Introduction au Langage C

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://claroline connect.univ-lyon1.fr` espace d'activités "M1102 M1103 C - Introduction au langage C"

46H prévues $\approx$ 42H de cours+TPs, 2H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Support de cours et livre de références pour le C

- Support de cours écrit par M. Samba Ndojh NDIAYE (pdf sur `http://claroline connect.univ-lyon1.fr`).
- Livre "Programmer en langage C - Cours et exercices corrigés" de Claude Delannoy.

Objectifs généraux du semestre

- **Sensibiliser les étudiants à l'écriture de programmes en langage C en *mode console*)**
- **Développer l'aptitude à transcrire un algorithme du cours d'algorithmique en langage C**
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont des langages assez populaires !

Objectifs généraux du semestre

- **Sensibiliser les étudiants à l'écriture de programmes en langage C en *mode console*)**
- **Développer l'aptitude à transcrire un algorithme du cours d'algorithmique en langage C**
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont des langages assez populaires !

Objectifs généraux du semestre

- **Sensibiliser les étudiants à l'écriture de programmes en langage C en *mode console*)**
- **Développer l'aptitude à transcrire un algorithme du cours d'algorithmique en langage C**
- **Développer des compétences attractives pour les entreprises** $\Longrightarrow$ le C et le C++ sont des langages assez populaires !

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Objectifs spécifiques du semestre

A la fin du semestre, vous devriez être en mesure au sein d'un programme en C :

- de choisir les types C adaptés (entiers, flottants etc.) ;
- de manipuler des variables et des constantes C ;
- d'utiliser les opérateurs (+,-,*,/ etc.), les expressions, les expressions-instructions et les instructions du langage C ;
- d'employer les entrées-sorties clavier du C ;
- d'utiliser des fonctions C (et vos propres fonctions !) ;
- de gérer dynamiquement la mémoire des variables C.

Plan

Plan

Le C est un langage de programmation

- créé en 1972 par Dennis Ritchie pour écrire l'OS UNIX ("père" de Linux).
- au succès international ;
- normalisé, d'abord par l'ANSI (*American National Standard Institute*), puis par l'ISO (*International Standards Organization*).

Programmation et Algorithmique

La programmation informatique

L'ensemble des activités qui permettent d'écrire des *programmes informatiques*.

$$\neq$$

L'algorithmique

L'ensemble des activités logiques qui relèvent des *algorithmes*.

Programme informatique et Algorithme

Programme informatique

Une transcription/traduction d'un algorithme dans un **langage de programmation**.

$$\neq$$

Algorithme

Une séquence d'actions (plus ou moins élémentaires) qui permet d'aller d'un état initial à un état final afin de résoudre *systématiquement* un problème donné.

Exemple de langages ?

Exemple de langages ?

ADN, anglais, langage machine (00111011000...), langage des signes etc.

Exemple de langages ?

ADN, anglais, langage machine (00111011000...), langage des signes etc.

Un langage est caractérisé par un **vocabulaire** et une **syntaxe**.

Exemple d'algorithme ?

Exemple d'algorithme ?

Conduire une voiture, marcher d'un point A à un point B, tracer une courbe 2D (*polyline*), rechercher une information dans un "conteneur" (page Web, fichier texte, feuille de calculs), trier/organiser des informations etc.

Exemple d'algorithme ?

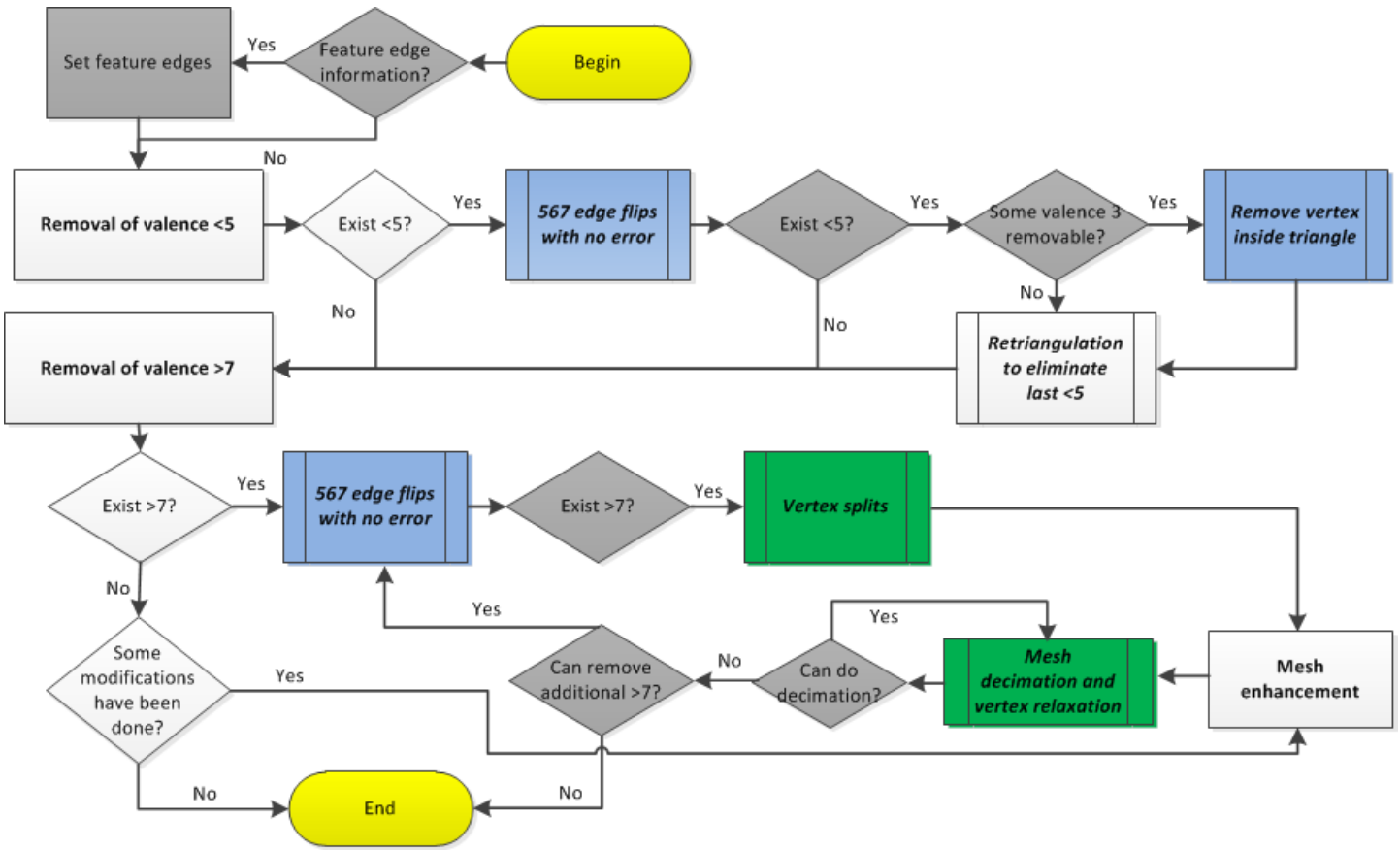
Conduire une voiture, marcher d'un point A à un point B, tracer une courbe 2D (*polyline*), rechercher une information dans un "conteneur" (page Web, fichier texte, feuille de calculs), trier/organiser des informations etc.

Un algorithme est caractérisé par une suite ordonnée de tests et de traitements (actions logiques, procédures) ayant un but précis.

Exemple de programme C : afficher *Hello world !*

```c
/* Affiche Hello World */
#include <stdio.h>
int main(void) {
    printf("Hello World!");
    return 0;
}
```

Exemple d'algorithme : modifier la géométrie d'un objet 3D



Programme et Algorithme : chronologie

1. Un problème (informatique) à résoudre (le "quoi") ;
2. On **écrit un algorithme** pour le résoudre (le "comment") ;
3. On **transcrit cet algorithme** dans un langage de programmation : c'est la base de la programmation informatique.

Différents algorithmes existent pour résoudre un problème.

Des milliers de langages de programmation : Pascal, Fortran, Ada, Java, C, C++,..., Go, rust, ... Chaque langage de programmation a sa propre syntaxe stricte sans ambiguïté.

Programme et Algorithme : chronologie

1. Un problème (informatique) à résoudre (le "quoi") ;
2. On **écrit un algorithme** pour le résoudre (le "comment") ;
3. On **transcrit cet algorithme** dans un langage de programmation : c'est la base de la programmation informatique.

Différents algorithmes existent pour résoudre un problème.

Des milliers de langages de programmation : Pascal, Fortran, Ada, Java, C, C++,..., Go, rust, ... Chaque langage de programmation a sa propre syntaxe stricte sans ambiguïté.

Programme et Algorithme : chronologie

1. Un problème (informatique) à résoudre (le "quoi") ;
2. On **écrit un algorithme** pour le résoudre (le "comment") ;
3. On **transcrit cet algorithme** dans un langage de programmation : c'est la base de la programmation informatique.

Différents algorithmes existent pour résoudre un problème.

Des milliers de langages de programmation : Pascal, Fortran, Ada, Java, C, C++,..., Go, rust, ... Chaque langage de programmation a sa propre syntaxe stricte sans ambiguïté.

Programme et Algorithme : chronologie

1. Un problème (informatique) à résoudre (le "quoi") ;
2. On **écrit un algorithme** pour le résoudre (le "comment") ;
3. On **transcrit cet algorithme** dans un langage de programmation : c'est la base de la programmation informatique.

Différents algorithmes existent pour résoudre un problème.

Des milliers de langages de programmation : Pascal, Fortran, Ada, Java, C, C++,..., Go, rust, ... Chaque langage de programmation a sa propre syntaxe stricte sans ambiguïté.

Programme et Algorithme : chronologie

1. Un problème (informatique) à résoudre (le "quoi") ;
2. On **écrit un algorithme** pour le résoudre (le "comment") ;
3. On **transcrit cet algorithme** dans un langage de programmation : c'est la base de la programmation informatique.

Différents algorithmes existent pour résoudre un problème.

Des milliers de langages de programmation : Pascal, Fortran, Ada, Java, C, C++,..., Go, rust, ... Chaque langage de programmation a sa propre syntaxe stricte sans ambiguïté.

