Programmation C

T. Place, M. Zeitoun

Université Bordeaux

Année 2026–2027, Licence 2 MI & CMI OSIA

Plan de ce diaporama

Dans cette partie…

1. Informations pratiques
   - Organisation de l'UE
   - Modalités de contrôle
   - Ressources

Organisation de l'UE

- **Amphi :** $2 \times$ 1h20 $+ 3 \times$ 1h20 cours supplémentaires de révision en option.
- **Cours-TD et TD machines** 12 semaines, à peu près $2 \times$ 1h20 chaque semaine.
- **TP : séances de 1h20 ou 2$\times$ 1h20.**
- **Web :** page du cours, moodle pour que vous déposiez du code.
- **Tutorat :** informations page Moodle.

Évaluation

- Session 1 : 50%CC + 50% examen.
- Session 2 : Max(Examen S2, 50%CC + 50% examen).

Épreuve CC	Durée	Période prévue	Coefficient
CC : Test	20 à 30 minutes	septembre	10%
CC : TP noté 1	1h20 à 1h30	octobre	30%
CC : TP noté 2	2h40 à 3h	novembre	60%

L'examen est aussi un TP noté (2h40 ou 3h).

Évaluation

- Session 1 : 50%CC + 50% examen.
- Session 2 : Max(Examen S2, 50%CC + 50% examen).

Épreuve CC	Durée	Période prévue	Coefficient
CC : Test	20 à 30 minutes	septembre	10%
CC : TP noté 1	1h20 à 1h30	octobre	30%
CC : TP noté 2	2h40 à 3h	novembre	60%

L'examen est aussi un TP noté (2h40 ou 3h).

- TP notés
  - LLM, outils IA **interdits**, de même qu'internet.
  - Flux vidéo des sessions enregistrés.
  - Correction seulement après vérification de chaque écran.
  - Retour personnalisé à chaque étudiant ou étudiante.

Évaluation

- Session 1 : 50%CC + 50% examen.
- Session 2 : Max(Examen S2, 50%CC + 50% examen).

Épreuve CC	Durée	Période prévue	Coefficient
CC : Test	20 à 30 minutes	septembre	10%
CC : TP noté 1	1h20 à 1h30	octobre	30%
CC : TP noté 2	2h40 à 3h	novembre	60%

L'examen est aussi un TP noté (2h40 ou 3h).

- Projet
  - Pas d'évaluation cette année (25% de zéros en 2025–26).
  - Mais 2/3 des TP portent sur la réalisation d'un projet (jeu).

Ressources

- Forum moodle. Réponse sous 24h, aide mutuelle ⇒ Utilisez-le !
- LLMs, type Claude ou ChatGPT. Copilot pas recommandé.
- FAQ (Foire Aux Questions)
  - `http://c-faq.com/`
  - `https://www.levenez.com/lang/c/faq/` (FR, 2012).
- Pages de manuel pour la bibliothèque standard.
  - Section 3. Exemple : `man 3 printf`.

Références conseillées

- Présentation claire et complète du langage C

  A. Braquelaire.
  Méthodologie de la programmation en langage C.
  Norme C99, API POSIX. Dunod, Paris, $4^{\text{ème}}$ éd. 2005.

- Référence du langage.

  S. P. Harbison et G. L. Steele Jr.
  C : a reference manual (5th ed.).
  Prentice-Hall, Upper Saddle River, NJ, USA, 2002.

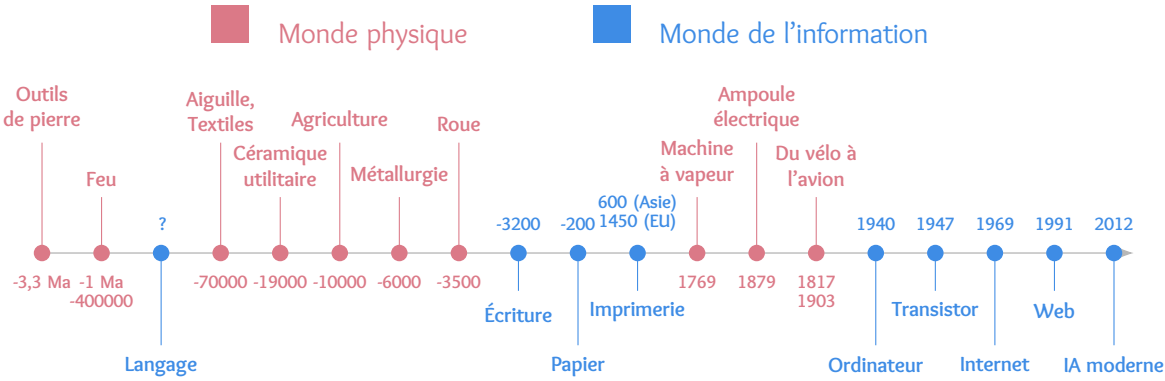
- Norme C23, pour rechercher un point précis (courage ☹).

  Draft de la norme C23.
  `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf`.

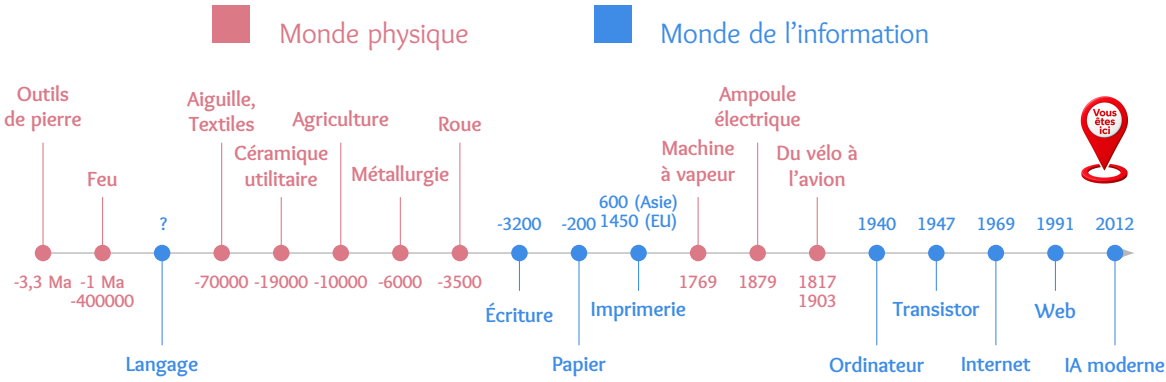
Dans cette partie…

2. L'IA dans ce cours
   - Une évolution profonde
   - Impact sur l'apprentissage de l'informatique
   - Objectifs et clés pour réussir

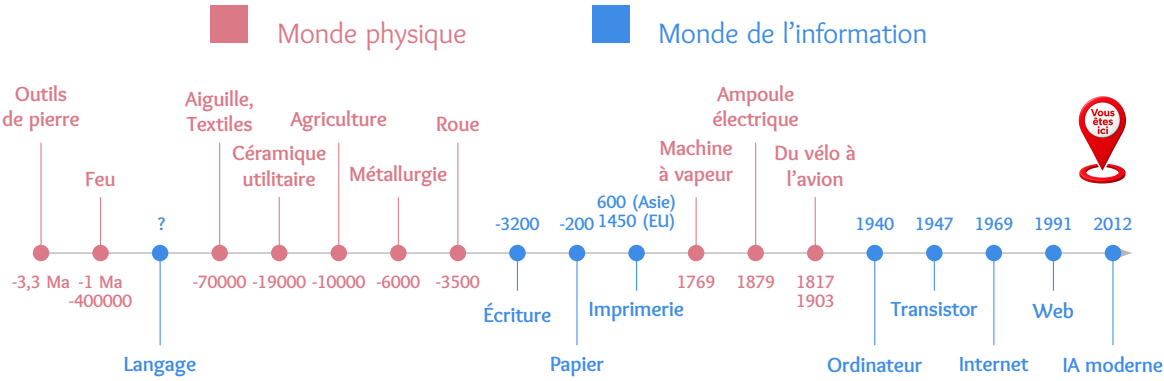
Ruptures « technologiques »



Ruptures « technologiques »

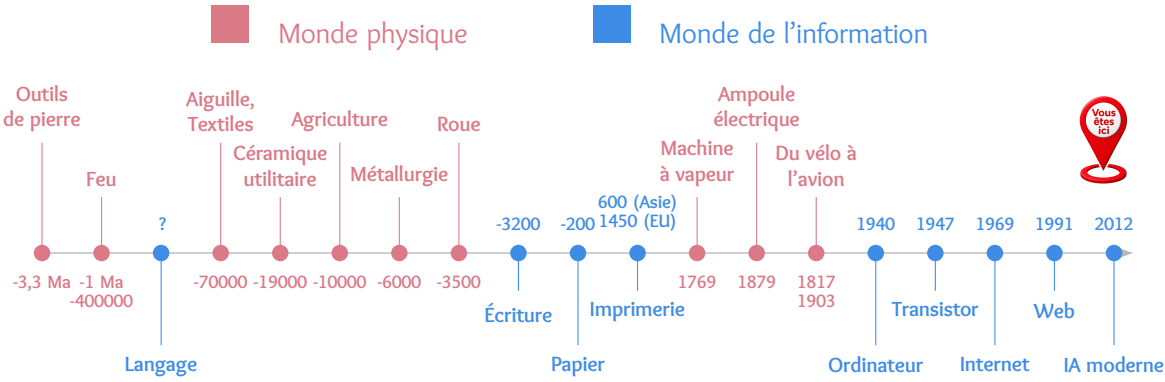


Ruptures « technologiques »



Dates approximatives… Avec échelle de temps linéaire :

Ruptures « technologiques »



Dates approximatives… Avec échelle de temps linéaire :

Avènement de l'IA = évolution majeure $\Longrightarrow$ **avantages + risques.**

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité
Information	Accès immédiat	Trop de données Manipulation, biais, deepfakes

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité
Information	Accès immédiat	Trop de données Manipulation, biais, deepfakes
Emploi	Augmentation productivité	Destruction d'emplois

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité
Information	Accès immédiat	Trop de données Manipulation, biais, deepfakes
Emploi	Augmentation productivité	Destruction d'emplois
Éducation	Tutorat personnalisé permanent	

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité
Information	Accès immédiat	Trop de données Manipulation, biais, deepfakes
Emploi	Augmentation productivité	Destruction d'emplois
Éducation	Tutorat personnalisé permanent	Dépendance, erreurs et biais Paresse, perte compétences

Quelques impacts de l'IA

Domaine	Avantages	Risques
Environnement	Optimisation consommation Aide à la recherche	Surconsommation (énergie, eau, métaux, CO_2, etc.)
Médecine	Aide diagnostic & recherche	Biais de diagnostics Erreurs et responsabilité
Information	Accès immédiat	Trop de données Manipulation, biais, deepfakes
Emploi	Augmentation productivité	Destruction d'emplois
Éducation	Tutorat personnalisé permanent	Dépendance, erreurs et biais Paresse, perte compétences

Liste non exhaustive (sécurité, vie privée, gouvernance, société, etc.) !

Est-il encore pertinent d'enseigner l'informatique ?

- Une IA actuelle réussit tout exercice de niveau licence sans problème, instantanément.

Est-il encore pertinent d'enseigner l'informatique ?

- Une IA actuelle réussit tout exercice de niveau licence sans problème, instantanément.
- Dans ce cas, quel est l'intérêt d'apprendre à programmer ?

Est-il encore pertinent d'enseigner l'informatique ?

- Une IA actuelle réussit tout exercice de niveau licence sans problème, instantanément.
- Dans ce cas, quel est l'intérêt d'apprendre à programmer ?

LOISIRS Article du Gorafi, Août 2026

Elle commence bêtement des cours de dessin alors qu'une IA peut mieux faire

Julie Tissot, 29 ans, a toujours eu un goût certain pour le dessin et la peinture, mais n'a jamais osé prendre des cours. Aujourd'hui, elle se décide enfin, et s'est inscrite à un atelier près de chez elle, dans le quartier des Beaux-Arts. Une décision ridicule, puisqu'elle va passer des heures à apprendre à dessiner des corps humains, des paysages, les ombres et les perspectives, alors que n'importe quelle IA peut le faire en une instruction. Une décision incompréhensible selon Benjamin Lavaud, expert en intelligence artificielle : « Ici, on frôle l'absurde. Passer autant de temps à gribouiller sur un carnet alors que Chat GPT ou Gemini fait ça en 10 secondes, c'est lunaire. Il y a vraiment des gens qui ont du temps à perdre, on croît rêver. »

Lever le malentendu sur la nature de l'informatique

- Faire de l'informatique $\neq$

Lever le malentendu sur la nature de l'informatique

- Faire de l'informatique $\neq$ écrire du code.

