Bases de la programmation impérative
Première partie

Y. Guesnet

Département d'informatique
Université de Rouen

1er septembre 2025

Plan

Plan

Plan

Le cours d'initiation à la programmation impérative

Objectifs

Ce cours a deux objectifs :

- Vous présenter les concepts généraux de la programmation logicielle.
- Vous initier à la programmation impérative au travers du langage C.

Les enseignements

Organisation

- 18 h de cours
- 24 h de TD
- 18 h de TP

Les enseignements

Évaluations

- Contrôle continu intégral.
- Deux contrôles.
- Un TP noté.
- La note finale est obtenue « *grosso modo* » en calculant la moyenne des trois notes.

Les enseignements

Évaluations

- Contrôle continu intégral.
- Deux contrôles.
- Un TP noté.
- La note finale est obtenue « *grosso modo* » en calculant la moyenne des trois notes.
- Pour être précis, la formule exacte est :
  ((CC1 + CC2) / 2) * 0.70 + TP * 0.30

Les enseignements

Évaluations

- Contrôle continu intégral.
- Deux contrôles.
- Un TP noté.
- La note finale est obtenue « *grosso modo* » en calculant la moyenne des trois notes.
- Pour être précis, la formule exacte est :
  ((CC1 + CC2) / 2) * 0.70 + TP * 0.30
- Un écrit de « seconde chance » en fin de semestre. La seconde chance peut remplacer une ou deux notes de contrôle (pas de TP).

Bibliographie

[1] Achille Braquelaire. *Méthodologie de la programmation en C - 4e édition*. Sciences SUP. Dunod, 2005. ISBN : 9782100490189.

[2] C. Delannoy. *Le guide complet du langage C*. Blanche. Eyrolles, 2020. ISBN : 9782212071030.

[3] *ISO/IEC 9899:2024, Information technology — Programming languages — C*. ISO/IEC. 2024. URL : https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf (visité le 25/08/2025).

[4] B.W. Kernighan et D.M. Ritchie. *Le langage C : norme ANSI - 2e édition*. Sciences SUP. Informatique. Dunod, 2004. ISBN : 9782100487349.

Plan

Programmation

La programmation informatique

- La programmation (informatique) désigne l'écriture de programmes informatiques.
- Les programmes sont écrits dans des langages informatiques décrivant l'ensembles des opérations à effectuer sous une forme lisibles par l'homme.
- L'ensemble du texte d'un programme (écrit dans un langage de programmation) forme le code source d'un programme. Le code source est généralement stocké dans des fichiers texte.
- Il existe des langages de plus ou moins haut niveau selon le degré d'abstraction les séparant des instructions reconnues par la machine.

Programmation impérative

Paradigme impératif

- Un paradigme de programmation correspond à la façon dont on va indiquer à la machine le travail à fournir ;
- Le paradigme impératif (ou programmation impérative) consiste à indiquer les instructions à exécuter sous la forme de suites d'opérations ;
- Un autre paradigme de programmation est la programmation déclarative dont la programmation fonctionnelle est un sous-type.

Plan

Le langage C : un langage impératif

- Nous découvrirons la programmation impérative au travers du langage C ;
- Le langage C (ou plus simplement le C) est un langage impératif procédural ;
- Cela signifie que les opérations à effectuer peuvent être regroupées sous la forme d'étapes de calcul contenant plusieurs opérations à effectuer chacunes ;
- Les étapes de calcul sont appelées procédures ou fonctions (ou routines ou sous-routine).

Le langage C : un langage structuré

- Le C est un langage structuré : il utilise des structures de contrôle qui sont des instructions pouvant modifier l'exécution séquentielle des instructions.
- Ces structures de contrôle offrent des constructions plus complexes que celles proposées par les instructions de la machine.
- Les langages structurés améliorent la lisibilité du code, le rendant plus facile à corriger et à maintenir dans le temps.

Le langage C : un langage compilé

- Le langage C est un langage compilé, cela signifie qu'un traitement du code source est nécessaire afin d'obtenir un fichier exécutable qui contiendra les instructions machine à exécuter.
- D'autres langages sont interprétés : le code source est alors interprété par un programme dédié afin d'exécuter les opérations qu'il contient (comme le php ou le javascript).
- Certains langages sont compilés pour produire du pseudo-code qui sera exécuté par une machine virtuelle (comme le java).

Le langage C

Un bref historique

- À l'origine le langage C est lié aux systèmes UNIX. Il fut conçu pour le développement d'UNIX. Il est dérivé du langage B qui a été développé par Ken Thompson et qui est aussi l'un des inventeurs d'UNIX.
  - La première version du C est achevée vers 1973 permettant de réécrire UNIX.
  - La première version officielle date de 1978, elle est décrite dans le livre « The C Programming Language », de Brian Kernighan et Dennis Ritchie (le fameux « Kernighan et Ritchie »).

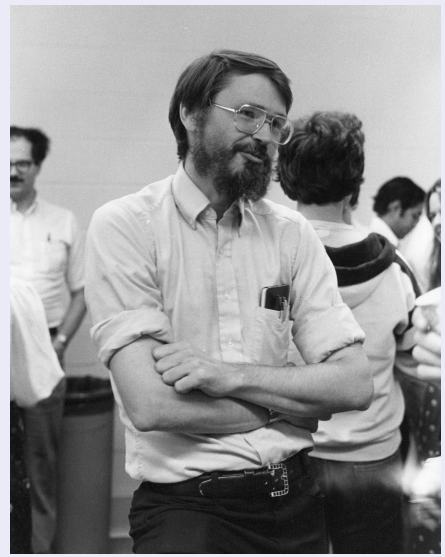
Plusieurs versions du langage C

Les normes du C

On peut distinguer trois déclinaisons du langages C :

- la version d'origine : le C K&R ;

K. Thompson (le langage B) et D. Ritchie



B. Kernighan

- la première normalisation : le C ANSI ;
- la deuxième normalisation : le C ISO.

