Hello C

Ressources

Cours

Voici quelques cours disponibles en ligne, complets et avec des exercices:

https://lucidar.me/fr/c-class/learn-c-programming/

https://zestedesavoir.com/tutoriels/755/le-langage-c-1/

https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c

https://moodle.insa-lyon.fr/course/view.php?id=5199

Comptes à suivre

https://x.com/7etsuo

Exemples en cybersecurité

https://wargames.ret2.systems/

Sommaire

I. Introduction au Langage C

A. Histoire et importance du langage

C Le langage de programmation C a été créé au début des années 1970 par Dennis Ritchie au sein des laboratoires Bell. Il a été conçu pour être utilisé avec

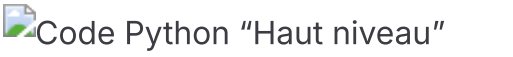
le système d'exploitation UNIX, également développé à Bell Labs. Depuis lors, le C est devenu l'un des langages les plus utilisés et a eu une influence considérable sur de nombreux autres langages de programmation, notamment C++, Java et Python.

L'une des raisons de la popularité du C est sa flexibilité et sa portabilité. Les programmes écrits en C peuvent être exécutés sur différents types de machines avec peu ou pas de modification. Cette caractéristique a fait du C le langage de choix pour le développement de systèmes d'exploitation, de langages de programmation, de compilateurs et de nombreux autres logiciels de bas niveau.

On considère le C comme un langage “mid-level”, car il n’est pas bas niveau à proprement parler car pas directement du langage machine, mais nécessite beaucoup d’efforts notamment en terme de gestion de la mémoire, donc il n’est plus considéré comme un langage de haut niveau.


Code Assembleur “Bas Niveau”

Code Assembleur “Bas Niveau”


Code Python “Haut niveau”

Code Python “Haut niveau”

B. Les domaines d'application du C

Où le C est utilisé

Le langage C est utilisé dans une variété de domaines en raison de sa rapidité, de son efficacité et du niveau de contrôle qu'il offre sur le matériel informatique. Voici quelques domaines clés où le C est particulièrement populaire :

- Développement de systèmes d'exploitation (Windows, Linux, MacOS (BSD), Android)

- Programmation de systèmes embarqués (voitures, avions, fusées, terminaux de paiements, communications...)
- Développement de drivers et de noyaux
- Programmation de jeux vidéo et de moteurs graphiques
- Applications nécessitant des performances élevées, comme la simulation et le traitement de données
- Outils de développement logiciel, y compris les compilateurs et les interpréteurs pour d'autres langages
- Développement de shaders: des mini-programmes graphiques exécutés en parallèle par le GPU *exemple*

Où le C n'est pas utilisé

N'étant pas un langage moderne et simple, le C n'est pas utilisé pour beaucoup de choses aujourd'hui. En effet, certaines choses très rapides dans certains langages peuvent rapidement devenir très fastidieux en C. Quand on peut se permettre d'utiliser plus de ressources (calcul, mémoire, pas en embarqué par exemple), les développeurs vont se porter vers des langages plus modernes tels que Python.

- Applications mobiles
- Sites Web
- Serveurs

C. Le C et la Cybersécurité

Le C est principalement utilisé pour les systèmes critiques, tels que les systèmes d'exploitation, drivers et systèmes embarqués. Ces systèmes se doivent d'être extrêmement sécurisés car ils sont la base de nombreux outils critiques.

En tant qu'expert en Cybersécurité, connaitre le langage C de manière approfondie est un atout incontestable. En effet, à mi-chemin entre un langage de bas niveau et de haut niveau, il permet d'écrire rapidement des programmes

tout en gardant un contrôle total sur sur comportement vis à vis de la machine sur lequel il tourne. Comprendre ce langage en profondeur permet de comprendre comment fonctionnent les ordinateurs en interne.

La plupart des failles critiques en informatiques exploitées par les hackers sont des vulnérabilités liées à la mauvaise gestion de la mémoire par les programmeurs (buffer-overflow, write-after-free, null-pointers). Apprendre le C permet de comprendre comment la mémoire d'un ordinateur fonctionne, mets en garde sur les erreurs à ne pas faire et comment se protéger des hackers.

Les langages dits de plus haut niveau (~python) sont plus lents, plus lourds et ne donne pas la main directe à la mémoire, ce qui résulte en des programmes moins sûrs, plus sujets aux vulnérabilités car dépendants à trop de couches entre eux et la machine.

Ce contrôle supplémentaire sur la mémoire qu'apporte le C doit cependant être parfaitement maitrisé, autrement il en résulte parfois des conséquences dramatiques.