Le (micro)processeur ne comprend que des 0 et des 1

Les suites de 0 et de 1 représentent les instructions élémentaires (e.g. fonctions arithmétiques et logiques) exécutables par le microprocesseur de la machine. Ces suites particulières de 0 et de 1 représentent le *langage machine*.

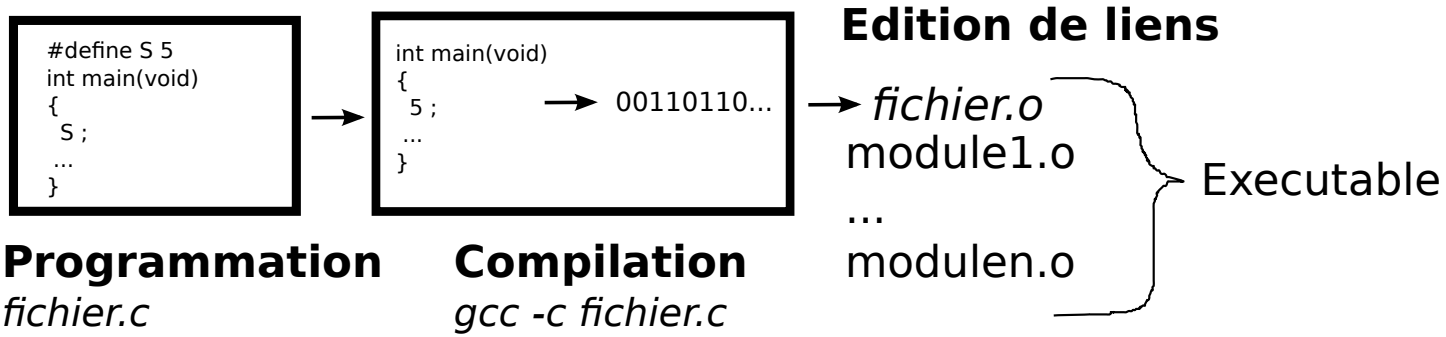
$\Longrightarrow$ Nécessité de traduire un programme informatique, lisible par un humain, en langage machine bas-niveau : c'est l'**étape de compilation** pour les langages compilés tels que le C.

Le (micro)processeur ne comprend que des 0 et des 1

Les suites de 0 et de 1 représentent les instructions élémentaires (e.g. fonctions arithmétiques et logiques) exécutables par le microprocesseur de la machine. Ces suites particulières de 0 et de 1 représentent le *langage machine*.

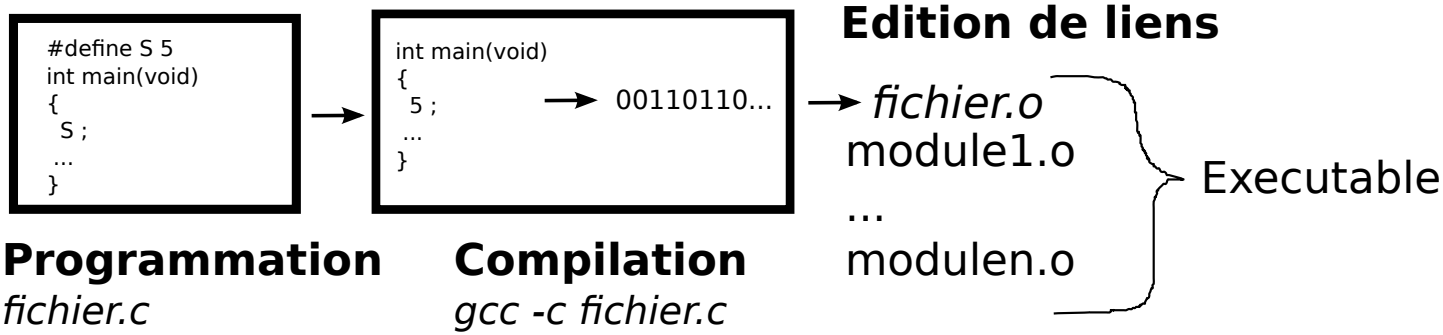
$\Longrightarrow$ Nécessité de traduire un programme informatique, lisible par un humain, en langage machine bas-niveau : c'est l'**étape de compilation** pour les langages compilés tels que le C.

Création d'un fichier exécutable à partir d'un programme en langage C (1/2)



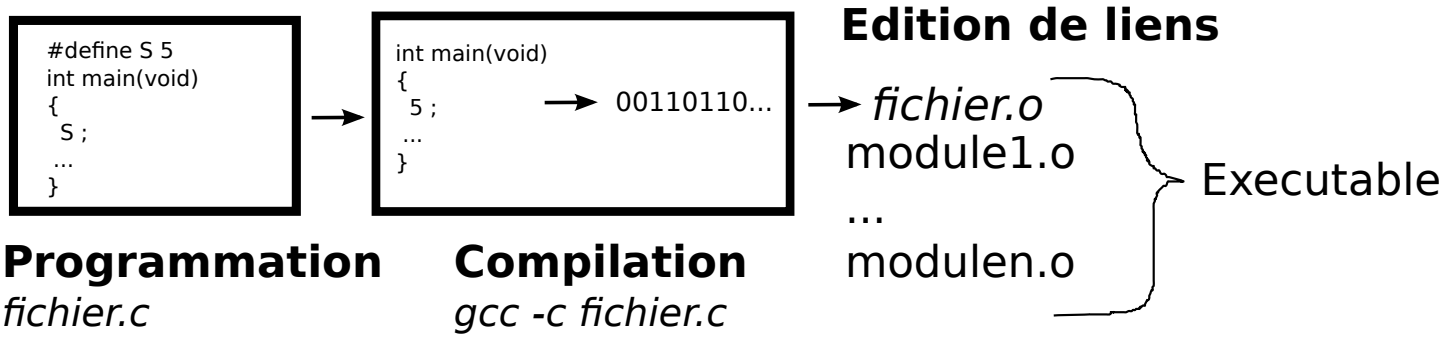
1. La **saisie du programme** dans un fichier source (.c)
2. La **compilation** :
   1. transformations textuelles réalisées par le **préprocesseur**.
   2. la compilation proprement dite, la traduction du texte généré par le préprocesseur en instructions machine (0 et 1).
3. L'**édition de liens**, pour inclure les modules objets (les .o) correspondant aux fonctions prédéfinies utilisées par le programme.

Création d'un fichier exécutable à partir d'un programme en langage C (1/2)



1. La **saisie du programme** dans un fichier source (.c)
2. La **compilation** :
   1. transformations textuelles réalisées par le **préprocesseur**.
   2. la compilation proprement dite, la traduction du texte généré par le préprocesseur en instructions machine (0 et 1).
3. L'**édition de liens**, pour inclure les modules objets (les .o) correspondant aux fonctions prédéfinies utilisées par le programme.

Création d'un fichier exécutable à partir d'un programme en langage C (1/2)



1. La **saisie du programme** dans un fichier source (.c)
2. La **compilation** :
   1. transformations textuelles réalisées par le **préprocesseur**.
   2. la compilation proprement dite, la traduction du texte généré par le préprocesseur en instructions machine (0 et 1).
3. L'**édition de liens**, pour inclure les modules objets (les .o) correspondant aux fonctions prédéfinies utilisées par le programme.

Création d'un fichier exécutable à partir d'un programme en langage C (2/2)

Les étapes de compilation et d'édition de liens sont réalisées à l'aide d'un **compilateur**.

Pour les TPs, nous utiliserons un EDI en C/C++ basé sur la version Mingw du compilateur GCC (soit Dev-C++, soit Code : :Blocks, soit NetBeans). GNU GCC : `https://gcc.gnu.org/`

Remarque : il existe des sites Web pour compiler et exécuter du code C en ligne :
`http://www.tutorialspoint.com/compile_c_online.php`
`http://www.pythontutor.com/c.html`

Création d'un fichier exécutable à partir d'un programme en langage C (2/2)

Les étapes de compilation et d'édition de liens sont réalisées à l'aide d'un **compilateur**.

Pour les TPs, nous utiliserons un EDI en C/C++ basé sur la version Mingw du compilateur GCC (soit Dev-C++, soit Code : :Blocks, soit NetBeans). GNU GCC : `https://gcc.gnu.org/`

Remarque : il existe des sites Web pour compiler et exécuter du code C en ligne :
`http://www.tutorialspoint.com/compile_c_online.php`
`http://www.pythontutor.com/c.html`

Types d'erreur de programmation

- Les erreurs *syntaxiques* et plus généralement les erreurs de compilation : ne compile pas.
- Les erreurs *sémantiques* : les erreurs d'exécution (interruption brutale du programme) et les erreurs logiques (résultat erroné).

Types d'erreur de programmation

- Les erreurs *syntaxiques* et plus généralement les erreurs de compilation : ne compile pas.
- Les erreurs *sémantiques* : les erreurs d'exécution (interruption brutale du programme) et les erreurs logiques (résultat erroné).

Respecter la norme ANSI

Il faut rajouter les options de compilation suivantes :
-ansi -pedantic-errors

```
gcc monFichier.c -ansi -pedantic-errors
```

Tous vos programmes C (TPs) devront respecter la norme ANSI (parfois on utilisera la norme ISO C99, mais pas la norme ISO C11).

Respecter la norme ANSI

Il faut rajouter les options de compilation suivantes :
-ansi -pedantic-errors

gcc monFichier.c -ansi -pedantic-errors

Tous vos programmes C (TPs) devront respecter la norme ANSI (parfois on utilisera la norme ISO C99, mais pas la norme ISO C11).

Respecter la norme ANSI

Il faut rajouter les options de compilation suivantes :
-ansi -pedantic-errors

gcc monFichier.c -ansi -pedantic-errors

Tous vos programmes C (TPs) devront respecter la norme ANSI (parfois on utilisera la norme ISO C99, mais pas la norme ISO C11).

Gestionnaire de version préconisé : Git

Gérer plusieurs versions de votre code (locales et distantes) :

- synchroniser votre travail sur plusieurs machines (dont un serveur Git) et avec votre binôme.
- suivi des modifications, revenir à une version précédente, tester plusieurs solutions dans plusieurs branches et ne conserver à la fin que la meilleure solution etc.

Plus d'information dans le document gestion_des_versions_de_vos_projets disponible sur clarolineconnect. *Vous aurez un cours dans 4 semaines.*
Pas obligatoire, mais si un jour vous n'avez pas votre code lors d'une évaluation... ça sera votre problème.

