

Initiation à la programmation

Université Bordeaux 1

Année 2006-2007, Licence semestre 2



Plan du cours

- 1 Informations pratiques
- 2 Langage C : présentation
- 3 Démarrage rapide en C
- 4 Constructions fréquentes
- 5 Types structurés 1 : tableaux
- 6 Les variables
- 7 Notion d'expression
- 8 Les fonctions
- 9 Les pointeurs
- 10 Types structurés 2 : structures
- 11 Compilation et modularité
- 12 Compléments : instructions

1– Informations pratiques

- ▶ Organisation de l'UE
- ▶ Modalités de contrôle
- ▶ Bibliographie
- ▶ Ressources internet
- ▶ Objectifs du cours

Organisation de l'UE

▶ Cours : 12 séances 1h20

- ▶ **Math** Giuliana Bianchi bianchi@labri.fr
- ▶ **Info** Marc Zeitoun mz@labri.fr

▶ TD-TP : 12 séances 2h40

- | | | | |
|-------------|---------------------------|-------------|---------------------|
| Info | ▶ Frédérique Carrère | Math | ▶ Gérard Giroux |
| | ▶ Marie-Christine Counilh | | ▶ Catherine Pannier |
| | ▶ Fabrice Dècle | | |
| | ▶ Stefka Gueorguieva | | |
| | ▶ Jean-Claude Ville | | |

▶ Tutorat : 12h-14h.

▶ Web : <http://dept-info.labri.fr/ENSEIGNEMENT/InitProg/>

▶ 1ère session

- ▶ CC
- ▶ Examen 1h30
- ▶ Note finale $\max(\text{Ex}, 3/4\text{Ex} + 1/4\text{CC})$

▶ 2ème session

- ▶ Examen 2 1h30
- ▶ Note finale $\max(\text{Ex2}, 3/4\text{Ex2} + 1/4\text{CC})$

- ▶ Présentation claire et complète du langage C



A. Braquelaire.

Méthodologie de la programmation en langage C.

Norme C99, API POSIX. Dunod, Paris, 4^{ème} éd. 2005.

- ▶ Plus denses, moins agréables mais de référence



B. W. Kernighan et D. M. Ritchie.

Le langage C, norme ANSI.

Masson, Paris, 1990.



S. P. Harbison et G. L. Steele Jr.

C : a reference manual (3rd ed.).

Prentice-Hall, Upper Saddle River, NJ, USA, 1991.

- ▶ Nombreuses introductions : <http://mapage.noos.fr/emdel/>
- ▶ FAQ (Foire Aux Questions)
 - ▶ <http://c-faq.com/>
 - ▶ <http://www.faqs.org/faqs/fr/comp/lang/> (FR, 2003).
- ▶ Bibliothèque standard
 - ▶ Pages de manuel section 3. Par exemple **man 3 printf**.
 - ▶ <http://ccs.ucsd.edu/c/>
 - ▶ <http://www.utas.edu.au/infosys/info/documentation/C/>
- ▶ Normes
 - ▶ ANSI <http://www.lysator.liu.se/c/rat/title.html>
 - ▶ Corrections tc1 et tc2 à la norme ANSI :
<ftp://dkuug.dk/JTC1/SC22/WG14/www/docs/tc1.htm>
 - ▶ C99 <http://wwwold.dkuug.dk/jtc1/sc22/open/n2794/>



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.



Objectifs du cours

- ▶ Introduction à la programmation impérative en langage C.
- ▶ Pouvoir comprendre ou écrire un programme C
 - ▶ **Correct** Utilisation du compilateur, du debugger.
 - ▶ **Sûr** Pas de failles de sécurité.
 - ▶ **Portable** Exploitable sur plusieurs machines.
 - ▶ **Modulaire** Découpage logique en fonctions simples
⇒ facilement modifiable et extensible.
 - ▶ **Efficace** Algorithmes, implémentation rapides.
 - ▶ **Lisible** Commentaires, indentation, noms bien choisis.

2– Langage C : présentation

- ▶ Caractéristiques du langage C
- ▶ Bref historique : de BPCL à C99
- ▶ Pourquoi des normes ?

Caractéristiques du langage C

- ▶ **Portable** Un programme C **conforme aux normes** est utilisable sur une grande variété de machines.
Normes : ANSI, **C-99**, **API POSIX**,...
- ▶ **Puissant** Nombreux domaines d'application
OS (Unix), réseau, BD, graphique...
- ▶ **Efficace** Permet de développer des programmes rapides.
- ▶ **Haut niveau** Détails hardware cachés au programmeur.
- ▶ **Souple** Très permissif, accès à la mémoire.
⇒ facile à aborder, **difficile** à bien maîtriser.

Caractéristiques du langage C

- ▶ **Portable** Un programme C **conforme aux normes** est utilisable sur une grande variété de machines.
Normes : ANSI, C-99, API POSIX,...
- ▶ **Puissant** Nombreux domaines d'application
OS (Unix), réseau, BD, graphique...
- ▶ **Efficace** Permet de développer des programmes rapides.
- ▶ **Haut niveau** Détails hardware cachés au programmeur.
- ▶ **Souple** Très permissif, accès à la mémoire.
⇒ facile à aborder, **difficile** à bien maîtriser.

Caractéristiques du langage C

- ▶ **Portable** Un programme C **conforme aux normes** est utilisable sur une grande variété de machines.
Normes : ANSI, **C-99**, **API POSIX**,...
- ▶ **Puissant** Nombreux domaines d'application
OS (Unix), réseau, BD, graphique...
- ▶ **Efficace** Permet de développer des programmes rapides.
- ▶ **Haut niveau** Détails hardware cachés au programmeur.
- ▶ **Souple** Très permissif, accès à la mémoire.
⇒ facile à aborder, **difficile** à bien maîtriser.

Caractéristiques du langage C

- ▶ **Portable** Un programme C **conforme aux normes** est utilisable sur une grande variété de machines.
Normes : ANSI, **C-99**, **API POSIX**,...
- ▶ **Puissant** Nombreux domaines d'application
OS (Unix), réseau, BD, graphique...
- ▶ **Efficace** Permet de développer des programmes rapides.
- ▶ **Haut niveau** Détails hardware cachés au programmeur.
- ▶ **Souple** Très permissif, accès à la mémoire.
⇒ facile à aborder, **difficile** à bien maîtriser.

Caractéristiques du langage C

- ▶ **Portable** Un programme C **conforme aux normes** est utilisable sur une grande variété de machines.
Normes : ANSI, **C-99**, **API POSIX**,...
- ▶ **Puissant** Nombreux domaines d'application
OS (Unix), réseau, BD, graphique...
- ▶ **Efficace** Permet de développer des programmes rapides.
- ▶ **Haut niveau** Détails hardware cachés au programmeur.
- ▶ **Souple** Très permissif, accès à la mémoire.
⇒ facile à aborder, **difficile** à bien maîtriser.



Bref historique : de BPCL à C99

- ▶ 1967 Langage BPCL (Richards, Bell Labs)
- ▶ 1970 Langage B (Thompson, BL)
*Pas de types, manipulation de mots machine
⇒ programmes non portables
Structures de contrôle, pointeurs, récursivité.*
- ▶ 1972 1er compilateur C (Kernighan, Ritchie, BL)
- ▶ 1978 1ère spécification publique du C
- ▶ 1989 Norme « ANSI » ⇒ C « ISO » ou « ANSI »
- ▶ 1999 Norme ISO/IEC 9899 : C99.



Pourquoi des normes ?

- ▶ Plusieurs vendeurs ont développé leur propre version du langage C.
- ▶ Ils ont proposé des extensions **incompatibles** entre elles.
- ▶ Norme : fournit une **signification précise et unique** des programmes.
- ▶ Un programme qui respecte la norme **C99** est **portable** : on peut le déployer sur n'importe quelle architecture actuelle.
- ▶ Les programmes ne respectant pas la norme **C99** seront (ici) considérés comme incorrects.

3– Démarrage rapide en C

- ▶ Du source à l'exécutable
- ▶ Variables
- ▶ Tests
- ▶ Boucles
- ▶ Fonctions
- ▶ Bonnes habitudes de programmation



1er exemple : Hello, world !

```
/* 1er programme source */
```

```
#include <stdio.h> /* pour la declaration de printf */
```

```
int main (void)
{
    printf("Hello world!\n");
}
```



Premières remarques

- ▶ Un programme C contient des définitions de fonctions.
 - ▶ L'une d'elles s'appelle `main`.
- ▶ Les instructions sont exécutées **séquentiellement** à partir de la 1ère instruction de `main`.
- ▶ Le `;` (point-virgule) est un terminateur d'instruction.
- ▶ Les commentaires sont placés entre `/*` et `*/`.
- ▶ Le programmeur indique les groupes d'instructions avec des accolades `{...}` (l'indentation aide juste à la lisibilité).
- ▶ Un programme doit être traduit (compilé) avant d'être exécuté.

Compiler pour créer l'exécutable

- ▶ Contrairement à python, C est un langage **compilé**.
- ▶ Cela signifie qu'un programme C ne peut pas être exécuté tel quel.
- ▶ Il est nécessaire de le traduire en langage machine.
- ▶ La phase de traduction

du langage C
vers
le langage machine

s'appelle la **compilation**.

- ▶ La traduction est faite par un logiciel : le compilateur

Programme source C

Compilateur

Fichier exécutable

Compiler pour créer l'exécutable

- ▶ Le programmeur écrit un programme source C.
- ▶ Il le traduit ensuite pour obtenir un exécutable :

Programme source C

Compilateur

Fichier exécutable

- ▶ Une compilation réussie crée un fichier : l'**exécutable** (l'application).
- ▶ Il dépend de la machine sur laquelle on veut exécuter le programme.
- ▶ \implies on doit compiler une fois par architecture d'utilisation.
- ▶ Si le compilateur rencontre une erreur,
 - ▶ Il continue pour rechercher d'autres erreurs.
 - ▶ Il ne **crée pas** l'exécutable.



Compiler pour créer l'exécutable

- ▶ Avant d'exécuter le programme, on doit le traduire pour créer un fichier exécutable. Cette traduction s'appelle la **compilation**.
- ▶ On utilise le **compilateur** (= logiciel de traduction) **gcc**.
- ▶ `$ emacs HelloWorld.c`
- ▶ `$ gcc -Wall -std=c99 -o HelloWorld HelloWorld.c`
compile le fichier HelloWorld.c dans lequel se trouve le source, et crée l'exécutable HelloWorld.



Compiler pour créer l'exécutable

- ▶ Avant d'exécuter le programme, on doit le traduire pour créer un fichier exécutable. Cette traduction s'appelle la **compilation**.
- ▶ On utilise le **compilateur** (= logiciel de traduction) **gcc**.
- ▶ `$ emacs HelloWorld.c`
- ▶ `$ gcc -Wall -std=c99 -o HelloWorld HelloWorld.c`
compile le fichier HelloWorld.c dans lequel se trouve le source, et crée l'exécutable HelloWorld.
 - ▶ **-Wall** : gcc donne les principaux avertissements.



Compiler pour créer l'exécutable

- ▶ Avant d'exécuter le programme, on doit le traduire pour créer un fichier exécutable. Cette traduction s'appelle la **compilation**.
- ▶ On utilise le **compilateur** (= logiciel de traduction) **gcc**.
- ▶ `$ emacs HelloWorld.c`
- ▶ `$ gcc -Wall -std=c99 -o HelloWorld HelloWorld.c`
compile le fichier HelloWorld.c dans lequel se trouve le source, et crée l'exécutable HelloWorld.
 - ▶ `-Wall` : gcc donne les principaux avertissements.
 - ▶ `-std=c99` : Conformité à la norme C99.



Compiler pour créer l'exécutable

- ▶ Avant d'exécuter le programme, on doit le traduire pour créer un fichier exécutable. Cette traduction s'appelle la **compilation**.
- ▶ On utilise le **compilateur** (= logiciel de traduction) **gcc**.
- ▶ `$ emacs HelloWorld.c`
- ▶ `$ gcc -Wall -std=c99 -o HelloWorld HelloWorld.c`
compile le fichier HelloWorld.c dans lequel se trouve le source, et crée l'exécutable HelloWorld.
 - ▶ `-Wall` : gcc donne les principaux avertissements.
 - ▶ `-std=c99` : Conformité à la norme C99.
 - ▶ `-o` : Permet de nommer l'exécutable.



Compiler pour créer l'exécutable

- ▶ Avant d'exécuter le programme, on doit le traduire pour créer un fichier exécutable. Cette traduction s'appelle la **compilation**.
- ▶ On utilise le **compilateur** (= logiciel de traduction) **gcc**.
- ▶ `$ emacs HelloWorld.c`
- ▶ `$ gcc -Wall -std=c99 -o HelloWorld HelloWorld.c`
compile le fichier HelloWorld.c dans lequel se trouve le source, et crée l'exécutable HelloWorld.
 - ▶ `-Wall` : gcc donne les principaux avertissements.
 - ▶ `-std=c99` : Conformité à la norme C99.
 - ▶ `-o` : Permet de nommer l'exécutable.
- ▶ `$./HelloWorld` exécute le fichier compilé.
- ▶ L'utilisation de l'exécutable ne nécessite plus le source.



Options importantes de gcc

- ▶ `-o` permet de nommer le fichier exécutable (`a.out` par défaut).
- ▶ `-Wall` indique des avertissements sur le code.
- ▶ `-Werror` produit une erreur à la place d'un avertissement.
- ▶ `-std=c99` se conforme à la norme C99.
- ▶ `-pedantic` avertit en cas de programme sortant de la norme.

Ces options aident le programmeur à détecter les erreurs.

On compilera **systématiquement** avec

```
-std=c99 -Wall -Werror -pedantic.
```

Autre option utile :

- ▶ `-g` permet d'utiliser ultérieurement le debugger.

- ▶ Contrairement à Python, on ne peut pas utiliser les variables avant de les avoir **déclarées**.
- ▶ Une déclaration permet entre autres de donner le **type** des variables.
- ▶ On utilisera au début des variables entières, de type **int**.
- ▶ Exemples de **déclaration** :
 - `int x;` déclaration d'une variable x de type **entier**.
 - `int x, y;` déclaration de deux variables x et y de type **entier**.