Faits divers

Crowdstrike

> La **panne informatique mondiale du July 19, 2024July 19, 2024July 19, 2024July 19, 2024July 19, 2024** est causée par la mise à jour de *Falcon Sensor*, un logiciel développé par la société de cybersécurité américaine CrowdStrike. Cette mise à jour défectueuse provoque partout à travers le monde le plantage d'environ 8,5 millions d'ordinateurs et serveurs utilisant le système d'exploitation Microsoft Windows, causant d'importantes perturbations, essentiellement au sein des entreprises. Le *cloud* de Microsoft est aussi partiellement tombé en panne ce jour là, mais un peu avant et donc sans rapport avec la mise à jour défectueuse.

Divers secteurs économiques sont affectés tels que les aéroports, les banques, les hôtels, les hôpitaux, la grande distribution, les marchés financiers, la

restauration, des services de diffusion gouvernementaux comme les numéros d'appels d'urgence et des sites internet.

https://x.com/perpetualmaniac/status/1814376668095754753?s=46&t=GrgUv84ThhjJMTYOeBX-2g

Ariane

Le **vol 501** est le vol inaugural du lanceur européen Ariane 5, qui a eu lieu le June 4, 1996. Il s'est soldé par un échec, causé par un dysfonctionnement informatique,

https://lucidar.me/fr/c-class/lesson-02-06-ariane-flight-501/

L'avenir du C ?

Le C va continuer à être utilisé pendant un moment car il représente encore une grosse part des outils critiques encore utilisé aujourd'hui. Cependant, le risque qu'il apporte en terme de gestion de mémoire est problèmatique et des solutions commencent à apparaitre. D'autres langages dits "memory-safe" existent, mais un en particulier, le **Rust**, commence à faire sa place. Certains composants du noyaux Linux ont déjà étés réécrits dans ce langage. Il a l'avantage d'être très similaire au C++, mais avec un atout majeur: il est impossible de faire des erreurs de mémoire avec (plus de "null pointer", pas d'ecriture en zone non autorisée)

D. Installation de l'environnement de développement

Pour commencer à programmer en C, vous aurez besoin d'un environnement de développement qui comprend un éditeur de texte et un compilateur.

Coder en ligne

Pour débuter simplement, il est possible d'utiliser un éditeur de code en ligne

https://www.programiz.com/c-programming/online-compiler/

Coder localement

Nous recommandons l'utilisation de Visual Studio Code avec l'extension C/C++ pour un acces rapide aux outils nécessaires.

Pour installer les outils nécessaires, suivre les instructions sur ce site:

https://code.visualstudio.com/docs/languages/cpp

Si on ne souhaite pas utiliser Visual Studio, il faut au moins un éditeur de texte et le compilateur `gcc` à disposition

Conseil: activer l'auto save: Aller dans Settings (Cmd/Ctrl + ,) chercher `auto save` et choisir `onFocusChange`

E. Structure d'un programme C

Un programme C est composé de fichiers texte avec l'extension `.c`

Voici un exemple simple d'un programme:

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

// Fonction principale
int main()
{
        /*
      Afficher un message dans le terminal
        */
    printf("Hello world!\n");