Plan

Plan

Notre premier programme

hello.c

```c
/**
 * Équipe pédagogique BPI
 * Un premier programme qui dit bonjour (ou plutôt au revoir)
 */

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

int main(void) {
    printf("Goobye cruel world.\n");
    return EXIT_SUCCESS;
}
```

Notre premier programme

Construction d'un programme

Les étapes pour passer d'un code source à un programme exécutable :

- Le prétraitement : le source est tout d'abord analysé par le préprocesseur qui traite toutes les directives présentes dans le source et les remplace par du code C.
- La compilation : le code est ensuite compilé pour produire un code assembleur.
- L'assemblage : l'assembleur est ensuite traduit en langage machine.
- L'édition des liens : les références externes (comme les appels aux fonctions présentes dans des bibliothèques et les réferences aux variables externes) sont remplacées par les instructions adéquates.

Notre premier programme

Exemple

Voici le résultat des différentes étapes appliquées à notre premier programme :

- Prétraitement : hello.i
- La compilation : hello.s
- L'assemblage (attention, ce n'est plus un fichier texte) : hello.o
- L'édition des liens : hello

« The GNU Compiler Collection »

gcc

- Pour construire nos programmes, nous utiliserons *gcc*.
- Il s'agit d'un programme qui permet de construire des programmes écrits dans divers langages dont le C.
- Pour compiler notre premier programme, nous pourrions nous contenter de la commande :

```
$ gcc hello.c
```

« The GNU Compiler Collection »

gcc

- Pour construire nos programmes, nous utiliserons *gcc*.
- Il s'agit d'un programme qui permet de construire des programmes écrits dans divers langages dont le C.
- Pour compiler notre premier programme, nous pourrions nous contenter de la commande :

```
$ gcc hello.c
```

- Quel est alors le nom du programme exécutable généré ?

« The GNU Compiler Collection »

gcc

- Pour construire nos programmes, nous utiliserons *gcc*.
- Il s'agit d'un programme qui permet de construire des programmes écrits dans divers langages dont le C.
- Pour compiler notre premier programme, nous pourrions nous contenter de la commande :

```
$ gcc hello.c
```

- Quel est alors le nom du programme exécutable généré ?
- Pour préciser le nom du programme à contruire, nous pouvons utiliser l'option « `-o` » :

```
$ gcc -o hello hello.c
```

Les options de compilations

Afin de détecter un maximum d'erreurs, nous pouvons transmettre à *gcc* des options modifiant son comportement. Ainsi, nous utiliserons pendant ce semestre les options suivantes :

- *-std=c2x -Wpedantic* pour s'assurer que notre source respecte la norme ISO ;
- *-Wall -Wconversion -Wextra -Wfatal-errors -Wwrite-strings* pour que *gcc* nous affiche un maximum d'avertissements s'il détecte du code suspect ;
- *-Werror* pour que les avertissements soient considérés comme des erreurs ;
- *-O2* pour améliorer la détection des erreurs.

Les options de compilations

Nous pourrons donc construire notre programme avec la commande :

```
gcc                                     \
    -std=c2x -Wpedantic                 \
    -Wall -Wconversion -Wextra          \
    -Wfatal-errors -Wwrite-strings      \
    -Werror                             \
    -O2                                 \
    -o hello hello.c
```

Les options de compilations

Nous pourrons donc construire notre programme avec la commande :

```
gcc                                  \
    -std=c2x -Wpedantic              \
    -Wall -Wconversion -Wextra       \
    -Wfatal-errors -Wwrite-strings   \
    -Werror                          \
    -O2                              \
    -o hello hello.c
```

Heureusement nous utiliserons un environnement de développement intégré (EDI ou IDE pour les anglophones) qui transmettra toutes ces options automatiquement à *gcc*.

Plan

La présentation du code

Les conventions de codage

- Afin d'être facilement lisible par tous, le source d'un programme se doit d'être correctement présenté.
- De même qu'un texte en langage naturel se doit de respecter les règles de typographie de la langue dans laquelle il est écrit, le source d'un programme se doit de respecter les conventions de codage de l'équipe de développement.
- Il est, par exemple, admis par toute la communauté qu'un source se doit d'être correctement indenté : les instructions appartenant au même bloc (délimité en C par une paire d'accolade) doivent débuter sur la même colonne. De plus l'indentation (le nombre d'espace débutant une ligne) doit croitre en fonction du niveau d'imbrication des blocs.

La présentation du code

Les conventions de codage

- Il est encore un peu tôt pour présenter l'ensemble de nos conventions de codage, elles seront introduites au fur et à mesure.
- Mais vous pouvez retrouver un document récapitualtif ici

La fonction principale

`int main(void)`

Un programme C doit contenir une (et une seule) fonction principale, il s'agit du point d'entrée du programme :

```
int main(void) {
  // Code qui sera exécuté au lancement du programme
}
```

La fonction principale

`int main(void)`

Un programme C doit contenir une (et une seule) fonction principale, il s'agit du point d'entrée du programme :

```
int main(void) {
  // Code qui sera exécuté au lancement du programme
}
```

Fichiers sources

- Le code C d'un programme sera placé dans des fichiers d'extension `.c` (plus tard, nous trouverons aussi des fichiers d'extension `.h`)
- Par convention, la fonction principale doit être placée le plus « haut » possible dans le fichier source.

Les commentaires

Les commentaires

Vous avez pu remarquer qu'on peut insérer des commentaires dans le code source. Ceux-ci seront ignorés par le compilateur.
Un commentaire peut être

- soit sur une seule ligne, il commence alors par « // » ;
- soit sur plusieurs lignes, le commentaire doit alors débuter par « /* » et se terminer par « */ ».

Les commentaires

Les commentaires

Vous avez pu remarquer qu'on peut insérer des commentaires dans le code source. Ceux-ci seront ignorés par le compilateur.
Un commentaire peut être

- soit sur une seule ligne, il commence alors par « *//* » ;
- soit sur plusieurs lignes, le commentaire doit alors débuter par « */** » et se terminer par « **/* ».