Gestionnaire de version préconisé : Git

Gérer plusieurs versions de votre code (locales et distantes) :

- synchroniser votre travail sur plusieurs machines (dont un serveur Git) et avec votre binôme.
- suivi des modifications, revenir à une version précédente, tester plusieurs solutions dans plusieurs branches et ne conserver à la fin que la meilleure solution etc.

Plus d'information dans le document gestion_des_versions_de_vos_projets disponible sur clarolineconnect. *Vous aurez un cours dans 4 semaines.*
Pas obligatoire, mais si un jour vous n'avez pas votre code lors d'une évaluation... ça sera votre problème.

Gestionnaire de version préconisé : Git

Gérer plusieurs versions de votre code (locales et distantes) :

- synchroniser votre travail sur plusieurs machines (dont un serveur Git) et avec votre binôme.
- suivi des modifications, revenir à une version précédente, tester plusieurs solutions dans plusieurs branches et ne conserver à la fin que la meilleure solution etc.

Plus d'information dans le document gestion_des_versions_de_vos_projets disponible sur clarolineconnect. *Vous aurez un cours dans 4 semaines.*

Pas obligatoire, mais si un jour vous n'avez pas votre code lors d'une évaluation... ça sera votre problème.

Gestionnaire de version préconisé : Git

Gérer plusieurs versions de votre code (locales et distantes) :

- synchroniser votre travail sur plusieurs machines (dont un serveur Git) et avec votre binôme.
- suivi des modifications, revenir à une version précédente, tester plusieurs solutions dans plusieurs branches et ne conserver à la fin que la meilleure solution etc.

Plus d'information dans le document gestion_des_versions_de_vos_projets disponible sur clarolineconnect. *Vous aurez un cours dans 4 semaines.*
Pas obligatoire, mais si un jour vous n'avez pas votre code lors d'une évaluation... ça sera votre problème.

Plan

Langage de programmation impérative / procédurale

Exécution d'instruction (action) étape par étape semblable à une recette de cuisine. Chaque étape est susceptible de modifier l'état actuel du programme (e.g. affectation).

Langage de programmation typé

Les *variables* et *variables constantes* utilisées dans un programme ont un type (e.g. entier, réel) et il est possible de créer des types composés à partir des types de base (e.g. tableau de 10 entiers).

Avez-vous déjà entendu parlé de variable et de constante en cours d'algo ?

Langage de programmation impérative / procédurale

Exécution d'instruction (action) étape par étape semblable à une recette de cuisine. Chaque étape est susceptible de modifier l'état actuel du programme (e.g. affectation).

Langage de programmation typé

Les *variables* et *variables constantes* utilisées dans un programme ont un type (e.g. entier, réel) et il est possible de créer des types composés à partir des types de base (e.g. tableau de 10 entiers).

Avez-vous déjà entendu parlé de variable et de constante en cours d'algo ?

Langage de programmation impérative / procédurale

Exécution d'instruction (action) étape par étape semblable à une recette de cuisine. Chaque étape est susceptible de modifier l'état actuel du programme (e.g. affectation).

Langage de programmation typé

Les *variables* et *variables constantes* utilisées dans un programme ont un type (e.g. entier, réel) et il est possible de créer des types composés à partir des types de base (e.g. tableau de 10 entiers).

Avez-vous déjà entendu parlé de variable et de constante en cours d'algo ?

Un nombre fini d'opérations

Opérateurs classiques : arithmétiques (+,−,*,/,%), relationnels (<,<=,>,>=,==, !=), logiques (&&,||, !), affectation (=), in/décrémentation (++, −−).
Opérateurs moins classiques : manipulation de bits (&,|,∧, ≪,≫), adresse d'une variable (&), taille en octets de la variable ou du type (**sizeof**). Pas d'opérateur puissance en C !

Un nombre fini d'instructions

Instructions *de choix* (**if**, **switch**), instructions *de boucle* (**do-while**, **while**, **for**).

Quelles opérations et instructions avez-vous vu en cours d'algo ?

Un nombre fini d'opérations

Opérateurs classiques : arithmétiques (+,−,*,/,%), relationnels (<,<=,>,>=,==,!=), logiques (&&,||,!), affectation (=), in/décrémentation (++, −−).
Opérateurs moins classiques : manipulation de bits (&,|,∧, ≪,≫), adresse d'une variable (&), taille en octets de la variable ou du type (**sizeof**). Pas d'opérateur puissance en C !

Un nombre fini d'instructions

Instructions *de choix* (**if**, **switch**), instructions *de boucle* (**do-while**, **while**, **for**).

Quelles opérations et instructions avez-vous vu en cours d'algo ?

Un nombre fini d'opérations

Opérateurs classiques : arithmétiques (+,−,*,/,%), relationnels (<,<=,>,>=,==,!=), logiques (&&,||,!), affectation (=), in/décrémentation (++, −−).
Opérateurs moins classiques : manipulation de bits (&,|,∧, ≪,≫), adresse d'une variable (&), taille en octets de la variable ou du type (**sizeof**). Pas d'opérateur puissance en C !

Un nombre fini d'instructions

Instructions *de choix* (**if**, **switch**), instructions *de boucle* (**do-while**, **while**, **for**).

Quelles opérations et instructions avez-vous vu en cours d'algo ?

Une bibliothèque standard

Elle contient **tout ce qui est interaction avec le système** (entrées-sorties, allocation de mémoire, etc.). Ces interactions sont réalisées par appel de *fonctions* (sous-programme).

Premier programme en C

```c
/* Affiche Hello World */
#include <stdio.h> /* Entrées et sorties
                      standards */
int main(void) {
   printf("Hello World!");
   /* printf : fonction prédéfinie, fournie
      avec le langage */
   return 0;
}
```

Le point d'entrée du programme est la fonction main

Le nom du programme principal main **est imposé**. Le main est délimité par les accolades "{" et "}" (**bloc**).
Voici les en-têtes usuelles pour le programme principal :

```
int main(void) ou int main() /* si on n'utilise
pas les arguments de la ligne de commande */

int main(int arg, char *argv[]) ou int main(int
arg, char **argv) /* si on utilise les arguments
de la ligne de commande */
```

Ordre de lecture d'un programme C

Un programme C est "lu" progressivement du haut (début de fichier) vers le bas (fin de fichier).

On pourra utiliser un symbole S à une ligne n seulement s'il a été défini à une ligne précédente !

Ordre de lecture d'un programme C

Un programme C est "lu" progressivement du haut (début de fichier) vers le bas (fin de fichier).

On pourra utiliser un symbole S à une ligne n seulement s'il a été défini à une ligne précédente !

Plan

structure_prog.c

```c
/* 1) Les directives adressées au préprocesseur. */
#include <stdio.h> /* fichier d'en-tête à insérer */
#define NB 5       /* définition d'un symbole NB */
int main(void)
{
    /* 2) Les déclarations de variables */
    int i;
    int j=NB;
    /* 3) Les instructions */
    j++;
    i=j+4;
    printf("%d\n",i);
    return 0;
}
```

Les directives

- Elles doivent être en début de ligne.
- Elles commencent par un #.
- Une directive par ligne.
- Elles s'appliquent à la partie du programme qui leur succède.

Il est conseillé de les mettre au début des fichiers. Le préprocesseur remplace les symboles par leur valeur.

Les directives

- Elles doivent être en début de ligne.
- Elles commencent par un **#**.
- Une directive par ligne.
- Elles s'appliquent à la partie du programme qui leur succède.

Il est conseillé de les mettre au début des fichiers. Le préprocesseur remplace les symboles par leur valeur.

Les directives

- Elles doivent être en début de ligne.
- Elles commencent par un **#**.
- Une directive par ligne.
- Elles s'appliquent à la partie du programme qui leur succède.

Il est conseillé de les mettre au début des fichiers. Le prépro-cesseur remplace les symboles par leur valeur.

Les directives

- Elles doivent être en début de ligne.
- Elles commencent par un **#**.
- Une directive par ligne.
- Elles s'appliquent à la partie du programme qui leur succède.

Il est conseillé de les mettre au début des fichiers. Le prépro-cesseur remplace les symboles par leur valeur.

Déclarations

```
int i;
int j=NB;
```

Ces deux instructions sont des déclarations de variables.

En C, comme dans la plupart des langages actuels, **les déclarations des types des variables sont obligatoires et doivent être regroupées au début du programme**.

Intérêt de la déclaration d'une variable ?

- Typer une variable, et donc spécifier sa taille mémoire ainsi que son codage ;
- Réserver une zone mémoire pour stocker cette variable (mémoire centrale), et pouvoir écrire ou lire une valeur dans cette zone ;
  - Pouvoir mémoriser des résultats de calculs utilisés plus tard ou à plusieurs endroits.
  - Pouvoir contrôler le déroulement d'une boucle (for, while, do-while).

Intérêt de la déclaration d'une variable ?

- Typer une variable, et donc spécifier sa taille mémoire ainsi que son codage ;
- Réserver une zone mémoire pour stocker cette variable (mémoire centrale), et pouvoir écrire ou lire une valeur dans cette zone ;
  - Pouvoir mémoriser des résultats de calculs utilisés plus tard ou à plusieurs endroits.
  - Pouvoir contrôler le déroulement d'une boucle (for, while, do-while).

Intérêt de la déclaration d'une variable ?

- Typer une variable, et donc spécifier sa taille mémoire ainsi que son codage ;
- Réserver une zone mémoire pour stocker cette variable (mémoire centrale), et pouvoir écrire ou lire une valeur dans cette zone ;
  - Pouvoir mémoriser des résultats de calculs utilisés plus tard ou à plusieurs endroits.
  - Pouvoir contrôler le déroulement d'une boucle (for, while, do-while).

Intérêt de la déclaration d'une variable ?

- Typer une variable, et donc spécifier sa taille mémoire ainsi que son codage ;
- Réserver une zone mémoire pour stocker cette variable (mémoire centrale), et pouvoir écrire ou lire une valeur dans cette zone ;
  - Pouvoir mémoriser des résultats de calculs utilisés plus tard ou à plusieurs endroits.
  - Pouvoir contrôler le déroulement d'une boucle (for, while, do-while).