Lever le malentendu sur la nature de l'informatique

- Faire de l'informatique $\neq$ écrire du code.
- C'est une science basée sur le raisonnement et la logique.

  « ***La science, c'est de la logique, et la logique, c'est merveilleux.*** »

Yves Coppens

Lever le malentendu sur la nature de l'informatique

- Faire de l'informatique $\neq$ écrire du code.
- C'est une science basée sur le raisonnement et la logique.

  « ***La science, c'est de la logique, et la logique, c'est merveilleux.*** »

Yves Coppens

Vos compétences à développer :

- Lecture et compréhension de consignes,
- Raisonnement logique,
- Expression en langage naturel ou formel (français, anglais, maths, code),
- Relecture, compréhension et amélioration des raisonnements/codes.
- Interprétation et critique des résultats fournis par une IA.
- …

Lever le malentendu sur la nature de l'informatique

- Faire de l'informatique $\neq$ écrire du code.
- C'est une science basée sur le raisonnement et la logique.

  « ***La science, c'est de la logique, et la logique, c'est merveilleux.*** »

Yves Coppens

Vos compétences à développer :

- Lecture et compréhension de consignes,
- Raisonnement logique,
- Expression en langage naturel ou formel (français, anglais, maths, code),
- Relecture, compréhension et amélioration des raisonnements/codes.
- Interprétation et critique des résultats fournis par une IA.
- …

Cela demande une **base théorique solide**. La capacité à écrire du code vient en bonus.

Comment utiliser et ne pas utiliser l'IA ?

- ✔ Ne pas utiliser l'IA.
- ✔ Faire vérifier son programme en cas de doute.
- ✔ Demander une explication sur une question qu'on n'arrive pas à faire.
- ✗ Copier-coller une question, copier-coller la réponse.

Comment utiliser et ne pas utiliser l'IA ?

- ✔ Ne pas utiliser l'IA.
- ✔ Faire vérifier son programme en cas de doute.
- ✔ Demander une explication sur une question qu'on n'arrive pas à faire.
- ✗ Copier-coller une question, copier-coller la réponse.

Le cerveau ne se forme que par l'entraînement, pas en regardant les autres s'entraîner.

Il est donc important de :

- ne pas déléguer la **difficulté** à l'IA,
- programmer souvent **soi-même**, en particulier au CREMI.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Objectifs du cours

- Introduction à la programmation en langage C.
- Pouvoir comprendre, évaluer, 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
    $\Rightarrow$ facilement modifiable et extensible.
  - Efficace — Algorithmes, implémentation rapides.
  - Lisible — Commentaires, indentation, noms bien choisis.

Dans cette partie…

3. Langage C : présentation
   - Caractéristiques du langage C
   - Bref historique : de BCPL à C23
   - Pourquoi des normes ?

Caractéristiques du langage C

- **Portable** — Un programme C conforme aux normes est utilisable sur une grande variété de machines.
  Normes : ANSI, C99, C11, C18, C23, API POSIX,...
- **Puissant** — Nombreux domaines d'application
  OS (Unix), réseau, BD, graphique, IA...
- **Efficace** — Permet de développer des programmes rapides.
- **Haut niveau** — Détails hardware cachés au programmeur.
- **Souple** — Très permissif, accès à la mémoire.
  $\Longrightarrow$ 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, C99, C11, C18, C23, API POSIX,...
- **Puissant** — Nombreux domaines d'application
  OS (Unix), réseau, BD, graphique, IA...
- **Efficace** — Permet de développer des programmes rapides.
- **Haut niveau** — Détails hardware cachés au programmeur.
- **Souple** — Très permissif, accès à la mémoire.
  $\Longrightarrow$ 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, C99, C11, C18, C23, API POSIX,...
- **Puissant** — Nombreux domaines d'application
  OS (Unix), réseau, BD, graphique, IA...
- **Efficace** — Permet de développer des programmes rapides.
- **Haut niveau** — Détails hardware cachés au programmeur.
- **Souple** — Très permissif, accès à la mémoire.
  $\Longrightarrow$ 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, C99, C11, C18, C23, API POSIX,...
- **Puissant** — Nombreux domaines d'application
  OS (Unix), réseau, BD, graphique, IA...
- **Efficace** — Permet de développer des programmes rapides.
- **Haut niveau** — Détails hardware cachés au programmeur.
- **Souple** — Très permissif, accès à la mémoire.
  $\Longrightarrow$ 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, C99, C11, C18, C23, API POSIX,...
◦	Puissant	Nombreux domaines d'application OS (Unix), réseau, BD, graphique, IA...
◦	Efficace	Permet de développer des programmes rapides.
◦	Haut niveau	Détails hardware cachés au programmeur.
◦	Souple	Très permissif, accès à la mémoire. $\Longrightarrow$ facile à aborder, difficile à bien maîtriser.

Bref historique : de BCPL à C23

1967	Langage BCPL	(Martin Richards)
1970	Langage B Pas de types, manipulation de mots machine $\Longrightarrow$ programmes non portables Structures de contrôle, pointeurs, récursivité	(Ken Thompson)
1972	1er compilateur C	(Dennis Ritchie)
1978	1ère spécification publique du C	
1989	Norme « ANSI » $\Longrightarrow$ on parle de C « ISO » ou « ANSI »	
1999	Norme ISO/IEC 9899 : C99.	
20xx	ISO/CEI 9899 C11, C18, C23 (norme actuelle).	

Hall of fame

Brian Kernighan

Co-auteur premier livre sur C

Co-concepteur de `awk`

Dennis Ritchie

Prix Turing 1983

Co-concepteur de C

Ken Thompson

Prix Turing 1983

Co-concepteur C, Unix, Go, UTF-8

Photo de Dennis Richie ©Bell Labs

Photos de Brian Kernighan et Ken Thompson ©Peter Adams, Projet “Faces of Open Source”

Pourquoi des normes ?

- Plusieurs vendeurs $\Rightarrow$ plusieurs versions de C $\Rightarrow$ extensions incompatibles entre elles.
- Norme : fournit une signification précise et unique des programmes.
- Un programme respectant une norme est portable.
  Déployable sur toute architecture ayant un compilateur implémentant la norme.
- Les programmes ne respectant pas la norme C23 seront considérés incorrects.

Dans cette partie…

4. Démarrage rapide en C
   - Du source à l'exécutable
   - Variables
   - Tests
   - Boucles
   - Fonctions
   - Quelques différences avec python
   - Bonnes habitudes de programmation

Premier exemple : Hello, world !

Un premier programme source

```c
#include <stdio.h>  // Pour la déclaration de printf.

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

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.
- Les instructions sont exécutées séquentiellement à partir de la première de `main`.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.
- Les instructions sont exécutées séquentiellement à partir de la première de `main`.
- Le `;` (point-virgule) est un terminateur d'instruction.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.
- Les instructions sont exécutées séquentiellement à partir de la première de `main`.
- Le `;` (point-virgule) est un terminateur d'instruction.
- Les commentaires sont placés entre `/*` et `*/`, ou de `//` à la fin de ligne.

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.
- Les instructions sont exécutées séquentiellement à partir de la première de `main`.
- Le `;` (point-virgule) est un terminateur d'instruction.
- Les commentaires sont placés entre `/*` et `*/`, ou de `//` à la fin de ligne.
- Le programmeur indique les groupes d'instructions avec des accolades `{…}` (l'indentation aide juste à la lisibilité).

Premières remarques

- Un programme C contient une suite de définitions de types, variables et fonctions.
- Chaque définition de fonction peut contenir
  - des définitions de variables.
  - des instructions.
- L'une des fonctions s'appelle `main`.
- Les instructions sont exécutées séquentiellement à partir de la première de `main`.
- Le `;` (point-virgule) est un terminateur d'instruction.
- Les commentaires sont placés entre `/*` et `*/`, ou de `//` à la fin de ligne.
- 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.

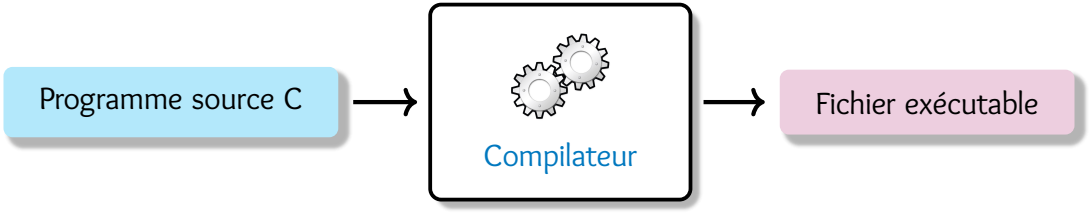
  Il est nécessaire de le traduire en langage machine.

- La phase de traduction

  du langage C
  vers
  le langage machine

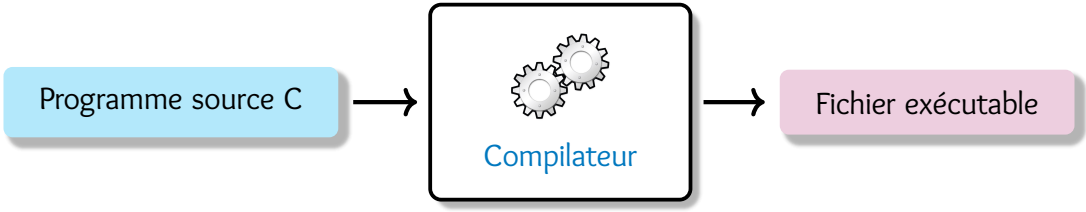
  s'appelle la compilation.

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



Compiler pour créer l'exécutable

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



- Une compilation réussie crée un fichier : l'exécutable (l'application).
- L'exécutable dépend de la machine sur laquelle on veut exécuter le programme.

  $\Longrightarrow$ on doit compiler une fois par architecture d'utilisation.
- Si le compilateur rencontre une erreur,
  - Il la signale mais 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`.

Exemple de compilation

```
$ gcc -Wall -std=c23 -o HelloWorld HelloWorld.c
$ ./HelloWorld
```

compile le fichier source `HelloWorld.c` 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`.

Exemple de compilation

```
$ gcc -Wall -std=c23 -o HelloWorld HelloWorld.c
$ ./HelloWorld
```

compile le fichier source `HelloWorld.c` 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`.

Exemple de compilation

```
$ gcc -Wall -std=c23 -o HelloWorld HelloWorld.c
$ ./HelloWorld
```

compile le fichier source `HelloWorld.c` et crée l'exécutable `HelloWorld`.

- `-Wall` : `gcc` donne les principaux avertissements.
- `-std=c23` : Conformité à la norme C23.

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`.

Exemple de compilation

```
$ gcc -Wall -std=c23 -o HelloWorld HelloWorld.c
$ ./HelloWorld
```

compile le fichier source `HelloWorld.c` et crée l'exécutable `HelloWorld`.

- `-Wall` : `gcc` donne les principaux avertissements.
- `-std=c23` : Conformité à la norme C23.
- `-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`.

Exemple de compilation

```
$ gcc -Wall -std=c23 -o HelloWorld HelloWorld.c
$ ./HelloWorld
```

compile le fichier source `HelloWorld.c` et crée l'exécutable `HelloWorld`.

  - `-Wall` : `gcc` donne les principaux avertissements.
  - `-std=c23` : Conformité à la norme C23.
  - `-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`

Option	Signification
`-std=c23`	Compile avec la norme C23
`-o`	Permet de nommer le fichier exécutable
`-c`	Construit juste le fichier objet `.o`, pas l'exécutable
`-Wall`	Active des avertissements classiques
`-Wextra`	Ajoute d'autres avertissements non couverts par -Wall
`-pedantic`	Signale les constructions non conformes à la norme sélectionnée
`-Werror`	Transforme tout avertissement en erreur
`-g`	Permet d'utiliser ultérieurement le débogueur

Ces options **aident** le programmeur à détecter les erreurs.
Pour compiler en TP, il suffira de taper `make`, qui activera ces options et d'autres.

Variables

- 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 `int` (entier).

  `int x, y;` déclaration de deux variables `x` et `y` de type `int` (entier).

Exemple : variables

Utilisation de variables

```c
#include <stdio.h>       // pour la déclaration de printf()
int lire_entier(void);   // déclatation de lire_entier()

int main(void) {         // définition de main()
  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.