Exemples

```
// Un commentaire sur une seule ligne.
/*
 * Un commentaire "multi-lignes"
 * (sur plusieurs lignes)
 */
```

Fin d'un programme

La terminaison

- Un programme C se termine lorsqu'il arrive à la fin de la fonction principale (nous verrons plus tard qu'il existe aussi d'autres moyens de terminer un programme)
- Lors de sa terminaison, le programme doit transmettre un code indiquant son état de terminaison. Ce code est soit *EXIT_SUCCESS*, soit *EXIT_FAILURE* :

```
int main(void) {
  // ...

  // Fin du programme, tout s'est bien déroulé
  return EXIT_SUCCESS;
}
```

Les fichiers d'en-tête

Les directives d'inclusion

Notre programme d'exemple contient deux directives pour le pré-processeur :

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

- Il s'agit de directives d'inclusion : le préprocesseur va ajouter au code source le contenu des fichiers *stdlib.h* et *stdio.h*.
- Ces fichiers font partie de la bibliothèque standard : il s'agit d'un ensemble d'outils proposé par défaut par le langage afin d'aider le programmeur.
- Par exemple, dans le fichier *stdlib.h* nous trouvons la définition des macros *EXIT_SUCCESS* et *EXIT_FAILURE*.
- Dans le fichier *stdio.h* nous trouvons la déclaration de la fonction *printf*.

Plan

Plan

Un programme, des données

Les variables

- De manière un peu simpliste, un programme peut être vu comme un composant qui reçoit des données en entrée, réalise des calculs et fournit des données en sortie.
- Afin de pouvoir mémoriser les données, les langages informatiques proposent la notion de variable.
- Une variable possède :
  1. un identificateur,
  2. un type,
  3. un emplacement réservé en mémoire permettant de stocker une valeur.

Les variables

Exemple

```
int n = 1;
```

Définition de variable

La ligne précédente permet de définir une variable d'identificateur `n`, de type entier `int` et dont l'emplacement mémoire contient la représentation de la valeur 1 (nous dirons que `n` a pour valeur 1).

Les variables

Déclaration, définition, initialisation

Pour pouvoir être utilisée, une variable doit être

- déclarée : on indique au compilateur le type et l'identificateur de la variable qu'on veut utiliser ;
- définie : on demande au compilateur de réserver un emplacement mémoire pour stocker la valeur de la variable ;
- initialisée : on stocke une première valeur dans la variable.

Les variables

Déclaration et définition

La déclaration et la définition d'une variable peuvent se faire en une ligne où on indique d'abord le type de la variable suivit de l'identificateur de la variable :

```
int n;
```

Initialisation

L'initialisation d'une variable est réalisée à l'aide de l'opérateur d'affectation :

```
n = 1;
```

Les variables

Remarques

- Si on le peut, il est préférable d'initialiser la variable au moment de sa déclaration :

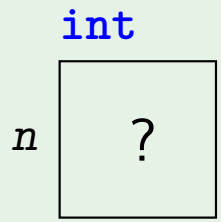
```
int n = 1;
```

- Une variable peut déclarée sans être définie mais ce n'est pas au programme de ce semestre.

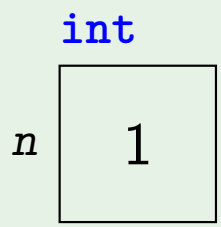
Les variables

Convention pour la représentation des variables

```
int n;
```




```
int n = 1;
```



Plan

Les identificateurs

Formation des identificateurs

Un identificateur est une chaine de caractères qui

- débute par un « _ » ou une lettre (majuscule ou minuscule) ;
- et peut se poursuivre par des « _ », des lettres (majuscules ou minuscules) ou des chiffres.

Exemples

Les chaines suivantes sont des identificateurs :

- `i`
- `count`
- `x`
- `_un_identificateur`

Les identificateurs

Remarque 1

- Le C autorise désormais les identificateurs étendus définis par Unicode, vous pourrez donc trouver dans des sources des identificateurs tels que `clé` ou `識別碼`.
- Dans ce cours, nous nous contenterons des identificateurs « standards » tels que définis dans le transparent précédent.

Remarque 2

Les constantes littérales de type caractère peuvent admettre un préfixe (`u8`, `u`, `U` ou `L`), mais ce ne sera pas abordé dans ce cours.

Les identificateurs

Remarque 3

On ne peut pas utiliser un mot clef comme identificateur. Les mots clefs du C sont :

alignas	*alignof*	**auto**	**bool**
break	**case**	**char**	**const**
constexpr	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
false	**float**	**for**	**goto**
if	*inline*	**int**	**long**
nullptr	**register**	*restrict*	**return**
short	**signed**	**sizeof**	**static**
static_assert	**struct**	**switch**	*thread_local*
true	**typedef**	*typeof*	*typeof_unqual*
union	**unsigned**	**void**	**volatile**
while	*_Atomic*	*_BitInt*	*_Complex*
_Decimal128	*_Decimal32*	*_Decimal64*	*_Generic*
_Imaginary	*_Noreturn*		

Les identificateurs

Remarque 4

Il y aussi des identificateurs réservés ou suceptibles de l'être par la norme (il ne faut donc pas les utiliser). Il s'agit :

- des identificateurs commençant par « __ » ou par « _ » puis une majuscule ;
- des identificateurs présent dans la norme ;

De plus les identifcateurs commençant par « _ » ne doivent être accessible qu'à l'intérieur du fichier où ils sont déclarés (*file scope*).

Remarque 5

Seules les premières lettres des identificateurs sont utilisées par le compilateur (au moins 63 pour les identificateurs internes et 31 pour les identificateurs externes)

Les identificateurs

À noter

Le C est « sensible à la casse », cela signifie qu'il fait la différence entre une minuscule et une majuscule.
Ainsi `foo`, `Foo` et `FOO` seront trois identificateurs différents.

Conventions de codage

- Les identificateurs de variables seront écrits en minuscule.
- Comme il ne peut y avoir d'espace dans un identificateur, nous séparerons les mots par des « _ » :

```
un_identificateur
```

Plan

Les types

Le type d'une variable

Le type d'une variable définit l'ensemble (fini) des valeurs que peut contenir cette variable.

