adresses :	0	1	2	...
octets (valeurs) :	01000110	11010111	00000001	...

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :
  - La déclaration `int toto;` aura pour effet de réserver l'espace mémoire nécessaire au stockage d'un entier.

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :
  - La déclaration `int toto;` aura pour effet de réserver l'espace mémoire nécessaire au stockage d'un entier.
  - Dans la suite du programme, l'adresse de cet espace mémoire sera utilisée partout où il est fait référence à cette variable (identificateur `toto`).

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :
  - La déclaration `int toto;` aura pour effet de réserver l'espace mémoire nécessaire au stockage d'un entier.
  - Dans la suite du programme, l'adresse de cet espace mémoire sera utilisée partout où il est fait référence à cette variable (identificateur `toto`).
  - C'est le codage machine des entiers en binaire qui sera employé pour manipuler cette donnée.

Mémoire et variables (rappels)

- Déclarer une variable a pour effet de réserver de la mémoire et de lui donner un usage particulier pour la suite du programme :
  - La déclaration `int toto;` aura pour effet de réserver l'espace mémoire nécessaire au stockage d'un entier.
  - Dans la suite du programme, l'adresse de cet espace mémoire sera utilisée partout où il est fait référence à cette variable (identificateur `toto`).
  - C'est le codage machine des entiers en binaire qui sera employé pour manipuler cette donnée.

Remarque. 

La taille d'un `int` est en principe exactement celle d'un mot mémoire, c'est à dire 4 ou 8 octets. Nous verrons au cours suivant d'autres types de données, leurs tailles et codages. Quoi qu'il en soit, le compilateur prend en charge ces aspects et nous aurons rarement à nous en soucier en programmant.

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Adresses :	...	344				348				352				...
Valeur :	...	1 ... 0				0... 1				1... 1				...

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Adresses :	...	344				348				352				...
Valeur :	...	1 ... 0				0... 1				1... 1				...

- C'est ce qu'on appelle un tableau, statique, unidimensionnel.

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Adresses :	...	344				348				352				...
Valeur :	...	1 ... 0				0... 1				1... 1				...

- C'est ce qu'on appelle un tableau, statique, unidimensionnel.
  - Sa taille doit être connue au moment de la compilation (statique)

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Adresses :	...	344				348				352				...
Valeur :	...	1 ... 0				0... 1				1... 1				...
Identificateur :	...	`toto[0]`				`toto[1]`				`toto[2]`				...

- C'est ce qu'on appelle un tableau, statique, unidimensionnel.
  - Sa taille doit être connue au moment de la compilation (statique)
  - Les cases sont accessibles, comme s'il s'agissait de variables, à l'aide des identificateurs `toto[0]`, `toto[1]`, `toto[2]`

Tableaux et mémoire

- En C, on peut réserver plusieurs espaces mémoires contigus pour des données de même type en une seule déclaration :

```
int toto[3];
```

Adresses :	...	344				348				352				...
Valeur :	...	1 ... 0				0... 1				1... 1				...
Identificateur :	...	toto[0]				toto[1]				toto[2]				...

- C'est ce qu'on appelle un tableau, statique, unidimensionnel.
  - Sa taille doit être connue au moment de la compilation (statique)
  - Les cases sont accessibles, comme s'il s'agissait de variables, à l'aide des identificateurs `toto[0]`, `toto[1]`, `toto[2]`
  - La numérotation commence à zéro. Si n est le nombre de cases du tableau la dernière case est donc numérotée $n - 1$.



Attention !

Il ne faut jamais accèder à une case au delà de la numérotation : `toto[3]`, `toto[-1]`, etc. Le compilateur ne vous préviendra pas de votre erreur, mais le programme va boguer.

Attention !

Il ne faut jamais accèder à une case au delà de la numérotation : `toto[3]`, `toto[-1]`, etc. Le compilateur ne vous préviendra pas de votre erreur, mais le programme va boguer.

L'erreur d'exécution `segmentation fault` signifie que le programme a effectué un accès à une case mémoire qui ne lui était pas réservée (mais il faut beaucoup s'écarter des bons indices du tableau).

Premier exemple

Premier exemple

```c
int main()
{
  /*Declaration et initialisation de variables*/
  int tableau[3];

  tableau[0] = 3;
  tableau[1] = 5;
  tableau[2] = tableau[0] + tableau[1];

  return EXIT_SUCCESS;
}
```

Premier exemple

```
int main()
{
  /*Declaration et initialisation de variables*/
  int tableau[3] = {3,5,8};

  tableau[0] = 3; ← inutile
  tableau[1] = 5; ← inutile
  tableau[2] = tableau[0] + tableau[1]; ← inutile

  return EXIT_SUCCESS;
}
```

Second exemple

```
int main()
{
  /* Declaration et initialisation de variables */
  int tab[3] = {3,5,8};
  int i; /* var. de boucle */
```