Exemple : variables (2)

Variables : affichage d'un nombre et son carré

```c
#include <stdio.h>       // pour la déclaration de printf()
int lire_entier(void);   // déclaration de lire_entier()

int main(void) {         // définition de main()
  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

```c
#include <stdio.h>
int lire_entier(void);

/* Affiche la valeur absolue d'un entier */
int main(void) {
  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

```c
#include <stdio.h>
int lire_entier(void);

/* Affiche le maximum entre deux entiers */
int main(void) {
  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

```c
#include <stdio.h>
int lire_entier(void);

/* Affiche la somme des entiers de 1 à n */
int main(void) {
  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

Premier programme source, deuxième version

```c
#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

```
// Renvoie la somme des entiers de 1 à n
int somme(int n) {
  int s = 0;

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

  return s;
}
```

Exemple 2 : fonctions, suite

```c
#include <stdio.h>

int lire_entier(void);

/* 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));
}
```

Quelques différences avec python

- On doit d'abord compiler un source C avant l'exécution.
- Les variables doivent être définies/déclarées avant d'être utilisées.
- Les fonctions doivent être définies/déclarées avant d'être utilisées.
- Un ; (point-virgule) termine chaque déclaration et chaque instruction simple.
- Les instructions sont groupées par les accolades `{...}`, pas par l'indentation.
- Certaines constructions sont ressemblantes mais cependant différentes.
  Exemple : boucle `for`.

Bonnes habitudes de programmation

- La compilation doit se faire sans erreur ni avertissement (warning).
- Une mauvaise indentation n'est pas une erreur ($\neq$ python).
  ⚠ Les programmes incorrectement formatés seront cependant sanctionnés.
  💡 Configurer et utiliser un formateur automatique (`clang-format`).
- Les noms des fonctions, variables, etc. doivent être
  - lisibles,
  - pertinents,
  - cohérents (conventions uniformes).
- Les programmes doivent être commentés.

Dans cette partie...

5. 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 `while`
- Boucle `for`

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 du compilateur).*
  - 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` : pour représenter des entiers éventuellement négatifs.
  - La taille de l'intervalle d'entiers réprésenté dépend de la machine et tend à augmenter (un nombre de type `int` est souvent codé, actuellement, sur 4 octets).

  $\Rightarrow$ Ne pas faire d'hypothèse sur la taille d'un `int` : c'est `sizeof{int}` bytes.

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

- **`int`** : pour représenter des entiers éventuellement négatifs.
  - La taille de l'intervalle d'entiers réprésenté 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`** : c'est **`sizeof`**`{`**`int`**`}` bytes.

- **`double`** : pour représenter les nombres flottants en double précision, norme IEEE 754.

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

- **`int`** : pour représenter des entiers éventuellement négatifs.
  - La taille de l'intervalle d'entiers réprésenté 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`** : c'est **`sizeof`**`{`**`int`**`}` bytes.

- **`double`** : pour représenter les nombres flottants en double précision, norme IEEE 754.

- **`char`** : permet de représenter le jeu de caractères de base.
  - Ce sont des entiers que `printf()` peut afficher sous leur forme de caractères.
  - Les constantes sont entre *quotes* : `'X'`.

Ne pas confondre

- `1` : représente l'entier 1.
- `'1'` : représente le caractère 1, dont la valeur entière est souvent 49,
- `"1"` : une chaîne de caractères (qui ne contient qu'un seul caractère : le caractère `'1'`).

Type `double` : un exemple

```c
#include <stdio.h>
double lire_flottant(void);

/* Calcul de la surface du cercle de rayon r */
int main(void) {
  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

```c
#include <stdio.h>
char lire_caractere(void);

/* Affiche le caractere suivant dans le code utilisé (souvent ASCII) */
int main(void) {
  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 C18,
- Il faut ajouter la directive (inutile en C23) :

```
#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 comportement sur entiers et flottants :
  - Si les deux opérandes sont entiers, le résultat est la division entière.
  - Si au moins un opérande est flottant, le résultat est la division en 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 $(\texttt{x} < \texttt{y}) == (\texttt{z} > \texttt{t})$

- En C, l'évaluation des opérateurs `&&` et `||` est « paresseuse ».
  - On évalue de gauche à droite et on s'arrête dès le résultat connu.
  - **Exemple :** `(b != 0 && a/b == c)` ne fait jamais de division par 0.

Affectation

- 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 : exemple

```
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 à a, b, ou c.
```

Tests

- 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`

Instruction `if`

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

- L'expression est évaluée. Si elle est
  - non nulle, le test réussit : `<instruction ou bloc>`$_{vrai}$ est exécuté.
  - nulle, le test échoue : on passe à l'instruction qui suit `<instruction ou bloc>`$_{vrai}$, qu'on n'exécute pas.
- Pour exécuter plusieurs instructions si le test réussit, on les met dans un bloc `{...}`.

Instruction `if` : exemple

```c
#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.
- Pour exécuter plusieurs instructions sous le **`if`** ou le **`else`**, on utilise un bloc `{...}`.

Instruction `if...else` : exemple

```c
#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 `while`

Instruction `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

```c
#include <stdio.h>

void afficher_puissances(int k, int nmax) {
  int i = 1; /* définition et 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;
}
```

Boucle `for`

Instruction `for`

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

- `<expr1>` est une expression d'initialisation exécutée une seule fois en début de boucle.
- `<expr2>` est un test (d'arrêt) exécuté avant chaque itération (même la première).
- `<expr3>` est une expression exécutée à la fin de chaque itération.

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

```
for ( ; ; ) // Boucle infinie !
  ;
```

Boucle `for` : exemple

```c
#include <stdio.h>

/* Affiche la somme des nombres impairs entre n et m inclus */
void afficher_somme_impairs(int n, int m) {
  // Initialisation.
  // Pourquoi ne peut-elle pas être faite dans le for ?
  int res = 0;

  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);
}
```

Que se passe-t-il si $n = 3, m = 4$? Si $n = 4, m = 5$?

Boucle `for` : exemple 2

```c
#include <stdio.h>

// Affiche les 26 caractères après 'A', en les supposant consécutifs.
void afficher_majuscules(void) {
  for (char x = 'A'; x <= 'Z'; x = x + 1) {
    printf("%c ", x);
  }
  printf("\n");
}
```

Que se passe-t-il si on remplace `"%c"` par `"%d"` ?

Boucle `for` : exemple 3

```c
#include <stdio.h>
int lire_entier(void);

// Calcule la somme de n entiers entrés au clavier
int somme_suite_entiers(int n) {
  int k;
  int somme = 0;

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

Dans cette partie…

6. Les variables

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

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 l'emplacement de la définition dans le source C.
- Exemple de définition :

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

Identificateurs

Le nom d'une variable est un identificateur.

- Commence par une lettre `a,...,z, A,...,Z` ou `_`.
- Peut contenir les caractères `a,...,z, A,...,Z, 0,...,9,_`.
- 31 caractères significatifs (63 pour variables locales et fonctions statiques).
- Les compilateurs actuels différencient majuscules et minuscules.
- La norme précise des limitations (rarement suivies).

⚠ Ne pas utiliser `_` en début d'identificateur.

Portée

- 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 definie dans un autre fichier, en donnant son type.
- Exemple de déclaration :

```
extern int x;
```

Variables locales

- 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.
- Chaque bloc est délimité par `{` et `}`.
- Chaque bloc contient ses définitions de variables, instructions et sous-blocs.

Portée d'une variable : définit **où** on peut l'utiliser.

- La portée d'une variable locale est limitée :
  - au bloc dans laquelle elle est définie, après sa définition,
  - aux sous-blocs de ce bloc se trouvant après sa définition.
- La position des définitions de variables dans le programme influe donc sur leur portée.
- En C, 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;          // Portée de x
void f(int a)
{
   int y;
   /* ..... */
   {
      int z;
      /* ..... */
   }
   /* ..... */
}

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

Portée : exemple

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

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

Portée : exemple

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

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

Portée : exemple

```
int x;
void f(int a)
{
   int y;                /* Portée 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;                    /* Portée de t */
void g(void)
{
   /* ..... */
}
```

Masquage

- 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

```c
#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
}
```

Définitions et déclarations

- Une variable doit être définie ou déclarée avant d'être utilisée.
- Une déclaration associe un type avec un nom de variable.
- La définition, en plus :
  - demande l'allocation mémoire pour la variable,
  - donne une valeur initiale,
  - doit être unique.
- Source de confusion : déclarations et définitions ont parfois la même forme.
  - Le mot-clé `extern` force une déclaration.

```
extern int x;
```

  - Une initialisation explicite force définition.

```
int x = 0;
```

Déclarations de variables globales

- Une déclaration de variable globale « 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.
- Ces 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.

Allocation

- 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 de 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

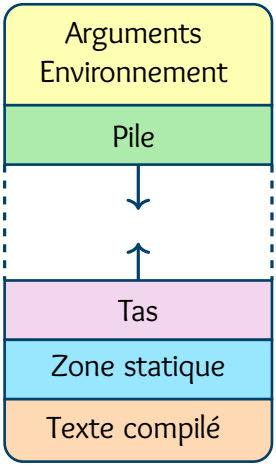
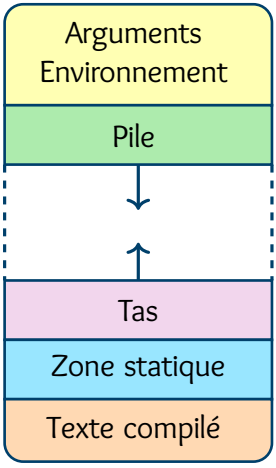
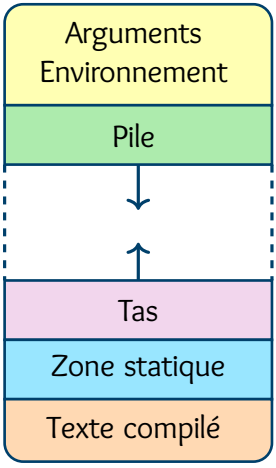


Schéma conceptuel de la mémoire



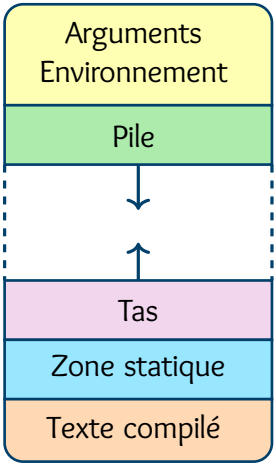
- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)...

Schéma conceptuel de la mémoire



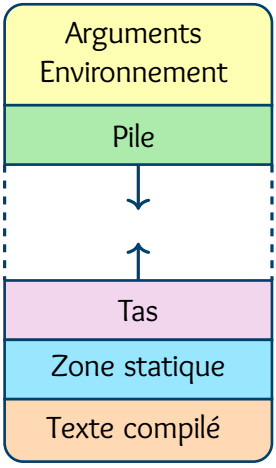
- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)...
- Variables locales non statiques

Schéma conceptuel de la mémoire



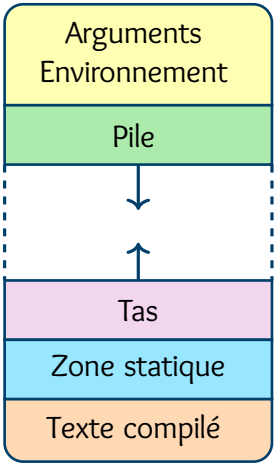
- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)...
- Variables locales non statiques
- Mémoire allouée par `malloc`

Schéma conceptuel de la mémoire



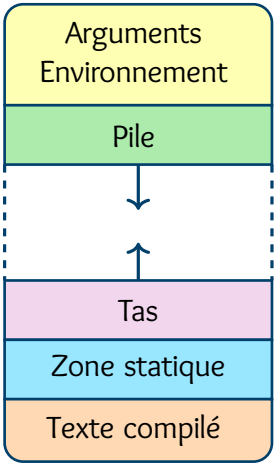
- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)...
- Variables locales non statiques
- Mémoire allouée par `malloc`
- Variables statiques et globales

Schéma conceptuel de la mémoire



- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)...
- Variables locales non statiques
- Mémoire allouée par `malloc`
- Variables statiques et globales
- Instructions du programme

Schéma conceptuel de la mémoire



- Arguments passés en ligne de commande (shell)
- Variables d'environnement héritées du shell (`PATH`, `HOME`)…
- Variables locales non statiques
- Mémoire allouée par `malloc`
- Variables statiques et globales
- Instructions du programme

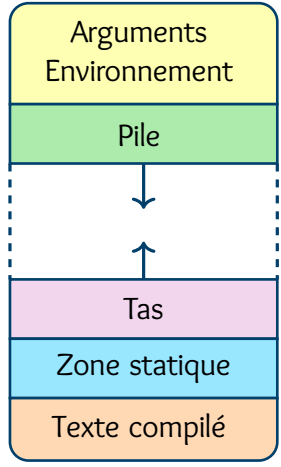
- La zone statique et le texte sont fixés avant le lancement.
- Les arguments sont initialisés au lancement du programme.
- La pile et le tas évoluent pendant l'exécution.

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,
  - à chaque appel de « sa » fonction,
  - 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.
  - Variables globales,
  - Variables explicitement déclarées `static`.
- Automatique : allouée seulement lors de l'entrée dans son bloc.
  - Variables locales non `static`
- Allouée dynamiquement : ce point sera vu plus tard (`malloc`).

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;
```

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

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
  - une seule fois,
  - à 0 par défaut (en l'absence d'initialisation explicite).