Exemple : variables

```
#include <stdio.h> /* pour la declaration de printf() */  
#include "inf101.h" /* ... et de lire_entier() */
```

```
int main (void)  
{  
    int x;  
  
    printf("Entrez un entier : ");  
    x = lire_entier();  
    printf("La variable x contient %d\n",x);  
}
```

- ▶ Ce programme suppose qu'on a une fonction `lire_entier()` écrite ailleurs permettant de lire un entier au clavier.



Variables : deuxième exemple

```
#include <stdio.h>
#include "inf101.h"

int main (void)
{
    int x;

    printf("Entrez un entier : ");
    x = lire_entier();
    printf("%d * %d = %d\n", x, x, x*x);
}
```



La fonction d'affichage printf

```
printf("Hello world !\n");
```

```
printf("Entrez un entier : ");
```

```
printf("%d\n",x);
```

```
printf("La variable x contient %d\n",x);
```

```
printf("%d * %d = %d\n", x, x, x*x);
```



Exemple : tests

```
#include <stdio.h>
#include "inf101.h"
int main (void) /* affiche la valeur absolue d'un entier */
{
    int x, y;

    printf("Entrez un entier : ");
    x = lire_entier();
    y = x;
    if (x < 0)
        y = -x;
    printf("|%d| = %d\n", x, y);
}
```

- ▶ Comme en python, il existe aussi une forme if...else.

Exemple : tests again

```
#include <stdio.h>
#include "inf101.h"
int main (void) /* affiche le maximum entre deux entiers */
{
    int x, y;

    printf("Entrez un entier : ");
    x = lire_entier();
    printf("Entrez un autre entier : ");
    y = lire_entier();

    if (x < y)
        printf("Le maximum est %d\n", y);
    else
        printf("Le maximum est %d\n", x);
}
```

Exemple : boucles

```
#include <stdio.h>
#include "inf101.h"
int main(void) /* affiche la somme des entiers de 1 a n */
{
    int n, s = 0;

    printf("Entrez un entier : ");
    n = lire_entier();

    for (int i = 1; i <= n; i = i + 1)
        s = s + i;

    printf("La somme des entiers de 1 a %d vaut %d\n", n , s);
}
```

- ▶ Il existe d'autres mécanismes pour les boucles (par ex. `while`).

Exemple : fonctions


```
/* 1er programme source, 2eme version */

#include <stdio.h>
void imprimer(void) /* DÉFINITION de imprimer() */
{
    printf("Hello world !\n");
}

int main(void)
{
    imprimer();          /* APPEL de imprimer() */
}
```



Ne pas confondre **définition** et **appel** (= **utilisation**) d'une fonction.



Exemple 2 : fonctions

```
int somme(int n) /* retourne la somme des entiers de 1 a n */
{
    int s = 0;

    for (int i = 1; i <= n; i = i + 1)
        s = s + i;

    return s;
}
```



Exemple 2 : fonctions, suite

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

```
#include "inf101.h"
```

```
/* Insérer ici la définition de la fonction somme */
```

```
int main (void)
```

```
{
```

```
    int n;
```

```
    printf("Entrez un entier : ");
```

```
    n = lire_entier();
```

```
    printf("La somme des entiers de 1 a %d vaut %d\n", n, somme(n));
```

```
}
```



Bonnes habitudes de programmation

- ▶ La compilation doit se faire **sans erreur ni avertissement (warning)**.
- ▶ Une mauvaise indentation n'est pas sanctionnée (\neq python).
Les programmes doivent cependant être **correctement indentés**.
- ▶ Les noms des fonctions, variables, etc. doivent être
 - ▶ **lisibles**,
 - ▶ **pertinents**,
 - ▶ **cohérents** (conventions **uniformes**).
- ▶ Les programmes doivent être **commentés**.

4– Constructions fréquentes

- ▶ Premiers types de base
- ▶ Les types `int`, `double`, `char`
- ▶ Exemples
- ▶ Le type `bool`
- ▶ Quelques opérateurs arithmétiques
- ▶ Quelques opérateurs logiques
- ▶ Affectation
- ▶ Tests : `if` et `if...else`
- ▶ Boucle `for`
- ▶ Boucle `while`



Premiers types de base

En C toute variable a un type.

Le **type** détermine :

- ▶ l'ensemble des **valeurs** qu'une variable peut prendre,
- ▶ la façon dont ces valeurs sont stockées en mémoire, *(souvent dépendant de l'implémentation)*.
- ▶ les **opérations** applicables à la variable.

Quelques types prédéfinis de base :

- ▶ **int** : entiers
- ▶ **double** : flottants (nombres avec partie décimale)
- ▶ **char** : caractères
- ▶ **bool** : Booléens

Il y a d'autres types entiers et d'autres types flottants.



Les types `int`, `double`, `char`

▶ `int` :

- ▶ La taille de l'intervalle d'entiers représenté dépend de la machine et tend à augmenter (un nombre de type `int` est souvent codé, actuellement, sur 4 octets).
- ▶ ⇒ ne pas faire d'hypothèse sur la taille d'un `int`.

▶ `double` : pour représenter les nombres flottants en double précision

- ▶ la précision dépend de la place utilisée pour coder la partie décimale

▶ `char` : permet de représenter le jeu de caractères de base.

- ▶ Constantes entre *quotes* : `'X'`. Ne pas confondre `'0'` et `0`.



Type double : un exemple

```
#include <stdio.h>
#include "inf101.h" /* pour la declaration de lire_flottant() */

int main (void) /* calcul de la surface du cercle */
{
    /* de rayon r */
    double r;
    double pi = 22.0 / 7; /* approximation de pi */

    printf("Entrez un nombre: ");
    r = lire_flottant();

    printf("Rayon = %g, Surface = %g\n", r, r * r * pi );
}
```



Type char : un exemple

```
#include <stdio.h>
#include "inf101.h" /* pour la declaration de lire_caractere() */

int main (void)
/* on affiche le caractere suivant dans le code ASCII */
{
    char x;

    printf("Entrez un caractere: ");
    x = lire_caractere();

    printf("%c est le caractere qui suit %c\n", x+1, x);
}
```




Le type bool

Le type Booléen (`bool`)

- ▶ n'a que 2 valeurs possibles : `true` et `false`,
- ▶ on l'utilise pour exprimer les résultats d'une expression logique,
- ▶ a été introduit dans la norme C99,
- ▶ il faut ajouter la directive :

```
#include <stdbool.h>
```

aux fichiers sources dans lesquels on l'utilise.

- ▶ en mémoire `true` est codé par l'entier 1, `false` est codé par l'entier 0.
- ▶ Dans une condition, toute expression
 - ▶ `non nulle` est considérée comme `vraie` ;
 - ▶ `nulle` est considérée comme `fausse`.

Quelques opérateurs arithmétiques

Dans l'ordre de priorité décroissante :

- ▶ + - unaires signe
- ▶ * / % opérateurs arithmétiques
- ▶ + - opérateurs arithmétiques
- ▶ = affectation

Donc $x + y * 2$ correspond à $x + (y * 2)$



La division / n'a pas le même sens sur entiers et flottants.

Quelques opérateurs logiques

Dans l'ordre de priorité décroissante :

- ▶ ! NON logique
- ▶ < > <= >= comparaisons (calcule un Booléen)
- ▶ == != comparaisons (calcule un Booléen)
- ▶ && ET logique
- ▶ || OU logique.

Donc $x < y == z > t$ correspond à $(x < y) == (z > t)$

En C, l'évaluation des opérateurs && et || est « **paresseuse** ».

- ▶ Permet de mémoriser une valeur.
- ▶ Affectation de variable :

`<nom_variable> = <expression>`

- ▶ Plus généralement :

`<l_valeur> = <expression>`

- ▶ `l_valeur` : expression qui a une adresse. Typiquement : variable.
- ▶ L'affectation
 - ▶ évalue (calcule) l'expression,
 - ▶ puis, range la valeur calculée à l'adresse de la `l_valeur`.
 - ▶ est elle même une expression qui s'évalue comme `<expression>`.

Affectation : exemples

```
int a, b, c;
```

```
a = 3;
```

```
b = 4;
```

```
c = 5;
```

```
c = a - 1;    /* c vaut 2 maintenant */
```

```
b = b + 1;    /* b vaut 5 maintenant */
```

```
a = a + b + c; /* a vaut 3+5+2, soit 10 maintenant */
```

```
a + b = 2;    /* INTERDIT ! */
```

```
/*
```

```
 * car a + b n'a pas d'adresse,
```

```
 * contrairement a a, b, ou c.
```

```
*/
```

- ▶ Deux instructions similaires : `if` et `if...else`.

```
if (<expression>
    <instruction ou bloc>vrai
```

```
if (<expression>
    <instruction ou bloc>vrai
else
    <instruction ou bloc>faux
```

- ▶ Permettent d'exécuter ou non des `instructions`, suivant la validité de la condition indiquée par l'`expression`.
- ▶ L'expression Booléenne peut utiliser les opérateurs
 - ▶ `&&` (ET logique)
 - ▶ `||` (OU logique).
 - ▶ `!` (NÉGATION logique).

Instruction if

```
if (<expression>
    <instruction ou bloc>vrai
```

- ▶ L'expression est évaluée. Si elle est
 - ▶ **non nulle**, le test réussit et `<instruction ou bloc>vrai` est exécuté.
 - ▶ **nulle**, le test échoue. On passe alors à l'instruction qui suit `<instruction ou bloc>vrai`, qu'on n'exécute pas.
- ▶ Si on veut exécuter **plusieurs** instructions en cas de test réussi, on doit les mettre **dans un bloc**, entre `{...}`.
- ▶ S'il n'y a qu'une instruction, les accolades ne sont pas nécessaires.



Instruction if : exemple

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

```
int maximum(int a, int b)
```

```
{
```

```
    int res;
```

```
    res = a;
```

```
    if (b > a)
```

```
        res = b;
```

```
    return res;
```

```
}
```




Instruction if : exemple

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

```
int maximum(int a, int b)
```

```
{  
    int res;  
  
    res = a;  
    if (b > a)  
    {  
        res = b;  
        printf("Le maximum de %d et %d est %d\n", a, b, res);  
    }  
    return res;  
}
```

Instruction if...else

```
if (<expression>)  
    <instruction ou bloc>vrai  
else  
    <instruction ou bloc>faux
```

- ▶ L'expression est évaluée. Si elle est
 - ▶ non nulle, le test réussit et <instruction ou bloc>_{vrai} est exécuté.
 - ▶ nulle, le test échoue et <instruction ou bloc>_{faux} est exécuté.
- ▶ Si on veut exécuter plusieurs instructions sous le **if** ou le **else**, on doit les mettre dans un bloc, entre {...}.
- ▶ Sinon, les accolades ne sont pas nécessaires.



Instruction `if...else` : exemple

```
#include <stdio.h>

void
est_divisible_par(int n, int i)
{
    if (n % i == 0)
        printf("%d est divisible par %d\n", n, i);

    else
        printf("%d n'est pas divisible par %d\n", n, i);
}
```

Quelle précaution prendre pour utiliser cette fonction ?



Boucle for

```
for (<expr1>;<expr2>;<expr3>)  
    <instruction ou bloc>
```

- ▶ <expr1> est une expression d'initialisation exécutée une seule fois (au début de l'exécution de la boucle.
- ▶ <expr2> est un test (d'arrêt) exécuté avant chaque itération.
- ▶ <expr3> est une expression, modifiant au moins une variable de <expr2>, exécutée à la fin de chaque itération.

<expr1>, <expr2> ou <expr3> peuvent être vides.

```
for ( ; ; ) /* boucle infinie ! */  
    ;
```



Boucle for : exemple

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

```
void
```

```
afficher_somme_impairs(int n, int m)
```

```
/* affiche la somme des nombres impairs entre n et m inclus */
```

```
{
```

```
    int res = 0; /* initialisation */
```

```
    for (int i = n + (1-n%2); i <= m; i = i + 2)
```

```
        res = res + i;
```

```
    printf("Somme des impairs entre %d et %d: %d\n", n, m, res);
```

```
}
```

Boucle for : exemple 2

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

```
void
```

```
afficher_majuscules (void)
```

```
/* affiche les 26 caractères consécutifs à partir de 'A' */
```

```
{
```

```
    for (char x = 'A' ; x <= 'Z'; x = x+1)
```

```
        printf("%c ", x);
```

```
    printf("\n");
```

```
}
```

Boucle for : exemple 3

```
#include <stdio.h>
#include "inf101.h"

int somme_suite_entiers (int n)  /* calcule la somme de n entiers */
{
    /* rentres au clavier */
    int k;
    int som = 0;

    for (int i = n ; i > 0 ; i = i-1)
    {
        printf("Entrez un entier: ");
        k = lire_entier();
        som = som + k;
    }
    return som;
}
```



Boucle while

```
while (<expression>)  
    <instruction ou bloc>
```

1. L'expression est évaluée.
2. Si elle est fausse, on sort du **while**, on passe à l'instruction suivante.
3. Si elle est vraie,
 - ▶ **<instruction ou bloc>** est exécuté,
 - ▶ puis on revient en 1.

Exemple

```
while (true) /* boucle infinie ! */  
    ;
```




Boucle while : exemple

```
#include <stdio.h>

void
afficher_puissances(int k, int nmax)
{
    int i = 1;    /* initialisation */

    while (i <= nmax)
    {
        printf("%d\n", i);
        i = i * k;
    }
}
```

Boucle while : exemple 2

```
bool est_premier(int n)
{
    int i = 2;
    while (i*i <= n)
    {
        if (n % i == 0)
        {
            printf("%d est divisible par %d\n", n, i);
            return false;
        }
        i++;
    }
    printf("%d est premier\n", n);
    return true;
}
```

5– Types structurés 1 : tableaux

- ▶ Tableaux
- ▶ Chaînes de caractères : principes
- ▶ Arguments de la fonction `main`

- ▶ Ils permettent de mémoriser plusieurs **éléments du même type**.