Exemple

En définissant la variable n par la ligne suivante

```
int n;
```

on spécifie que la variable n contiendra une valeur entière comprise entre -32768 et 32767 (au moins).

Les types

Types de bases

Pour ce semestre, nous nous contenterons des types de bases standards suivants :

- `bool`
- `char`
- `signed char`, `short int`, `int`, `long int`, `long long int`
- `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`
- `float`, `double`, `long double`

Les types

Les booleéns

- Les variables booléennes seront de type `bool`.
- Le type `bool` n'autorise que deux valeurs : *`false`* et *`true`*.

Exemple

```
bool b = true;
```

Remarques

- Dans une expression arithmétique, *`true`* vaudra 1 et *`false`* vaudra 0.
- Dans une expression booléenne, 0 vaudra *`false`* et toute valeur différente de 0 vaudra *`true`*.

Les types

Les caractères

- Le type `char` représente le plus petit type possible pour les entiers.
- Les valeurs de ce type sont représentées à l'aide de 8 bits (au moins).
- Il peut contenir le plus petit objet représentable en mémoire (les anglais parleront de *byte*).
- Il peut servir à stocker des entiers mais aussi des caractères. La ligne suivante est donc valide :

```
char c = 'a';
```

- La norme ne spécifie pas si le type `char` est signé ou non. Si besoin, on pourra utiliser `unsigned char` ou `signed char`. Ces types utilisent le même nombre de bits que le type `char`.

Les caractères

Les constantes littérales de type caractère

- Une constante littérale est une valeur constante qu'on spécifie explicitement. Par exemple, 1 est une constante littérale alors que $1 + 1$ ne l'est pas.
- Les constantes littérales de type caractère sont exprimées en entourant le caractère d'apostrophes. Comme dans la ligne suivante :

```
char c = 'a';
```

Les caractères

Les caractères spéciaux

Il existe des caractères dit spéciaux qu'on ne peut désigner à l'aide d'une lettre. Il sont alors exprimés à l'aide d'une barre oblique suivie d'un lettre :

- `'\a'` (*alert*) alerte sonore ou visuelle
- `'\b'` (*backspace*) retour arrière
- `'\f'` (*form feed*) saut de page
- `'\n'` (*new line*) saut de ligne
- `'\r'` (*carriage return*) retour en début de ligne
- `'\t'` (*horizontal tab*) tabulation horizontale
- `'\v'` (*vertical tab*) tabulation verticale

Les caractères

Autres séquences d'échappement

Le caractère « \ » permet aussi d'indiquer d'autres caractères :

`'\''`	pour l'apostrophe
`'\"'`	pour les guillemets
`'\?'`	pour le point d'interrogation
`'\\'`	pour la barre oblique inversée
`'\0nnn'`	pour spécifier que le caractère a pour valeur *nnn* (en octal)
`'\txnn'`	pour spécifier que le caractère a pour valeur *nn* (en hexadécimal)

Remarque

Les caractères « " » et « ? » peuvent aussi bien s'écrire `'"'` et `'?'` que `'\"'` et `'\?'`. Par contre pour indiquer un « ' » ou un « \ » il faut passer par `'\''` et `'\\'`

Les types

Les entiers

Pour représenter des valeurs entières plus grandes que celles autorisées par les types `char`, nous pourrons utiliser les types suivants :

- `short int` et `unsigned short int` : au moins 16 bits ($[-32768, 32767]$ ou $[0, 65535]$)
- `int` et `unsigned int` : au moins 16 bits
- `long int` et `unsigned long int` : au moins 32 bits ($[-2147483648, 2147483647]$ ou $[0, 4294967295]$)
- `long long int` et `unsigned long long int` : au moins 64 bits ($[-9223372036854775808, 9223372036854775807]$ ou $[0, 18446744073709551615]$)

Les entiers

Remarque

- Depuis C23, la norme impose que les entiers soient représentés en notation complément à 2.
- Par contre le boutisme (*endianness* en anglais) n'est pas imposé.

Les entiers

Les constantes littérales de type entier

Il existe de nombreuses façons d'écrire une constante litérale de type entier. Nous en donnons dans ce qui suit quelques exemples.

- On peut écrire un nombre entier en base 10 :

  ```
  1234567890
  ```

- On peut séparer les chiffres par des « ' » (rend la lecture plus facile) :

  ```
  1'234'567'890
  ```

Les entiers

Les constantes littérales de type entier (suite)

- Si l'écriture du nombre débute par un 0, alors le nombre est ecrit en octal (base 8) :

  ```
  011145401322
  ```

- Si elle commence par `0x` ou `0X` alors elle est en hexadécimale (base 16) :

  ```
  0x499602D2
  ```

  Les lettres peuvent être en minuscules ou en majuscules.

- Si elle commence par `0b` ou `0B` alors elle est en binaire (depuis C23) :

  ```
  0b100'1001'1001'0110'0000'0010'1101'0010
  ```

Les entiers

Les constantes littérales de type entier (suite)

On peut également suffixer les écritures des nombres afin d'indiquer le type de la constante littérale :

- `u` pour un entier non signé
- `l` pour un entier long
- `ll` pour un `long long`

Les suffixes peuvent être en minuscules ou majuscules (par contre on ne peut pas mélanger la casse à l'intérieur d'un seul suffixe, par exemple `lL` n'est pas autorisé).

Les entiers

Suffixes et types des enters

Le type de la constante littérale est déterminé en fonction du suffixe et de la valeur de l'entier. Par exemple, sur nos machines :

- `12345u` sera un `unsigned int`
- `1234567890` sera un `int`
- `1234567890u` sera un `unsigned int`
- `12345678900u` sera un `unsigned long int`
- `0x7FFF'FFFF` sera un `int`
- `0xFFFF'FFFF` sera un `unsigned int`
- `2147483647` sera un `int`
- `4294967295` sera un `long int`

Les types

Les flottants

Pour stocker des valeurs décimales nous pourrons utiliser les types suivants :

- `float`
- `double`
- `long double`

Classification des flottants

- L'ensembles des valeurs du type `float` est un sous-ensemble des valeurs du type `double` qui est lui-même un sous-ensemble du type `long double`
- Sauf indication contraire, nous utiliserons le type `double` pour stocker des nombres décimaux.