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
  - une seule fois,
  - à 0 par défaut (en l'absence d'initialisation explicite).
- Les variables locales automatiques (non statiques)

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
  - une 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,

  $\Longrightarrow$ doivent être initialisées ou affectées avant chaque 1ère utilisation.
- Pour les variables globales/statiques, les valeurs d'initialisations sont constantes.

Types de base

- 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.

Taille des types de base

- L'espace mémoire qu'occupe un type dépend de la machine et du compilateur.
- L'opérateur `sizeof` permet de connaître la taille d'un type ou d'une valeur.
- On a toujours :
  - `sizeof(short)` $\leqslant$ `sizeof(int)` $\leqslant$ `sizeof(long)` $\leqslant$ `sizeof(long long)`
  - `sizeof(float)` $\leqslant$ `sizeof(double)` $\leqslant$ `sizeof(long double)`

Taille des types, limites

- Pour un type de base, les limites indiquent la plus petite/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` vaut -2^{16},
> - `SHRT_MAX` vaut $2^{16} - 1$.

- **unsigned** : calcul modulo le plus grand entier représentable.
- **signed** : un dépassement de capacité provoque typiquement
  - un « cycle » positifs $\longleftrightarrow$ négatifs,
  - une erreur.

> ⚠ Ne pas mélanger entiers signés et non signés.
>
> ⚠ Il faut avoir une bonne raison pour utiliser les entiers non signés.

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 $>$ `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 chaîne\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`.

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

1	0	1	1	0	0	0	1	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	représente
$2^7 +$		$2^5 +$	$2^4 +$				2^0	$= 177$

- Sur $n+1$ bits, $a_n ... a_1 a_{0_{(2)}}$ représente $a_0 + 2a_1 + ... + 2^n a_n$
  Ex : $base_2(177) = 10110001_{(2)}$ = représentation de 177.
- Sur n bits, ce codage permet de représenter les entiers allant
  - de 0 : $\underbrace{000...0_{(2)}}_{n \text{ fois}}$
  - à $2^n - 1$: $\underbrace{111...1_{(2)}}_{n \text{ fois}}$.

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 $k > 0$:
  - $code(k) = base_2(k)$
  - $code(0) = base_2(0) = 00 \cdots 0$
  - $code(-k) = base_2([(2^n - 1) - k] + 1))$
- 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.

Dans cette partie…

7. Notion d'expression
   - L'affectation
   - Expressions et effets de bord
   - Opérateurs
   - Opérateurs logiques

Notion d'expression

- Une expression est

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :
  - des variables : `x`,
  - des constantes : 2, 3
  - des opérateurs : `+`, `*`, `!=`.
- Une expression a donc un type. L'exemple calcule

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :
  - des variables : `x`,
  - des constantes : 2, 3
  - des opérateurs : `+`, `*`, `!=`.
- Une expression a donc un type. L'exemple calcule un Booléen.

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :
  - 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 font des effets de bord. Exemple ?

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :
  - 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 font des effets de bord. Exemple ? `x++`, `++x`.

Notion d'expression

- Une expression est « quelque chose » qui peut être calculé.
- Exemple : `(((x + 3) * 2)!= 0)`.
- Une expression peut faire intervenir, entre autres :
  - 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 font des effets de bord. Exemple ? `x++`, `++x`.
- Si <e> est une expression, <e>; est une instruction consistant à évaluer (calculer) l'expression, et effectuer les effets de bord.
- Les opérateurs ont des priorités pour enlever l'ambiguïté.

L'affectation

Une affectation

```
x = <expression>
```

1. calcule l'expression `<expression>`,
2. range la valeur calculée à l'adresse (emplacement mémoire) de `x`.
3. a elle-même une valeur, celle de `<expression>`.

- 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`.
- Dans les deux cas, **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.

> Cela dépend si, quand il calcule `<expr1> - <expr2>`, il évalue d'abord l'expression `<expr1>` ou d'abord l'expression `<expr2>`.

Priorité des opérateurs

Opérateur	Signification
`++`,`--` **postfixe**	Post-incrémentation, post-décrémentation
`++`,`--` **préfixe**	Pré-incrémentation, pré-décrémentation
`!`	Non logique
`+`,`-` **unaires**	Signe
`(<nom_de_type>)`	Forçage de type (cast)
`*`, `/`, `%`	Opérateurs arithmétiques multiplicatifs
`+`, `-`	Opérateurs arithmétiques additifs
`<`, `>`, `<=`, `>=`	Comparaisons
`==`, `!=`	Comparaisons
`&&`	Et logique
`||`	Ou logique
`=`, `+=`, `=`, `*=`, `/=`, `%=`	Affectations

Opérateurs logiques

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

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

!	
true	false
true	true

- Toute valeur non nulle est considérée vraie (`true`)
- Toute valeur nulle est considérée fausse (`false`).
- En C, l'évaluation des opérateurs `&&` et `||` est « paresseuse ».

Évaluation 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)) {
//...
}
```

Évaluation 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)) {
//...
}
```

- 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.

Dans cette partie…

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

Tableaux

- Ils permettent de mémoriser plusieurs éléments du même type.

  $$\texttt{<type> <identificateur> [n}_1\texttt{][n}_2\texttt{]}\ldots\texttt{[n}_k\texttt{]}$$

  - C90 : $\texttt{n}_1, \texttt{n}_2, \ldots, \texttt{n}_\texttt{k}$ sont des expressions constantes.
  - C99 et après : $\texttt{n}_1, \texttt{n}_2, \ldots, \texttt{n}_\texttt{k}$ peuvent être des expressions ou `*`.
  - Le tableau a k dimensions.
  - Les indices de la j-ème dimension varient de 0 à $\texttt{n}_j - 1$.
  - L'allocation se fait sur la pile.

- 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 l'archive fournie sous moodle.

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'`

- Il n'y a pas de type spécifique pour les chaînes en C : ce sont juste des tableaux de caractères.
- 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.

Chaînes de caractères : bibliothèque

- 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`.

La fonction `main()`

- Pour permettre le passage d'arguments à l'exécutable, on définit la fonction `main` ainsi :

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

Comme `main()` n'est habituellement pas appelée directement, ses arguments `argc` et `argv` sont initialisés automatiquement au lancement du programme.

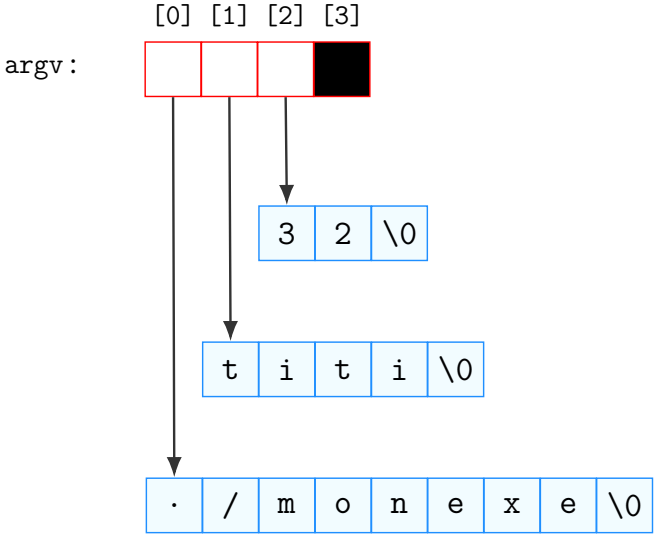
`argc` Entier initialisé par le nombre d'arguments + 1 avec lesquels l'exécutable est lancé.

`argv` Tableau de chaînes de caractères initialisé par le nom de la commande (en case 0) suivie de chacun des arguments avec lesquels l'exécutable a été lancé.

Ce mécanisme permet donc au programme de savoir comment il a été appelé.

Arguments de `main()`

Au lancement de la commande shell `./monexe titi 32`, la variable `argc` vaut 3.



Arguments de `main()`

⚠ Ne pas confondre :

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

- Leur nom est souvent `argc` et `argv`, mais ce n'est pas obligatoire. Si on écrit :

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

  La fonction `main()` a alors deux arguments :
  - `nb_a` (entier) et
  - `a` (tableau de chaînes de caractères).
- Le nom du fichier exécutable par lequel est lancée la commande est alors `a[0]`.

Arguments de `main()`, exemple

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  printf("Commande %s lancée avec %d arguments.\n", argv[0], argc - 1);
  for (int num = 1; num < argc; num++) {
    printf("Argument %2d : %s\n", num, argv[num]);
  }
  return 0;
}
```

```
mz@habanero:~$ gcc -Wall -Wextra argsmain.c
mz@habanero:~$ ./a.out un dos tre four
Commande ./a.out lancée avec 4 arguments.
Argument 1 : un
Argument 2 : dos
Argument 3 : tre
Argument 4 : four
```

Dans cette partie…

9. Les fonctions

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

Fonctions et modularité

- 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.

Fonctions et modularité

- 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

- 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

Fonction f

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

Fonction g

```
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

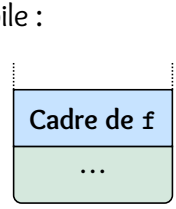
Fonction f

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

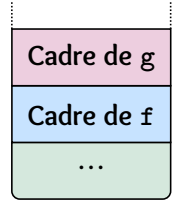
Fonction g

```
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 : exemple

Fonction f

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

Fonction g

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

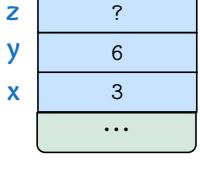
Appel de fonction : exemple

Fonction f

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

Fonction g

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



Avant appel de g

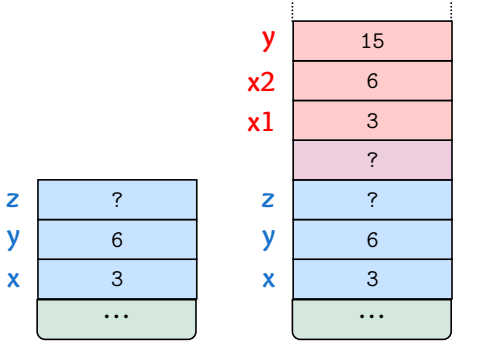
Appel de fonction : exemple

Fonction f

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

Fonction g

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



Avant appel de g Pendant l'appel à g

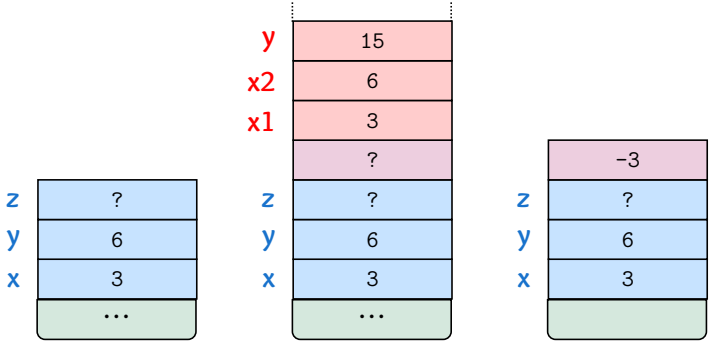
Appel de fonction : exemple

Fonction f

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

Fonction g

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



Avant appel de g | Pendant l'appel à g | Après retour de g

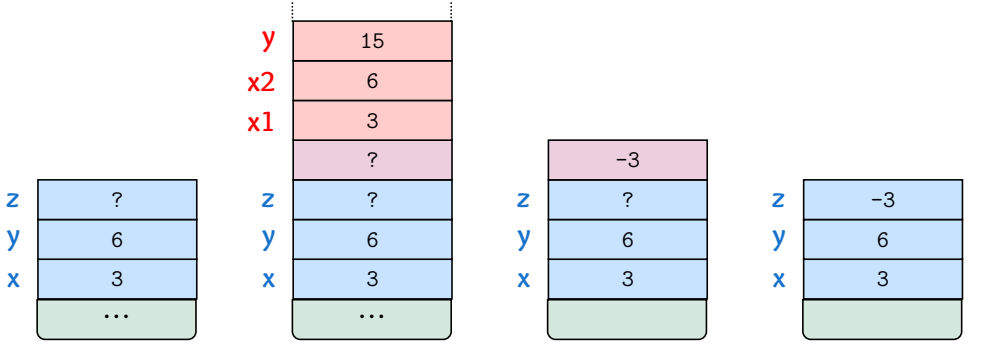
Appel de fonction : exemple

Fonction f

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

Fonction g

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



Avant appel de g — Pendant l'appel à g — Après retour de g — Après $z = g(x, y)$

Appel de fonction : récapitulatif

- Les arguments de la fonction sont calculés,
- Les valeurs sont transmises à la fonction appelée,

⚠ Conséquence :

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 des valeurs.

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 des valeurs.

⚠ Une fonction ne peut pas modifier la valeur des variables d'appel.

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 des valeurs.

⚠ Une fonction ne peut pas modifier la valeur des variables d'appel.