`<type> <identificateur> [n1][n2]. . . . [nk]`

- ▶ **C90** : n_1, n_2, \dots, n_k sont des expressions constantes.
 - ▶ **C99** : n_1, n_2, \dots, n_k peuvent être des expressions ou `*`.
 - ▶ Le tableau a k dimensions.
 - ▶ Les indices de la j -ème dimension varient de 0 à $n_j - 1$.
- ▶ Exemples

```
char T[10]; // tableau T de 10 caracteres.
```

```
int mat[5][3]; // "matrice" d'entiers mat, 5 lignes, 3 colonnes.
```

```
T[0] = 'a'; // affectation de la 1ere case de T.
```

```
mat[4][2] = 12; // affectation de la "derniere" case de mat.
```



Tableaux : un exemple

- ▶ Remplir le tableau T pour que T[i] contienne $i * i$

```
int T[10];
```

```
for(int i = 0; i < 10; i++)  
    T[i] = i * i;
```

- ▶ Remplir la matrice 5×3 pour que mat[i][j] contienne $i + j$

```
int mat[5][3];
```

```
for(int i = 0; i < 5; i++)  
    for(int j = 0; j < 3; j++)  
        mat[i][j] = i + j;
```



Tableaux : premières utilisations

- ▶ Saisir un tableau d'entiers.
- ▶ Afficher un tableau d'entiers.
- ▶ Rechercher si une valeur est dans un tableau d'entiers.
- ▶ Rechercher le premier indice d'un tableau d'entiers contenant la valeur maximale.
- ▶ Échanger le contenu de la première case contenant la valeur maximale avec le contenu de la dernière case.
- ▶ Trier un tableau (tri par sélection).
- ▶ Trier un tableau (tri à bulles).

Fonctions écrites dans les fichiers suivants (cliquer sur les liens) :

- ▶ [TrisTableau.c](#),
- ▶ [TrisTableau.h](#).

Chaînes de caractères : principes

- ▶ Les chaînes sont mémorisées dans des tableaux de caractères terminés par `'\0'`.

- ▶ Par exemple la chaîne "toto" est mémorisée comme :

't'	'o'	't'	'o'	'\0'
-----	-----	-----	-----	------

- ▶ Un type possible pour représenter les chaînes est `char []` (tableau de caractères).
- ▶ On rencontre aussi `char *`, dont on verra la signification plus tard.
- ▶ Plusieurs fonctions sont prédéfinies pour les chaînes.
- ▶ Ex. comparaison : `strcmp`, longueur `strlen`. (inclure `string.h`).



Chaînes : exemples

- ▶ Calculer la longueur d'une chaîne (reprogrammer `strlen`).
- ▶ Comparer 2 chaînes.
- ▶ Copie de chaînes.
- ▶ Passage d'une chaîne alphabétique en minuscules.

- ▶ La bibliothèque prédéfinit plusieurs fonctions de manipulation de chaînes. Parmi elles
 - ▶ `strlen` : longueur d'une chaîne.
 - ▶ `strcmp`, `strncmp` : comparaison de chaînes.
 - ▶ `strncat` : concaténation de chaînes.
 - ▶ `strcpy`, `strncpy` : copie de chaînes.



Ne pas confondre caractères et chaînes de caractères.

```
char c = 'A' ;
```

```
char *s = "A" ;
```

- ▶ Des fonctions de manipulation de caractères sont aussi utiles. Voir les pages de manuel de `isalpha` et `tolower`.

- ▶ Pour permettre le passage d'arguments à la fonction `main`, on la définit de la façon suivante :

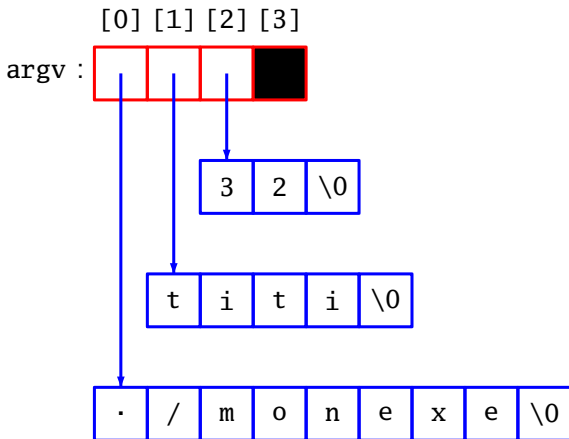
```
int main (int argc, char *argv[])
{
    .....
}
```

`argc` initialisé par le nombre d'arguments avec lesquels l'exécutable correspondant est lancé.

`argv` tableau de chaînes de caractères initialisé par les arguments avec lesquels l'exécutable correspondant est lancé.

Arguments de main

Au lancement de la commande shell `./monexe titi 32`





Ne pas confondre

- ▶ les arguments avec lesquels l'utilisateur appelle le programme,
- ▶ les arguments de main.

▶ Formellement, si `main` est définie par

```
int main (int nb_a, char *a[]){...}
```

`main` a deux arguments :

- ▶ `nb_a` (entier) et
 - ▶ `a` (tableau de chaînes de caractères).
- ▶ Le nom du fichier exécutable par lequel est lancée la commande est considéré comme le premier argument.
- ▶ `argv[0]` contient donc ainsi le nom de l'exécutable lancé.

6– Les variables

- ▶ Qu'est-ce qu'une variable ?
- ▶ Identificateurs
- ▶ Portée
- ▶ Masquage
- ▶ Définitions et déclarations
- ▶ Représentation de l'information
- ▶ Classes d'allocation
- ▶ Initialisation
- ▶ Types de base
- ▶ Types de base : tailles, valeurs


Qu'est-ce qu'une variable ?

- ▶ Une variable permet de mémoriser des valeurs.
- ▶ Un emplacement en mémoire (adresse) lui est attribué.
- ▶ Une définition de variable définit :
 - ▶ son nom : un « identificateur ».
 - ▶ sa portée : les portions de code où on peut l'utiliser.
 - ▶ son type : les valeurs qu'elle peut prendre, les opérations possibles.
 - ▶ sa classe d'allocation : indique la zone mémoire où elle est stockée.
 - ▶ sa valeur initiale, éventuellement.
- ▶ Certaines de ces caractéristiques peuvent être définies, en partie ou complètement, par la place de la définition dans le source C.
- ▶ Exemple de définition :

```
static int x = 1234;
```

Le **nom** d'une variable est un **identificateur**.

- ▶ commence par une lettre a, \dots, z, A, \dots, Z ou $_$.
- ▶ peut contenir les caractères $a, \dots, z, A, \dots, Z, 0, \dots, 9, _$.
- ▶ jusqu'à 31 caractères significatifs.
- ▶ les compilateurs actuels différencient majuscules e minuscules.
- ▶ La norme précise des limitations (rarement suivies).

 Ne pas utiliser $_$ en début d'identificateur.

- ▶ **Portée** d'une variable = partie du programme où on peut l'utiliser.
- ▶ Une variable **définie hors** du corps de toute fonction est **globale**.
- ▶ Une variable **définie dans** le corps d'une fonction est **locale**.
- ▶ On ne peut **utiliser** une variable que dans le corps d'une fonction.
- ▶ On peut utiliser une variable globale
 - ▶ dans toute fonction du fichier après sa définition, et
 - ▶ dans un autre fichier en la **déclarant** avec le mot réservé **extern**.
- ▶ Une déclaration annonce au compilateur qu'une variable (globale) est définie dans un autre fichier, en donnant son type.
- ▶ Exemple de déclaration :

```
extern int x;
```


- ▶ Une variable définie dans le corps d'une fonction est dite **locale**.
- ▶ Le corps d'une fonction contient des blocs emboîtés les uns dans les autres, délimités par **{** et **}**.
- ▶ Chaque bloc contient ses définitions de variables, suivies d'instructions et de sous-blocs.
- ▶ La portée d'une variable locale est **limitée au bloc** dans laquelle elle est définie, et à ses sous-blocs.
- ▶ La **position** des définitions de variables dans le programme influence donc sur leur **portée**.
- ▶ En C99, on peut définir des variables de boucles.



Portée : exemple

```
int x;
void f(int a)
{
    int y;
    /* ..... */
    {
        int z;
        /* ..... */
    }
    /* ..... */
}

int t;
void g(void)
{
    /* ..... */
}
```

Portée : exemple

```
int x;                                /* Portee de x */
void f(int a)
{
    int y;
    /* ..... */
    {
        int z;
        /* ..... */
    }
    /* ..... */
}
```

```
int t;
void g(void)
{
    /* ..... */
}
```

Portée : exemple

```
int x;  
void f(int a)           /* Portee de a */  
{  
    int y;  
    /* ..... */  
    {  
        int z;  
        /* ..... */  
    }  
    /* ..... */  
}
```

```
int t;  
void g(void)  
{  
    /* ..... */  
}
```

Portée : exemple

```
int x;  
void f(int a)  
{  
    int y;                /* Portee de y */  
    /* ..... */  
    {  
        int z;  
        /* ..... */  
    }  
    /* ..... */  
}
```

```
int t;  
void g(void)  
{  
    /* ..... */  
}
```

Portée : exemple

```
int x;
void f(int a)
{
    int y;
    /* ..... */
    {
        int z;           /* Portee de z */
        /* ..... */
    }
    /* ..... */
}

int t;
void g(void)
{
    /* ..... */
}
```

Portée : exemple

```
int x;
void f(int a)
{
    int y;
    /* ..... */
    {
        int z;
        /* ..... */
    }
    /* ..... */
}

int t;                               /* Portee de t */
void g(void)
{
    /* ..... */
}
```

- ▶ On dit qu'une variable est **visible** dans les blocs de sa portée.
- ▶ Si 2 variables de même nom sont visibles dans le même bloc, le nom fait référence à la variable définie dans le bloc le plus interne.
- ▶ En cas de conflit de nom, la variable définie dans le bloc le plus interne **masque** l'autre.

Masquage : exemple

```
#include <stdio.h>
int x = 7;
int main(void)
{
    printf("x = %d\n", x);           // affiche 7
    {
        int x = 10;
        printf("x = %d\n", x);     // affiche 10
        {
            int x = 3;
            printf("x = %d\n", x); // affiche 3
        }
        printf("x = %d\n", x);     // affiche 10
    }
    printf("x = %d\n", x);         // affiche 7
}
```

- ▶ Une variable doit être définie ou déclarée avant d'être utilisée.
- ▶ la déclaration associe un type avec un nom de variable.
- ▶ la définition, **en plus**
 - ▶ demande l'allocation mémoire pour la variable,
 - ▶ donne une valeur initiale.



Déclarations de variables globales

- ▶ Une déclaration « promet » au compilateur qu'il y a une variable ayant ce type et ce nom, définie
 - ▶ soit dans un autre fichier source,
 - ▶ soit dans une bibliothèque.
- ▶ Les déclarations permettent d'utiliser les variables globales dans plusieurs fichiers.
- ▶ Pour chaque variable, il y a **une** définition et éventuellement plusieurs déclarations.
- ▶ Une déclaration fait référence à une définition dans une autre partie du programme.

Représentation de l'information

Unités

- ▶ **Bit** = Binary digIT, unité de stockage pouvant coder 2 valeurs.
- ▶ **Octet** = paquet de 8 bits
- ▶ **Byte** : plus petit paquet adressable :
 - ▶ Un byte doit pouvoir coder le jeu de caractères de base.
 - ▶ Un byte est formé d'une séquence contiguë de bits.
- ▶ **Mot** : un processeur peut traiter simultanément plusieurs bytes,
- ▶ Souvent (actuellement) :
 - ▶ 1 byte = 8 bits = 1 octet.
 - ▶ Données codées sur un nombre entier d'octets.
 - ▶ Exemple : représentation des caractères. Codes ASCII, EBCDIC, ...
 - ▶ ASCII 'A' est codé par 65, '0' par 48, ...
 - ▶ EBCDIC 'A' est codé par 193, '0' par 240, ...
- ▶ Le programmeur n'a pas besoin de connaître ces valeurs.

Un codage des entiers non signés

▶ Base 2

$$\begin{array}{cccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 2^7 + & & 2^5 + & 2^4 + & & & & 2^0 \\ & & & & & & & \text{représente} \\ & & & & & & & = 177 \end{array}$$

▶ Sur $n + 1$ bits, $a_n \dots a_1 a_0_{(2)}$ représente $a_0 + 2a_1 + \dots + 2^n a_n$
Ex : $base_2(177) = 10110001_{(2)} = \text{représentation de } 177.$

▶ Sur n bits, ce codage permet de représenter les entiers allant

▶ de 0 : $\underbrace{000 \dots 0}_{n \text{ fois}}_{(2)}$

▶ à $2^n - 1$: $\underbrace{111 \dots 1}_{n \text{ fois}}_{(2)}$.

Un codage des entiers signés

- ▶ Entiers > 0 ou < 0 : codage en complément à 2.
- ▶ Permet de coder les entiers de -2^{n-1} à $2^{n-1} - 1$
- ▶ Sur n bits, pour $k > 0$:
 - ▶ $code(k) = base_2(k)$
 - ▶ $code(0) = base_2(0) = 00 \dots 0$
 - ▶ $code(-k) = base_2([(2^n - 1) - k] + 1)$
- ▶ Exemple : sur $n = 3$ bits

	100	101	110	111	000	001	010	011
Signés	-4	-3	-2	-1	0	1	2	3
Non signés	4	5	6	7	0	1	2	3



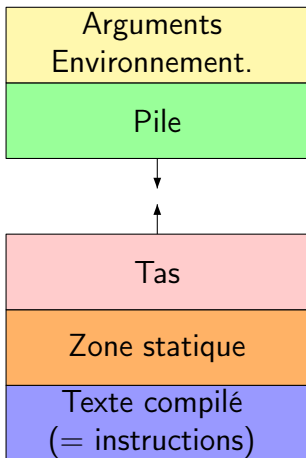
Débordement arithmétique.



Ne pas mélanger signés et non signés dans les calculs.

- ▶ **Allouer** une variable, c'est lui **réserver** un emplacement en mémoire.
- ▶ Une définition est en particulier une demande d'allocation.
- ▶ La place d'une variable peut être, selon les cas, réservée dans :
 - ▶ la zone statique,
 - ▶ la pile,
 - ▶ le tas, ...
- ▶ Les **arguments** d'une fonction et les **variables locales non statiques** sont alloués sur la **pile**.
- ▶ Les variables **globales** et celles déclarées **static** sont allouées en **zone statique**.

Schéma conceptuel de la mémoire



- ▶ La zone statique et le texte sont **fixés avant le lancement**.
- ▶ Les arguments sont initialisés au lancement du programme.
- ▶ La pile et le tas évoluent pendant l'exécution.