Variables locales ou temporaires

Une **variable locale**, appelée aussi *variable temporaire*, est une variable déclarée à l'intérieur d'un bloc **{ ... }**. Sa durée de vie est limitée à son bloc.

Une variable déclarée à l'extérieur de tout bloc { ... } est une **variable globale**. Sa durée de vie est celle de la durée de vie du programme.

Variables locales ou temporaires

Une **variable locale**, appelée aussi *variable temporaire*, est une variable déclarée à l'intérieur d'un bloc **{ ... }**. Sa durée de vie est limitée à son bloc.

Une variable déclarée à l'extérieur de tout bloc **{ ... }** est une **variable globale**. Sa durée de vie est celle de la durée de vie du programme.

Identificateurs

Ils servent à **désigner les différents "objets" manipulés par le programme** tels que les *variables* et les *fonctions*.

En C :

- Les identificateurs sont formés d'une suite de lettres (caractères anglais !) ou de chiffres, le premier d'entre eux étant forcément une lettre. "_" est considéré comme une lettre.
- Seuls les 32 1ers caractères sont significatifs.
- Les minuscules et majuscules ne sont pas équivalentes.

Identificateurs

Ils servent à **désigner les différents "objets" manipulés par le programme** tels que les *variables* et les *fonctions*.

En C :

- Les identificateurs sont formés d'une suite de lettres (caractères anglais !) ou de chiffres, le premier d'entre eux étant forcément une lettre. "_" est considéré comme une lettre.
- Seuls les 32 1ers caractères sont significatifs.
- Les minuscules et majuscules ne sont pas équivalentes.

Identificateurs

Ils servent à **désigner les différents "objets" manipulés par le programme** tels que les *variables* et les *fonctions*.

En C :

- Les identificateurs sont formés d'une suite de lettres (caractères anglais !) ou de chiffres, le premier d'entre eux étant forcément une lettre. "_" est considéré comme une lettre.
- Seuls les 32 1ers caractères sont significatifs.
- Les minuscules et majuscules ne sont pas équivalentes.

Identificateurs

Ils servent à **désigner les différents "objets" manipulés par le programme** tels que les *variables* et les *fonctions*.

En C :

- Les identificateurs sont formés d'une suite de lettres (caractères anglais !) ou de chiffres, le premier d'entre eux étant forcément une lettre. "_" est considéré comme une lettre.
- Seuls les 32 1ers caractères sont significatifs.
- Les minuscules et majuscules ne sont pas équivalentes.

Les mots-clés du langage C qui ne peuvent pas être utilisés comme *identificateur*

Mots-clés ***types*** : void, char, int, float, double, short, long, const, signed, unsigned, enum, typedef, struct, union, extern, static, auto, register, sizeof, volatile ;

Mots-clés ***instructions*** : for, while, if, else, switch, case, default, break, continue, return, goto

Les séparateurs

En C, 2 identificateurs ou 1 mot-clé et 1 identificateur ou 2 mots-clés sont séparés :

- soit par un caractère particulier , = () [] {} ; * .
- soit par un espace ou une fin de ligne.

Les commentaires /* */

Ils sont formés de caractères quelconques étendus sur une ou plusieurs lignes et placés entre les symboles /* et */.

Ils peuvent apparaître à tout endroit du programme où un séparateur pourrait être placé.

Les commentaires fin de ligne //

La norme ISO C99 autorise une seconde forme de commentaire, dit "fin de ligne". Un tel commentaire est introduit par // et tout ce qui suit le // jusqu'à la fin de ligne est un commentaire.

Règles d'un programme C

- Chaque fichier source peut contenir des directives, des déclarations de variables (et fonctions).
- Le programme C doit contenir **une et une seule fonction main**.
- Les espaces, tabulations, fins de lignes sont ignorées lors de la compilation.
- Les commentaires /* ... */ et // sont ignorés lors de la compilation.
- Les mots-clés du langage C sont réservés et ne peuvent pas être utilisés comme identificateur (de variable ou de fonction).

Règles d'un programme C

- Chaque fichier source peut contenir des directives, des déclarations de variables (et fonctions).
- Le programme C doit contenir **une et une seule fonction main**.
- Les espaces, tabulations, fins de lignes sont ignorées lors de la compilation.
- Les commentaires /* ... */ et // sont ignorés lors de la compilation.
- Les mots-clés du langage C sont réservés et ne peuvent pas être utilisés comme identificateur (de variable ou de fonction).

Règles d'un programme C

- Chaque fichier source peut contenir des directives, des déclarations de variables (et fonctions).
- Le programme C doit contenir **une et une seule fonction main**.
- Les espaces, tabulations, fins de lignes sont ignorées lors de la compilation.
- Les commentaires /* ... */ et // sont ignorés lors de la compilation.
- Les mots-clés du langage C sont réservés et ne peuvent pas être utilisés comme identificateur (de variable ou de fonction).

Règles d'un programme C

- Chaque fichier source peut contenir des directives, des déclarations de variables (et fonctions).
- Le programme C doit contenir **une et une seule fonction main**.
- Les espaces, tabulations, fins de lignes sont ignorées lors de la compilation.
- Les commentaires /* ... */ et // sont ignorés lors de la compilation.
- Les mots-clés du langage C sont réservés et ne peuvent pas être utilisés comme identificateur (de variable ou de fonction).

Règles d'un programme C

- Chaque fichier source peut contenir des directives, des déclarations de variables (et fonctions).
- Le programme C doit contenir **une et une seule fonction main**.
- Les espaces, tabulations, fins de lignes sont ignorées lors de la compilation.
- Les commentaires /* ... */ et // sont ignorés lors de la compilation.
- Les mots-clés du langage C sont réservés et ne peuvent pas être utilisés comme identificateur (de variable ou de fonction).

Questions

- *A quoi sert une directive C ?*
- *Donner 2 mots qui ne sont pas des mots-clés ni des identificateurs.*
- *Quel est l'intérêt des commentaires laissés par le programmeur ?*

Plan

3 notions importantes liées :

- L'encodage et le stockage de l'information.
- Les types de base.
- Les variables.

Mémoire centrale et codage de l'information (1/2)

- La mémoire centrale (RAM) est un ensemble de 0 et 1 (positions binaires). **L'unité de codage est le bit**.
- Les bits sont regroupés en octets (8 bits). **Le plus petit espace mémoire adressable est l'octet**. L'information est donc organisée et manipulée par paquets d'octets.
- Une information en mémoire centrale est repérée par son adresse (l'@ du 1er octet à lire) et son type (qui spécifie le nombre d'octets à lire).

Que représente l'octet 01001101 en mémoire centrale ? Un nombre, un caractère, une partie d'une instruction machine ?

Mémoire centrale et codage de l'information (1/2)

- La mémoire centrale (RAM) est un ensemble de 0 et 1 (positions binaires). **L'unité de codage est le bit**.
- Les bits sont regroupés en octets (8 bits). **Le plus petit espace mémoire adressable est l'octet**. L'information est donc organisée et manipulée par paquets d'octets.
- Une information en mémoire centrale est repérée par son adresse (l'@ du 1er octet à lire) et son type (qui spécifie le nombre d'octets à lire).

Que représente l'octet 01001101 en mémoire centrale ? Un nombre, un caractère, une partie d'une instruction machine ?

Mémoire centrale et codage de l'information (1/2)

- La mémoire centrale (RAM) est un ensemble de 0 et 1 (positions binaires). **L'unité de codage est le bit**.
- Les bits sont regroupés en octets (8 bits). **Le plus petit espace mémoire adressable est l'octet**. L'information est donc organisée et manipulée par paquets d'octets.
- Une information en mémoire centrale est repérée par son adresse (l'@ du 1er octet à lire) et son type (qui spécifie le nombre d'octets à lire).

Que représente l'octet 01001101 en mémoire centrale ? Un nombre, un caractère, une partie d'une instruction machine ?

Mémoire centrale et codage de l'information (1/2)

- La mémoire centrale (RAM) est un ensemble de 0 et 1 (positions binaires). **L'unité de codage est le bit**.
- Les bits sont regroupés en octets (8 bits). **Le plus petit espace mémoire adressable est l'octet**. L'information est donc organisée et manipulée par paquets d'octets.
- Une information en mémoire centrale est repérée par son adresse (l'@ du 1er octet à lire) et son type (qui spécifie le nombre d'octets à lire).

Que représente l'octet 01001101 en mémoire centrale ? Un nombre, un caractère, une partie d'une instruction machine ?

Mémoire centrale et codage de l'information (1/2)

- La mémoire centrale (RAM) est un ensemble de 0 et 1 (positions binaires). **L'unité de codage est le bit**.
- Les bits sont regroupés en octets (8 bits). **Le plus petit espace mémoire adressable est l'octet**. L'information est donc organisée et manipulée par paquets d'octets.
- Une information en mémoire centrale est repérée par son adresse (l'@ du 1er octet à lire) et son type (qui spécifie le nombre d'octets à lire).

Que représente l'octet 01001101 en mémoire centrale ? Un nombre, un caractère, une partie d'une instruction machine ?

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  $2^n * 8$ = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).

$2^n * 8$ = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

Mémoire centrale et codage de l'information (2/2)

Combien de valeurs sont codables avec :

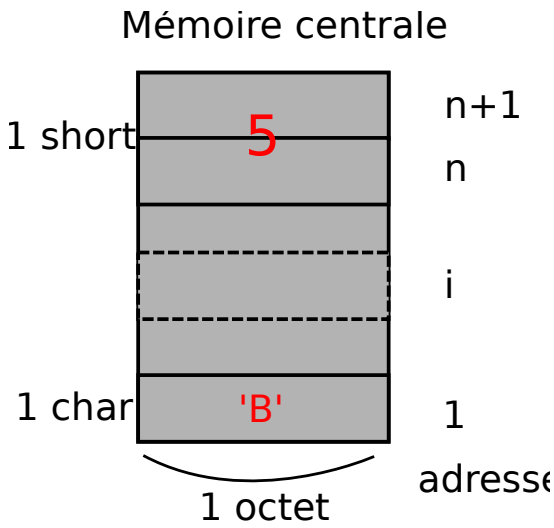
- 1 bit ? 2. [0, 1]
- 2 bits ? $4 = 2^2$. [00, 01, 10, 11]
- n bits ? 2^n.