```
void f(int x) {
  x = x + 1;
}
```

```
void g(int x, int y) {
  f(y); // Ne change pas la valeur de y.
  f(x); // Ne change pas la valeur de x.
}
```

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 des valeurs.

⚠ Une fonction ne peut pas modifier la valeur des variables d'appel.

```
void f(int x) {
  x = x + 1;
}
```

```
void g(int x, int y) {
  f(y); // Ne change pas la valeur de y.
  f(x); // Ne change pas la valeur de x.
}
```

- Lors d'un appel de fonction, un emplacement est alloué sur la pile d'exécution pour chaque variable automatique (= locale non statique) et chaque argument.

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 des valeurs.

⚠ Une fonction ne peut pas modifier la valeur des variables d'appel.

```
void f(int x) {
  x = x + 1;
}
```

```
void g(int x, int y) {
  f(y); // Ne change pas la valeur de y.
  f(x); // Ne change pas la valeur de x.
}
```

- Lors d'un appel de fonction, un emplacement est alloué sur la pile d'exécution pour chaque variable automatique (= locale non statique) et chaque argument.
- Cette mémoire est désallouée quand la fonction exécute `return`.

⇒ Si une fonction g est appelée plusieurs fois, chaque variable automatique 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`,
  - termine la fonction qui l'appelle,
  - revient dans la fonction appelante et transmet la valeur retour éventuelle.
- Si la fonction ne renvoie rien, l'instruction est utilisée seule.

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

- Si la fonction renvoie une valeur, `return` est suivi d'une expression de type compatible.

```
int f (...) {
  ...
  return 42;
}
```

Retour implicite

- Si une fonction ne renvoie rien, et que l'exécution arrive sur la dernière accolade fermante, un `return` implicite est effectué.
- Si la fonction renvoie 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` utilise `return` explicitement.

- 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`.

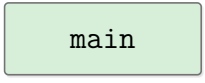
⚠ Ne pas confondre non plus **`return`** et `printf()` !

Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

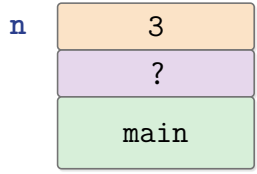


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

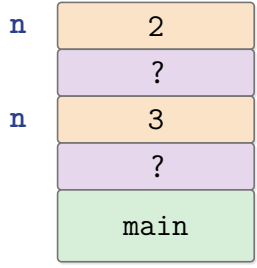


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

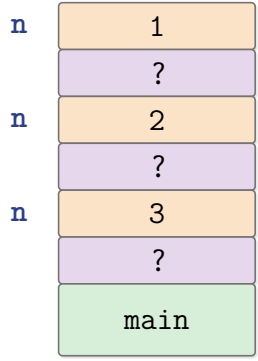


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

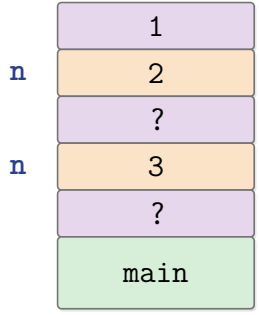


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

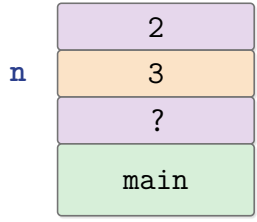


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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

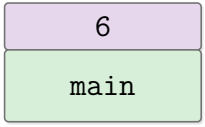


Récursivité

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

Fonction factorielle

```
unsigned long fact(short n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}

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



Récursivité

- 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 : suite de Fibonacci.

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

- Complexité $\approx \phi^n$ où $\phi = \frac{1+\sqrt{5}}{2}$, alors que $O(n)$ et même $O(\log n)$ sont possibles !
- Note. Il est toujours possible de « dérécursifier » une fonction.

Récursivité : efficacité

- 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);
}
```

Récursivité : efficacité

- 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 $2^n = (2^{n/2})^2 \times 2^{n\%2}$ 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.

Récursivité croisée

- Il est possible aussi d'écrire `f` qui appelle `g` qui elle-même appelle `f`.

Exemple

- Parité : `pair()` et `impair()`.
- Récurrences doubles.

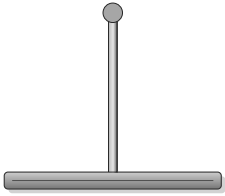
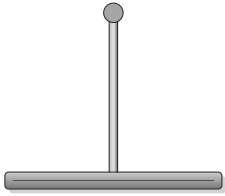
$$\begin{cases} u_0 = 1, & u_n = 2u_{n-1} - v_{n-1} \qquad (n \geq 1) \\ v_0 = 5, & v_n = u_{n-1} - v_{n-1} \qquad (n \geq 1) \end{cases}$$

⚠ On doit déclarer toute fonction utilisée avant sa définition.

☠ Attention à nouveau à l'efficacité !

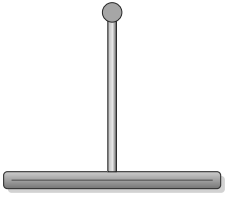
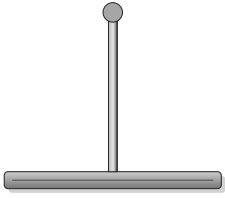
- L'utilisation naïve donne une complexité exponentielle.
- On a $\begin{pmatrix} u_n \\ v_n \end{pmatrix} = A \begin{pmatrix} u_{n-1} \\ v_{n-1} \end{pmatrix} = A^n \begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ avec $A = \begin{pmatrix} 2 & -1 \\ 1 & -1 \end{pmatrix}$.
- Pour calculer efficacement A^n,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

Récursivité : tours de Hanoi

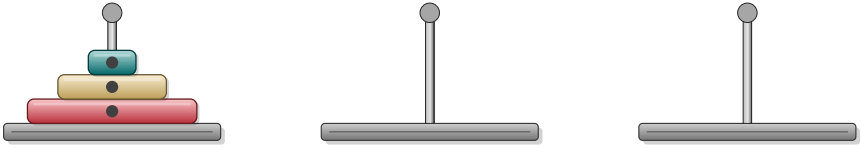




- Jeu En déplaçant les disques sur barres
  - un à un
  - sans poser un disque sur un disque plus petit,

arriver à la configuration suivante :


Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

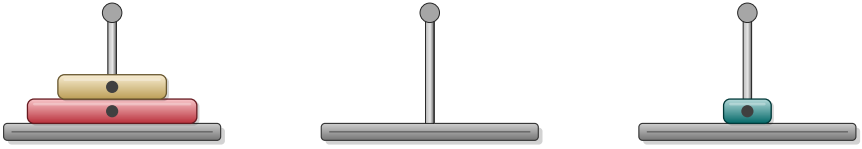
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

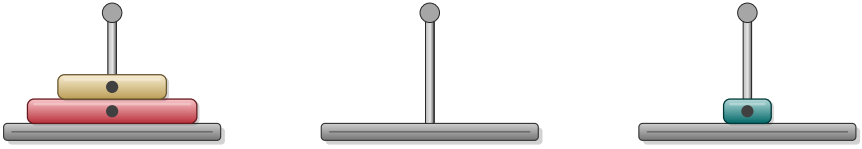
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

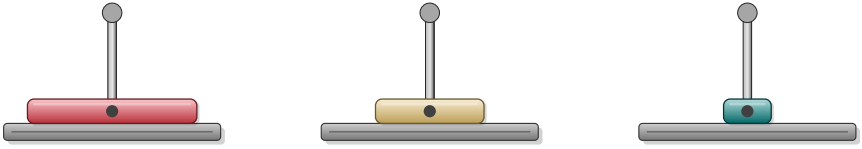
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

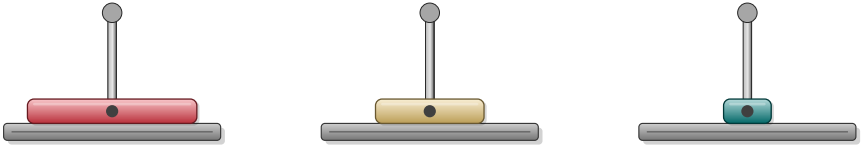
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

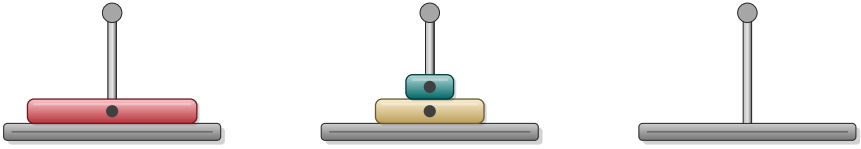
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

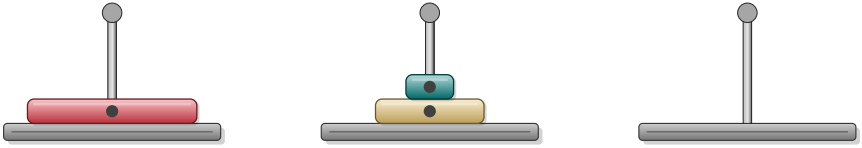
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

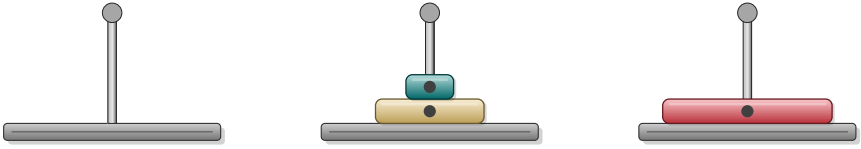
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

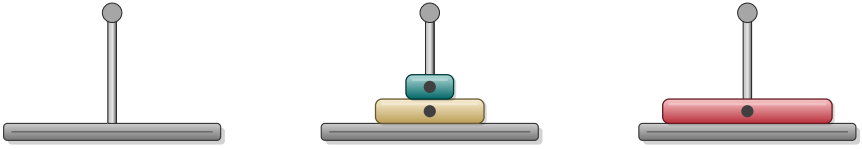
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

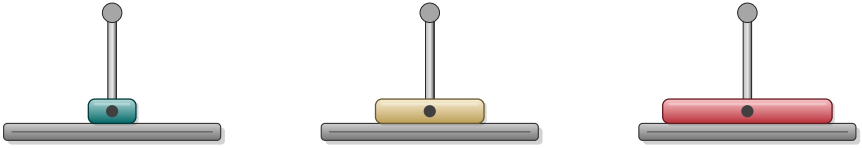
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

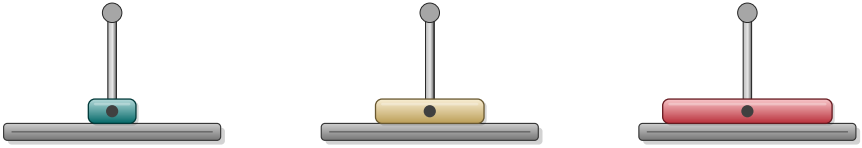
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

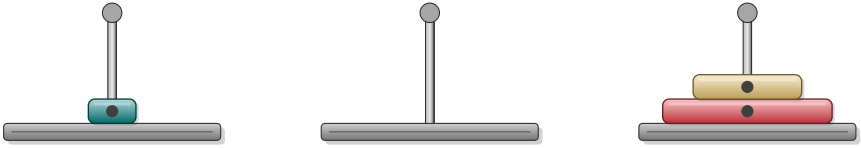
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

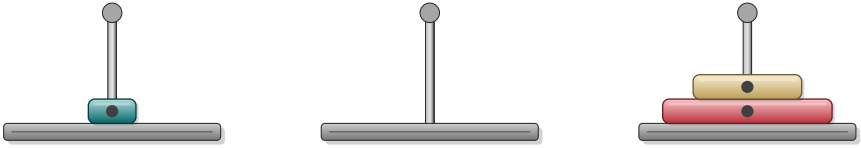
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

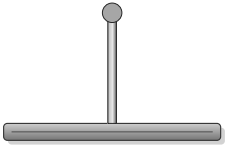
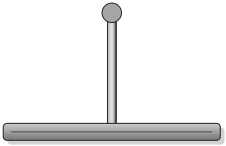
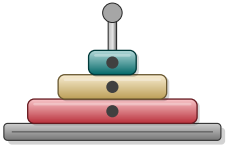
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques

- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

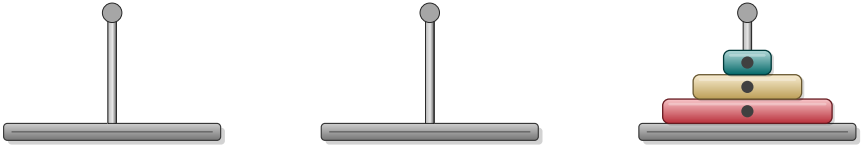
- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi : exemple avec 3 disques



- Déplacer $1 \to 3$
- Déplacer $1 \to 2$
- Déplacer $3 \to 2$

} Déplacement $1 \to 2$ des 2 disques supérieurs

- Déplacer $1 \to 3$

Déplacement $1 \to 3$ du plus grand disque

- Déplacer $2 \to 1$
- Déplacer $2 \to 3$
- Déplacer $1 \to 3$

} Déplacement $2 \to 3$ des 2 disques supérieurs

Tours de Hanoi

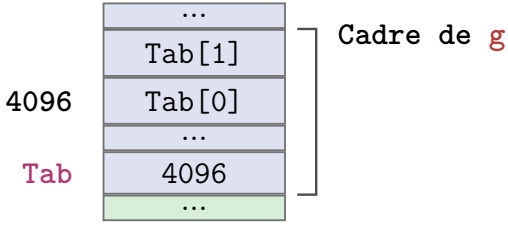
- On sait faire pour un disque.
- Si on sait faire pour $n-1$ disques, alors on sait faire pour n.

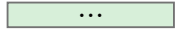
```c
#include <stdio.h>