    return 0;
}
```

Chaque programme doit avoir **une seule** fonction `main` qui est executée au demarrage du programme. La structure de base d'un programme C comprend des directives de préprocesseur, des déclarations de variables, des définitions de fonctions et des commentaires.

Décomposons cet exemple:

Inclusions

Les directives commençant par un symbole `#` sont appelés des instructions de **préprocesseurs**. Nous y reviendrons plus tard, mais ici nous avons affaire à des inclusions de librairies standard.

Commentaires

Il est possible d'ecrire des remarques dans le code qui ne serviront qu'a la personne qui le lira et n'affectera en rien le programme. Cela peut aider pour expliquer ce qui se passe dans le programme à d'autres personnes. Les commentaires sont precedés de `//` sur une seule ligne, ou entourées de `/*` et `*/` quand on ecrit sur plusieurs lignes. Ces parties sont généralement colorés en gris dans les éditeurs.

Fonction principale

Nous avons ensuite la fonction `main` qui est déclarée et qui sera lancée au début du programme. Une fonction contient un ensemble de code entre ses accolades `{` et `}`

Affichage

L'appel de la fonction `printf` permet d'afficher quelque chose dans le terminal qui execute le programme. Elle appartient a la librairie standard `stdio`

Retour

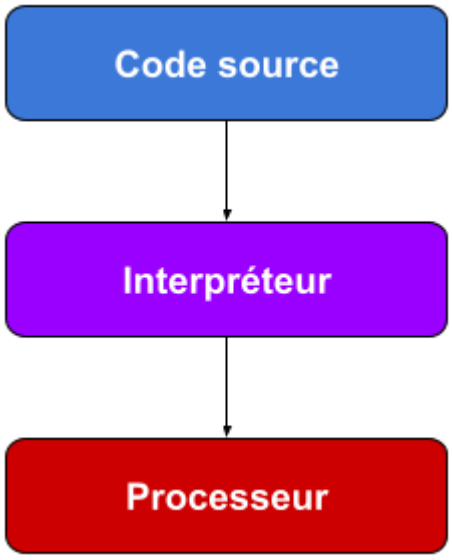
La fonction `main` retournera la valeur 0, ce qui annonce que le programme s'est bien déroulé.

F. Compilation et exécution d'un programme

La compilation est le processus par lequel le code source C est transformé en code machine que l'ordinateur peut exécuter.

Les langages interprétés

Un langage interprété nécessite l'utilisation d'un interpréteur qui va exécuter le code source. L'interpréteur est un programme intermédiaire qui analyse et exécute les lignes (généralement une par une). Les langages interprétés les plus populaires sont : MATLAB, PHP, Python, Java ... Chacun de ces langages nécessite l'installation de l'interpréteur associé pour fonctionner.

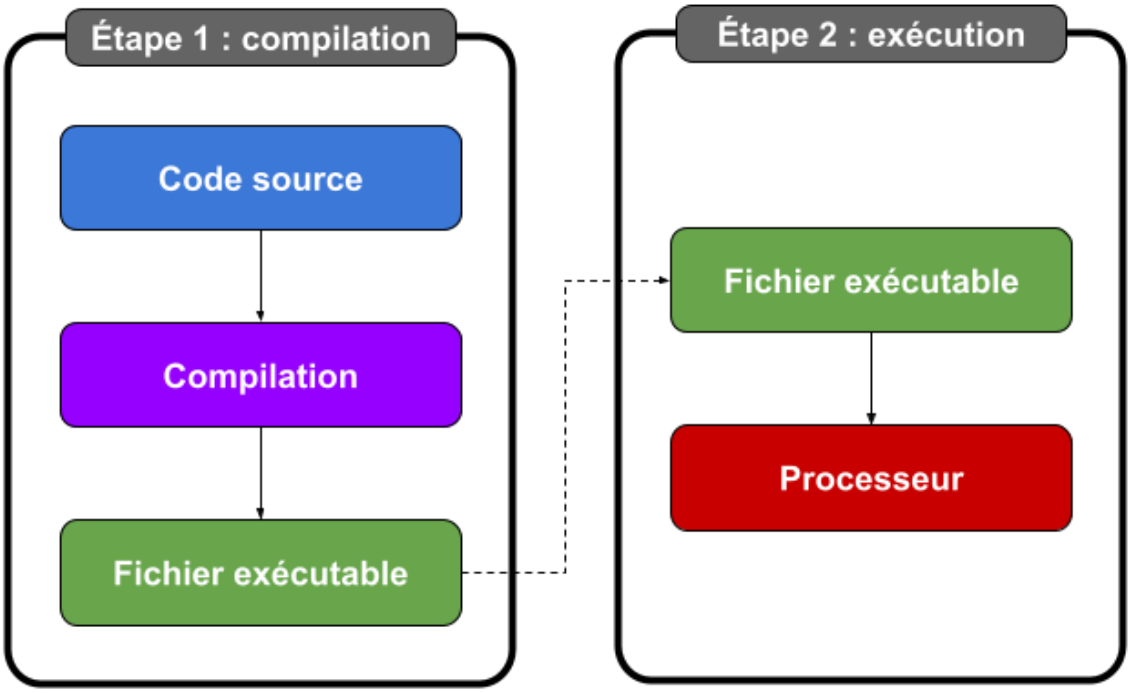


Les langages compilés

Avec les programmes compilés, l'exécution se fait en deux étapes :

1. Le code source est d'abord transformé en langage machine.
2. Ensuite le code transformé est directement exécuté dans le processeur

La première étape, la transformation du code source en langage machine est appelée **compilation**. La sortie de la compilation est un fichier exécutable, typiquement un fichier `.exe` sous Windows. Voici quelques exemples de languages compilés : C, C++, C#, swift, Pascal...



De manière générale, les langages compilés sont plus performants, car ils sont exécutés directement dans le processeur. Pour avoir un ordre de grandeur, cette page présente une comparaison entre Python et C++. Le même programme dure 15 minutes en Python, contre 30 secondes en C++. Le graphique ci-dessous donne un ordre de grandeur des ratios d'exécution des différents langages. On s'aperçoit que le C est 10 fois plus rapide que PHP 7 et 40 fois plus rapide que le même programme en Python.

image.png

La contrepartie est la durée de compilation : si elle est généralement instantanée, elle peut parfois être beaucoup plus longue sur de gros projets. Par exemple la compilation d'un noyau Linux peut durer une dizaine d'heure. Évidemment, si le code source n'est pas modifié, il est inutile de relancer la compilation avant chaque exécution.

Compilation

Le compilateur le plus connu pour le C s'appelle **gcc**. Il est exécutable en ligne de commande.

💡

Ne pas confondre éditeur de code et compilateur:

L'éditeur de code (Visual Studio Code) est un simple éditeur de texte qui enregistre des fichiers texte. Grace à ses outils et plugin il permet de simplifier le travail du développeur.

Le compilateur (gcc) va transformer les fichiers texte en fichiers exécutables par l'ordinateur.

Pour compiler un fichier, il faut lancer la commande suivante:

Linux/Mac