Les flottants

Remarque

Les flottants ne permettent souvent que de stocker une approximation des nombres décimaux. Par exemple, si on demande d'afficher la valeur `4.1` avec 16 chiffres après la virgule, nous obtiendrons (sur nos machines) :

```
4.0999999999999996
```

Les flottants

Les constantes littérales de type flottant

Les valeurs de type flottant pourront s'écrire sous la forme :

- une partie fractionnaire de la forme :
  - une nombre entier
  - une virgule (à l'anglaise) : « . »
  - un nombre entier
- un exposant (facultatif) sous la forme :
  - un « *e* » ou « *E* »
  - un signe (facultatif) « + » ou « - »
  - un entier
- un suffixe indiquant le type de la constante : « *f* » ou « *F* » pour `float`, « *l* » ou « *L* » pour `long double` (rien pour `double`).

Les flottants

Valeur des constantes

Une valeur de la forme $nepp$ représente le nombre $n \times 10^p$.

Exemples

Les écritures suivantes représentent toutes le nombre 1234 :

- 1234.
- .1234e4
- 12340e-1
- 1234f (de type `float`)
- 12.34e+2l (de type `long double`)

Les flottants

Remarques

1. L'un des entiers de la partie fractionnaire peut être omis.
2. La virgule n'est pas obligatoire mais il faut soit une virgule, soit un exposant pour que le nombre soit vu comme un flottant.
3. Il existe d'autres écritures et d'autres types flottants mais ils ne seront pas abordés dans ce cours.

Plan

Les expressions

Les expressions en C

- Les expressions permettent d'exprimer un calcul. Elles sont contituées
  - d'opérateurs : indiquent les opérations à effectuer ;
  - d'opérandes : indiquent les valeurs, au sens large, sur lesquelles portent les opérations.
- Les expressions possèdent :
  - un type ;
  - une valeur.

Les expressions

Exemple

L'expression

```
1 + 2 * 3
```

a

- `+` et `*` comme opérateurs ;
- `1`, `2` et `3` comme opérandes ;
- `int` comme type ;
- 7 comme valeur.

Les opérateurs

Arité des opérateurs

Nous appelons l'arité d'un opérateur le nombre d'opérandes qu'il admet. En C, il existe :

- des opérateurs unaires, par exemple dans l'expression `-1`, l'opérateur `-` a pour opérande `1` ;
- des opérateurs binaires, par exemple dans l'expression `1 + 2`, l'opérateur `+` a pour opérandes `1` et `2` ;
- un opérateur ternaire `  ?  :  `.

Remarque

Un même lexème peut, selon le contexte, représenter différents opérateurs. Par exemple

- dans l'expression `-1`, `-` désigne l'opérateur moins unaire ;
- dans l'expression `1 - 2`, `-` désigne l'opérateur de soustraction.

Les opérateurs

Les opérateurs arithmétiques

- Les types arithmétiques désignent les types entiers (type `bool` compris) et flottants.
- Les opérateurs arithmétiques sont les opérateurs binaires suivants : `+`, `-`, `*`, `/` et `%`.

Les opérateurs arithmétiques

- L'opérateur `/` désigne la division euclidienne lorsque ses deux opérandes sont entières et la division algébrique sinon.
- L'opérateur `%` désigne le reste de la division euclidienne, il n'admet que des opérandes entières.

Les opérateurs

Le C et la division euclidienne

Il est utile de préciser la définition retenue par la norme de la division euclidienne pour les entiers signés : le quotient est la troncature du quotient algébrique. La définition du reste en découle directement : pour deux entiers a et b de quotient q, le reste r est :

$$r = a - (a/b) * b$$

Exemples

Avec cette définition, nous avons :

- `5 / 2 == 2` et `5 % 2 == 1` ;
- `-5 / 2 == -2` et `-5 % 2 == 1` ;
- `5 / -2 == -2` et `5 % -2 == 1` ;
- `-5 / -2 == 2` et `-5 % -2 == -1`.

Les expressions

Types des expressions

Le type d'une expression dépend du type de ses opérandes. Lorsque les opérandes ont des types différents, elles sont généralement converties en un type commun qui deviendra celui de l'expression, on parle alors de conversions implicites.

Exemples

- L'expression `6.4 / 2` sera de type `double` ;
- L'expression `6ul / 2` sera de type `unsigned long int` ;
- L'expression `'a' / true` sera de type `int` ;

Les expressions

Conversions implicites pour les types arithmétiques

- Promotion entière : toute operande de type entier de précision moindre (en nombre de bits) que le type `int` est convertie en `int` ou en `unsigned int` si le type `int` ne peut recevoir toutes les valeurs du type de l'opérande
- Les règles suivantes sont ensuite appliquées :
  - si l'une des opérandes est de type flottant alors les opérandes seront converties dans le type flottant de plus grande précision rencontré (`long double`, `double` ou `float`, dans cet ordre) ;
  - sinon un type entier pour les conversions est choisi en fonction des types des opérandes.

Les expressions

Remarque 1

- Notez que s'il y a un type signé et un type non signé, le type de l'expression peut être non signé si le type signé ne peut accueillir toutes les valeurs du type non signé. Ainsi l'expression

  ```
  1u / -1
  ```

  aura pour type `unsigned int`.

- Sur nos machines, cela produira un avertissement de *gcc* :

  ```
  la conversion non-signée de « int » vers
  « unsigned int » change la valeur de « -1 » en
  « 4294967295 »
  ```

Les expressions

Remarque 2

- Les calculs sur les entiers non signés ne produiront pas de débordements car ils sont effectués modulo 2^N (pour des types sur N bits) ;
- Par contre les calculs sur des entiers peuvent produire des débordements et alors le comportement dépendra de la machine (le débordement peut-être silencieusement ignoré ou une exception peut être levée).