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 ?

Récursivité et tableaux

- On verra qu'un tableau se comporte presque comme 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 renvoyer `Tab` dans `g`.
- En effet, `Tab` désigne un emplacement dans le cadre de `g`, et ce cadre est désalloué au retour de `g`.





Compléments : `atexit()`

- 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.

```c
#include <stdio.h>

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

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

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

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

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

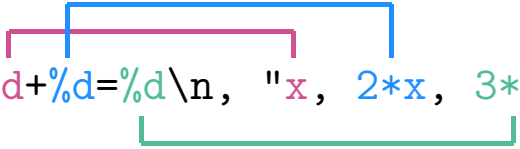
Rappels : la fonction `printf()`

- 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	Affichage
`printf("OK\n");`	`OK`
`printf("C'est ÖK\"\n");`	`C'est "OK"`
`printf("TVA: 19,6%%\n");`	`TVA : 19,6%`
`printf("Pi=%g\n",acos(1));`	`Pi=3.14159`
`printf("%ce%cl%c\n", 'H','l','o');`	`Hello`
`printf("%+06.3Lg\n",7.234L);`	`+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

- Le caractère suivant le % d'une directive donne l'interprétation de « son » argument.

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 y a d'autres spécificateurs (pour entiers dans d'autres bases, non signés, longs etc.).

Nombre variable d'arguments

- 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`.

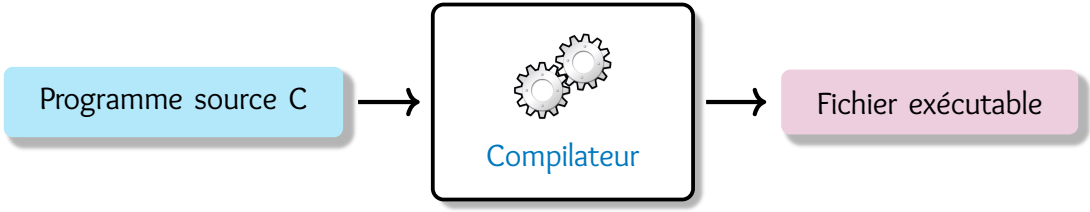
Dans cette partie...

10. Compilation et modularité

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

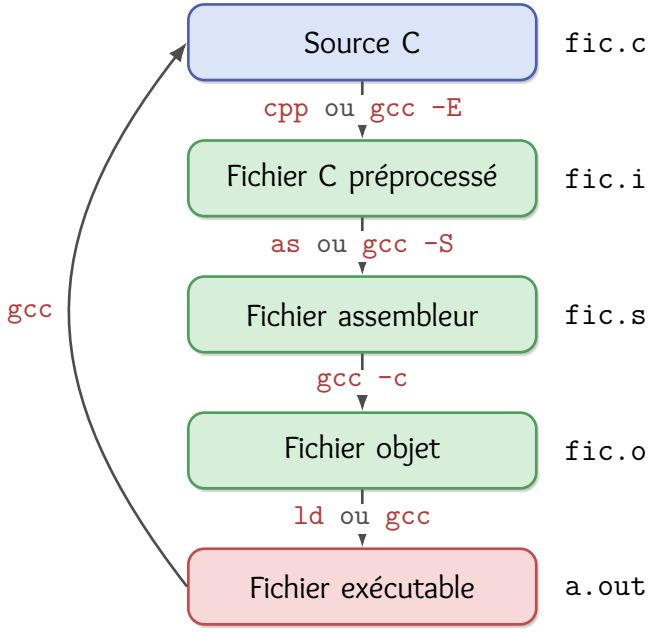
La compilation (rappel)

- Le programmeur ou la programmeuse écrit un programme source C
- Il est ensuite traduit pour obtenir un exécutable.



- Après compilation, on obtient un nouveau fichier, appelé exécutable, qui est l'application.
- L'exécutable dépend de la machine sur laquelle on veut exécuter le programme.
- $\Longrightarrow$ on doit compiler autant de fois qu'on veut d'exécutables sur machines différentes.
- Voir le ⏩ premier cours pour des rappels sur le compilateur `gcc`.

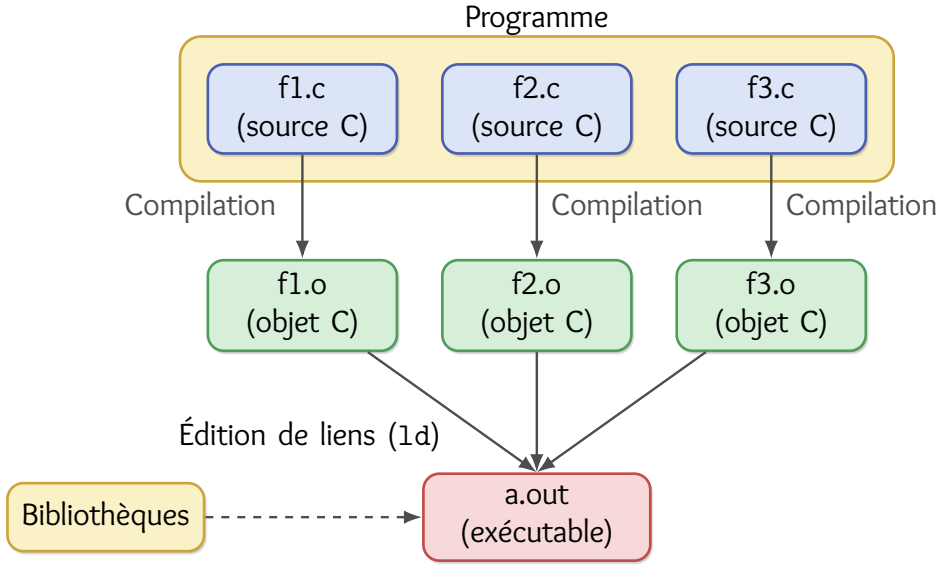
Étapes de la compilation



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.

L'édition de liens

- 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.

Dans cette partie...

11. 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

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

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.

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

Instruction `continue`

- L'instruction `continue` change le déroulement de la boucle la plus proche la contenant.
- Elle fait passer à l'itération suivante, sans finir le corps de la boucle.

```c
#include <stdio.h>
int lire_entier();

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)
         );
    }
```

☠ (pourquoi ?)

```
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 proche qui la contient.
- Exemple : calcul de l'indice minimum d'une case contenant 0 dans les n premières cases d'un tableau :

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

Expression `b?e1:e2`

- Dans une expression `b?e1:e2`, `e1` et `e2` sont des expressions de types compatibles.
- Pour calculer l'expression :
  - On évalue `b`.
  - Si `b` est vrai :
    - l'expression `e2` n'est pas évaluée.
    - l'expression `e1` est évaluée et sa valeur est celle de `b?e1:e2`.
  - Si `b` est faux :
    - l'expression `e1` n'est pas évaluée ;
    - l'expression `e2` est évaluée et sa valeur est celle de `b?e1:e2`.
- Exemple : `return x>y? x: y` renvoie 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 pratique dans les boucles :

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

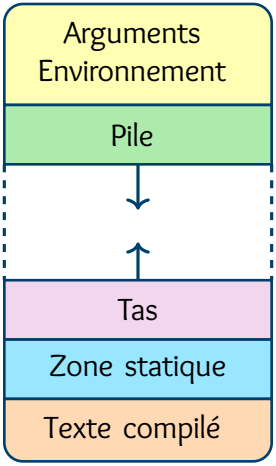
Dans cette partie…

12. Les pointeurs

- Rappel : organisation mémoire d'un processus
- Variables et adresses
- Pointeurs : valeurs et déclarations
- Emplacement pointé
- Tableaux vs. pointeurs
- Arithmétique des pointeurs
- 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 :

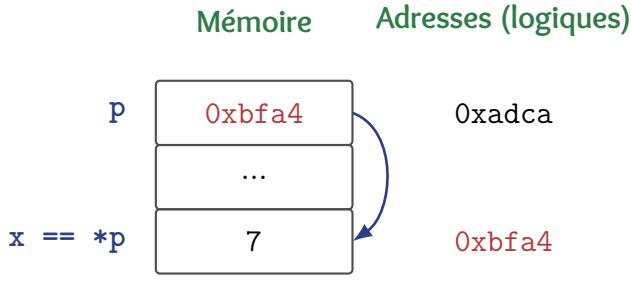
```
<type> *<nom de variable>;
```

- 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;
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;
```

Emplacement pointé *(pointee)*

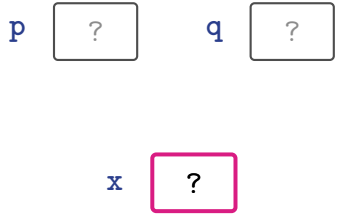
- Si `p` est un pointeur de type `T *`, la donnée de taille `sizeof(T)` octets commençant à l'adresse `p` se note `*p`.
- `*p` référence **l'emplacement pointé** par `p`.

```
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`.

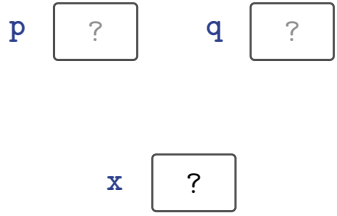
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

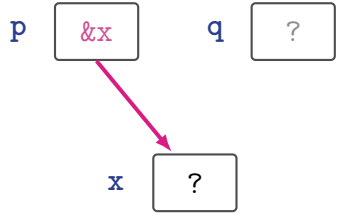
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

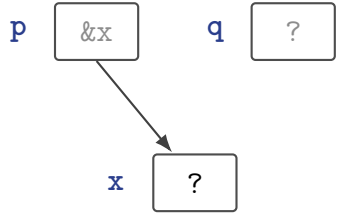
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

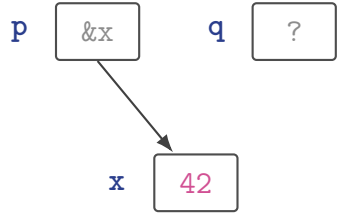
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

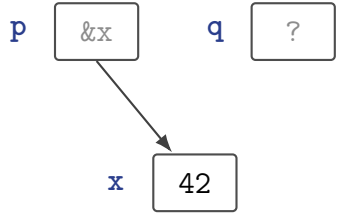
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

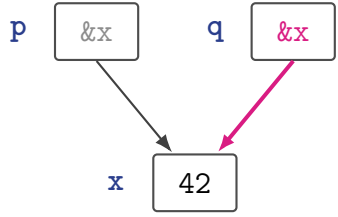
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

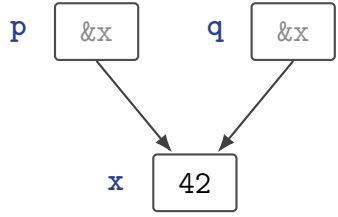
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

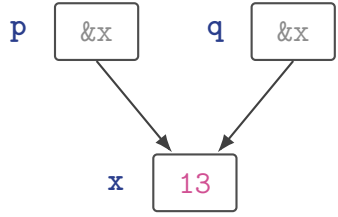
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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`.