```
gcc -Wall main.c -o main
```

Windows

```
gcc -Wall main.c -o main.exe
```

- `gcc` : appel du programme
- `-Wall` Afficher tous les warnings
- `main.c` : fichiers source à compiler
- `-o main` : enregistrer le resultat (**o**utput) dans le fichier `main`

Nous avons maintenant un fichier `main` (linux/mac) ou `main.exe` (windows).

Execution

Nous créerons ici principalement des programmes qui seront exécutés en *ligne de commande*.

Une fois que votre terminal est ouvert et que vous êtes dans le repertoire du programme, vous pouvez le lancer avec ces commandes:

`./main` *// Linux/Mac*

`./main.exe` *// Windows*

Vous verrez alors s'afficher les sorties de votre programme.

Sous Linux/Mac on peut faire les deux en même temps via l'opérateur `&&`

`gcc main.c -o main && ./main`

Erreurs de compilation

Si le code que l'on essaie de compiler contient des erreurs, la compilation échouera et le compilateur nous donnera des indications sur les problemes pour nous aider à les resoudre. Bien décoder ces messages d'erreur sera d'une grande aide pour corriger les problemes.

```jsx
/tmp/wBHb2sextG.c: In function 'main':
ERROR!
/tmp/wBHb2sextG.c:6:32: error: expected ';' before 'return'
6 |     printf("Try [programiz.pro](http://programiz.pro/)")
|                                ^
|                                ;
7 |
8 |     return 0;
|     ~~~~~~
```

Ici le message nous indique que nous avons oublié un `;` ligne 6

II. La Syntaxe en C

Un code en C est régi par des règles strictes qu'il faut impérativement respecter pour que le programme aie le comportement que l'on désire et même qu'il compile.

Les mots-clés et identificateurs

Les **mots-clés** sont des mots réservés en C qui ont une signification spéciale pour le compilateur. Ils ne peuvent pas être utilisés comme noms de variables

ou de fonctions. Des exemples de mots-clés incluent `int`, `return`, `if`, et `while`.

Les **identificateurs**, en revanche, sont les noms donnés par le programmeur aux variables, fonctions, tableaux, etc. Ils doivent commencer par une lettre ou un underscore et peuvent être suivis de lettres, de chiffres ou d'underscores.

Instructions

Celui ci est séparé en instructions, elles mêmes séparées par des points-virgules `;`

Les instructions sont:

- déclaration de variable
- affectation de variable
- appel de fonction

Les instructions ne sont pas:

- conditions
- boucles
- préprocesseur

```c
// premiere instruction
int a = 0;
// deuxieme instruction
float b = 1.0;
// troisieme instruction
if(a > 2) {
    printf("oui");
}
```

Les sauts de ligne et les espacements n'ont pas d'impact sur l'interprétation du code, en revanche il aide grandement à la lisibilité.

Le code précedent est strictement équivalent au suivant, mais ce dernier est bien moins lisible:

```c
int a=0;float b=1.0;if(a>2){printf("oui");}
```

Bloc

Un bloc est un morceau de code englobé par des accolades `{` et `}`

Il est utilisé pour les fonctions, boucles et conditions.

Les variables qui sont déclarées dans un bloc **ne sont valides que dans ce bloc**. Cela signifie que en dehors de ses limites (accolades), elles n'existent pas.

Lors de l'entrée dans un bloc, on indente d'une colonne (voir paragraphe suivant).

```c
int a = 5;
if(a < 2) {
    int b = 5;
    printf(b);
}
if(a > 2) {
    printf(b); // b n'existe pas ici
}
```

```c
int a = 5;
if(a < 2) {
    int b = 5;
    printf(b);
}
if(a > 2) {
    int b = 4;
    printf(b); // b existe ici mais est differente
}
```

Ici dans le deuxième bloc on tente de lire b, mais elle a été déclarée dans un autre bloc, on aura donc une erreur de compilation.

Il est en revanche possible d'imbriquer des blocs, et d'acceder aux variables du bloc supérieur (pas inférieur)

```c
int a = 5;

if(a < 10) {
    int b = 5;
    printf(b);

    if(a > 2) {
        printf(b); // b existe
        int c = 12;
    }
    printf(c); // c n'existe pas ici
}
```

Indentation

L'indentation est l'alignement vertical du code, qui doit correspondre au bloc dans lequel est le code.

Quand on ouvre un bloc, on indente d'un cran vers la droite, quand on ferme un bloc, désindente vers la gauche.