Donc avec n bits on peut :

- coder un entier naturel appartenant à $[0; 2^n - 1]$.
- coder un entier relatif appartenant à $[-2^{n-1}; 2^{n-1} - 1]$.
- adresser 2^n octets différents (n = 32 ou 64).
  2^n * 8 = le nombre total de bits utilisables en mémoire = $\frac{2^n}{10^9} Go$. Avant 1998 on obtenait un Go en divisant par 2^{30}.

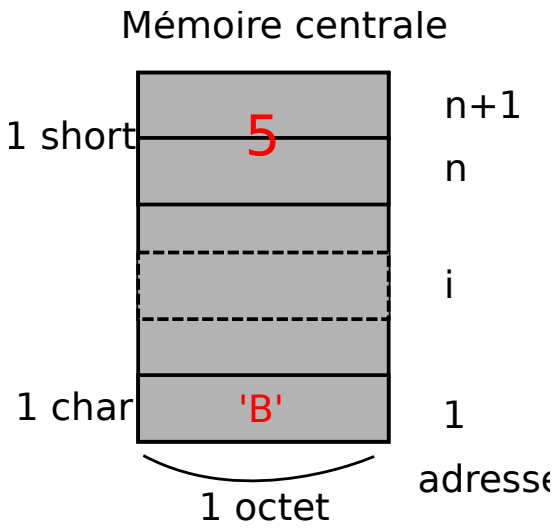
Intérêts des types ?

- **Codage de l'information** dans la mémoire centrale : à chaque type un codage particulier est associé.
  En particulier le nombre d'octets utilisés dépend du type.
- Optimisation de la **complexité mémoire**.
- **Contrôle** sur les opérations possibles (et sur les arguments d'une fonction).



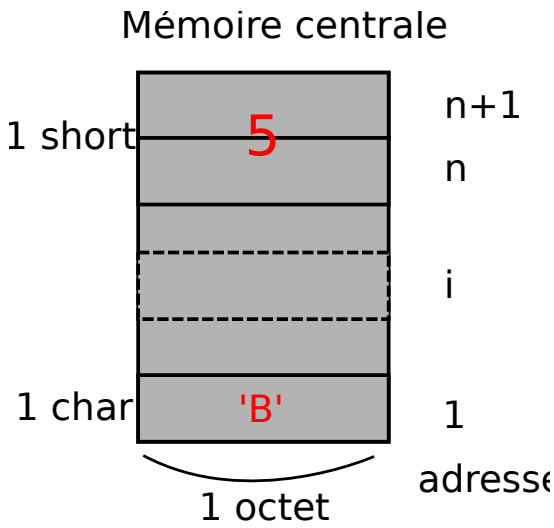
Intérêts des types ?

- **Codage de l'information** dans la mémoire centrale : à chaque type un codage particulier est associé. En particulier le nombre d'octets utilisés dépend du type.
- Optimisation de la **complexité mémoire**.
- **Contrôle** sur les opérations possibles (et sur les arguments d'une fonction).



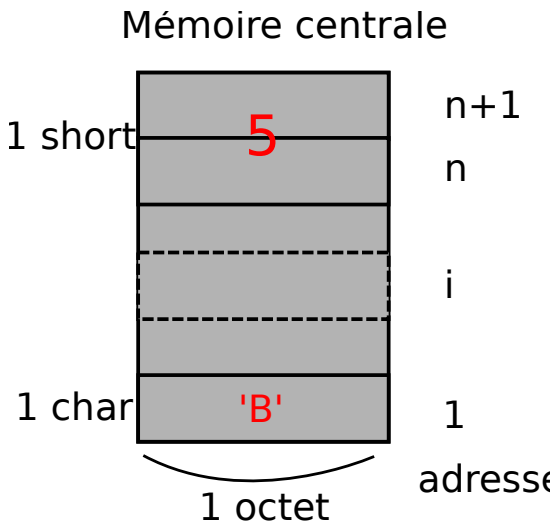
Intérêts des types ?

- **Codage de l'information** dans la mémoire centrale : à chaque type un codage particulier est associé. En particulier le nombre d'octets utilisés dépend du type.
- Optimisation de la **complexité mémoire**.
- **Contrôle** sur les opérations possibles (et sur les arguments d'une fonction).



Intérêts des types ?

- **Codage de l'information** dans la mémoire centrale : à chaque type un codage particulier est associé. En particulier le nombre d'octets utilisés dépend du type.
- Optimisation de la **complexité mémoire**.
- **Contrôle** sur les opérations possibles (et sur les arguments d'une fonction).



Les types de bases du langage C

- **Types scalaires ou simples** : les nombres entiers, les nombres flottants et les caractères.
- Aucun type de base en C ne permet de représenter un nombre aussi grand qu'on veut.
- A partir des types de base, d'autres, dits **types dérivés** pourront être construits tels que des tableaux.

Les types de bases du langage C

- **Types scalaires ou simples** : les nombres entiers, les nombres flottants et les caractères.
- Aucun type de base en C ne permet de représenter un nombre aussi grand qu'on veut.
- A partir des types de base, d'autres, dits **types dérivés** pourront être construits tels que des tableaux.

Les types de bases du langage C

- **Types scalaires ou simples** : les nombres entiers, les nombres flottants et les caractères.
- Aucun type de base en C ne permet de représenter un nombre aussi grand qu'on veut.
- A partir des types de base, d'autres, dits **types dérivés** pourront être construits tels que des tableaux.

Les types entiers (1/4)

Le mot-clé **int** correspond à la représentation des nombres **entiers relatifs**.

1 bit est réservé pour représenter le signe du nombre (0 positif / 1 négatif). $n \geq 0$ est ensuite codé par sa représentation binaire. $n < 0$ est codé comme suit : 1) on "inverse" la représentation binaire de $|n|$, 2) on ajoute 1 (ce qui correspond à la représentation binaire de $2^{32} - |n|$).

Le codage des nombres signés est dit 'complément à 2'.

Les types entiers (1/4)

Le mot-clé **int** correspond à la représentation des nombres **entiers relatifs**.

1 bit est réservé pour représenter le signe du nombre (0 positif / 1 négatif). $n \geq 0$ est ensuite codé par sa représentation binaire. $n < 0$ est codé comme suit : 1) on "inverse" la représentation binaire de $|n|$, 2) on ajoute 1 (ce qui correspond à la représentation binaire de $2^{32} - |n|$).

Le codage des nombres signés est dit 'complément à 2'.

Les types entiers (1/4)

Le mot-clé **int** correspond à la représentation des nombres **entiers relatifs**.

1 bit est réservé pour représenter le signe du nombre (0 positif / 1 négatif). $n \geq 0$ est ensuite codé par sa représentation binaire. $n < 0$ est codé comme suit : 1) on "inverse" la représentation binaire de $|n|$, 2) on ajoute 1 (ce qui correspond à la représentation binaire de $2^{32} - |n|$).

Le codage des nombres signés est dit 'complément à 2'.

Les types entiers (2/4)

Autre types d'entiers relatifs :

- **short int** (qu'on peut abréger en **short**).
- **long int** (qu'on peut abréger en **long**).

En général, un **short** est codé sur 2 octets, un **long** sur 4 octets et un **int** sur 2 ou 4 octets. Cela dépend des architectures.

Avec 16 bits, on peut représenter un entier relatif dans l'intervalle [-32768 ; 32767]. Avec 32 bits, dans l'intervalle [-2147483648 ; 2147483647].

Les types entiers (2/4)

Autre types d'entiers relatifs :

- **short int** (qu'on peut abréger en **short**).
- **long int** (qu'on peut abréger en **long**).

En général, un **short** est codé sur 2 octets, un **long** sur 4 octets et un **int** sur 2 ou 4 octets. Cela dépend des architectures.

Avec 16 bits, on peut représenter un entier relatif dans l'intervalle [-32768 ; 32767]. Avec 32 bits, dans l'intervalle [-2147483648 ; 2147483647].

Les types entiers (2/4)

Autre types d'entiers relatifs :

- **short int** (qu'on peut abréger en **short**).
- **long int** (qu'on peut abréger en **long**).

En général, un **short** est codé sur 2 octets, un **long** sur 4 octets et un **int** sur 2 ou 4 octets. Cela dépend des architectures.

Avec 16 bits, on peut représenter un entier relatif dans l'intervalle [-32768 ; 32767]. Avec 32 bits, dans l'intervalle [-2147483648 ; 2147483647].

Les types entiers (3/4)

Les types entiers peuvent être nuancés par l'utilisation du qualificatif **unsigned** pour représenter seulement des entiers positifs (plus de bit utilisé pour le signe) :

- **unsigned int**
- **unsigned short**
- **unsigned long**

Avec 16 bits, on peut représenter un entier naturel dans l'intervalle [0 ; 65535]. Avec 32 bits, dans l'intervalle [0 ; 4294967295].

Les types entiers (3/4)

Les types entiers peuvent être nuancés par l'utilisation du qualificatif **unsigned** pour représenter seulement des entiers positifs (plus de bit utilisé pour le signe) :

- **unsigned int**
- **unsigned short**
- **unsigned long**

Avec 16 bits, on peut représenter un entier naturel dans l'intervalle [0 ; 65535]. Avec 32 bits, dans l'intervalle [0 ; 4294967295].

Les types entiers (4/4)

Exemple de constantes entières : 5 -45 +237 0477 (constante octale) 0x5a2b 0X5a2b (constantes hexadécimales).

Pour forcer une constante positive a être **unsigned**, utiliser le suffixe `u` ou `U`. Utiliser le suffixe `l` ou `L` pour forcer une constante en **long**. Et `ul` ou `UL` pour du **unsigned long**.

Les types entiers (4/4)

Exemple de constantes entières : 5 -45 +237 0477 (constante octale) 0x5a2b 0X5a2b (constantes hexadécimales).

Pour forcer une constante positive a être **unsigned**, utiliser le suffixe u ou U. Utiliser le suffixe l ou L pour forcer une constante en **long**. Et ul ou UL pour du **unsigned long**.

Les types flottants (1/3)

Les types flottants permettent de représenter, **de manière approchée**, une partie des nombres réels.

Un nombre réel sera représenté en flottant en déterminant 2 quantités **M** (mantisse) et **E** (exposant) telles que $M.B^E$ représente une approximation de ce nombre.
La base B est généralement unique pour une machine donnée (2 ou 16) et ne figure pas explicitement dans la représentation de ce nombre.