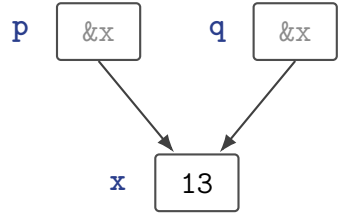
```
int *p, *q;
int x,
p  = &x;
*p = 42;
q  = p;
*q = 13;
```



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;
```



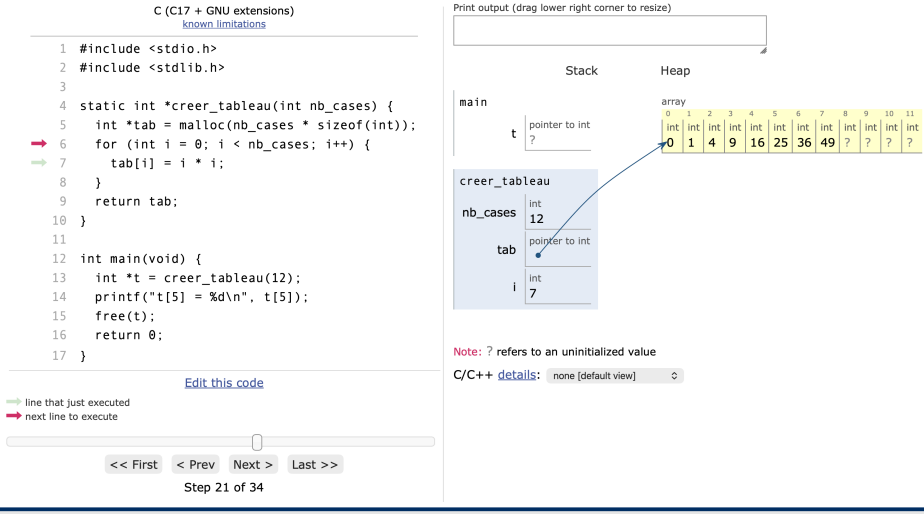
⚠ `int *r;` réserve la place pour la variable `r`, PAS pour l'emplacement `*r` sur lequel pointera `r`.

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

👉 Lien sur la célèbre animation pointerfun de Stanford.

Pythontutor

- Pour bien programmer, il **faut** dessiner la mémoire manipulée.
- Pour vérifier, utilisez Pythontutor !



Tableaux vs. pointeurs

- Un tableau se comporte presque comme 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, équivalent à tab[1] = 2;
tab = tab2;     // ILLÉGAL
```

⚠ `int p[10]` réserve de la place `p` et pour 10 entiers (contrairement à `int *p`).

- Il y a d'autres différences entre tableaux et pointeurs. Par exemple, `sizeof tab2` renvoie la place occupée par les cases du tableau `tab2`, et non la taille d'un pointeur.

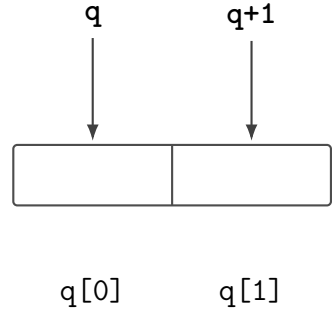
Arithmétique des pointeurs

- 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 `sizeof(char)=1` et `sizeof(short)=2`.

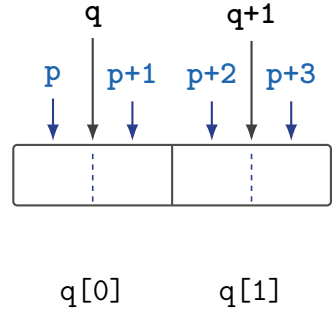
```
short q[2];
char *p;
p = (char *)q;
```



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 `sizeof(char)=1` et `sizeof(short)=2`.

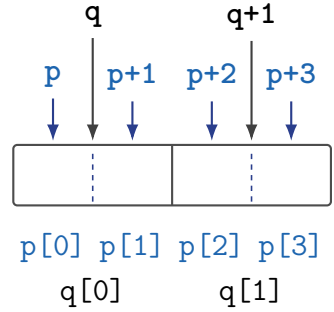
```
short q[2];
char *p;
p = (char *)q;
```



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 `sizeof(char)=1` et `sizeof(short)=2`.

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

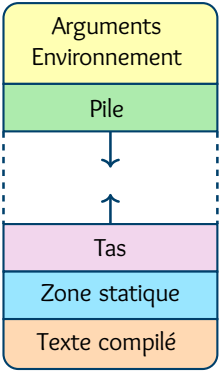


Qu'appelle-t-on « allocation dynamique » ?

Problème de l'allocation sur la pile ?

Une fonction créant de la mémoire sur la pile ne peut pas la transmettre. En effet, cette mémoire est détruite au retour de la fonction.

```
int *f(){
  int tab[10];
  //...
  return tab; // NON ! NO WAY! NEVER!
}
```

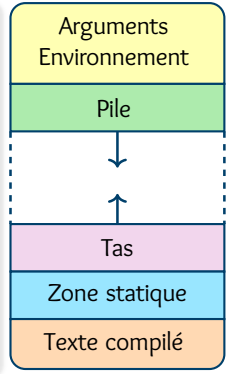


Qu'appelle-t-on « allocation dynamique » ?

Problème de l'allocation sur la pile ?

Une fonction créant de la mémoire sur la pile ne peut pas la transmettre. En effet, cette mémoire est détruite au retour de la fonction.

```
int *f(){
  int tab[10];
  //...
  return tab; // NON ! NO WAY! NEVER!
}
```



L'allocation dynamique sert à contourner ce problème. On demande explicitement que :

- de la mémoire nous soit allouée (et combien) sur le « tas »,
- la mémoire qui nous a été allouée ainsi soit détruite.

La mémoire allouée sur le tas reste allouée tant qu'on ne demande pas de la détruire.

Les fonctions d'allocation dynamique (1)

Si on compare la mémoire à une ville et le gestionnaire mémoire à une agence immobilière :

- Deux fonctions de réservation d'appartement :

```
// Trouve et réserve-moi un appartement de S m2. Renvoie-moi son adresse.
void *malloc(size_t S);

// Trouve et réserve-moi un appartement de NB pièces, chacune de S m2,
// chacune initialement vide, et renvoie-moi son adresse.
void *calloc(size_t NB, size_t S);
```

Les fonctions d'allocation dynamique (1)

Si on compare la mémoire à une ville et le gestionnaire mémoire à une agence immobilière :

- Deux fonctions de réservation d'appartement :

```
// Trouve et réserve-moi un appartement de S m2. Renvoie-moi son adresse.
void *malloc(size_t S);

// Trouve et réserve-moi un appartement de NB pièces, chacune de S m2,
// chacune initialement vide, et renvoie-moi son adresse.
void *calloc(size_t NB, size_t S);
```

- Une fonction de déménagement :

```
// Déménage mon appartement se trouvant à l'adresse ptr dans un nouvel
// appartement de S m2. Renvoie-moi son adresse.
void *realloc(void *ptr, size_t S);
```

Les fonctions d'allocation dynamique (1)

Si on compare la mémoire à une ville et le gestionnaire mémoire à une agence immobilière :

- Deux fonctions de réservation d'appartement :

```
// Trouve et réserve-moi un appartement de S m2. Renvoie-moi son adresse.
void *malloc(size_t S);

// Trouve et réserve-moi un appartement de NB pièces, chacune de S m2,
// chacune initialement vide, et renvoie-moi son adresse.
void *calloc(size_t NB, size_t S);
```

- Une fonction de déménagement :

```
// Déménage mon appartement se trouvant à l'adresse ptr dans un nouvel
// appartement de S m2. Renvoie-moi son adresse.
void *realloc(void *ptr, size_t S);
```

- Une fonction de libération d'appartement :

```
// Je rends mon appartement se trouvant à l'adresse ptr.
void free(void *ptr);
```

Les fonctions d'allocation dynamique (2)

Même chose, avec le vocabulaire informatique et un groupement différent :

- Deux fonctions de base.

```
// Demande l'allocation d'une zone de taille S (en « bytes »).
void *malloc(size_t S);

// Demande la désallocation d'une zone allouée précédemment
void free(void *ptr);
```

- Deux fonctions auxiliaires.

```
// Demande l'allocation d'une zone initialisée à 0,
// pour stocker NB objets, chacun de taille S.
void *calloc(size_t NB, size_t S);

// Demande le redimmensionnement de la zone déjà allouée
// (par ces fonctions) à la nouvelle taille S.
void *realloc(void *ptr, size_t S);
```

Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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

Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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

Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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

Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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

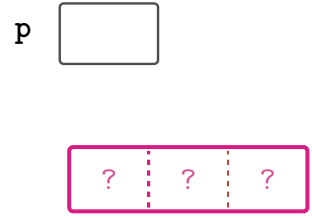
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



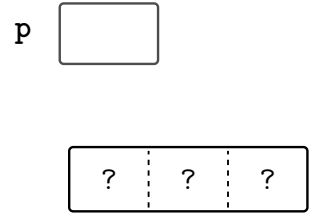
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



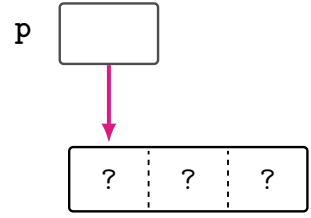
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



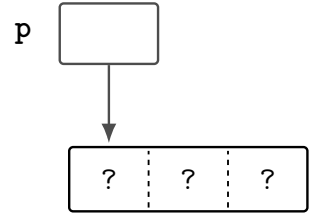
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



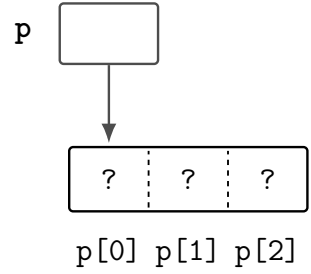
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



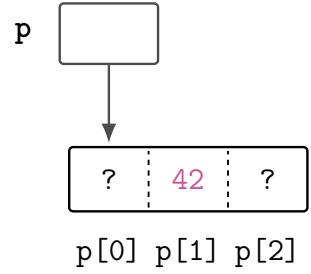
Allocation dynamique : malloc()

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

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

- La fonction alloue l'espace non initialisé de taille `S` bytes (souvent, 1 byte = 1 octet).
- La fonction `malloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `malloc()` renvoie `NULL` si l'allocation échoue.

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



Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.

Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.

Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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

Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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

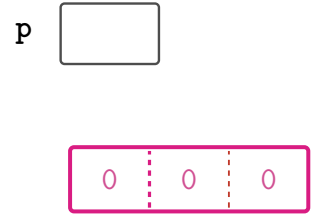
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



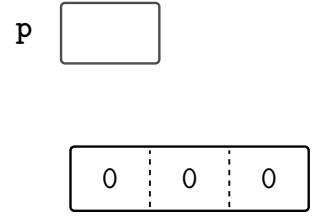
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



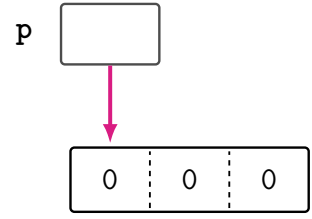
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



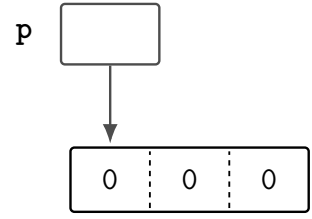
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



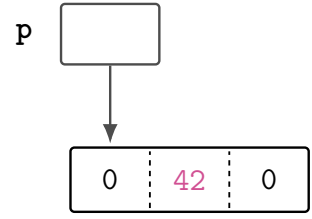
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



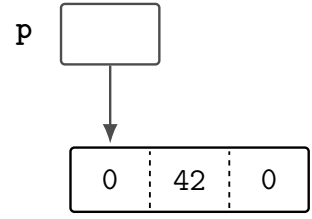
Allocation dynamique : calloc()

- La fonction `calloc()` a un objectif similaire à `malloc()`. Deux différences :

```
void *calloc(size_t NB, size_t S);
```

- Au lieu de préciser la taille globale de la mémoire demandée, on précise le nombre de cases `NB` et la taille individuelle de chaque case `S`.
- La fonction `calloc()` initialise à zéro l'espace alloué.
- La fonction `calloc()` renvoie l'adresse du début de la zone allouée en cas de succès.
- La fonction `calloc()` renvoie `NULL` si l'allocation échoue.