```c
int main() {    // nouveau bloc
    int a = 0;    // Indenté d'un cran
    int b = 0;    // meme bloc, meme cran

    if(a == b) {  // nouveau bloc
        int c = 0;  // Indenté d'un cran
    }             // bloc fermé, on désindente

    int d = 0;
}               // bloc fermé, on désindente
```

Sur Visual studio, vous pouvez activer l'indentation automatique.

Aller dans Settings (Cmd + ,) chercher `format on save` et activer Editor Format On Save

💡

Lors des évaluations il est impératif de rendre un code propre, c'est a dire clair, bien indenté, bien commenté, sans superflu

III. Les variables

Comme dans tous les langages de programmation, le C utilise des variables. Les variables peuvent contenir des entiers, des réels, du texte .... Contrairement aux *constantes*, le contenu des *variables* est amené à changer au cours de l'exécution d'un programme.

Ces variables serviront a stocker et manipuler les informations que nous devront calculer, telles que des données, des messages, des images, ou des équations. Un ordinateur est avant tout une machine a compter (*computer*), notre rôle de programmateur est de faire faire aux machines les opérations qui nous rendront les services que l'on attend.

A. Un langage typé

Le C est un langage typé, c'est-à-dire que chaque variable est définie pour un type donné (entier, réel, texte...) :

- Pour utiliser une variable, on doit **déclarer** son **type** *et* son **nom:**

```c
int nombre;
```

Ici on a déclaré une variable de type `int`, qui s'appelle `nombre`

La déclaration sert à allouer la mémoire nécessaire au stockage de la variable.

- Pour stocker une valeur dans une variable, on effectue une **affectation** via l'opérateur `=`

```c
nombre = 10;
```

- On **peut** affecter une valeur d'une variable lors de sa déclaration

```c
int nombre = 10;
```

B. Types de variables

En C, on peut distinguer 3 classes principales de types de variables :

- Les entiers (`int`) qui servent à mémoriser des nombres entiers, sans virgule : 0, 1, 25, -1024 ...
- Les flottants (`float`) qui servent à mémoriser des nombres réels, donc à virgule : 0.2 , 15.5, -3.1415 ...
  - Dans la plupart des langages, les virgules sont représentées par des points `.` et non pas des virgules comme en français, qui elles représentent plutôt des séparateurs.
- Les caractères (`char`) qui servent à mémoriser des caractères alpha-numérique : A, B, C ... Z, 0, 1, 2 ...9, ?, /, %, # ...
  - Les caractère sont écrits entre simple apostrophes en C : `'a'`

Tailles

Chaque type de variable a un usage different et surtout prendra une place différente dans la mémoire. Plus il prendra de place dans la mémoire, plus ses valeurs pourront êtres elevées.

💡

Avec 1 bit on peut compter jusqu'a 1, avec 2 bits, jusqu'à 3, 3 bits jusqu'à 7, etc...

Avec X bits on peut compter jusqu'a `2^X-1`

Signes

Chaque type peut être précédé du mot clé `unsigned` qui signifiera que ses valeurs démarreront de zero, donc pas de valeurs négatives. Cela doublera donc leur capacité.

Si une variable est signée, sa plage de valeurs est divisée par deux et partagée entre les nombres négatifs et positifs

Tailles et plages de valeurs

Type	Taille (bits)	Taille (octets)	Borne inférieure	Borne supérieure
`char`	8 bits	1	-128	+127
`unsigned char`	8 bits	1	0	255
`int`	32 bits	4	-2,147,483,648	+2,147,483,647
`unsigned int`	32 bits	4	0	4,294,967,295
`long`	64 bits	8	-9223372036854775808	+92233720368547
`float`	32 bits	4	1.2E-38	3.4E+38 (6 décima
`double`	64 bits	8	2.3E-308	1.7E+308 (15 décim

Ces informations sont données a titre d'exemple, elles peuvent changer selon le compilateur et le système d'exploitation cible

Pour rappel 1 octet = 8 bits

Booléens

Les booléens sont des chiffres ne pouvant représenter que deux valeurs, à l'image d'un bit. On nomme ces valeurs soit "vrai" ou "faux" ("true/false"), représentés par l'entier 1 (vrai) ou 0 (faux)

C. La mémoire

Dépassement

Si la valeur d'une variable dépasse ses bornes, sa valeur fera le tour et rejoindra l'autre borne.

Par exemple:

```c
char heure = 127;
heure = heure + 1;
// heure == -128
```

Lire plus sur le dépassement d'entiers

Représentation en mémoire

Un ordinateur ne connait que les bits, et ne sait pas compter jusqu'a 10. Les variables seront donc représentées en une série de bit en mémoire qui représentera sa valeur en fonction de son type.

Voici comment sont représentées les valeurs d'un `unsigned char` de 8 bits:

Base 10	Base 2
0	00000000

Base 10	Base 2
1	00000001
2	00000010
...	
254	11111110
255	11111111

C'est pour cela que si on additionne 255 + 1 on obtient 0

D. Nomenclature

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
>
> - *- Phil Karlton*

Noms

Afin que le compilateur puisse distinguer le nom des variables du reste du code, il existe des restrictions sur le nommage des variables.

Les noms de variables:

- doivent représenter ce qu'elles stockent
- peuvent être composés de plusieurs caractères.
- peuvent contenir des lettres minuscules ou majuscule (`a à z et A à Z`).
- peuvent contenir des chiffres (`0 à 9`).
- peuvent contenir un tiret bas (`_`).
- ne peuvent pas commencer par un chiffre.
- ne peuvent pas etre des **mots clés** du langage (`main` , `return` , `int` ...).

Casing

Il est préférable de garder une consistence dans la manière dont on nomme les variables qu'on appelle la casse: lire plus ici

Par exemple:

```c
int couleurVoiture;   // camelCase
int couleur_voiture;  // snake_case
int CouleurVoiture;   // PascalCase
#DEFINE COULEUR_VOITURE;  // UPPER_CASE
```

IV. Entrées et sorties

Afin de pouvoir être interactif, un programme peut recevoir et envoyer des informations à l'utilisateur où à d'autre programmes.

Pour les débuts nous utiliseront principalement des programmes en ligne de commande, donc les sorties seront simplement ce que l'on verra apparaitre dans le terminal et les entrées ce qu'on pourra taper au clavier dedans.

Le C utilise des fonctions de la bibliothèque standard pour l'entrée et la sortie de données. La bibliothèque `<stdio.h>` contient des fonctions comme `printf()` pour l'affichage de sortie et `scanf()` pour la saisie d'entrée. Ces fonctions permettent d'interagir avec l'utilisateur en lisant des données du clavier et en affichant des résultats à l'écran.

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

A. Affichage avec `printf`

Pour afficher du texte dans le terminal, nous utiliserons principalement la fonction `printf` qui signifie "print formatted" ou afficher de manière formatée. On va lui donner une chaine de caractères qui pourra contenir des variables.

Les variables dans la chaine de caractères seront symbolisée par `%d` où `d` représentera le type de la variable, puis on passera les variables en argument de la fonction dans l'ordre dans lequel elles apparaissent dans la chaine de caractère.

Pour sauter une ligne, nous écrirons `\n`

Exemple

```c
printf("Bonjour !\n");

int age = 25;
printf("%d", age);

// > 25
```

Cette syntaxe sera plus claire quand les bases seront connues entièrements, mais expliquons ce qui se passe ici:

- `printf(...)` on appelle la fonction `printf`
- `"%d"` le premier argument de la fonction représente ce que l'on veut afficher et dans quel format. Ici on va juste afficher un entier
- `age` est l'entier que l'on veut afficher

Exemple plus complet:

```c
int age = 25;
int departement = 33;
printf("Age: %d Département: %d", age, departement);
// > Age: 25 Département: 33
```

Ici on a mélangé du texte brut avec des variables.

Types de variables à afficher

Type	Code de format	Types
entiers signés	`%d`	`char`, `int`, `long`
entiers non signés	`%u`	`unsigned char`, `unsigned int`, `unsigned long`
flottants	`%f`	`float`, `double`
caractère	`%c`	`char`, `unsigned char`

B. Entrée avec `scanf`

Dans le sens inverse, si on veut récupérer une valeur à l'utilisateur, on va utiliser `scanf`

Son utilisation est très similaire à printf, sauf qu'on utilisera des `&` avant les variables à enregistrer.

Ce détail sera vu en profondeur dans le module de C avancé, mais pour faire court, c'est un moyen qui permettra a `scanf` de modifier les valeurs des variables qu'on lui passe. Sans le symbole `&` , les fonction ne peuvent que lire les variables qui lui sont passés en arguments.

Exemple

```c
#include <stdio.h>

int age;
printf("Quel est votre age ?\n");
scanf("%d", &age);
printf("Vous avez %d ans.\n", age);
```

🚧

Faire les exercices “Basiques”

V. Opérateurs

Les opérateurs permettent d’effectuer des opérations mathématiques entre plusieurs valeurs, telles que des additions ou des comparaisons.

A. Opérations arithmétiques

Ce sont les opérations mathématiques de bases telles que l’addition. On va juste additionner deux nombres et on aura un résultat.