Les types flottants (1/3)

Les types flottants permettent de représenter, **de manière approchée**, une partie des nombres réels.

Un nombre réel sera représenté en flottant en déterminant 2 quantités **M** (mantisse) et **E** (exposant) telles que $M.B^E$ représente une approximation de ce nombre.

La base B est généralement unique pour une machine donnée (2 ou 16) et ne figure pas explicitement dans la représentation de ce nombre.

Les types flottants (1/3)

Les types flottants permettent de représenter, **de manière approchée**, une partie des nombres réels.

Un nombre réel sera représenté en flottant en déterminant 2 quantités **M** (mantisse) et **E** (exposant) telles que $M.B^E$ représente une approximation de ce nombre.
La base B est généralement unique pour une machine donnée (2 ou 16) et ne figure pas explicitement dans la représentation de ce nombre.

Les types flottants (2/3)

Le C prévoit 3 types de flottants de tailles différentes :

- **float** : 4 octets, précision de 6 chiffres après la virgule.
- **double** : 8 octets, précision de 15 chiffres après la virgule.
- **long double** : nombre d'octets et précision qui dépendent des architectures, au moins ceux d'un double.

L'ensemble des nombres représentables à l'erreur de troncature près est au moins $[10^{-37} ; 10^{+37}]$ (*domaine couvert*).

Les types flottants (3/3)

Deux notations pour les constantes flottantes :

- **décimale** : obligatoirement un "." : 12.43 -0.38 -.38 4. .27
- **exponentielle**, en puissance de 10 : e ou E : 4.25E4 4.25e+4 42.5E3 48.e13 5427e-34

Par défaut, une constante flottante est de type **double** :

- Pour du type **float**, faire suivre son écriture de la lettre f (ou F) : gain de place mémoire en contrepartie d'une éventuelle perte de précision.
- Pour du type **long double**, faire suivre son écriture de la lettre l (ou L) : gain éventuel de précision, en contrepartie d'une perte en place mémoire.

Les types flottants (3/3)

Deux notations pour les constantes flottantes :

- **décimale** : obligatoirement un "." : 12.43 -0.38 -.38 4. .27
- **exponentielle**, en puissance de 10 : e ou E : 4.25E4 4.25e+4 42.5E3 48.e13 5427e-34

Par défaut, une constante flottante est de type **double** :

- Pour du type **float**, faire suivre son écriture de la lettre f (ou F) : gain de place mémoire en contrepartie d'une éventuelle perte de précision.
- Pour du type **long double**, faire suivre son écriture de la lettre l (ou L) : gain éventuel de précision, en contrepartie d'une perte en place mémoire.

Les types caractères

Le C permet de manipuler des caractères codés en mémoire sur un octet (8 bits) avec le type **char**. Certains caractères, nommés *caractères de contrôle*, ne sont pas associés à un graphisme.

Les constantes de type **char** (caractère), se notent en écrivant entre apostrophes (ou quotes) le caractère voulu : 'a' 'Y' '+' '£' '\n' '\t' '\a'

Le type **char** est parfois utilisé pour représenter des "petits" entiers sur [-128 ; 127] (**signed**) ou [0 ; 255] (**unsigned**). Par contre cela introduit une conversion vers le type **int** si des opérations sont effectuées dessus.

Les types caractères

Le C permet de manipuler des caractères codés en mémoire sur un octet (8 bits) avec le type **char**. Certains caractères, nommés *caractères de contrôle*, ne sont pas associés à un graphisme.

Les constantes de type **char** (caractère), se notent en écrivant entre apostrophes (ou quotes) le caractère voulu : 'a' 'Y' '+' '£' '\n' '\t' '\a'

Le type **char** est parfois utilisé pour représenter des "petits" entiers sur [-128 ; 127] (**signed**) ou [0 ; 255] (**unsigned**). Par contre cela introduit une conversion vers le type **int** si des opérations sont effectuées dessus.

Les types caractères

Le C permet de manipuler des caractères codés en mémoire sur un octet (8 bits) avec le type **char**. Certains caractères, nommés *caractères de contrôle*, ne sont pas associés à un graphisme.

Les constantes de type **char** (caractère), se notent en écrivant entre apostrophes (ou quotes) le caractère voulu : 'a' 'Y' '+' '£' '\n' '\t' '\a'

Le type **char** est parfois utilisé pour représenter des "petits" entiers sur [-128 ; 127] (**signed**) ou [0 ; 255] (**unsigned**). Par contre cela introduit une conversion vers le type **int** si des opérations sont effectuées dessus.

Les variables

Une variable

Un emplacement mémoire dans lequel est codé une information modifiable et utilisable grâce à un identificateur. En C, on peut aussi manipuler des informations via leur adresse (pointeur).

- **Syntaxe** : <*type*> <*identificateur* [= *valeur initialisation*] >.
- **Initialisation** : attribution d'une valeur à la variable dès sa création.

Les variables constantes

Qualificatif **const** : la valeur d'une variable ne doit pas changer lors de l'exécution du programme. **L'initialisation à la déclaration est alors obligatoire**.

```
int x ; /* déclaration d'une variable */
int a = 7 ; /* déclaration et initialisation d'une
    variable */
const int nb = 35 ; /* déclaration d'une constante */
```

L'opérateur **sizeof**

Syntaxe :

- **sizeof**(<*type*>)
- **sizeof**(<*variable*>)

sizeof renvoie la taille en octets (un entier naturel) d'un type ou d'une variable.

Questions

- *Quel est le choix de type le plus judicieux pour une variable destinée à contenir une valeur toujours dans [0 ;255] ?*
- *Quelle est la signification de "int a ;" du point de vue de la machine ?*

Plan

Pour afficher des informations : la fonction printf (1/3)

- Fontion de la bibliothèque standard stdio.
- **Syntaxe** :
  - printf("chaine de caractères") ;
  - printf("texte1%code1 texte2%code2 ... ", <expression1>, <expression2>,...) ;

Exemples :

```
printf("Bonjour\n vous allez
bien?"); /* caractères à afficher tels quels */
printf("%d+%o=%x",5,10,5+10);
printf(" un entier : %d ; un
flottant : %f ; un caractere : %c ; une chaine
de caractères : %s",5, 4.2, 'c', "hello");
```

Pour afficher des informations : la fonction printf (1/3)

- Fontion de la bibliothèque standard stdio.
- **Syntaxe** :
  - printf("chaine de caractères") ;
  - printf("texte1%code1 texte2%code2 ... ", <expression1>, <expression2>,...) ;

Exemples :

```
                printf("Bonjour\n vous allez
bien?"); /* caractères à afficher tels quels */
                printf("%d+%o=%x",5,10,5+10);
                printf(" un entier : %d ; un
flottant : %f ; un caractere : %c ; une chaine
de caractères : %s",5, 4.2, 'c', "hello");
```

Pour afficher des informations : la fonction printf (2/3)

- Les guillemets " ... " servent à délimiter une *chaîne de caractères*.
- La notation \n est conventionnelle, elle représente le caractère *fin de ligne* qui provoque le passage à la ligne.

Principaux codes format (écriture) :

- **int** : %d (**long** : %ld)
- **unsigned int** : %u (**unsigned long** : %lu)
- **float** ou **double** : %f (décimale 6 chiffres après .) ou %e (exponentielle)
- chaîne de caractères : %s

Pour afficher des informations : la fonction printf (2/3)

- Les guillemets " ... " servent à délimiter une *chaîne de caractères*.
- La notation **n** est conventionnelle, elle représente le caractère *fin de ligne* qui provoque le passage à la ligne.

Principaux codes format (écriture) :

- **int** : %d (**long** : %ld)
- **unsigned int** : %u (**unsigned long** : %lu)
- **float** ou **double** : %f (décimale 6 chiffres après .) ou %e (exponentielle)
- chaîne de caractères : %s

Pour afficher des informations : la fonction printf (2/3)

- Les guillemets " ... " servent à délimiter une *chaîne de caractères*.
- La notation **n** est conventionnelle, elle représente le caractère *fin de ligne* qui provoque le passage à la ligne.

Principaux codes format (écriture) :

- **int** : %d (**long** : %ld)
- **unsigned int** : %u (**unsigned long** : %lu)
- **float** ou **double** : %f (décimale 6 chiffres après .) ou %e (exponentielle)
- chaîne de caractères : %s

Pour afficher des informations : la fonction printf (3/3)

- **Fixer un nombre minimal de caractères à utiliser** (chiffre, . et espace) en plaçant ce nombre à droite du %. Ex : printf("%3d", 2) ; printf("%03d", 2) ;
- **Fixer le nombre de chiffres à afficher après la virgule** d'un flottant en plaçant ce nombre à gauche du code de conversion et en le précédant d'un point. Ex : printf("%.2f", 3.14159) ;
- Exemple les 2 ensembles : printf("%7.3f", 3.14159) ;

Que se passe-t-il dans les cas suivants ?

```
printf("%d ; %d \n",4.3f, 2) ;
printf("%f ; %d \n",4, 2) ;
printf("%d ; %d \n", 2) ;
printf("%d ; \n", 2, 7) ;
```

Pour afficher des informations : la fonction printf (3/3)

- **Fixer un nombre minimal de caractères à utiliser** (chiffre, . et espace) en plaçant ce nombre à droite du %. Ex : printf("%3d", 2) ; printf("%03d", 2) ;
- **Fixer le nombre de chiffres à afficher après la virgule** d'un flottant en plaçant ce nombre à gauche du code de conversion et en le précédant d'un point. Ex : printf("%.2f", 3.14159) ;
- Exemple les 2 ensembles : printf("%7.3f", 3.14159) ;

Que se passe-t-il dans les cas suivants ?

```
printf("%d ; %d \n",4.3f, 2) ;
printf("%f ; %d \n",4, 2) ;
printf("%d ; %d \n", 2) ;
printf("%d ; \n", 2, 7) ;
```

Pour afficher des informations : la fonction printf (3/3)

- **Fixer un nombre minimal de caractères à utiliser** (chiffre, . et espace) en plaçant ce nombre à droite du %. Ex : printf("%3d", 2) ; printf("%03d", 2) ;
- **Fixer le nombre de chiffres à afficher après la virgule** d'un flottant en plaçant ce nombre à gauche du code de conversion et en le précédant d'un point. Ex : printf("%.2f", 3.14159) ;
- Exemple les 2 ensembles : printf("%7.3f", 3.14159) ;