Les expressions

Ordre d'évaluation

Les expressions ont un ordre d'évaluation. Cet ordre dépend de :

- la priorité des opérateurs, par exemple l'opérateur `*` est prioritaire par rapport à l'opérateur `+` ;
- pour les opérateur de même priorité, l'évaluation se fait selon l'associativité des opérateurs, par exemple les opérateurs arithmétiques ont une associativité de la gauche vers la droite.

Exemple

L'expression

```
1 + 2 + 3 * 4
```

est évaluée comme l'expression

```
(1 + 2) + (3 * 4)
```

Les expressions

Ordre d'évaluation

Comme en Maths, l'ordre dévaluation peut être modifié à l'aide de parenthèses :

```
1 + (2 + 3)
```

Les expressions

Opérateurs d'incrémentation/décrémentation

- Il existe des opérateurs spécifiques pour réaliser des incrémentation et des décrémentations : `++` et `--`
- Ces opérateurs existent en deux versions : préfixe et postfixe.

Exemple

Si `i` vaut 1 avant l'évaluation de l'expression :

- `n = ++i;` `i` et `n` vaudront 2 ;
- `n = i++;` `i` vaudra 2 et `n` vaudra 1 ;

Les expressions

Opérateurs d'affectation

Nous avons déjà rencontré l'opérateur d'affectation simple = mais il en existe également des opérateurs d'affectation combinée :

- `+=`
- `-=`
- `*=`
- `/=`
- `%=`
- `<<=`
- `>>=`
- `&=`
- `^=`
- `|=`

L'affectation

Exemple

```
n += 2;
```

équivaut à

```
n = n + 2;
```

ainsi, si `n` vaut 1 avant l'évaluation de l'expression, il vaudra 3 après.

L'affectation

À noter

1. = est un opérateur, `n = 1` est donc une expression et, à ce titre, à un type et une valeur : `int` et 1.
2. = est un opérateur à effet secondaire : lors de son évaluation, en plus de retourner une valeur il modifie la valeur de l'opérande de gauche.
3. Par convention, il est interdit de combiner plusieurs opérateurs à effet secondaire dans la même expression. Ainsi, même si elle sont parfaitement valides, les expressions suivantes ne doivent pas être utilisées :

```
k = j = 1;
n = ++k;
```

4. Lorsque les types des opérandes sont différents, il y a une conversion implicite du type de l'expression de droite.

L'affectation

Définition

L'opérande de gauche de l'opérateur d'affectation doit être une *l-value* (pour *locator value* ou *left value*) : il s'agit d'une expression qui fait référence à une zone mémoire.

Exemple

- `n = 1 + 2` est une expression valide si `n` est l'identificateur d'une variable de type entier.
- `1 = n + 2` n'est pas une expression valide.
- `n = n + 2` est une expression valide, le `n` de gauche sera traité comme une *l-value* alors que celui de droite sera traité comme une *r-value* : il sera remplacé par la valeur qu'il mémorise.

Les expressions

Opérateurs de comparaisons

Le C propose quatre opérateurs relationnels et deux opérateurs d'égalité :

- `<`, `<=`, `>` et `>=`,
- `==`, `!=` (différent)

Exemple

- `1 >= 2` vaut 0 (*`false`*)
- `1 != 2` vaut 1 (*`true`*)

Les expressions

Opérateurs arithmétiques unaires

Il existe quatre opérateurs arithmétiques unaires :

- `+` : `+e` vaut `e` (le type dépend de la promotion entière)
- `-` : `-e` vaut l'opposé de `e`
- `~` : `~n` où `n` est de type entier, vaut le complément bit à bit de `n`. Par exemple si le type `int` est sur 16 bits, `~0b11'0011u` vaut `0b1111'1111'1100'1100`
- `!` : négation logique, `!e` est équivalent à `0 == e`

Les expressions

Opérateurs logiques

Le C propose la conjonction et la disjonction logique :

- `&&` : le « et » logique, par exemple `true && false` vaut 0 (`false`) et `true && true` vaut 1 (`true`)
- `||` : le « ou » logique, par exemple `true || false` vaut 1 (`true`) et `false || false` vaut 0 (`false`)

Les expressions

Remarque 1

- L'évaluation des opérandes se fait de la gauche vers la droite et elle est paresseuse, ainsi

  ```
  0 && ++n
  ```

  vaudra 0 et n sera inchangé.
- Notez que pour des raisons de lisibilité, une telle expression ne doit pas être utilisée.

Les expressions

Remarque 2

- La conjonction est prioritaire par rapport à la disjonction, ainsi

  ```
  e1 || e2 && e3
  ```

  est équivalent à `e1 || (e2 && e3)`.
- Même si elle est valide, l'expression sans parenthèse produira un avertissement de *gcc* :

  ```
  parenthèses suggérées autour de « && » à l'intérieur de « || »
  ```

  Il faut donc lui préferer sa forme parenthésée.

Les expressions

Les opérateurs

Il existe bien d'autres opérateurs qui vous seront présentés (ou pas) plus tard dans ce cours ou au second semestre.
L'ensemble des opérateurs du C sont présentés dans le transparent suivant selon l'ordre de priorité décroissant (un niveau par ligne, sauf pour les affectations qui tiennent sur deux lignes) et leur associativité ($\rightarrow$ pour gauche et $\leftarrow$ pour droite).

Les opérateurs du C

Plan

Plan

Les entrées/sorties formatées

Entrées/sorties

Par défaut, à son lancement, un programme dispose :

- d'une entrée standard qui permet au programme de lire des données à traiter ;
- d'une sortie standard qui permet au programme d'afficher des informations (comme des résultats ou des questions) ;
- d'une sortie des erreurs permettant au programme d'indiquer s'il a rencontré une erreur.

Les entrées/sorties formatées

Le terminal

Par défaut, les entrées/sorties standards correspondent au terminal :

- tout ce qui écrit sur la sortie standard ou la sortie des erreurs apparaitra dans le terminal (la fenêtre correspondant au terminal sur l'écran) ;
- tout ce qui est écrit dans le terminal (*via* le clavier) peut être lu sur l'entrée standard.