```c
// addition
c = a + b;
// soustration
c = a - b;
// division
c = a / b;
// multiplication
c = a * b;
// modulo
c = a % b;
```

Opérateurs unaires

Certains opérateurs ne prennent qu’un une opérande, tel que l’inversion:

```c
c = -a;
```

Opérateurs d’affectation

Pour écrire du code plus vite, il existe des raccourcies appelés opérateurs d’affectation:

```c
a += b;   /* == */   a = a + b;
a -= b;   /* == */   a = a - b;
a *= b;   /* == */   a = a * b;
a /= b;   /* == */   a = a / b;
a %= b;   /* == */   a = a % b;
```

Opérateurs d'incrémentation

Il arrive très souvent de devoir augmenter ou diminuer de `1` une variable, il y a donc un opérateur spécial pour cela:

```c
i++;    /* == */  i = i + 1;
i--;    /* == */  i = i - 1;
```

On peut aussi inverser le sens de l'opérateur, ce qui changera la valeur de retour: `++i` ou `--i`

Pour `i++` la valeur retournée est la valeur non incrémentée, alors que pour `++i` la valeur incrémentée est retournée.

Exemple

```c
int a = 2;
int b = a++; // a == 3, b == 2
int c = ++a; // a == 4, c == 4
```

Modulo

Le modulo est un opérateur qui retourne le **reste de la division euclidienne** de deux nombres.

C'est un opérateur très important en programmation et est très souvent utilisé, il est important de bien le maitriser.

💡

Rappel

Une division **traditionnelle** peut retourner des valeurs décimales. Par exemple, 3/2 retourne 1,5.

Une division **euclidienne** retournera deux valeurs entières distinctes: un **quotient** et un **reste.**

Par exemple la division euclidienne de 7 par 2 donne: `7 = 2 * 3 *(quotient)* + 1 *(reste)*`

Lire plus sur wikipedia

Exemples:

```c
3 % 2 == 1
5 % 2 == 1
4 % 3 == 1
5 % 3 == 2
```

Si le reste de la division euclienne de `a` par `b` vaut 0, alors cela signifie que `a` est divisible par `b`

`8 % 4 == 0` ⇒ 8 est divisible par 4

Dans le cas d'un dépassement de valeurs d'une variable, on peut expliquer une nouvelle valeur via le modulo.

Par exemple, pour un `unsigned char` qui a ses valeurs bornées entre 0 et 255, une affectation en valeur peut être succédée d'un modulo 256:

```c
unsigned char a;

a = 1;              // équivalent à
a = 1 % 256;        // a == 1

a = 256;            // équivalent à
a = 256 % 256;      // a == 0
```

```
a = 257;            // équivalent à
a = 257 % 256;      // a == 1
```

Résultats des opérations et types de variables

Ne pas oublier le type de variable dans lequel on stocke le résultat d'une variable. Il peut arriver que le résultat soit modifié pour rentrer dans le type de variable de destination, tel qu'un changement de flottant vers entier ou un dépassement de taille

```c
int a = 3/2; // a == 1
float b = 3/2; // b == 1.5
unsigned char c = 200 * 2; // c == 144 (400 % 256)
```

B. Opérations booléennes

Les opérations booléennes sont des opérations qui prendront plusieurs valeurs et retourneront toujours un booléen qui indiquera un résultat qui ne peut être que soit vrai ou faux.

Opérations de comparaison

Ce sont des opérateurs de comparaison. Nous allons ici prendre deux valeurs, qui pourront êtres de n'importe quel type, les comparer, et retourner un booléen correspondant au résultat de la comparaison.

Prenons l'égalité `a == b` : si `a` est égal à `b` alors l'opération retournera `1` (vrai), sinon `0` (faux)

```c
// égalité
a == b
// non-égalité
```

```
a != b

// a strictement supérieur à b
a > b
// a supérieur ou égal à b
a >= b

// a strictement inférieur à b
a < b
// a inférieur ou égal à b
a <= b
```

On peut tester des opérateurs booléens en les utilisant en tant qu'entiers:

```c
int a = 2;
int b = 3;

int c1 = a == b;  // c1 == 0
int c2 = a != b;  // c2 == 1
int c3 = a < b;   // c3 == 1
int c4 = a <= b;  // c4 == 1
int c5 = a > b;   // c5 == 0
int c6 = a >= b;  // c6 == 0
```

Attention aux comparaisons de valeurs de types différents, on peut avoir des surprises !

Opérateurs logiques

Les opérateurs logiques fonctionnent avec deux booléens pour en retourner un troisième. Cela va permettre de combiner plusieurs comparaisons pour former des **conditions** plus complexes.