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



Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.

Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.

Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

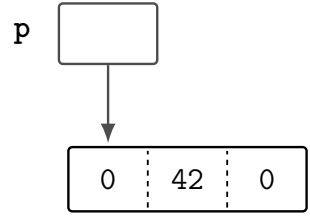
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



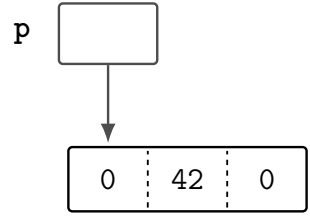
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



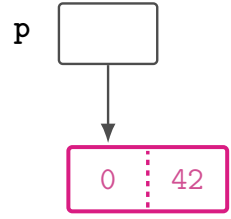
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



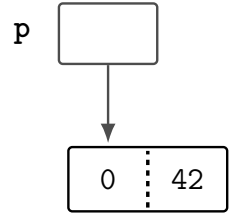
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



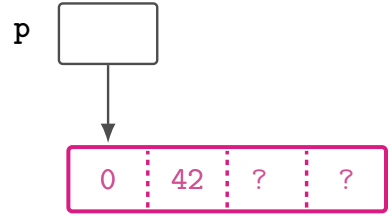
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



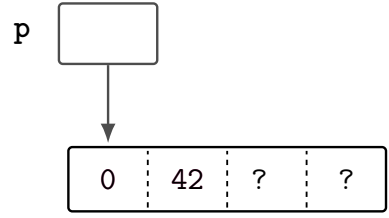
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



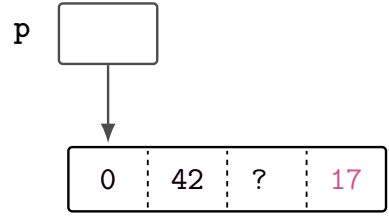
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



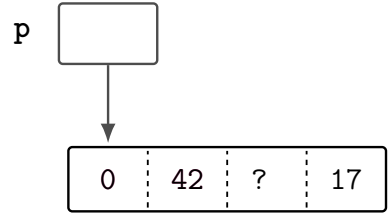
Allocation dynamique : realloc()

- La fonction `realloc()` permet de modifier la taille d'une zone déjà allouée.

```
void *realloc(void *ptr, size_t new_S);
```

- Le premier argument `ptr` de la fonction `realloc()` doit pointer sur une zone déjà allouée par une des fonctions `malloc`, `calloc` ou `realloc`.
- La fonction `realloc()` alloue une nouvelle zone mémoire de taille `new_S`, et copie l'ancienne zone dans la nouvelle, en tronquant si la nouvelle taille est plus petite.
- Elle renvoie l'adresse du début de la zone allouée, ou `NULL` en cas d'échec.

```
int *p;
p = calloc(3, sizeof(int));
p[1] = 42;
p = realloc(2, sizeof(int));
p = realloc(4, sizeof(int));
p[3] = 17;
```



(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```c
void free(void *p);
```

(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```
void free(void *p);
```

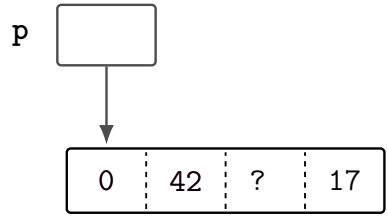
```
// Suite de l'exemple précédent.
// ...
p = realloc(4, sizeof(int));
p[3] = 17;
free(p);
```

(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```
void free(void *p);
```

```
// Suite de l'exemple précédent.
// ...
p = realloc(4, sizeof(int));
p[3] = 17;
free(p);
```

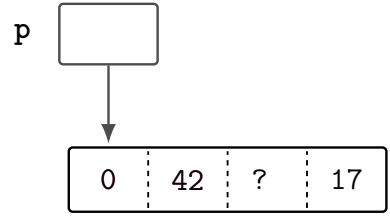


(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```
void free(void *p);
```

```
// Suite de l'exemple précédent.
// ...
p = realloc(4, sizeof(int));
p[3] = 17;
free(p);
```



(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```
void free(void *p);
```

```
// Suite de l'exemple précédent.
// ...
p = realloc(4, sizeof(int));
p[3] = 17;
free(p);
```

(Dés)Allocation dynamique : free()

- Il est important de désallouer (= rendre) la mémoire qu'on n'utilise plus.
- La fonction `free()` permet de désallouer de la mémoire allouée par `m/c/realloc()`.

```
void free(void *p);
```

```
// Suite de l'exemple précédent.
// ...
p = realloc(4, sizeof(int));
p[3] = 17;
free(p);
```

Important

- On ne doit libérer chaque zone allouée qu'une seule fois.
  On ne peut pas rendre les clés d'un appartement qu'on a déjà rendues.
- Moins important : `free(NULL);` n'a aucun effet.

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 = 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()/realloc()/calloc()`.
- Film d'animation artisanal de l'université de Stanford

https://www.youtube.com/watch?v=5VnDaHBi8dM

Passage de paramètres : rappel

- Lors d'un appel de fonction :
  - Les paramètres d'appel sont évalués (dans un ordre dépendant du compilateur),
  - 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

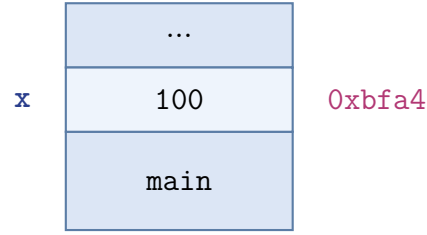
- Une fonction appelée peut modifier une variable automatique de la fonction appelante, si elle connaît l'adresse de la variable.

```
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.

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

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

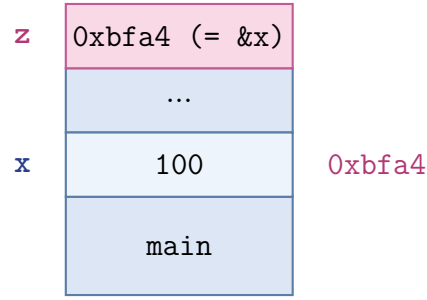


Dans `main`

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);
}
```

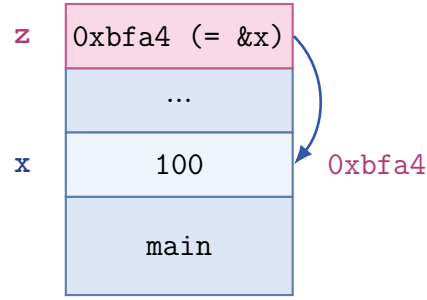


À l'appel de `f`

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.

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

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

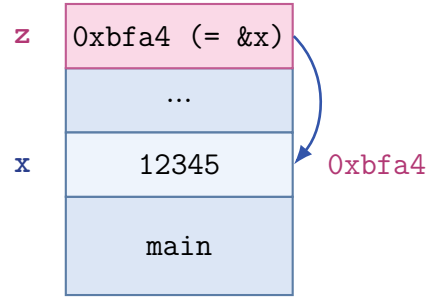


À l'appel de `f`

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);
}
```

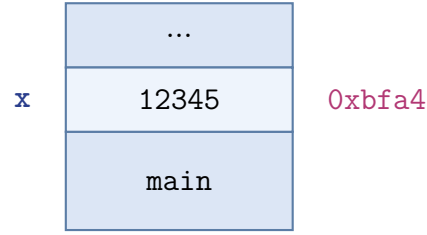


Après l'affectation `*z = 12345`

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.

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

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

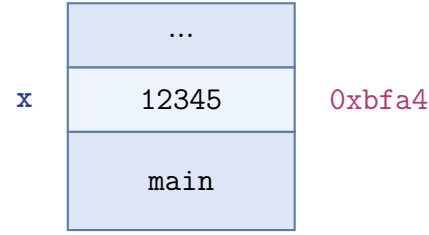


Après retour de `f`

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.

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

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



⚠ Rappel : ne jamais renvoyer l'adresse d'une variable locale non `static` !

Dans cette partie...

13. Structures et unions
    - Construction de nouveaux types
    - Énumérations
    - Structures
    - Abréviations de types

Construction de nouveaux types

- On construit de nouveaux types à partir des types de base en utilisant des constructeurs.

Type construit	Constructeur	Description
Tableaux	`[]`	Valeurs de même type rangées consécutivement
Pointeurs	`*`	Contient une adresse
Énumérations	`enum`	Valeur parmi un ensemble fini d'entiers
Structures	`struct`	« Vecteur » de valeurs de types quelconques
Unions	`union`	Choix entre valeurs de plusieurs types

Énumérations

- Un type énuméré contient un nombre fini de valeurs.
- Chaque valeur a un nom et est codée par un entier.
- Par défaut, ces entiers sont 0, 1, 2, etc. dans l'ordre.
- La définition du type se fait hors du corps de toute fonction.

```
enum couleur {     // définition du type
  BLEU, ROUGE, VERT, JAUNE, NOIR, BLANC, NB_COULEURS
};

int main(void) {
  for(enum couleur c = 0; c < NB_COULEURS){
    // Faire quelque chose avec la couleur c.
    if (c == BLEU)
      //...
  }
}
```

Structures

- Une structure regroupe plusieurs champs de types différents.

```
struct <nom> {
  <suite de déclarations 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 etu;     // declaration de variable
etu.age = 20;
```

Structures emboîtées

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

```c
// définitions de types
struct point {
  double abscisse;
  double ordonnee;
};

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

```c
// définitions 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; // Affectation !
c.rayon    = 5.0;
```

Structures : initialisations

```c
// définitions de types
struct point {
  double abscisse;
  double ordonnee;
};

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

```c
// définitions avec initialisations
struct point  p = {2.0, 3.0};
struct cercle c = {{2.0, 3.0}, 1.0};

// En C18 et après, on peut aussi écrire
struct cercle c1 = {p, 1.0};
```

Deux interdictions avec les structures

- On ne peut pas les comparer par `==`. Il faut comparer champ par champ.
- L'initialisation `struct point p = {2.0, 3.0}` est légale. Par contre, pour faire une affectation, on doit convertir explicitement la structure constante : `p = (struct point){2.0, 3.0}`.

Abréviations de types

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

```
typedef <type> <nom_choisi>
```

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

typedef struct point Point;

// Maintenant on peut utiliser Point
// comme abréviation de struct point

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

```
// Définit à la fois la
// structure et l'abréviation.
typedef struct point {
  double abscisse;
  double ordonnee;
} Point;
```

Abréviations de types

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

```
typedef <type> <nom_choisi>
```

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

typedef struct point Point;

// Maintenant on peut utiliser Point
// comme abréviation de struct point

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

```
// Définit à la fois la
// structure et l'abréviation.
typedef struct point {
  double abscisse;
  double ordonnee;
} Point;
```

Une habitude fréquente

Les abréviations des structures ont souvent le nom de la structure. `typedef struct point point;`

Dans cette partie...

Pointeurs et structures

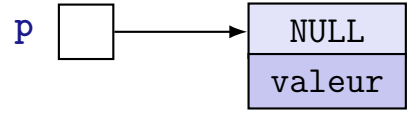
- Si `p` est un pointeur sur une structure et `champ` un champ de la structure, `p->champ` désigne `(*p).champ`.
- Un champ d'une structure peut être un pointeur sur une structure de même type.

```
typedef struct cell {
int valeur;
struct cell *suivant;
} cell;

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

Pointeurs et structures

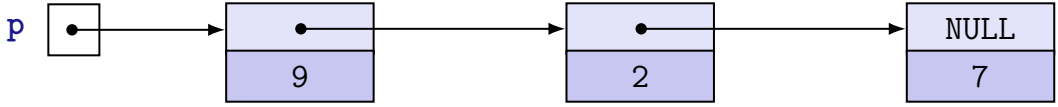
- Si `p` est un pointeur sur une structure et `champ` un champ de la structure, `p->champ` désigne `(*p).champ`.
- Un champ d'une structure peut être un pointeur sur une structure de même type.

```
typedef struct cell {
int valeur;
struct cell *suivant;
} cell;

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



Listes chaînées

- Cette construction permet d'implémenter les listes chaînées, chaque cellule de la liste comtenant 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 chaines de caractères, prenant en argument :
  - le tableau,
  - un pointeur sur une fonction de comparaison de deux chaînes.