Classe d'allocation

- ▶ La classe d'allocation détermine
 - ▶ où en mémoire est conservée la variable,
 - ▶ si l'allocation est permanente ou temporaire.
- ▶ une variable automatique (locale non statique) est
 - ▶ allouée sur la pile d'exécution,
 - ▶ seulement pendant que son bloc est actif.
- ▶ une variable globale ou statique
 - ▶ est allouée en zone statique,
 - ▶ a une existence permanente en mémoire.



Classes d'allocation

Une variable peut être :

- ▶ **statique** : allouée dès la compilation, possède un emplacement mémoire pendant toute la durée d'exécution du programme.
 - ▶ Les variables **globales**,
 - ▶ Les variables explicitement déclarées **static**.
- ▶ **automatique** : allouées seulement lors de l'entrée dans leur bloc.
 - ▶ Les autres variables locales.
- ▶ **allouée** : ce point sera vu plus tard (`malloc.h`).

Les variables automatiques

- ▶ sont plus économiques que des variables globales/statiques
 - ▶ mémoire allouée seulement lors de l'exécution d'une fonction.
- ▶ rendent le programme plus lisible.
- ⚠ Ne pas utiliser des variables globales inutilement.

Variables statiques

- ▶ On peut déclarer une variable statique par le mot-clé `static`.
- ▶ Les variables statiques ont une existence (une adresse allouée) pendant toute la durée d'exécution du programme.
- ▶ Elles sont initialisées à 0 sauf si initialisation explicite.
- ▶ Différence entre

```
void f(void)
{
    static int x;
    printf("x = %d\n", x++);
}
```

```
void f(void)
{
    int x = 0;
    printf("x = %d\n", x++);
}
```

Durée de vie vs. portée

- ▶ La **durée de vie** des variables allouées en zone statique est celle de l'exécution du programme.
- ▶ La **durée de vie** des variables d'une fonction **f** allouées sur la pile est la même que la durée d'appel de **f** (en fait de leur bloc).



Ne pas confondre

- ▶ Durée de vie (temporelle, à l'exécution)
- ▶ Portée (spatiale, dans le source)

Initialisation

- ▶ On peut initialiser une variable lors de sa définition.
- ▶ La variable sera alors initialisée à chacune de ses allocations.
- ▶ Exemple

```
int x = 3;
```

- ▶ Les variables globales/statiques sont initialisées
 - ▶ 1 seule fois,
 - ▶ à 0 par défaut (en l'absence d'initialisation explicite).
- ▶ Les variables locales automatiques (non statiques)
 - ▶ ne **sont pas** initialisées automatiquement.
 - ▶ en l'absence d'initialisation explicite, elles contiennent donc à leur création une valeur **arbitraire**,
 - ⇒ doivent être initialisées ou affectées avant **chaque** 1ère utilisation.
- ▶ On utilisera pour l'instant des initialiseurs constants.

- ▶ Plusieurs types prédéfinis sont utilisables.
 - ▶ `char` : caractère.
 - ▶ `short int`, `int`, `long int`, `long long int` : entier.
 - ▶ `float`, `double`, `long double`.
 - ▶ `bool` : Booléens (`stdbool.h`).
 - ▶ `float complex`, `double complex`, `long double complex` (`complex.h`).
 - ▶ `float imaginary`, `double imaginary`, `long double imaginary`.
- ▶ Les types caractère ou entier peuvent être `signed` ou `unsigned`.
- ▶ Les types entiers sont `signed` par défaut.
- ▶ Le type `char` est `signed` ou `unsigned` selon l'implémentation.

Types de base

- ▶ **char** : permet de représenter le **jeu de caractères de base**.
 - ▶ Codés en général sur un octet (mais pas nécessairement).
 - ▶ Même comportement que soit **unsigned char**, soit **signed char**.
 - ▶ Constantes entre *quotes* : 'X'. **Ne pas confondre '0' et 0.**
- ▶ Les différents types entiers diffèrent par
 - ▶ l'intervalle qu'ils permettent de représenter,
 - ▶ l'arithmétique, signée ou non.
- ▶ Les **flottants** représentent des nombres avec parties décimales. La précision dépend de la place utilisée pour les coder.
 - ▶ **float** : simple précision
 - ▶ **double** : double précision
 - ▶ **long double** : précision étendue
- ▶ **void** : type vide pour les fonctions sans argument ou sans retour.

- ▶ L'espace mémoire qu'occupent les différents types dépend de la machine et du compilateur.
- ▶ L'opérateur `sizeof` permet de connaître la taille d'un type.
- ▶ On a toujours :
 - ▶ `sizeof(short) ≤ sizeof(int)`
 - `≤ sizeof(long)`
 - `≤ sizeof(long long)`
 - ▶ `sizeof(float) ≤ sizeof(double)`
 - `≤ sizeof(long double)`

- ▶ Pour tout type de base, les limites indiquent la plus petite et la plus grande valeur autorisée.
- ▶ Ces valeurs se trouvent dans le fichier `<limits.h>`
- ▶ **Exemple** : les valeurs minimale et maximale du type `short int` sont données par les constantes. Typiquement :
 - ▶ `SHRT_MIN` valant -2^{16} ,
 - ▶ `SHRT_MAX` valant $2^{16} - 1$.
- ▶ `unsigned` : calcul modulo le plus grand entier représentable.
- ▶ `signed` : un dépassement de capacité provoque typiquement
 - ▶ un « cycle » positifs \iff négatifs,
 - ▶ une erreur.

Tailles typiques

- ▶ `char` 8 bits = 1 octet
- ▶ `short` 16 bits = 2 octets
- ▶ `long` 32 bits = 4 octets
- ▶ `float` 32 bits = 4 octets
- ▶ `double` 64 bits = 8 octet

- ▶ Des conversions peuvent se produire pendant les calculs.
- ▶ Par ex, si on affecte un entier $> \text{USHRT_MAX}$ à un `unsigned short`.
- ▶ La fonction `printf` effectue aussi des conversions.
- ▶ Un `cast` demande une conversion explicite :

```
(unsigned short)(123456789)
```

Constantes

- ▶ Chaque type fournit ses propres constantes.
 - ▶ entier décimal `int` `1234`
 - ▶ entier octal `int` `02322`
 - ▶ entier hexadécimal `int` `0x4d2`
 - ▶ entier hexadécimal `long` `0x4d2L`
 - ▶ entier hexa `unsigned long` `0xffffffff`
 - ▶ flottant `double` `12.3, 123e-1, .123e2`
 - ▶ caractère `char` `'a' '%' '\n' '\0' '0'`
 - ▶ chaîne de caractères `char *` `"une chaine\n"`
- ▶ Des règles déterminent le type d'une constante entière.
- ▶ Par exemple, une constante décimale est du premier type qui peut la représenter, parmi `int`, `long` ou `long long`.

7– Notion d'expression

- ▶ L'affectation
- ▶ Expressions et effets de bord
- ▶ Opérateurs
- ▶ Opérateurs logiques

Notion d'expression


- ▶ Une expression est une entité qui se calcule.
- ▶ Exemple : $((x + 3) * 2) != 0$.
- ▶ Une expression peut faire intervenir
 - ▶ des variables : x ,
 - ▶ des constantes : $2, 3$
 - ▶ des opérateurs : $+, *, !=$.
- ▶ Une expression a donc un type. L'exemple calcule un Booléen.
- ▶ **En plus du calcul**, certaines expressions peuvent faire autre chose
 - ▶ changer la valeur de variables,
 - ▶ provoquer des affichages,...
- ▶ On dit qu'elles sont **à effet de bord**.
- ▶ Si $\langle e \rangle$ est une expression, $\langle e \rangle ;$ est une instruction consistant à évaluer (calculer) l'expression, et effectuer les effets de bord.
- ▶ Les opérateurs ont des priorités pour enlever l'ambiguïté.

- ▶ Une affectation


$x = \langle \text{expression} \rangle$

1. calcule l'expression $\langle \text{expression} \rangle$,
2. range la valeur calculée à l'adresse (emplacement mémoire) de x .
3. a elle-même une valeur, celle de $\langle \text{expression} \rangle$.

- ▶ Une affectation est donc une expression à effet de bord.
- ▶ On ne peut mettre à gauche de $=$ qu'une entité ayant une adresse.
Pour l'instant : un nom de variable.

 Ne pas confondre initialisation et affectation.

Des expressions à effets de bord

- ▶ Si x est une variable entière :
 - ▶ $x++$ est une expression qui s'évalue comme x .
 - ▶ $++x$ est une expression qui s'évalue comme $x+1$.
- ▶ Après évaluation, la valeur de x est incrémentée.
- ▶ Idem avec $x--$ et $--x$.
- ▶  Ne pas utiliser plusieurs de ces expressions en même temps :
- ▶ **Exemple** Si x vaut 0, l'expression

$$x++ - x++$$

peut valoir 1 ou -1 , suivant le compilateur.

Priorité des opérateurs

- ▶ ++, -- postfixes
- ▶ ++, -- préfixes
- ▶ ! non logique
- ▶ -, + unaires signe
- ▶ (<nom_de_type>) cast
- ▶ *, /, % opérateurs arithm. multiplicatifs
- ▶ -, + opérateurs arithm. additifs
- ▶ <, >, <=, >= comparaisons
- ▶ ==, != comparaisons
- ▶ && et logique
- ▶ || ou logique
- ▶ =, +=, =, *=, /=, =% affectations



La division / n'a pas le même sens sur entiers et les flottants.

Opérateurs logiques

- ▶ On a déjà vu les opérateurs Booléens ET `&&`, OU `||`, NON `!`.

<code>&&</code>	true	false
true	true	false
false	false	false

<code> </code>	true	false
true	true	true
false	true	false

	true	false
<code>!</code>	false	true

- ▶ Toute valeur non nulle est considérée vraie (true), et toute expression s'évaluant en 0 est considérée fausse (false).
- ▶ En C, l'évaluation des opérateurs `&&` et `||` est « **paresseuse** ».

- ▶ On évalue une expression `&&` ou `||` de gauche à droite.
- ▶ On s'arrête dès qu'on connaît le résultat.
- ▶ Exemple :

```
if ((x != 0) && (y/x == 2))  
{  
    //...  
}
```

- ▶ On évalue une expression `&&` ou `||` de gauche à droite.
- ▶ On s'arrête dès qu'on connaît le résultat.
- ▶ Exemple :

```
if ((x != 0) && (y/x == 2))  
{  
    //...  
}
```

- ▶ Ici, même si `x` est nul, on n'effectue pas de division par 0.
- ▶ En effet, si `x != 0` est `false`, l'expression du `if` est fausse.
- ▶ On n'a pas besoin de tester `(y/x == 2)` et **on ne le fait pas**.
- ▶ `if ((y/x == 2) && (x != 0))` provoque une erreur si `x` est nul.

8– Les fonctions

- ▶ Fonctions et modularité
- ▶ Définition, déclaration, appel
- ▶ Définition de fonction : syntaxe
- ▶ Appel de fonction
- ▶ Retour d'une fonction
- ▶ `return` vs. `exit`
- ▶ Récursivité
- ▶ Compléments

- ▶ Les fonctions structurent le programme en entités logiques.
 - ▶ code réutilisable.
 - ▶ code plus facile à corriger.
 - ▶ code plus facilement modifiable et maintenable.
 - ▶ code générique. **Exemple** : fonction de tri **paramétrée** par un ordre.

- ▶ Les fonctions structurent le programme en entités logiques.
 - ▶ code réutilisable.
 - ▶ code plus facile à corriger.
 - ▶ code plus facilement modifiable et maintenable.
 - ▶ code générique. **Exemple** : fonction de tri **paramétrée** par un ordre.
- ▶ Elles induisent des ruptures dans l'aspect séquentiel.
- ▶ Quand l'instruction courante **I** contient un appel à une fonction **f** :
 - ▶ Les arguments de **f** sont évalués pour lui être transmis,
 - ▶ L'exécution du programme passe à la première instruction de **f**.
 - ▶ Lorsque dans **f** on rencontre l'instruction **return** ou la dernière accolade, le programme reprend juste après l'appel de **f** dans **I**.

Définition, prototype, appel

- ▶ Un **prototype** (ou **déclaration**) de fonction décrit
 - ▶ son nom,
 - ▶ le type de ses paramètres, et
 - ▶ le type de la valeur qu'elle calcule.
- ▶ Une **définition** de fonction comporte en plus la liste de ses instructions, dans son corps (entre { et }).
- ▶ Pour pouvoir compiler un fichier source (.c), toute fonction **utilisée** doit être, **avant toute utilisation**
 - ▶ soit **définie**,
 - ▶ soit **déclarée** explicitement. Typiquement :

```
<type retour> <nom> (<liste types>);
```
 - ▶ soit **déclarée** via une directive **#include**.



Définitions de fonctions : syntaxe

- ▶ Schématiquement, une définition de fonction est constituée de :
 - ▶ son **interface** :
 - ▶ le type et le nom de la fonction,
 - ▶ les types et les noms des paramètres.
 - ▶ son **corps**, ou bloc principal, constitué de
 - ▶ une accolade ouvrante `{`.
 - ▶ des définitions de variables (locales), des instructions, des blocs.
 - ▶ une accolade fermante `}`.

- ▶ Définir une fonction dans le corps d'une autre fonction est interdit.

Appel de fonction

```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```

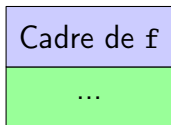
- ▶ On dit que **f** appelle **g**.
- ▶ **f** est la fonction appelante et **g** est la fonction appelée.
- ▶ Lors de l'appel de **g**, on dit parfois que **g** a une invocation active.
- ▶ Au cours d'un programme, une fonction peut
 - ▶ être appelée puis appelante,
 - ▶ s'appeler elle-même (récursivité).
 - ▶ avoir à un instant donné plusieurs invocations en cours. Par ex,
 - ▶ f1 appelle f1, ou
 - ▶ f1 appelle f2 qui elle-même appelle f1.

Appel de fonction

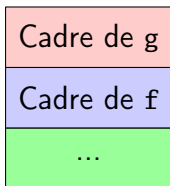
```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```

- ▶ Schéma conceptuel de la pile :



Avant appel de g



Pendant l'appel à g

- ▶ Un cadre d'appel de fonction permet de mémoriser (entre autres)
 - ▶ les valeurs des arguments et variables automatiques.
 - ▶ la valeur retour transmise à la fonction appelante.