Les entrées/sorties formatées

stdio.h

Les fonctions d'entrées/sorties formatées sont déclarées dans le fichier d'en-tête *stdio.h*. Pour les utiliser, nous devrons donc insérer en début de programme la ligne :

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

Plan

Les sorties formatées

La sortie standard

Pour écrire du texte sur la sortie standard nous utiliserons la fonction *printf*. Dans le premier exemple de ce cours, nous avions :

```
printf("Goobye cruel world.\n");
```

qui donne le résultat suivant dans le terminal :

```
$ ./hello
Goobye cruel world.
$
```

Les sorties formatées

La fonction *printf*

Le premier argument de *printf* est le format : il indique ce qu'il faut écrire. Dans l'exemple précédent il s'agit d'une chaine simple mais le format peut aussi contenir des spécifications de conversion :

```
printf("Le résultat est : %d.\n", n);
```

Si *n* est une variable de type `int` qui vaut 1 alors nous aurons en sortie :

```
Le résultat est : 1.
```

Les sorties formatées

La fonction *printf*

Une spécification de conversion commence par « % » et se termine par un caractère spécificateur de conversion. Pour chaque conversion, la fonction *printf* attend un argument supplémentaire correspondant à la valeur à convertir. Par exemple :

- `c` écrit le caractère correspondant à la valeur entière fournit (un `int` converti en un `unsigned char`) ;
- `d` écrit un entier ;
- `u`, pour un argument entier non signé, écrit un entier ;
- `f`, écrit le flottant transmis en paramètre sous la forme *[-]dddd.dddd* ;
- `e`, écrit le flottant transmis en paramètre sous la forme *[-]d.ddde±dd* ;
- `s`, écrit une chaine de caractères.

La fonction *printf*

Exemples

```
printf("Bonjour %s %c\n", "tout le monde", '!');
printf("%d + %u = %f ?\n", 1, 1u, 1.0);
```

produira :

```
Bonjour tout le monde !
1 + 1 = 1.000000 ?
```

La fonction *printf*

Modificateur de longueur

Par défaut, les indicateurs de conversion attendent un `int` pour *d*, un `unsigned int` pour *u*, un `float` ou un `double` pour *f* ou *g*. On peut cependant modifier le type attendu en ajoutant avant le spécificateur de conversion modificateur de longueur, par exemple :

- *l* pour `long int` ou `unsigned long int` ;
- *ll* pour `long long int` ou `unsigned long long int`
- *L* pour `long double`.

La fonction *printf*

Exemples

```
long int n = 1l;
long long int k = 1llu;
long double x = 1.0L;

printf("%ld + %llu = %Lf ?\n", n, k, x);
```

produira :

```
1 + 1 = 1.000000 ?
```

La fonction *printf*

Largeur de champ et précision

On peut également ajouter une largeur de champ et/ou une précision pour modifier l'affichage avant le modificateur de longeur :

- la largeur de champ est un naturel qui indique le nombre minimum de caractères qui seront écrits ;
- la précision est un « . » suivit d'un naturel modifiant l'affichage : elle indique le nombre de chiffres après la virgule pour les flottants ou le nombre de chiffres minimal devant être affichés pour les entiers.

La fonction *printf*

Exemples

```
printf("%5.3ld\n", 1l);
printf("%.3f\n", 1.);
printf("%5c\n", '*');
```

produira :

```
  001
1.000
    *
```

La fonction *printf*

Les drapeaux

On peut enfin ajouter juste après le « % » ajouter des drapeaux qui modifieront aussi l'affichage :

- `-` pour justifier l'affichage à gauche ;
- `+` pour forcer l'affichage du signe (« + ») pour les nombres positifs des types signés ;
- `0`, pour les nombres, ajoute des `0` pour compléter la largeur de champ.

La fonction *printf*

Exemples

```
printf("%-+5ld%05.1f\n", 1l, 1.);
```

produira :

```
+1    001.0
```

La fonction *printf*

Le format

- Pour afficher un « % », nous écrirons « %% » dans le format ;
- la largeur de champ ainsi que la précision peuvent être « déportés » dans les paramètres si on les remplace par un « * » dans le format.

La fonction *printf*

Exemples

- `printf("100%%\n");`

  produira :

  ```
  100%
  ```

  `printf("%*.*d\n", 5, 2, 1);`

  produira :

  ```
     01
  ```

La fonction *printf*

Remarque

Il existe bien d'autres possibilités pour le format. En voici quelques exemples :

```
printf("%lg\n", 1.2);                  // 1.2
printf("%lg\n", 10'000'000'000.);      // 1e+10
printf("%b\n", 15);                    // 1111
printf("%X\n", 15);                    // F
printf("%#x\n", 15);                   // 0xf
```

Plan

Les entrées formatées

L'entrée standard

Pour lire des données sur l'entrée standard nous utiliserons la fonction *scanf*. Par exemple :

```
printf("Entrez un entier :\n");
int n;
scanf("%d", &n);
printf("Vous avez entré : %d\n", n);
```

produira le résultat suivant :

```
Entrez un entier :
5
Vous avez entré : 5
```

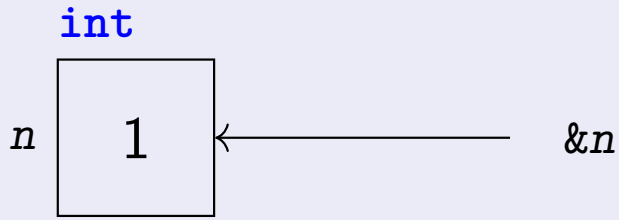
Les entrées formatées

L'opérateur de référencement

Un nouvel opérateur apparait dans l'expression :

```
scanf("%d", &n);
```

Il s'agit de l'opérateur « `&` », opérateur de référencement (ou d'adresse). Il s'agit d'un opérateur unaire dont l'opérande doit être une l-value. Il retourne l'adresse de son opérande :



Ainsi, en transmettant l'adresse de la variable à `scanf`, cela lui permet de connaitre l'emplacement mémoire où il doit stocker la valeur lue.