Que se passe-t-il dans les cas suivants ?

```
printf("%d ; %d \n",4.3f, 2) ;
printf("%f ; %d \n",4, 2) ;
printf("%d ; %d \n", 2) ;
printf("%d ; \n", 2, 7) ;
```

Pour afficher des informations : la fonction printf (3/3)

- **Fixer un nombre minimal de caractères à utiliser** (chiffre, . et espace) en plaçant ce nombre à droite du %. Ex : printf("%3d", 2) ; printf("%03d", 2) ;
- **Fixer le nombre de chiffres à afficher après la virgule** d'un flottant en plaçant ce nombre à gauche du code de conversion et en le précédant d'un point. Ex : printf("%.2f", 3.14159) ;
- Exemple les 2 ensembles : printf("%7.3f", 3.14159) ; " 3.142"

Que se passe-t-il dans les cas suivants ?

```
printf("%d ; %d \n",4.3f, 2) ;
printf("%f ; %d \n",4, 2) ;
printf("%d ; %d \n", 2) ;
printf("%d ; \n", 2, 7) ;
```

Pour lire des informations : la fonction scanf (1/3)

- Fontion de la bibliothèque standard stdio.
- **Syntaxe** : scanf("%code1%code2...", &<*variable1*>, &<*variable2*>, ...) ;

Principaux codes format (lecture) :

- **int** : %d (pour un **short** %hd, un **long** %ld)
- **unsigned int** : %u (pour un **unsigned short** %hu, un **unsigned long** %lu)
- **float** : %f ou %e
- **double** : %lf
- **char** : %c
- chaîne de caractères : %s

Pour lire des informations : la fonction scanf (1/3)

- Fontion de la bibliothèque standard stdio.
- **Syntaxe** : scanf("%code1%code2...", &<*variable1*>, &<*variable2*>, ...) ;

Principaux codes format (lecture) :

- **int** : %d (pour un **short** %hd, un **long** %ld)
- **unsigned int** : %u (pour un **unsigned short** %hu, un **unsigned long** %lu)
- **float** : %f ou %e
- **double** : %lf
- **char** : %c
- chaîne de caractères : %s

Pour lire des informations : la fonction scanf (2/3)

- Notez que si x est une variable d'un certain type, alors $\&x$ est l'adresse de cette variable. **&** est l'opérateur *adresse de*.
- scanf lit au fur et à mesure le tampon "entrée clavier". Dès que scanf n'arrive pas à lire une valeur, il s'arrête.
- Si scanf doit lire un nombre, alors elle va "sauter" les caractères espaces et fin de ligne jusqu'à tomber sur un nombre. Si scanf doit lire un caractère, alors scanf lit le 1er caractère quel qu'il soit et va pointer sur le suivant.

Pour lire des informations : la fonction scanf (2/3)

- Notez que si x est une variable d'un certain type, alors $\&x$ est l'adresse de cette variable. **&** est l'opérateur *adresse de*.
- scanf lit au fur et à mesure le tampon "entrée clavier". Dès que scanf n'arrive pas à lire une valeur, il s'arrête.
- Si scanf doit lire un nombre, alors elle va "sauter" les caractères espaces et fin de ligne jusqu'à tomber sur un nombre. Si scanf doit lire un caractère, alors scanf lit le 1er caractère quel qu'il soit et va pointer sur le suivant.

Pour lire des informations : la fonction scanf (2/3)

- Notez que si x est une variable d'un certain type, alors $\&x$ est l'adresse de cette variable. **&** est l'opérateur *adresse de*.
- scanf lit au fur et à mesure le tampon "entrée clavier". Dès que scanf n'arrive pas à lire une valeur, il s'arrête.
- Si scanf doit lire un nombre, alors elle va "sauter" les caractères espaces et fin de ligne jusqu'à tomber sur un nombre. Si scanf doit lire un caractère, alors scanf lit le 1er caractère quel qu'il soit et va pointer sur le suivant.

Pour lire des informations : la fonction scanf (3/3)

```
char c;
int i;
double d;
float x,y;
scanf(" %c", &c); /* lecture d'un caractère non
                     séparateur grâce au ' ' */
scanf("%d",&i);
scanf("%d%lf%f%f",&i,&d,&x,&y);
scanf("%f",x); /* utilisation erronée */
scanf("%f%f",&x,&x);
```

Plan

Définitions

Expression

Entité syntaxiquement correcte qui possède une "valeur" mais ne réalise aucune action, en particulier aucune affectation à une variable. *Sa valeur a un type qui est un des types de base.* Exemples : 5, $6 - 9$.

Instruction

Réalise une "action". Elle peut éventuellement faire intervenir des expressions. Exemples : l'instruction nulle (;), les instructions conditionnelles.

Instruction-expression

Cas d'une instruction qui possède une "valeur". Exemples : l'affectation, l'in/décrémentation.

Définitions

Expression

Entité syntaxiquement correcte qui possède une "valeur" mais ne réalise aucune action, en particulier aucune affectation à une variable. *Sa valeur a un type qui est un des types de base.* Exemples : 5, $6 - 9$.

Instruction

Réalise une "action". Elle peut éventuellement faire intervenir des expressions. Exemples : l'instruction nulle (;), les instructions conditionnelles.

Instruction-expression

Cas d'une instruction qui possède une "valeur". Exemples : l'affectation, l'in/décrémentation.

Définitions

Expression

Entité syntaxiquement correcte qui possède une "valeur" mais ne réalise aucune action, en particulier aucune affectation à une variable. *Sa valeur a un type qui est un des types de base.* Exemples : 5, $6 - 9$.

Instruction

Réalise une "action". Elle peut éventuellement faire intervenir des expressions. Exemples : l'instruction nulle (;), les instructions conditionnelles.

Instruction-expression

Cas d'une instruction qui possède une "valeur". Exemples : l'affectation, l'in/décrémentation.

Les expressions peuvent être formées à l'aide d'opérateurs, et elles interviennent souvent au sein d'instructions :

```
char c = 'A'; /* 'A' est une expression constante */
int n = 2; /* 2 est une expression constante */

float x = (4-1)/(8-2) ; /* expression constante */
float y = 3 * x + 1 ; /* expression non constante */
printf("valeur %f", 3 * x + 1);
printf("valeur booleenne fausse %d et vraie %d",
    5<4, 5>4);
```

$'A'$, 2, $(4-1)/(8-2)$, $3*x+1$, $5<4$ et $5>4$ sont des expressions, car elles ont une valeur.

Précision sur les opérateurs et les conversions implicites (1/2)

- Syntaxe opérateur unaire : **OP** <*expression*>
- Syntaxe opérateur binaire : <*expression1*> **OP** <*expression2*>

Les opérations binaires ne sont définies (en machine) que pour 2 opérandes ayant le même type.

Un mécanisme de **conversion implicite** est mis en place lorsque les 2 opérandes sont de type différent.

La règle est le **respect de l'intégrité des données** : le type retenu est celui qui ne fait pas perdre de précision supplémentaire. Le compilateur prévoit des instructions de conversion.

Précision sur les opérateurs et les conversions implicites (1/2)

- Syntaxe opérateur unaire : **OP** <*expression*>
- Syntaxe opérateur binaire : <*expression1*> **OP** <*expression2*>

Les opérations binaires ne sont définies (en machine) que pour 2 opérandes ayant le même type.

Un mécanisme de **conversion implicite** est mis en place lorsque les 2 opérandes sont de type différent.

La règle est le **respect de l'intégrité des données** : le type retenu est celui qui ne fait pas perdre de précision supplémentaire. Le compilateur prévoit des instructions de conversion.

Précision sur les opérateurs et les conversions implicites (1/2)

- Syntaxe opérateur unaire : **OP** <*expression*>
- Syntaxe opérateur binaire : <*expression1*> **OP** <*expression2*>

Les opérations binaires ne sont définies (en machine) que pour 2 opérandes ayant le même type.

Un mécanisme de **conversion implicite** est mis en place lorsque les 2 opérandes sont de type différent.

La règle est le **respect de l'intégrité des données** : le type retenu est celui qui ne fait pas perdre de précision supplémentaire. Le compilateur prévoit des instructions de conversion.

Précision sur les opérateurs et les conversions implicites (1/2)

- Syntaxe opérateur unaire : **OP** <*expression*>
- Syntaxe opérateur binaire : <*expression1*> **OP** <*expression2*>

Les opérations binaires ne sont définies (en machine) que pour 2 opérandes ayant le même type.

Un mécanisme de **conversion implicite** est mis en place lorsque les 2 opérandes sont de type différent.

La règle est le **respect de l'intégrité des données** : le type retenu est celui qui ne fait pas perdre de précision supplémentaire. Le compilateur prévoit des instructions de conversion.

Précision sur les opérateurs et les conversions implicites (2/2)

- Attention au débordement : une expression donnera son résultat dans son type, même si son type ne peut pas représenter le résultat.
- Les opérateurs arithmétiques ne sont pas définis pour les types **char** et **short**, le compilateur va les convertir en **int** (*promotion numérique* ou *conversion systématique*).
- Il faut éviter d'appliquer des opérateurs binaires entre deux opérandes si l'un est signé et l'autre non-signé, car le résultat peut être dénué de sens.
- Les opérateurs relationnels (<,<=,>,>=,==,!=) donnent des expressions de type entier (0 pour faux et 1 pour vrai).

Précision sur les opérateurs et les conversions implicites (2/2)

- Attention au débordement : une expression donnera son résultat dans son type, même si son type ne peut pas représenter le résultat.
- Les opérateurs arithmétiques ne sont pas définis pour les types **char** et **short**, le compilateur va les convertir en **int** (*promotion numérique* ou *conversion systématique*).
- Il faut éviter d'appliquer des opérateurs binaires entre deux opérandes si l'un est signé et l'autre non-signé, car le résultat peut être dénué de sens.
- Les opérateurs relationnels (<,<=,>,>=,==,!=) donnent des expressions de type entier (0 pour faux et 1 pour vrai).