Appel de fonction

```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```

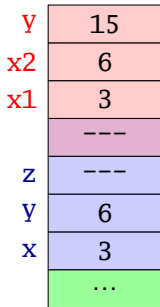
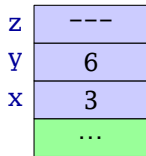
z	---
y	6
x	3
	...

Avant appel de g

Appel de fonction

```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```

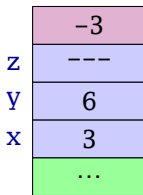
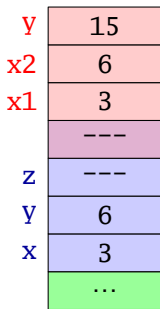
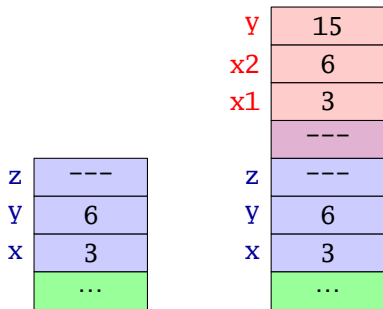


Avant appel de g Pendant appel à g

Appel de fonction

```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```

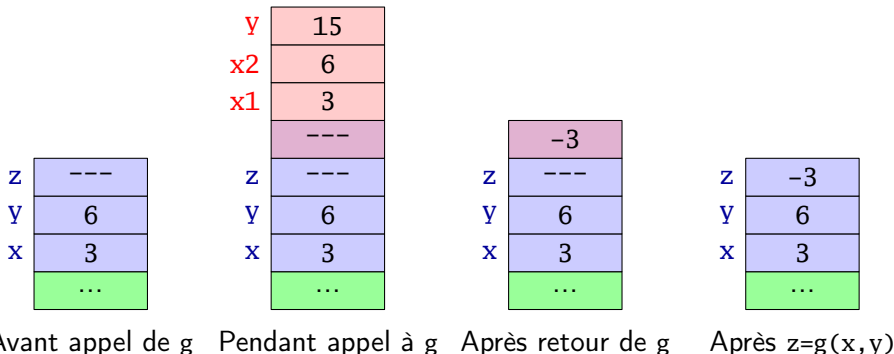


Avant appel de g Pendant appel à g Après retour de g

Appel de fonction

```
void f(void)
{
    int x = 3, y = 6, z;
    z = g(x,y);
}
```

```
int g(int x1, int x2)
{
    int y = 15;
    return y - x1 * x2;
}
```



Appel de fonction : récapitulatif

- ▶ Les arguments de la fonction sont calculés,
- ▶ Les **valeurs** sont transmises à la fonction appelée,
- ⚠ **Conséquence** la fonction appelée travaille sur les **valeurs**.
- ⚠ Avec la définition `void g(int x) { x = x+1; }` l'appel `g(x)` ne change pas la valeur de `x`.
- ▶ Lors de chaque appel d'une fonction, un emplacement est alloué pour chaque **variable automatique** et chaque **argument**.
Rappel une variable statique est allouée une seule fois (zone statique).
- ⇒ Si une fonction `g` est invoquée plusieurs fois, chaque variable automatique locale et argument de `g` a un emplacement associé pour **chaque** invocation de `g` (dans chaque cadre de `g`).

Retour d'une fonction

- ▶ L'instruction `return` demande la terminaison de la fonction et le retour dans la fonction appelante.
- ▶ Si la fonction retourne une valeur, l'instruction est suivie d'une expression de type compatible.

```
int f (...)  
{  
    ...  
    return 0;  
}
```

- ▶ Si la fonction ne retourne rien, l'instruction est utilisée seule.

```
void f (...)  
{  
    ...  
    return;  
}
```


- ▶ Si une fonction ne retourne rien, et que l'exécution arrive sur la dernière accolade fermante, un `return` implicite est effectué.
- ▶ Si la fonction retourne une valeur, on doit utiliser `return`.
- ▶ Exception : si la fonction `main` ne rencontre pas d'instruction `return`, un `return 0;` implicite est effectué.



- Il est cependant conseillé que la fonction `main` retourne une valeur **explicitement** (par `return`) (c'est une obligation en C90).
- ▶ Il faut aussi tester systématiquement le code retour des fonctions pour savoir si elles se sont déroulées correctement.

return vs. exit

- ▶ Sauf dans la première invocation de la fonction `main`, l'instruction `return` ne provoque pas l'arrêt du programme.
 - ▶ Pour terminer le programme, on peut utiliser la fonction de bibliothèque `void exit(int status);`.
 - ▶ Dans la 1ère invocation de `main`, `return <exp> ≡ exit(<exp>)`.
 - ▶ Les constantes symboliques `EXIT_SUCCESS` et `EXIT_FAILURE` peuvent être passées à `exit()` pour indiquer respectivement une terminaison normale ou anormale.
 - ▶ On peut enregistrer par `atexit()` des fonctions à exécuter avant une fin de programme due à un `exit()` ou un `return` du 1er `main`.
- ⚠ Ne pas confondre non plus `return` et `printf()` !

Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

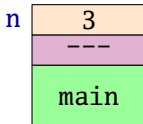
main

Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

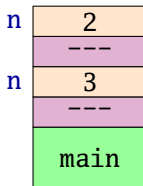


Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

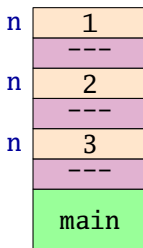


Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

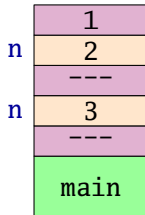


Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

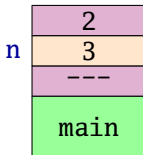


Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```

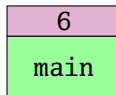


Récurtivité

- ▶ On peut, dans une définition de fonction f , appeler la fonction f .
- ▶ Les règles d'appel de fonction s'appliquent normalement.

```
int Factorielle(int n)
{
    if (n <= 1)
        return 1;
    return n * Factorielle(n-1);
} /* Factorielle */
```

```
int main(void)
{
    return Factorielle(3);
}
```



- ▶ Il faut toujours s'assurer que la condition d'arrêt est traitée.
- ▶ Les programmes récursifs sont
 - 😊 souvent plus courts, mais
 - ⚠ parfois beaucoup moins efficaces si écrits sans soin.
- ▶ Exemple : Fibonacci.

```
long long Fibonacci (int n)
{
    if (n <= 1)
        return 1;
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```

- ▶ Complexité $\approx \phi^n$, alors que $O(n)$ et même $O(\log n)$ sont possibles !
- ▶ **Note** Il est toujours possible de « dérécurifier » une fonction.

- ▶ Exemple : calcul de 2^n pour $n \geq 0$.

```
// version 1 :  $O(2^n)$ 
```

```
int pow2 (int n)
{
    if (n <= 0)
        return 1;
    return pow2(n-1) + pow2(n-1);
}
```

```
// version 2 :  $O(n)$ 
```

```
int pow2lin (int n)
{
    if (n <= 0)
        return 1;
    return 2 * pow2lin(n-1);
}
```

- ▶ Exemple : calcul de 2^n pour $n \geq 0$.

```
// version 1 :  $O(2^n)$ 
```

```
int pow2 (int n)
{
    if (n <= 0)
        return 1;
    return pow2(n-1) + pow2(n-1);
}
```

```
// version 2 :  $O(n)$ 
```

```
int pow2lin (int n)
{
    if (n <= 0)
        return 1;
    return 2 * pow2lin(n-1);
}
```

▶ Exercice

- ▶ Utiliser le fait que $2^n = (2^{n/2})^2 \cdot 2^{n\%2}$ pour calculer 2^n en $O(\log n)$.
- ▶ Adapter la technique au calcul du $n^{\text{ème}}$ nombre de Fibonacci.
- ▶ Donner une version non récursive de ces fonctions.

- ▶ Il est possible aussi d'écrire f qui appelle g qui elle-même appelle f .
- ▶ Exemple :
 - ▶ Parité : `pair()` et `impair()`.
 - ▶ Récurrences doubles.

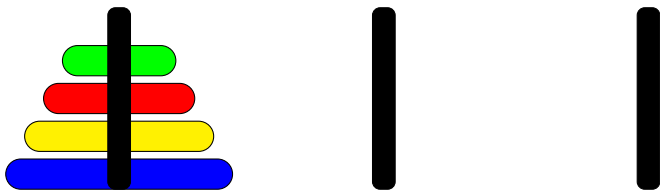
$$\begin{cases} u_0 = 1, & u_n = 2u_{n-1} - v_{n-1} & (n \geq 1) \\ v_0 = 5, & v_n = u_{n-1} - v_{n-1} & (n \geq 1) \end{cases}$$

 On doit déclarer toute fonction utilisée avant sa définition.

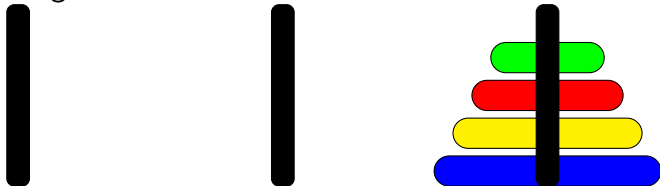
 Attention à nouveau à l'efficacité !

- ▶ L'utilisation naïve donne une complexité exponentielle.
- ▶ On a $\begin{pmatrix} u_n \\ v_n \end{pmatrix} = A \begin{pmatrix} u_{n-1} \\ v_{n-1} \end{pmatrix} = A^n \begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ avec $A = \begin{pmatrix} 2 & -1 \\ 1 & -1 \end{pmatrix}$.
- ▶ Pour calculer efficacement A^n , on peut s'inspirer du calcul de 2^n .

Récurtivité : tours de Hanoi

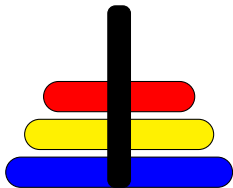


- ▶ **Jeu** En déplaçant les disques sur barres
 - ▶ un à un
 - ▶ sans poser un disque sur un disque plus petit,arriver à la configuration suivante :



Tours de Hanoi : exemple

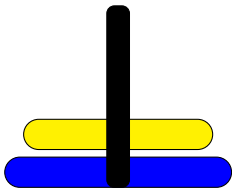
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

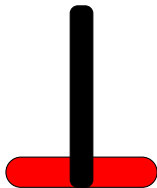
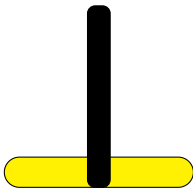
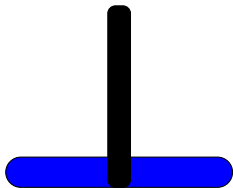
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

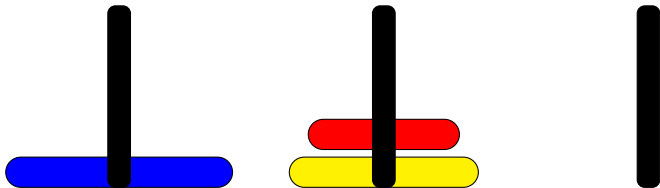
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

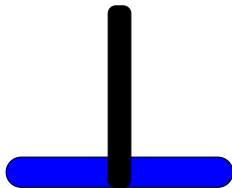
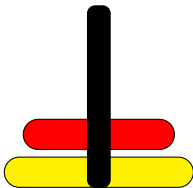
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ **Déplacer 1 \rightarrow 3**
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

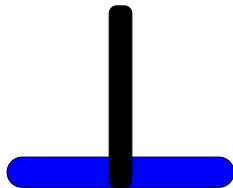
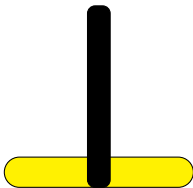
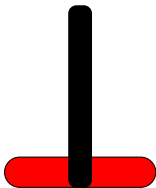
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

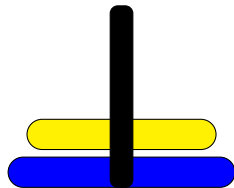
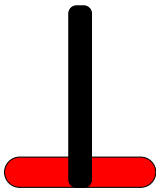
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

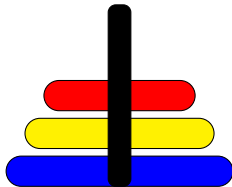
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

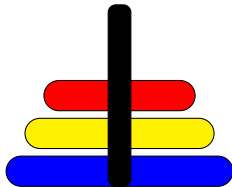
Avec 3 disques



- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

Tours de Hanoi : exemple

Avec 3 disques

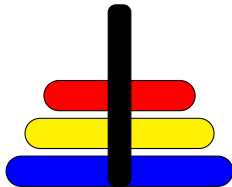


- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 2
- ▶ Déplacer 3 \rightarrow 2
- ▶ Déplacer 1 \rightarrow 3
- ▶ Déplacer 2 \rightarrow 1
- ▶ Déplacer 2 \rightarrow 3
- ▶ Déplacer 1 \rightarrow 3

} Déplacement 1 \rightarrow 2 des 2 disques supérieurs

Tours de Hanoi : exemple

Avec 3 disques



▶ Déplacer 1 \rightarrow 3

▶ Déplacer 1 \rightarrow 2

▶ Déplacer 3 \rightarrow 2

▶ Déplacer 1 \rightarrow 3

▶ Déplacer 2 \rightarrow 1

▶ Déplacer 2 \rightarrow 3

▶ Déplacer 1 \rightarrow 3

} Déplacement 1 \rightarrow 2 des 2 disques supérieurs

} Déplacement 2 \rightarrow 3 des 2 disques supérieurs



Tours de Hanoi

- ▶ On sait faire pour un disque.
- ▶ Si on sait faire pour $n - 1$ disques, on sait faire pour n .

```
void hanoi(int nb, int depart, int arrivee)
{
    if (nb == 1)
    {
        printf("Déplacer %d --> %d\n", depart, arrivee);
        return;
    }
    int intermediaire = 6 - (depart + arrivee); // piquet restant
    hanoi(nb-1, depart, intermediaire);
    hanoi(1, depart, arrivee);
    hanoi(nb-1, intermediaire, arrivee);
}
```

- ▶ Complexité? Peut-on faire mieux?

Récurtivité et tableaux

- ▶ On verra qu'un tableau est un pointeur (constant).
- ▶ Un tableau **Tab** contient l'adresse de sa première case **Tab[0]**.
- ▶ Si un tableau **Tab** est une variable automatique locale à une fonction **g**, alors on ne doit pas retourner **Tab** dans **g**.
- ▶ En effet, **Tab** désigne un emplacement dans le cadre de **g**, et ce cadre est désalloué au retour de **g**.