Second exemple

```c
int main()
{
  /* Declaration et initialisation de variables */
  int tab[3] = {3,5,8};
  int i; /* var. de boucle */

  for (i = 0; i < 3; i = i + 1) /* pour chaque case */
  {
    printf("tab[%d] = %d\n", i, tab[i]);
  }
  return EXIT_SUCCESS;
}
```

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5
10				2	

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5
10				2	
9					tab[2] = 8

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5
10				2	
9					tab[2] = 8
10				3	

Trace du second exemple

```
1   int main()
2   {
3     /* Declaration et initialisation de variables */
4     int tab[3] = {3,5,8};
5     int i; /* var. de boucle */
6
7     for (i = 0; i < 3; i = i + 1) /* pour chaque case */
8     {
9       printf("tab[%d] = %d\n", i, tab[i]);
10    }
11    return EXIT_SUCCESS;
12  }
```

ligne	tab[0]	tab[1]	tab[2]	i	sortie écran
initialisation	3	5	8	?	
7				0	
9					tab[0] = 3
10				1	
9					tab[1] = 5
10				2	
9					tab[2] = 8
10				3	
11	Renvoie EXIT_SUCCESS				

Pour aller plus loin

- La taille d'un tableau statique gagne à être fixée par une constante symbolique (`#define N 3`).

Pour aller plus loin

- La taille d'un tableau statique gagne à être fixée par une constante symbolique (`#define N 3`).
- L'identificateur du tableau (*ie* `tab` dans la déclaration `int tab[3];`) est lui même une variable. Sa valeur est l'adresse de la première case du tableau `tab[0]`.

Pour aller plus loin

- La taille d'un tableau statique gagne à être fixée par une constante symbolique (`#define N 3`).
- L'identificateur du tableau (*ie* `tab` dans la déclaration `int tab[3];`) est lui même une variable. Sa valeur est l'adresse de la première case du tableau `tab[0]`.
  - Les variables dont la valeur est une adresse (les pointeurs) seront vues en détail au second semestre.
  - La notation *esperluette*, `&x`, donne accès à l'adresse d'une variable ;
  - La notation *étoile*, `*tab`, ne s'applique qu'à une adresse, elle donne alors accès à la valeur contenue à cette adresse.
  - Les expressions `tab[i]` et `*(tab + i)` sont identiques en C.

Démos

L'instruction de contrôle while

Syntaxe :

`while (`*condition*`) { `*bloc*` }`.

L'instruction de contrôle while

Syntaxe :

`while (`*condition*`) { `*bloc*` }`.

Code source

```
/* avant */
while ( ... )
{
  ...
}
/* après */
```

L'instruction de contrôle while

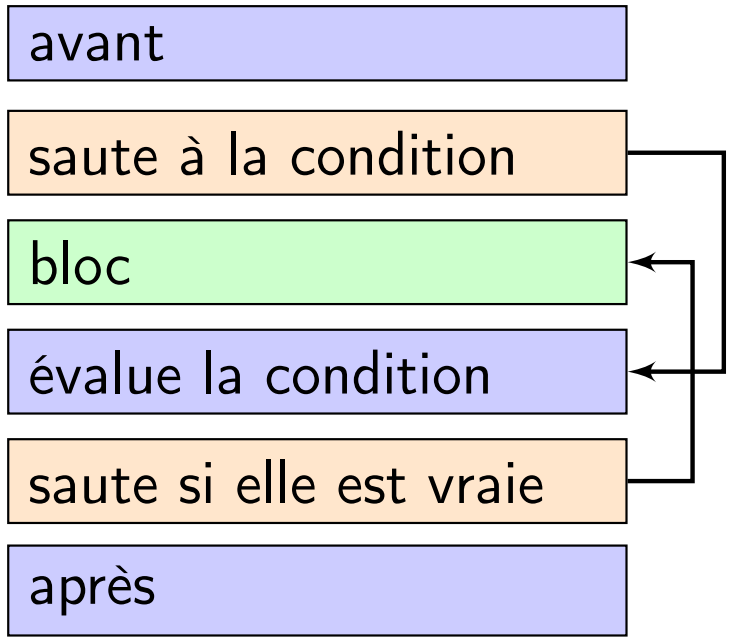
Syntaxe :

`while (`*condition*`) { `*bloc*` }`.

Code source

```
/* avant */
while ( ... )
{
  ...
}
/* après */
```

Schéma de traduction



Trace

```
1  int main()
2  {
3      /* Declaration et initialisation des variables */
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer␣un␣nombre␣positif␣");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce␣nombre␣a␣%d␣chiffres␣decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran

```
1  int main()
2  {
3      /* Declaration et initialisation des variables */
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer un nombre positif ");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	

```
1  int main()
2  {
3      /* Declaration et initialisation des variables */
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer un nombre positif ");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...

```
1  int main()
2  {
3      /* Declaration et initialisation des variables */
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer un nombre positif ");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer␣un␣nombre␣positif␣");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce␣nombre␣a␣%d␣chiffres␣decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		

Trace

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de chiffres */
6
7      printf("Entrer un nombre positif ");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre positif ");
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	

Trace

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que x a plus d'un chiffre */
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	
12	6		

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	
12	6		
13		4	

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que
11     {
12         x = x / 10; /* enlever un chiffre a x */
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	
12	6		
13		4	
15			ce nombre a 4...