Les entrées formatées

La fonction *scanf*

Tout comme *printf*, *scanf* attend une chaine de format comme premier paramètre. Elle lui indique ce qu'il doit lire. Le format peut contenir des spécifications de conversion. Les spécifications de conversion diffèrent un peu de celles de *printf*, elles contiennent (dans cet ordre) :

- un caractère « % » ;
- un caractère « * » optionnel qui supprime l'affectation dans l'un des paramètres ;
- un entier strictement positif indiquant le nombre maximal de caractères à lire ;
- un modificateur de longueur optionnel ;
- un spécificateur de conversion.

Les entrées formatées

Le format

Les modificateurs de longueur et les spécificateurs de conversion peuvent être les mêmes que ceux vus pour *printf*, cependant :

- un `double` doit forcément être lu avec `%lf` ;
- `%f` et `%e` ont le même effet : ils lisent un flottant quelque soit son écriture ;
- `%s` lit une suite de caractères « non blancs », elle s'arrête donc avant le premier caractère d'espacement rencontré (espace, tabulations horizontales et verticales, nouvelle ligne, retour à la ligne et saut de page).

Les entrées formatées

Exemples

- Pour lire un entier, nous écrirons :

```
int n;
scanf("%d", &n);
```

- Pour lire un caractère, nous écrirons :

```
char c;
scanf("%c", &c);
```

- Pour lire un flottant, nous écrirons :

```
double x;
scanf("%lf", &x);
```

Les entrées formatées

Exemples

Le code

```
int n;
char c;
double x;
scanf("%2d%c%lf", &n, &c, &x);
```

avec l'entrée `123456` affectera 12 à `n`, `'3'` à c et 456 à `x`.

Les entrées formatées

Format avec des caractères « ordinaires » ou d'espacement

En dehors des spécifications de conversion, un format peut contenir :

- des caractères « ordinaires », dans ce cas *scanf* s'attend à les trouver tels quels sur l'entrée standard ;
- la suite « %% », dans ce cas *scanf* s'attend à trouver un « % » ;
- des caractères « blancs » (caractères d'espacement), *scanf* va alors lire l'entrée jusqu'à rencontrer un caractère non blanc (qui restera non lu).

Les entrées formatées

Remarque

Lors de l'exécution dans un terminal, le code suivant :

```
printf("Entrez un pourcentage :\n");
double x;
scanf("%lf%%\n", &x);
printf("Vous avez entré : %.0lf%%\n", x);
```

va sembler « bloquer » avec l'entrée suivante pourquoi ?

```
$ Entrez un pourcentage :
12%
```

Les entrées formatées

Remarque

Il existe bien d'autres spécifications de conversion autorisées. Par exemple, nous pouvons trouver dans la norme quelque chose comme :

```
scanf("%*[^\n]");
```

Pouvez-vous deviner quel est son effet ?

Les entrées formatées

Valeur de retour

scanf est une fonction : elle retourne le nombre d'éléments correctement lus et affectés (de type `int`). Elle peut retourner 0 si aucun élément n'a été affecté, voire *EOF* en cas d'échec avant la première conversion.

Les entrées formatées

Exemple

Dans le code

```
int n;
double x;
scanf("%5lf%d", &x, &n);
```

avec *gcc*, la fonction *scanf* retournera :

- 2 pour l'entrée `12e345` ;
- 1 pour l'entrée `12e34a` ;
- 1 pour l'entrée `12ea34` (la norme indique qu'elle devrait retourner 0) ;
- 0 pour l'entrée `a12e34`
- *EOF* si la fin de l'entrée est atteinte (`[^D]` sous Linux) ;

Les entrées formatées

Valeur de retour de *scanf*

Selon les versions de *gcc* et avec les options de compilations présentées en début de ce cours, il se peut que la compilation des exemples précédents échoue avec le message d'erreur suivant :

```
erreur: la valeur retournée par « scanf » est ignorée
alors qu'elle est déclarée avec l'attribut
« warn_unused_result »
```

Les entrées formatées

Valeur de retour de *scanf*

La valeur de retour de *scanf* doit toujours être traitée : il faut s'assurer que la lecture a permis d'affecter des valeurs aux variables dont on a transmis l'adresse.

Exemple

Considérons le code suivant :

```
printf("Entrez un pourcentage :\n");
double x;
scanf("%lf%%", &x);
printf("Vous avez entré : %.0lf%%\n", x);
```

Que penser de l'exécution suivante ?

```
Entrez un pourcentage :
coucou
Vous avez entré : 5%
```

Les entrées formatées

Valeur de retour de *scanf*

Pour ce début de cours, nous pourrons contrôler que nous avons bien réussi à lire ce qui était demandé à l'aide de la fonction *assert* :

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

int main(void) {
  printf("Entrez un pourcentage :\n");
  double x;
  assert(scanf("%lf%%", &x) == 1);
  printf("Vous avez entré : %.0lf%%\n", x);

  return EXIT_SUCCESS;
}
```

Les entrées formatées

scanf et options de compilation

Avec les versions « récentes » de *gcc* (et sous Linux), nous pourrons utiliser les options de compilations suivantes afin de recevoir un avertissement pour toute valeur de retour de *scanf* non utilisée :

```
-O2 -D_FORTIFY_SOURCE=2
```

Valeur de retour

Remarque

Question : *printf* est aussi une fonction, que retourne-t-elle ?

Redirections

Entrées et sorties standard

Lorsqu'on exécute un programme dans un terminal, on peut modifier l'entrée et la sortie standard de façon à ce qu'elles soient redirigées vers des fichiers. Pour cela on utilise les opérateurs « < » pour rediriger l'entrée et « > » pour rediriger la sortie. Ainsi avec la commande suivante :

```
$ ./hello > out.txt < in.txt
```

Le programme va lire dans le fichier `in.txt` et écrire dans le fichier `out.txt`.