Précision sur les opérateurs et les conversions implicites (2/2)

- Attention au débordement : une expression donnera son résultat dans son type, même si son type ne peut pas représenter le résultat.
- Les opérateurs arithmétiques ne sont pas définis pour les types **char** et **short**, le compilateur va les convertir en **int** (*promotion numérique* ou *conversion systématique*).
- Il faut éviter d'appliquer des opérateurs binaires entre deux opérandes si l'un est signé et l'autre non-signé, car le résultat peut être dénué de sens.
- Les opérateurs relationnels (<,<=,>,>=,==,!=) donnent des expressions de type entier (0 pour faux et 1 pour vrai).

Précision sur les opérateurs et les conversions implicites (2/2)

- Attention au débordement : une expression donnera son résultat dans son type, même si son type ne peut pas représenter le résultat.
- Les opérateurs arithmétiques ne sont pas définis pour les types **char** et **short**, le compilateur va les convertir en **int** (*promotion numérique* ou *conversion systématique*).
- Il faut éviter d'appliquer des opérateurs binaires entre deux opérandes si l'un est signé et l'autre non-signé, car le résultat peut être dénué de sens.
- Les opérateurs relationnels (<,<=,>,>=,==, !=) donnent des expressions de type entier (0 pour faux et 1 pour vrai).

Priorité des opérateurs arithmétiques

1. Les opérateurs unaires + et −.
2. Les opérateurs binaires *, / et %.
3. Les opérateurs binaires + et −.
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

Priorité des opérateurs arithmétiques

1. Les opérateurs unaires + et −.
2. Les opérateurs binaires *, / et %.
3. Les opérateurs binaires + et −.
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

Priorité des opérateurs arithmétiques

1. Les opérateurs unaires + et −.
2. Les opérateurs binaires *, / et %.
3. Les opérateurs binaires + et −.
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

Priorité des opérateurs arithmétiques

1. Les opérateurs unaires + et −.
2. Les opérateurs binaires *, / et %.
3. Les opérateurs binaires + et −.
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

Priorité des opérateurs arithmétiques

1. Les opérateurs unaires + et −.
2. Les opérateurs binaires *, / et %.
3. Les opérateurs binaires + et −.
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

Priorité des opérateurs relationnels

1. Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
2. Les opérateurs binaires <, <=, >, >=.
3. Les opérateurs binaires == (égalité) et != (différence).
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$x + y < a + 2$?

Priorité des opérateurs relationnels

1. Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
2. Les opérateurs binaires <, <=, >, >=.
3. Les opérateurs binaires == (égalité) et != (différence).
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$x + y < a + 2$? $(x + y) < (a + 2)$

Priorité des opérateurs relationnels

1. Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
2. Les opérateurs binaires <, <=, >, >=.
3. Les opérateurs binaires == (égalité) et != (différence).
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$x + y < a + 2$?

Priorité des opérateurs relationnels

1. Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
2. Les opérateurs binaires <, <=, >, >=.
3. Les opérateurs binaires == (égalité) et != (différence).
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$x + y < a + 2$?

Priorité des opérateurs relationnels

1. Les opérateurs relationnels sont moins prioritaires que les opérateurs arithmétiques.
2. Les opérateurs binaires <, <=, >, >=.
3. Les opérateurs binaires == (égalité) et != (différence).
4. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
5. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$x + y < a + 2$?

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b \,?\, (!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs logiques

1. L'opérateur unaire ! (négation) est le plus prioritaire. 0 signifie faux, tout autre valeur vrai.
2. Les opérateurs binaires || et && sont de priorité inférieure aux opérateurs arithmétiques ou relationnels.
3. L'opérateur || est de priorité inférérieure à l'opérateur &&.
4. **Évaluation feignante des opérateurs** || (arrêt si rencontré un vrai à gauche) **et** && (arrêt si rencontré un faux à gauche).
5. En cas de priorités identiques, les calculs s'effectuent de gauche à droite.
6. Les parenthèses forcent le calcul de l'expression qu'elles contiennent.

$!a == b$? $(!a) == b$ 5.0&&2||0 (5.0&&2)||0

Priorité des opérateurs : récapitulatif

1. **référence** () [] -> .
2. **unaire** + − ++ −− ! * & (cast) **sizeof**
3. **arithmétique** * / %
4. **arithmétique** + −
5. **décalage** « »
6. **manipulation de bits** &
7. **manipulation de bits** ∧
8. **manipulation de bits** |
9. **relationnel** < <= > >=
10. **relationnel** == !=
11. **logique** &&
12. **logique** ||
13. **conditionnel** ? :
14. **affectation** = += -= *= /= %= &= ∧= |= «= »=
15. **séquentiel** ,

Exercice

Donner les valeurs des variables a et b à la fin du programme suivant :

```c
#include <stdio.h>
int main(void) {
    double a, b;
    a = 2.1 * !(8%5 - 3) && ( 5/2-2.5f || 5<=4 );
    b = 43/5/2*3 - 100 % 88;
    printf("a = %f ; b = %f\n", a, b);
    return 0;
}
```

L'affectation (ordinaire) ou assignation (1/2)

Syntaxe : <*variable*> = <*expression*>
L'affectation se réalise grâce à l'opérateur binaire **=** et elle consiste à attribuer une valeur (opérande de droite) à une variable (opérande de gauche). Elle est de priorité la plus faible possible (sauf opérateur séquentiel ,).

```
int i, j, k ;
i = 3 ;
j = k = 5 ; /* interprété comme j = (k = 5) ; */
i = 4*k + 7 ; /* à droite du signe =, une expression
    de type int */
```

L'affectation (ordinaire) ou assignation (2/2)

- La valeur à gauche de l'opérateur d'affectation est appelée *lvalue* (une variable ou une case d'une variable tableau).
- Les affectations en série s'évaluent de la droite vers la gauche (associativité de droite à gauche).
- Si le type de l'expression de droite ne correspond pas au type de la lvalue, alors le type de l'expression est converti dans le type de la lvalue, ce qui peut conduire à une dégradation plus ou moins importante de l'information.

L'affectation (ordinaire) ou assignation (2/2)

- La valeur à gauche de l'opérateur d'affectation est appelée *lvalue* (une variable ou une case d'une variable tableau).
- Les affectations en série s'évaluent de la droite vers la gauche (associativité de droite à gauche).
- Si le type de l'expression de droite ne correspond pas au type de la lvalue, alors le type de l'expression est converti dans le type de la lvalue, ce qui peut conduire à une dégradation plus ou moins importante de l'information.

L'affectation (ordinaire) ou assignation (2/2)

- La valeur à gauche de l'opérateur d'affectation est appelée *lvalue* (une variable ou une case d'une variable tableau).
- Les affectations en série s'évaluent de la droite vers la gauche (associativité de droite à gauche).
- Si le type de l'expression de droite ne correspond pas au type de la lvalue, alors le type de l'expression est converti dans le type de la lvalue, ce qui peut conduire à une dégradation plus ou moins importante de l'information.

L'incrémentation et la décrémentation (1/2)

L'incrémentation ++ et la décrémentation −− ont la priorité la plus élevée dans les expressions, car on ne peut que les appliquer sur des lvalues et pas sur des expressions.

- Si l'opérateur est placé à gauche : préin/prédé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée après application de l'opérateur.
- Si l'opérateur est placé à droite : postin/postdé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée avant application de l'opérateur.

L'incrémentation et la décrémentation (1/2)

L'incrémentation ++ et la décrémentation −− ont la priorité la plus élevée dans les expressions, car on ne peut que les appliquer sur des lvalues et pas sur des expressions.

- Si l'opérateur est placé à gauche : préin/prédé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée après application de l'opérateur.
- Si l'opérateur est placé à droite : postin/postdé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée avant application de l'opérateur.

L'incrémentation et la décrémentation (1/2)

L'incrémentation ++ et la décrémentation −− ont la priorité la plus élevée dans les expressions, car on ne peut que les appliquer sur des lvalues et pas sur des expressions.

- Si l'opérateur est placé à gauche : préin/prédé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée après application de l'opérateur.
- Si l'opérateur est placé à droite : postin/postdé-crémentation : si la lvalue est dans une expression, alors la valeur de l'expression est calculée avant application de l'opérateur.

L'incrémentation et la décrémentation (2/2)

```
int j, i=1, a, b = 7, c = 3;
j = i++ ; /* équivalent à j=i; i=i+1; */
j = ++i ; /* équivalent à i=i+1; j=i; */
a = ++b - c--;
```

Quelle est la valeur de a ?

L'incrémentation et la décrémentation (2/2)

```
int j, i=1, a, b = 7, c = 3;
j = i++ ; /* équivalent à j=i; i=i+1; */
j = ++i ; /* équivalent à i=i+1; j=i; */
a = ++b - c--;
```

Quelle est la valeur de a ? 5

L'affectation généralisée/élargie

Le C permet de remplacer $i = i + k$ par $i \mathrel{+}= k$, $i = i * k$ par $i \mathrel{*}= k$... Cette possibilité concerne seulement les opérateurs binaires arithmétiques et de manipulation de bits.

L'opérateur de cast

Forcer la conversion d'une expression quelconque dans le type de son choix, à l'aide d'un opérateur nommé en anglais *cast*.

Syntaxe : (type désiré) *expression simple* ou (type désiré)(*expression composée*). La priorité du cast est moindre que les opérateurs unaires, mais supérieure aux opérateurs binaires.

```
int n = 4, p = 5;
(double) ( n/p ); /* le cast s'applique au résultat
    de n/p soit 0 */
(double) n/p; /* le cast s'applique à n */
printf("%f\n",(float)n);
```

L'opérateur séquentiel ,

L'opérateur séquentiel , élargit la notion d'expression, en permettant d'exprimer *plusieurs calculs successifs* au sein d'une même expression. Les expressions successives sont évaluées de la gauche vers la droite et la valeur de cette expression "élargie" est celle de la dernière expression évaluée (la plus à droite).

```
a * b, i+j, 5-3 ;
a = (i++, j++*i) ;
a = 4, b=8 ;
for(i=0, j=0 ; ... ; ...) ...
for(i=0, j=0, printf("on commence le for") ; ... ;
    ...) ...
```

L'opérateur conditionnel ? :

L'opérateur conditionnel **? :** est le seul opérateur ternaire du C.
Syntaxe : <expression> **?**<expression_vrai> **:**<expression_faux>
Si *expression* est vrai, alors l'expression ou l'instruction-expression *expression_vrai* est exécutée, sinon c'est *expression_faux*.

```
max = a>b ? a : b ;
```