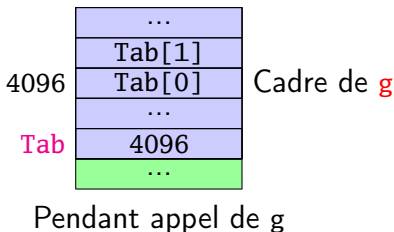


Diagram illustrating the state of memory after the function call. A single green box contains "...". Below it, the text "Après l'appel à g" and "Plus rien à l'adresse 4096 !" is written.

Compléments : atexit()

(Complément hors programme examen)

- ▶ La fonction `atexit()` permet d'enregistrer des fonctions sans argument et ne renvoyant pas de valeur.
- ▶ Elles seront exécutées en cas d'`exit()` (ou `return` de la première invocation de `main`) en ordre inverse de leur enregistrement.

```
void f(void)
{
    printf("f()\n");
}
```

```
void g(void)
{
    printf("g()\n");
}
```

```
int main(void)
{
    atexit(f);
    atexit(f);
    atexit(g);

    printf("-----\n");
    exit(EXIT_SUCCESS);
}
```



Rappels : la fonction `printf()`

- ▶ La fonction `printf` sert à réaliser des affichages.
- ▶ Son 1er argument est une chaîne de caractères.
- ▶ Ce 1er argument sert de support à ce qui est affiché.
- ▶ La plupart de ses caractères sont affichés tels quels, sauf
 - ▶ des séquences d'échappement : `\n`, `\t`, `\\`, `\"`, ...
 - ▶ des directives formées du caractère `%` suivi d'autres caractères, permettant un affichage formaté.
 - ▶ Le nombre de paramètres suivant le 1er argument dépend du nombre de séquences `%`...

La fonction printf() : exemples

Instruction

Affichage

▶ `printf("x = %d\n", x);`

`x = 7`

▶ `printf("x+1=%d\n", x+1);`

`x = 8`

▶ `printf("%d+%d=%d\n", x, 2*x, 3*x);`

`7+14=21`



La fonction printf() : exemples

Instruction

```
printf("OK\n");  
printf("C'est \"OK\"\\n");  
printf("TVA: 19,6%%\n");  
printf("Pi=%g\n",acos(1));  
printf("%ce%cl%c\n", 'H', 'l', 'o');  
printf("%+06.3Lg\n",7.234L);
```

Affichage

```
OK  
C'est "OK"  
TVA: 19,6%  
Pi=3.14159  
Hello  
+07.23
```

Remarque À compiler avec l'option `-lm` pour édition de liens avec la bibliothèque `libm`, contenant la définition de `acos`.



La fonction printf : directives

- ▶ Les caractères pouvant suivre le % d'une directive donnent l'interprétation de l'argument correspondant

Séquence	Interprétation de l'argument correspondant
%d,	Nombre décimal
%c	Caractère
%s	Chaîne de caractères
%g	Nombre flottant en double précision
%p	Pointeur
%%	Aucun argument correspondant. Imprime %.

- ▶ Il existe d'autres spécificateurs (pour entiers dans d'autres bases, non signés, longs etc.).



Nombre variable d'arguments

(Complément hors programme examen)

- ▶ La fonction `printf` prend un nombre variable d'arguments.
- ▶ Il est possible d'écrire des fonctions à nombre variable d'arguments.
- ▶ L'un des arguments doit permettre de retrouver le nombre et le type des arguments restants.
- ▶ Pour la fonction `printf`, c'est la chaîne de format qui joue ce rôle : les séquences `%...` permettent de retrouver cette information.
- ▶ Le prototype d'une telle fonction se termine par `....`
- ▶ Par exemple, un prototype de `printf` pourrait être

```
int printf(const char *fmt, ...);
```

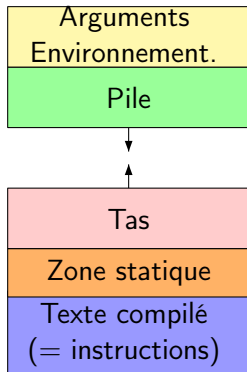
- ▶ Pour plus de détails, cf. `man va_arg`.

9– Les pointeurs

- ▶ Rappel : organisation mémoire d'un processus
- ▶ Variables et adresses
- ▶ Pointeurs : valeurs et déclarations
- ▶ Emplacement pointé
- ▶ Tableaux vs. pointeurs
- ▶ Arithmétique des pointeurs
- ▶ Allocation dynamique
- ▶ Passage de paramètres

Organisation mémoire d'un processus

- ▶ Un programme en cours d'exécution s'appelle un **processus**.
- ▶ Sa mémoire peut être schématisée en distinguant plusieurs zones.
- ▶ Certaines (texte, statique) sont de taille fixée à la compilation.
- ▶ D'autres (pile, tas) ont une taille évoluant au cours de l'exécution.



Variables et adresses

- ▶ Une variable active à un instant donné possède une **adresse logique**.
- ▶ Une **adresse** est un entier désignant un emplacement mémoire
 - ▶ sur la pile,
 - ▶ en zone statique,
 - ▶ dans le tas, ...
- ▶ L'**adresse** d'une variable **x** se note **&x**.
- ▶ Si le type de **x** est **T**, le type de **&x** se note **T ***.
- ▶ Un **pointeur** est une variable qui sert à mémoriser des adresses.
- ▶ Le type d'un pointeur **p** dépend du type des variables dont **p** contiendra l'adresse.
- ▶ **Exemples** `char *p;` définit **p** comme un pointeur sur un **char**.
`int **q;` \rightsquigarrow **q** est un pointeur sur un pointeur sur un **int**.



Pointeurs : valeurs

- ▶ Pointeur = variable pouvant contenir une adresse.
- ▶ Si `p` contient l'adresse `&x` de la variable `x`, on dit que `p` pointe sur `x`.
- ▶ Les pointeurs mémorisent des adresses codées par des entiers.
- ▶ En pratique, on ne s'intéresse pas aux valeurs de ces entiers.
- ▶ Ces adresses logiques correspondent à la vue locale que le processus a de sa mémoire (pas à une adresse physique).
- ▶ La constante `NULL` de `<stddef.h>` n'est l'adresse d'aucun objet.
- ▶ Avec le prototype `int main(int argc, char *argv[])`, le tableau `argv` a `argc+1` cases, et la dernière contient `NULL`.



Pointeurs : déclarations

- ▶ Déclaration, cas simples :

`<type> *<nom de variable>;`

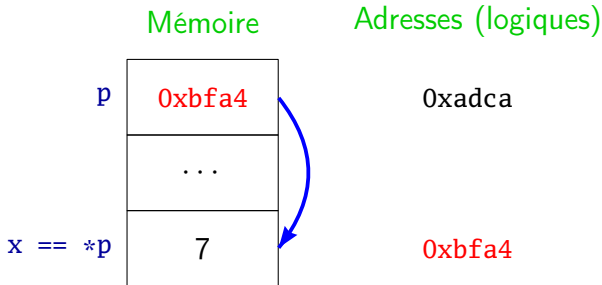
- ▶ Exemples

- ▶ `char *p;` pointeur sur un caractère.
 - ▶ `char **p;` pointeur sur pointeur sur un caractère.
 - ▶ `char *tab[10];` tableau de 10 pointeurs sur caractère.
 - ▶ `char (*tab)[10];` pointeur sur un tableau de 10 char.
-
- ▶ Le constructeur `*` est moins prioritaire que `[]`.

Pointeurs : un exemple

- ▶ Pointeur `p` pouvant contenir l'adresse d'une variable de type `short`.

```
short *p;           // p: pointeur sur short
short x = 7;
p = &x;            // p pointe maintenant sur x
```



Emplacement pointé (*pointee*)

- ▶ Si `p` est un pointeur de type `T *`, la donnée de taille `sizeof(T)` octets commençant à l'adresse `p` se note `*p`.
- ▶ `*p` référence l'emplacement pointé par `p`.

```
int *p, *q;
```



Emplacement pointé (*pointee*)

- ▶ Si **p** est un pointeur de type **T ***, la donnée de taille `sizeof(T)` octets commençant à l'adresse **p** se note ***p**.
- ▶ ***p** référence l'emplacement pointé par **p**.

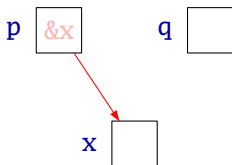
```
int *p, *q;  
int x;
```



Emplacement pointé (*pointee*)

- ▶ Si p est un pointeur de type $T *$, la donnée de taille $\text{sizeof}(T)$ octets commençant à l'adresse p se note $*p$.
- ▶ $*p$ référence l'emplacement pointé par p .

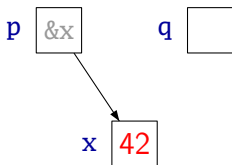
```
int *p, *q;  
int x;  
p = &x;
```



Emplacement pointé (*pointee*)

- ▶ Si **p** est un pointeur de type **T ***, la donnée de taille `sizeof(T)` octets commençant à l'adresse **p** se note ***p**.
- ▶ ***p** référence l'emplacement pointé par **p**.

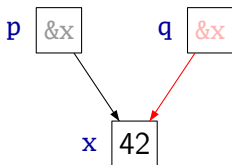
```
int *p, *q;  
int x;  
p = &x;  
*p = 42;
```



Emplacement pointé (*pointee*)

- ▶ Si **p** est un pointeur de type **T ***, la donnée de taille `sizeof(T)` octets commençant à l'adresse **p** se note ***p**.
- ▶ ***p** référence l'emplacement pointé par **p**.

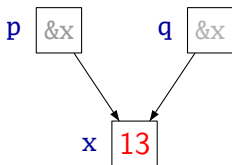
```
int *p, *q;  
int x;  
p = &x;  
*p = 42;  
q = p;
```



Emplacement pointé (*pointee*)

- ▶ Si **p** est un pointeur de type **T ***, la donnée de taille `sizeof(T)` octets commençant à l'adresse **p** se note ***p**.
- ▶ ***p** référence l'emplacement pointé par **p**.

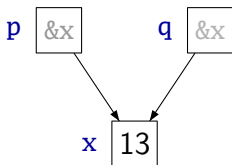
```
int *p, *q;  
int x;  
p = &x;  
*p = 42;  
q = p;  
*q = 13;
```



Emplacement pointé (*pointee*)

- ▶ Si **p** est un pointeur de type **T ***, la donnée de taille `sizeof(T)` octets commençant à l'adresse **p** se note ***p**.
- ▶ ***p** référence l'emplacement pointé par **p**.

```
int *p, *q;  
int x;  
p = &x;  
*p = 42;  
q = p;  
*q = 13;
```



! `int *r;` réserve de la place pour la variable `r`, **PAS** pour l'emplacement `*r` sur lequel pointera `r`.

```
int *r;  
*r = 3; // NON ! Il faut d'abord  
// faire pointer r vers un emplacement legal
```

Tableaux vs. pointeurs

- ▶ Un tableau est un pointeur **constant**.
- ▶ Ceci explique (a posteriori) pourquoi une fonction qui reçoit un tableau en argument peut modifier les valeurs de ses cases.
- ▶ ***p** est équivalent à **p[0]**
- ▶ ***(p+i)** est équivalent à **p[i]**.

```
int tab[10];  
int tab2[20];
```

```
*(tab+1) = 2;    // OK, equivalent a tab[1] = 2;  
tab = tab2;     // ILLEGAL
```



int p[10] demande au compilateur de réserver de la place p et pour 10 entiers (contrairement à int *p).



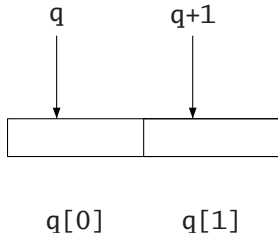
Arithmétique des pointeurs

- ▶ Si p est un pointeur et n une expression entière, la valeur numérique de $p+n$ dépend du type de p .
- ▶ Si p , de type $T *$, pointe sur la i -ème case d'un tableau de type T , $p+n$ pointe sur la $i+n$ -ème case (si elle est définie [...]).

Arithmétique des pointeurs

- ▶ Si p est un pointeur et n une expression entière, la valeur numérique de $p+n$ dépend du type de p .
- ▶ Si p , de type $T *$, pointe sur la i -ème case d'un tableau de type T , $p+n$ pointe sur la $i+n$ -ème case (si elle est définie [...]).
- ▶ Exemple, en supposant $\text{sizeof}(\text{char})=1$ et $\text{sizeof}(\text{short})=2$.

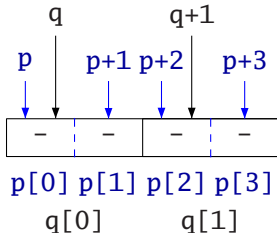
```
short q[2];  
char *p;  
p = (char *)q;
```



Arithmétique des pointeurs

- ▶ Si p est un pointeur et n une expression entière, la valeur numérique de $p+n$ dépend du type de p .
- ▶ Si p , de type $T *$, pointe sur la i -ème case d'un tableau de type T , $p+n$ pointe sur la $i+n$ -ème case (si elle est définie [...]).
- ▶ Exemple, en supposant $\text{sizeof}(\text{char})=1$ et $\text{sizeof}(\text{short})=2$.

```
short q[2];  
char *p;  
p = (char *)q;
```





Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶

```
void *calloc (size_t nb, size_t size);
```

alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacuns, et renvoie un pointeur sur le début de ce tableau.

Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶

```
void *calloc (size_t nb, size_t size);
```

alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacuns, et renvoie un pointeur sur le début de ce tableau.

- ▶ Exemple

```
int *p;
```

p

Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶

```
void *calloc (size_t nb, size_t size);
```

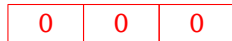
alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacuns, et renvoie un pointeur sur le début de ce tableau.

- ▶ Exemple

```
int *p;  
    (int *)calloc(3, sizeof(int));
```

p

--



Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

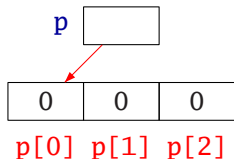
- ▶

```
void *calloc (size_t nb, size_t size);
```

alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacun, et renvoie un pointeur sur le début de ce tableau.

- ▶ Exemple

```
int *p;  
p = (int *)calloc(3, sizeof(int));
```



Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

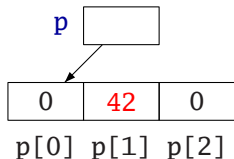
- ▶

```
void *calloc (size_t nb, size_t size);
```

alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacun, et renvoie un pointeur sur le début de ce tableau.