Trace

```
1  int main()
2  {
3      /* Declaration et initiali
4      int x;
5      int nb = 1; /* nombre de c
6
7      printf("Entrer un nombre p
8      scanf("%d", &x);
9
10     while (x > 9) /* tant que
11     {
12         x = x / 10; /* enlever
13         nb = nb + 1; /* compter un chiffre de plus */
14     }
15     printf("ce nombre a %d chiffres decimaux\n", nb);
16
17     /* Valeur fonction */
18     return EXIT_SUCCESS;
19 }
```

L'utilisateur saisit 6071.

ligne	x	nb	sortie écran
initialis.	?	1	
7			Entrer...
8	6071		
12	607		
13		2	
12	60		
13		3	
12	6		
13		4	
15			ce nombre a 4...
18	Renvoie EXIT_SUCCESS		

For ou while ?

- Un `for` peut toujours être simulé par un `while` et le code machine sera identique. Il suffit d'introduire un compteur de boucle (la variable de boucle du `for`).

For ou while ?

- Un `for` peut toujours être simulé par un `while` et le code machine sera identique. Il suffit d'introduire un compteur de boucle (la variable de boucle du `for`).
- Par convention, les programmeurs préfèrent utiliser un `for` lorsque le nombre d'itérations est connu à l'avance. Par exemple, pour faire la somme des éléments d'un tableau. Dans le cas contraire, les programmeurs utilisent un `while`. Par exemple, pour chercher un élément dans un tableau.

For ou while ?

- Un `for` peut toujours être simulé par un `while` et le code machine sera identique. Il suffit d'introduire un compteur de boucle (la variable de boucle du `for`).
- Par convention, les programmeurs préfèrent utiliser un `for` lorsque le nombre d'itérations est connu à l'avance. Par exemple, pour faire la somme des éléments d'un tableau. Dans le cas contraire, les programmeurs utilisent un `while`. Par exemple, pour chercher un élément dans un tableau.
- Maintenant que nous avons le `while`, il est possible qu'un programme ne termine jamais (`Ctrl-C`).

Expressions booléennes

Les *conditions* employées dans les structures de contrôle (`if`, `for` ou `while`) sont des expressions booléennes, pouvant être *Vrai*, *Faux* ou :

Expressions booléennes

Les *conditions* employées dans les structures de contrôle (`if`, `for` ou `while`) sont des expressions booléennes, pouvant être *Vrai*, *Faux* ou :

- des inégalités entre expressions arithmétiques

$$\begin{aligned} \textit{inégalité} := & \; e_1 < e_2 \mid e_1 > e_2 \mid e_1 \; != e_2 \\ & \mid e_1 <= e_2 \mid e_1 >= e_2 \mid e_1 == e_2 \end{aligned}$$

Expressions booléennes

Les *conditions* employées dans les structures de contrôle (`if`, `for` ou `while`) sont des expressions booléennes, pouvant être *Vrai*, *Faux* ou :

- des inégalités entre expressions arithmétiques

$$\begin{aligned} \textit{inégalité} := {} & e_1 < e_2 \mid e_1 > e_2 \mid e_1 \texttt{ != } e_2 \\ & \mid e_1 \texttt{ <= } e_2 \mid e_1 \texttt{ >= } e_2 \mid e_1 \texttt{ == } e_2 \end{aligned}$$

- ou des combinaisons logiques d'expressions booléennes :

$$\begin{aligned} \textit{condition} := {} & (\textit{condition}) \texttt{ \&\& } (\textit{condition}) && \text{(et)} \\ & \mid (\textit{condition}) \texttt{ || } (\textit{condition}) && \text{(ou)} \\ & \mid \texttt{!}(\textit{condition}) && \text{(non)} \end{aligned}$$

Constantes booléennes

- Certains langages possédent un type booléen (admettant deux valeurs *true* et *false*) pour les expressions booléennes.

Constantes booléennes

- Certains langages possédent un type booléen (admettant deux valeurs *true* et *false*) pour les expressions booléennes.
- En langage C, les expressions booléennes sont de type entier (`int`), l'entier *zéro* joue le rôle du Faux, l'entier *un* joue le rôle du Vrai et tout entier différent de zéro est évalué a vrai.

Constantes booléennes

- Certains langages possédent un type booléen (admettant deux valeurs *true* et *false*) pour les expressions booléennes.
- En langage C, les expressions booléennes sont de type entier (`int`), l'entier *zéro* joue le rôle du Faux, l'entier *un* joue le rôle du Vrai et tout entier différent de zéro est évalué a vrai.
- On se donne deux constantes symboliques :

```
/* Declaration des constantes et types utilisateur */
#define TRUE 1
#define FALSE 0

int main()
{
    int continuer = TRUE; /* faut-il continuer ?*/

    while (continuer)
    {
        ...
```

Démos