- ▶ Exemple

```
int *p;  
p = (int *)calloc(3, sizeof(int));  
p[1] = 42;
```



Allocation dynamique : calloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

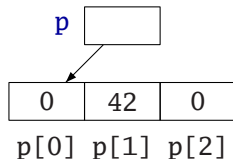
- ▶

```
void *calloc (size_t nb, size_t size);
```

alloue l'espace initialisé à 0 pour un tableau de `nb` éléments de `size` octets chacun, et renvoie un pointeur sur le début de ce tableau.

- ▶ Exemple

```
int *p;  
p = (int *)calloc(3, sizeof(int));  
p[1] = 42;
```



- ▶ `calloc` renvoie `NULL` si l'allocation échoue.



Allocation dynamique : malloc

► Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

► `void *malloc (size_t size);`

alloue l'espace non initialisé de taille `size` octets.

Allocation dynamique : malloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶

```
void *malloc (size_t size);
```

alloue l'espace non initialisé de taille `size` octets.

- ▶ Exemple

```
int *p;
```

p

Allocation dynamique : malloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

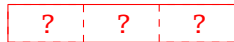
- ▶ `void *malloc (size_t size);`

alloue l'espace non initialisé de taille `size` octets.

- ▶ Exemple

```
int *p;  
    (int *)malloc(3 * sizeof(int));
```

p 



Allocation dynamique : malloc

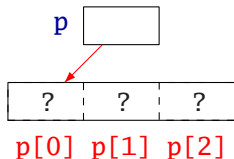
- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶ `void *malloc (size_t size);`

alloue l'espace non initialisé de taille `size` octets.

- ▶ Exemple

```
int *p;  
p = (int *)malloc(3 * sizeof(int));
```



Allocation dynamique : malloc

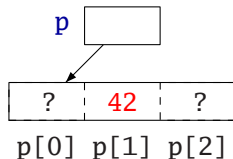
- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

- ▶ `void *malloc (size_t size);`

alloue l'espace non initialisé de taille `size` octets.

- ▶ Exemple

```
int *p;  
p = (int *)malloc(3 * sizeof(int));  
p[1] = 42;
```



Allocation dynamique : malloc

- ▶ Pour allouer à l'exécution (« *dynamiquement* ») de la mémoire, on utilise `calloc`, `malloc` ou `realloc`.

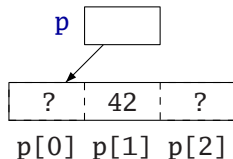
- ▶

```
void *malloc (size_t size);
```

alloue l'espace non initialisé de taille `size` octets.

- ▶ Exemple

```
int *p;  
p = (int *)malloc(3 * sizeof(int));  
p[1] = 42;
```



- ▶ `malloc` renvoie `NULL` si l'allocation échoue.



Allocation dynamique : realloc

▶ `void *realloc (void *p, size_t size);`

permet de réallouer une zone de `size` octets.

- ▶ La zone pointée par `p` doit avoir été allouée avec `m/c/realloc`.
- ▶ Si `size` est plus grand que la taille de cette zone, le contenu de la zone est conservé, sinon il est tronqué.

Allocation dynamique : realloc

▶ `void *realloc (void *p, size_t size);`

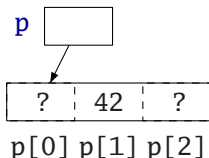
permet de réallouer une zone de `size` octets.

▶ La zone pointée par `p` doit avoir été allouée avec `m/c/realloc`.

▶ Si `size` est plus grand que la taille de cette zone, le contenu de la zone est conservé, sinon il est tronqué.

▶ Suite de l'exemple précédent

```
int *p;  
p = (int *)malloc(3 * sizeof(int));  
p[1] = 42;
```



Allocation dynamique : realloc

▶ `void *realloc (void *p, size_t size);`

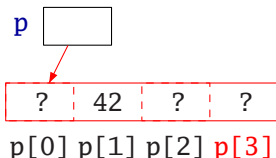
permet de réallouer une zone de `size` octets.

▶ La zone pointée par `p` doit avoir été allouée avec `m/c/realloc`.

▶ Si `size` est plus grand que la taille de cette zone, le contenu de la zone est conservé, sinon il est tronqué.

▶ Suite de l'exemple précédent

```
int *p;  
p = (int *)malloc(3 * sizeof(int));  
p[1] = 42;  
p = (int *)realloc(p, 4 * sizeof(int));
```



Allocation dynamique : realloc

▶ `void *realloc (void *p, size_t size);`

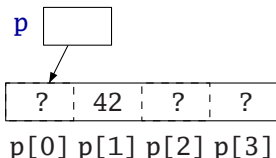
permet de réallouer une zone de `size` octets.

▶ La zone pointée par `p` doit avoir été allouée avec `m/c/realloc`.

▶ Si `size` est plus grand que la taille de cette zone, le contenu de la zone est conservé, sinon il est tronqué.

▶ Suite de l'exemple précédent

```
int *p;  
p = (int *)malloc(3 * sizeof(int));  
p[1] = 42;  
p = (int *)realloc(p, 4 * sizeof(int));
```



▶ `realloc` renvoie `NULL` en cas d'échec, laissant intact le bloc original.



Allocation dynamique : free

▶ Il est important de désallouer la mémoire qu'on n'utilise plus.

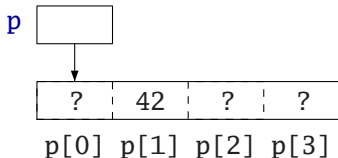
▶ `void free (void *p);`

permet de désallouer la mémoire allouée par `m/c/realloc`.

Allocation dynamique : free

- ▶ Il est important de désallouer la mémoire qu'on n'utilise plus.
- ▶ `void free (void *p);`
permet de désallouer la mémoire allouée par `m/c/realloc`.
- ▶ Suite de l'exemple précédent

```
int *p;  
//...
```



Allocation dynamique : free

- ▶ Il est important de désallouer la mémoire qu'on n'utilise plus.

- ▶ `void free (void *p);`

permet de désallouer la mémoire allouée par `m/c/realloc`.

- ▶ Suite de l'exemple précédent

```
int *p;
```

```
//...
```

```
free(p);
```

p



Affectation de pointeurs : résumé

- ▶ Pour affecter à un pointeur un emplacement pointé, on utilise
 - ▶ soit l'adresse d'une variable existante : $p = \&x$.
 - ▶ soit la constante NULL : $p = \text{NULL}$.
 - ▶ soit une adresse stockée dans un autre pointeur $p = q$.
 - ▶ soit, plus généralement, une expression de type pointeur $p = q+1$.
 - ▶ soit une fonction demandant la création à l'exécution d'une zone mémoire : `malloc()/calloc()/realloc()`.
- ▶ Film <http://www.cs.stanford.edu/cslibrary/PointerFunCBig.avi>

Passage de paramètres : rappel

- ▶ Lors d'un appel de fonction :
 - ▶ Les paramètres d'appel sont évalués (dans un ordre dépendant de l'implémentation),
 - ▶ les valeurs calculées sont empilées,
 - ▶ la fonction appelée **travaille sur ces valeurs empilées**.
- ▶ Un appel de fonction de la forme $f(x)$, avec

```
void f(int z)
{
    z = 12345;
}
```

ne peut donc pas modifier la valeur de la variable x .

Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

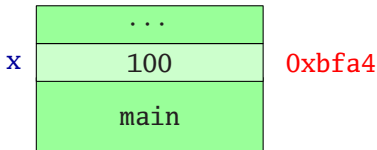
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

Dans main



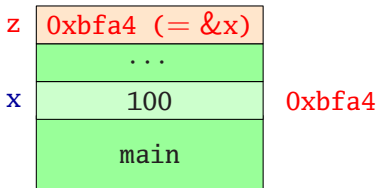
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

À l'appel de f



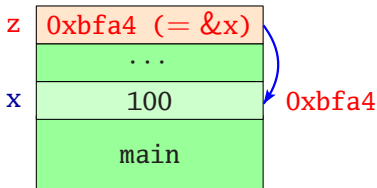
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

À l'appel de f



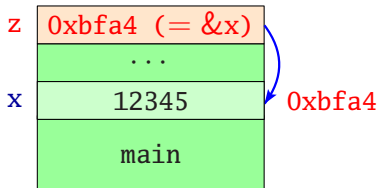
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

Après l'affectation *z=12345



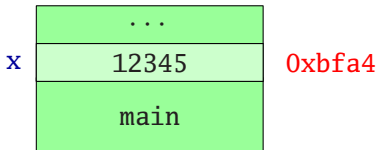
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

Après retour de f



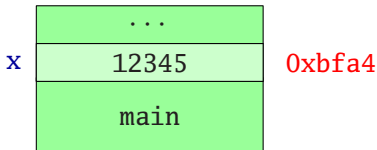
Pointeurs et passage de paramètres

- ▶ Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
void f(int *z)
{
    *z = 12345;
    return;
}
```

```
int main(void)
{
    int x = 100;
    f(&x);
    printf("Valeur de x : %d", x) ;
}
```

Après retour de f



⚠ Ne pas retourner l'adresse d'une variable locale automatique !

10– Types structurés 2 : structures

- ▶ Construction de nouveaux types
- ▶ Énumérations
- ▶ Structures
- ▶ Abréviations de types
- ▶ Pointeurs et structures
- ▶ Listes chaînées
- ▶ Pointeurs sur fonctions

- ▶ On peut construire des nouveaux types à partir des types de base en utilisant des **constructeurs**.
- ▶ Premiers types “construits” :
 - ▶ les énumérations.
 - ▶ les tableaux.
 - ▶ les structures.
 - ▶ les unions (non présentées dans ce cours).
 - ▶ les pointeurs.

Énumérations

- ▶ Un type énuméré contient un nombre **fini** de valeurs.
- ▶ Chaque valeur a un nom et est codée par un entier.
- ▶ La définition du type se fait hors du corps de toute fonction.

```
enum jour                // def. du type
{
    lun, mar, mer, jeu, ven, sam, dim
};

enum jour    x, y;      // def. des variables

int main(void)
{
    x = lun;
    y = x + 1;          // y vaut mar
}
```


Structures

- ▶ Une structure regroupe plusieurs champs de types différents.

```
struct <nom>
{
    <suite de declarations de champs>
};
```

- ▶ Exemple :

```
struct individu
{
    char nom[25];
    char prenom[20];
    int age;
};
```

- ▶ Pour accéder aux champs d'une structure on utilise l'opérateur . :

```
struct individu etud;    // declaration de variable
etud.age = 20;
```

Structures emboîtées

- ▶ Un champ d'une structure peut être lui-même une structure, si son type a déjà été défini.

```
// definitions de types
```

```
struct point
{
    double abscisse;
    double ordonnee;
};

struct cercle
{
    struct point centre;
    double rayon;
};
```

```
// definitions de variables
```

```
struct cercle c;

c.centre.abscisse = 12.5;
c.centre.ordonnee = 0.0;
c.rayon = 5.0;
// ou bien...

struct point p;
p.abscisse = 12.5;
p.ordonnee = 0.0;
c.centre = p;
c.rayon = 5.0;
```



Structures : initialisations

// definitions de types

```
struct point
{
    double abscisse;
    double ordonnee;
};
```

```
struct cercle
{
    struct point centre;
    double rayon;
};
```

// definitions avec initialisations

```
struct point p = {2.0, 3.0};
struct cercle c = {{2.0, 3.0}, 1.0};

// En C99, on peut aussi ecrire
struct cercle c1 = {p, 1.0};
```

Abréviations de types

- ▶ On peut donner un nom abrégé à un type avec `typedef`.

```
struct point
{
    double abscisse;
    double ordonnee;
};
```

```
typedef struct point Point;
```

```
// Maintenant on peut utiliser Point  
// comme abreviation de la structure
```

```
Point p;  
p.abscisse = 5.0;  
.....
```

Pointeurs et structures

- ▶ Si `p` est un pointeur sur une structure et `champ` un champ de la structure, `p->champ` désigne `(*p).champ`.
- ▶ Un champ d'une structure peut être un pointeur, qui pointe sur une structure de même type.

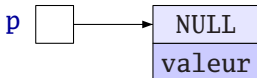
```
struct cell                cell *creer_cellule(int valeur)
{
    int valeur;           {
    struct cell *suivant; cell *p;
};                          p = (cell *)malloc(sizeof(cell));
                             p->valeur = valeur;
                             p->suivant = NULL;
typedef struct cell cell;   return p;
                             }

```

Pointeurs et structures

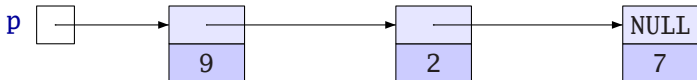
- ▶ Si `p` est un pointeur sur une structure et `champ` un champ de la structure, `p->champ` désigne `(*p).champ`.
- ▶ Un champ d'une structure peut être un pointeur, qui pointe sur une structure de même type.

```
struct cell                cell *creer_cellule(int valeur)
{
    int valeur;           {
    struct cell *suivant; cell *p;
};                          p = (cell *)malloc(sizeof(cell));
                             p->valeur = valeur;
                             p->suivant = NULL;
typedef struct cell cell;   return p;
                             }
```



Listes chaînées

- ▶ Cette construction permet d'implémenter les listes chaînées, chaque cellule de la liste contenant par exemple un entier.
- ▶ **Exemple** : la liste à 3 éléments (9,2,7) pourra être représentée par



- ▶ Écrire des fonctions implémentant :
 - ▶ la création d'une liste vide (transparent précédent),
 - ▶ le calcul de la longueur d'une liste,
 - ▶ l'ajout d'un élément en tête de liste,
 - ▶ la suppression de l'élément de tête s'il existe,
 - ▶ l'ajout d'un élément dans une liste supposée triée,
 - ▶ la suppression du 1er élément ayant une clé donnée,
 - ▶ le tri d'une liste selon les valeurs des clés.



Pointeurs sur fonctions

- ▶ Un **nom de fonction** est un pointeur constant sur la fonction.
- ▶ On peut définir des tableaux de pointeurs sur fonctions.
- ▶ On peut aussi passer des pointeurs sur fonctions en paramètre.
- ▶ `int (*f) (char *)` ; pointeur sur une fonction
retournant un `int`, dont l'argument
est un pointeur sur un `char`.
- ▶ Les pointeurs de fonctions peuvent être utilisés, par exemple, pour écrire des fonctions génériques.
- ▶ Exemple : fonction de tri d'un tableau de chaînes de caractères, prenant en argument :
 - ▶ le tableau,
 - ▶ un pointeur sur une fonction de comparaison de deux chaînes.

11– Compilation et modularité

- ▶ La compilation
- ▶ Étapes de la compilation
- ▶ Compilation de plusieurs fichiers
- ▶ Bibliothèques et en-têtes

La compilation (rappel)

- ▶ Le programmeur écrit un programme source C
- ▶ Il le traduit ensuite pour obtenir un exécutable

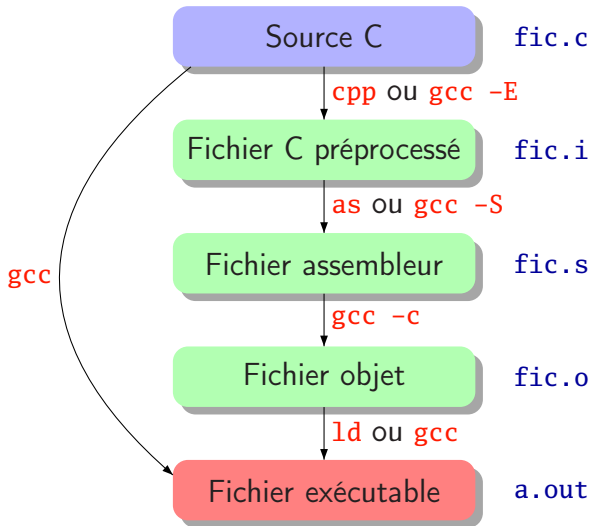
Programme source C

Compilateur

Fichier exécutable

- ▶ Une fois qu'on a compilé, on obtient un nouveau fichier, appelé **exécutable**, qui est l'application.
- ▶ L'exécutable dépend de la machine sur laquelle le programmeur veut exécuter le programme.
- ▶ \implies on doit compiler autant de fois qu'on veut créer d'exécutables pour des machines différentes.
- ▶ Voir le [premier cours](#) pour des rappels sur le compilateur **gcc**.

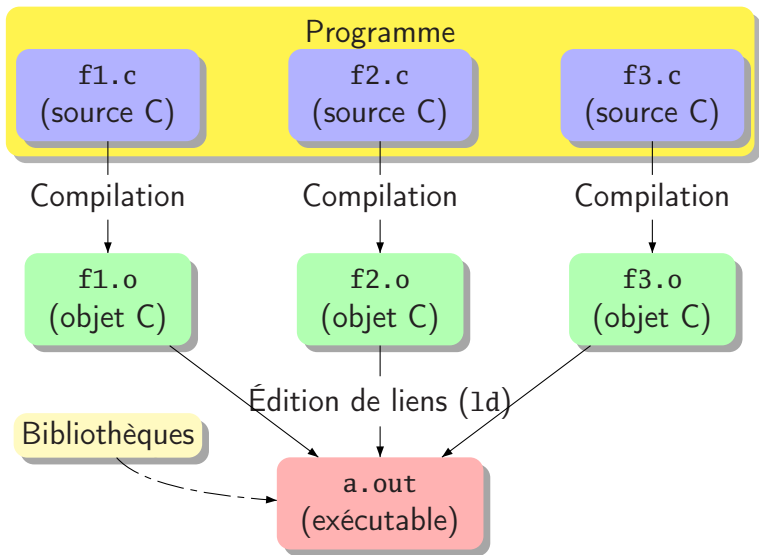
Étapes de la compilation



Pourquoi plusieurs étapes ?

- ▶ Chaque étape a une tâche précise et distincte.
- ▶ Le compilateur lui-même est **modulaire**.
- ▶ Un gros programme est réparti sur plusieurs fichiers sources.
 - ▶ Chaque fichier rassemble des fonctions travaillant « au même niveau ».
 - ▶ Si on modifie un fichier, on doit recréer son fichier objet (.o).
 - 😊 On n'est pas obligé de recréer les autres, qui existent de la précédente compilation.
- ▶ **Question** :
 - ▶ Peut-on utiliser une fonction (par ex.) définie dans un autre fichier ?
 - ▶ Réponse : Oui ! Mais il faut la déclarer.

Compilation de plusieurs fichiers



Bibliothèques vs. en-têtes

- ▶ Une bibliothèque contient des **définitions** (par ex. de fonctions).
 - ▶ La bibliothèque standard (`libc`) est utilisée par défaut.
 - ▶ Exemple : `printf()` est déjà écrite dans la bibliothèque standard.
 - ▶ L'option `-l` de `gcc` permet d'ajouter des bibliothèques.
-
- ▶ Une **déclaration** avertit le compilateur qu'on utilise une entité (comme une fonction) qui n'est pas définie dans le fichier compilé.
 - ▶ Elle doit être définie
 - ▶ Soit dans une bibliothèque avec laquelle on compile.
 - ▶ Soit dans un autre fichier de l'utilisateur.
 - ▶ Un fichier en-tête (`.h`), lu par `#include`, contient des **déclarations**.
 - ▶ La page de manuel d'une fonction fournit les en-têtes à inclure.



Les fichiers d'en-tête

- ▶ Nom des fichiers d'en-tête (headers) terminé par `.h`.
- ▶ Ils contiennent des
 - ▶ des « `#definitions` ».
 - ▶ des déclarations
 - ▶ de fonctions,
 - ▶ de variables,
 - ▶ de types
- ▶ Ils sont **pas** faits pour contenir des instructions.
- ▶ Se trouvent souvent dans le répertoire `/usr/include`.
- ▶ Sont lus par une directive `#include <xxx.h>`.
- ▶ `man 3 printf` donne l'en-tête avec la déclaration de `printf()`.
- ▶ On peut inclure un `fic.h` personnel par `#include "fic.h"`.
- ▶ L'option `-I` ajoute des répertoires de recherche de fichiers en-tête.

- ▶ Étape de la création d'exécutable qui :
 - ▶ réunit un ensemble de fichiers objets (.o) pour créer un fichier,
 - ▶ pour cela, on doit choisir et fixer l'adresse (dans l'exécutable) de chaque variable ou fonction du programme.
- ▶ Pour que l'édition de liens réussisse, il faut que
 - ▶ une fonction appelée `main` soit définie.
 - ▶ toute fonction utilisée dans le programme soit définie, soit une fois dans l'un des fichiers source soit dans la bibliothèque standard.
 - ▶ idem pour les variables.



Bibliothèques

- ▶ Une bibliothèque est un ensemble de fonctions déjà précompilées.
- ▶ La bibliothèque standard, toujours chargée par `gcc`, s'appelle `libc`.



Bibliothèques

- ▶ Une bibliothèque est un ensemble de fonctions déjà précompilées.
- ▶ La bibliothèque standard, toujours chargée par `gcc`, s'appelle `libc`.
- ▶ L'option `-ltoto` charge la bibliothèque du fichier `libtoto.so`.
- ▶ **Exemple** On compile avec `-lm` pour charger la bibliothèque `libm` contenant les fonctions mathématiques usuelles (trigonométrie,...)



Bibliothèques

- ▶ Une bibliothèque est un ensemble de fonctions déjà précompilées.
- ▶ La bibliothèque standard, toujours chargée par `gcc`, s'appelle `libc`.
- ▶ L'option `-ltoto` charge la bibliothèque du fichier `libtoto.so`.
- ▶ **Exemple** On compile avec `-lm` pour charger la bibliothèque `libm` contenant les fonctions mathématiques usuelles (trigonométrie,...)
- ▶ On peut créer ses propres bibliothèques avec l'option `-shared`.

Bibliothèques

- ▶ Une bibliothèque est un ensemble de fonctions déjà précompilées.
- ▶ La bibliothèque standard, toujours chargée par `gcc`, s'appelle `libc`.
- ▶ L'option `-ltoto` charge la bibliothèque du fichier `libtoto.so`.
- ▶ **Exemple** On compile avec `-lm` pour charger la bibliothèque `libm` contenant les fonctions mathématiques usuelles (trigonométrie,...)
- ▶ On peut créer ses propres bibliothèques avec l'option `-shared`.
- ▶ `gcc` cherche les bibliothèques dans des répertoires prédéfinis.
- ▶ On peut ajouter un répertoire à la liste avec l'option `-L`.
- ▶ **Exemple** `gcc -L/repertoire/ou/est/ma/bibliotheque fichier.c`

Bibliothèques

- ▶ Une bibliothèque est un ensemble de fonctions déjà précompilées.
- ▶ La bibliothèque standard, toujours chargée par `gcc`, s'appelle `libc`.
- ▶ L'option `-ltoto` charge la bibliothèque du fichier `libtoto.so`.
- ▶ **Exemple** On compile avec `-lm` pour charger la bibliothèque `libm` contenant les fonctions mathématiques usuelles (trigonométrie,...)
- ▶ On peut créer ses propres bibliothèques avec l'option `-shared`.
- ▶ `gcc` cherche les bibliothèques dans des répertoires prédéfinis.
- ▶ On peut ajouter un répertoire à la liste avec l'option `-L`.
- ▶ **Exemple** `gcc -L/repertoire/ou/est/ma/bibliotheque fichier.c`
- ▶ Certaines bibliothèques sont nécessaires pour l'exécution.
La variable shell `LD_LIBRARY_PATH` indique où les chercher.

12– Compléments : instructions

- ▶ Instruction switch
- ▶ Instruction do...while
- ▶ Instruction continue
- ▶ Instruction break
- ▶ Expression b?e1:e2
- ▶ Expression e1,e2

Instruction switch

- ▶ Permet d'exécuter des instructions selon la valeur d'une expression,

```
switch (<expression>
{
  case <expr_constante_1>:
    <suite instructions 1>
    .....
  case <expr_constante_n>:
    <suite instructions n>
  default:
    <suite instructions n+1>
}
```

- ▶ L'expression est comparée aux expressions constantes, dans l'ordre.
- ▶ En cas d'égalité, la suite d'instructions correspondante est exécutée puis la comparaison reprend avec l'expression constante suivante.
- ▶ L'instruction `break` permet de sortir du switch.

Instruction switch : exemple

```
int main(int argc, char *argv[])
{
    switch (argc)
    {
        case 1:
            printf("Commande sans argument\n");
            break;
        case 2:
            printf("Commande avec un argument\n");
            break;
        default:
            fprintf(stderr, "Usage : %s [arg. optionnel]\n", argv[0]);
            exit (EXIT_FAILURE);
            break;
    }
    //...
```


Instruction `do...while`

- ▶ Permet de répéter une séquence, suivant la valeur d'une expression.
- ▶ Contrairement au comportement de la boucle `while`, la séquence d'instructions est `d'abord` exécutée, `puis` l'expression est testée pour éventuellement recommencer.

```
do  
    <instruction ou bloc>  
while (<expression>;
```

1. L'`<instruction ou bloc>` est exécuté.
2. L'expression est évaluée.
 - a. Si elle est fausse, on sort de la boucle `do...while`.
 - b. Si elle est vraie, on revient en 1.

Instruction `do...while` : exemple

- ▶ Demander à l'utilisateur d'entrer un entier compris entre 1 et 10.

```
int a;
do
{
    printf("Entrez un nombre entre 1 et 10\n");
    a = lire_entier();
}
while (a < 1 || a > 10);
```

- ▶ **Exercice** Réécrire ce bout de programme avec une boucle `while`.

Rupture dans les boucles : continue

- ▶ L'instruction `continue` modifie le déroulement de la boucle la plus interne qui la contient.
- ▶ Elle fait passer à l'itération suivante, sans finir le corps de la boucle.

```
int main(void)
{
    printf("Valeurs de 1/(x-k) pour 0 <= x <= N\nEntrez k puis N\n");
    int k = lire_entier();
    int N = lire_entier();

    for (double x = 0.0; x <= N; x++)
        if (x == k)
            continue;
        else
            printf("x = %g, 1/(x-%d)=%g\n", x, k, 1/(x-k));
}
```

Instruction continue et boucles

⚠ Des boucles qui se comportent habituellement de même façon peuvent avoir un comportement différent avec `continue`.



```
for (double x = 0; x <= N; x++)  
  
    if (x == k)  
        continue;  
    else  
        printf(  
            "x = %g, 1/(x-%d)=%g\n",  
            x, k, 1/(x-k)  
        );
```



```
double x = 0.;  
while (x <= N)  
{  
    if (x == k)  
        continue;  
    else  
        printf(  
            "x = %g, 1/(x-%d)=%g\n",  
            x, k, 1/(x-k)  
        );  
    x++;  
}
```



Rupture dans les boucles : `break`

- ▶ L'instruction `break` dans une boucle provoque la `sortie` de la boucle la plus interne qui la contient.
- ▶ **Exemple** : calcul de l'indice minimum d'une case contenant 0 dans les n premières cases d'un tableau :

```
for (i = 0; i < n; i = i + 1)
    if (tab[i] == 0)
        break;
if (i == n)
    printf("Pas de 0 dans les %d 1eres cases\n", n);
else
    // tab[i] est le 1er 0
```



Expression $b ? e_1 : e_2$

- ▶ Dans une expression $b ? e_1 : e_2$, e_1 et e_2 sont des expressions de types compatibles.
- ▶ Pour calculer l'expression :
 - ▶ On évalue b .
 - ▶ Si b est vrai :
 - ▶ l'expression e_2 n'est pas évaluée.
 - ▶ l'expression e_1 est évaluée et sa valeur est celle de $b ? e_1 : e_2$.
 - ▶ Si b est faux :
 - ▶ l'expression e_1 n'est pas évaluée ;
 - ▶ l'expression e_2 est évaluée et sa valeur est celle de $b ? e_1 : e_2$.
- ▶ Exemple : `return x > y ? x : y` retourne le maximum de x et y .



Expression e1,e2

- ▶ Pour évaluer l'expression `e1,e2`
 - ▶ on évalue d'abord `e1`,
 - ▶ on évalue ensuite `e2`, qui donne le type et la valeur de l'expression globale `e1,e2`.
- ⚠ Assigner/modifier la valeur de `e1,e2` est **non portable**.
- ▶ Utilisation dans les boucles :

```
int i,j;  
for (i=-10,j=20; i < j; i++,j--)  
    ...
```