```c
// AND (ET logique)
a && b

// OR (OU logique)
```

```
a || b

// NOT (NON logique)
!a
```

Tableau Opérateurs logiques AND et OR

A	B	ET	OU
VRAI	VRAI	VRAI	VRAI
VRAI	FAUX	FAUX	VRAI
FAUX	FAUX	FAUX	FAUX
FAUX	VRAI	FAUX	VRAI

Tableau Opérateur logique NOT

A	NON
VRAI	FAUX
FAUX	VRAI

Exemple

```c
int a = 0;
int b = 1;

int c1 = a && b;    // c1 == 0
int c2 = a || b;    // c2 == 1
int c3 = !a;        // c3 == 1
int c4 = !b;        // c4 == 0
```

On va en général utiliser des comparaisons de chaque cotés:

```c
int a = 5;
int b = 7;
int c = 9;

int c = (a < b) && (b < c);
```

Le **NON** sert à indiquer que l'on ne veut pas qu'une condition se réalise

```c
!(a < b)  == (a >= b) // on ne veut pas que a soit strictement plus
!(a == b) == (a != b) // on ne veut pas que a égal à b
```

Loi de Morgan

Elle permet d'inverser des opérateur booléens

```c
(a && b) == !(!a || !b)

// Par exemple:
(p>=0 && p<=100) == !(p<0 || p>100)
```

Priorité des opérateurs

Comme en mathématiques, les opérateurs ont des priorités pour être sur dans quel ordre ils vont êtres calculés.

Les règles sont strictes mais complexes (voir ici), il est donc fortement recommandé d'utiliser des parenthèses pour séparer plusieurs opérations afin de ne pas se perdre et garder un code lisible. Si l'opération est trop complexe, séparer en plusieurs opérations

```c
x =  3 * 2 < 4;     // dans quel ordre ?
x =  (3 * 2) < 4;   // d'abord la multiplication !

y = a + b * 2 % 4 >= 3 + c;  // ...
```

```c
y = a + (((b*2) % 4)  > 3 ) + c; // (un peu) plus clair

y1 = (b*2) % 4;
y = a + (y1 > 3) + c;  // mieux en plusieurs lignes
```

🚧

Faire les exercices “Opérateurs”

VI. Contrôles de flux

Les contrôles de flux permettront de réagir conditionnellement au valeurs des variables et répéter des opérations un nombre déterminé de fois

---

A. Structures conditionnelles

Les structures conditionnelles permettent d’exécuter ou non certaines parties d’un programme selon les valeurs des variables

if...else

L'instruction `if...else` permet d'exécuter un bloc d'instructions selon le résultat d'un test

:

Syntaxe

```c
if (/* condition*/) {
  // Bloc d'instruction 1
}
else {
  // Bloc d'instruction 2
}
```

Si la *condition* est vraie alors on exécutera le *bloc 1,* sinon le *bloc 2*.

Il est à noter que :

- il ne faut pas mettre de point virgule à la fin du `if` (ce n'est pas une instruction)
- les parenthèses dans le test sont obligatoires
- le `else{...}` est facultatif
- les accolades peuvent être omises si le bloc ne contient qu'une instruction

Si on veut tester une deuxième condition si la première n'est pas satisfaite on peut utiliser `else if`

Syntaxe

```c
if (/* condition 1 */) {
  // Bloc d'instruction 1
}
else if (/* condition 2*/) {
  // Bloc d'instruction 2
else {
  // Bloc d'instruction 3
}
```

Si la *condition 1* est vraie alors on exécutera le *bloc 1*, sinon si la *condition 2* est vraie on exécutera le *bloc 2*, sinon le *bloc 3*

⚠

Attention: dans une structure `if ... else if ... else ...` **un seul des blocs sera executé**: le premier dont la condition est valide.

Si il y a une condition valide (`if` ou `else if`), le prochain bloc sera executé, mais si la condition suivante (`else if`) est valide aussi elle ne sera pas executée.

Exemples

```c
int a = 12;
int b = 24;

if(a > b) {
    printf("a est strictement plus grand que b");
} else {
    printf("a est inferieur ou égal à b");
}

if(a == b)
    printf("a est égal à b");

if(a < b) {
    printf("a est strictement inférieur à b");
}
else if (a == b) {
    printf("a est égal à b");
} else {
    printf("a est strictement plus grand que b");
}
```

Si on veut combiner des conditions, on peut utiliser les opérateurs logiques:

```c
if(a < b) {
    if(b < c) {
        // ...
    }
}

// Equivalent à
if( (a < b) && (b < c) ) {
    // ...
}
```

Switch

L'opérateur conditionnel `switch` permet de tester différentes valeurs d'une variable

Exemple

```c
char lettre = 'a';
switch (lettre) {
  case 'a':
  {
      printf("la lettre a");
      break;
  }
  case 'b':
  {
      printf("la lettre b");
      break;
  }
  default:
  {
      printf("une autre lettre");
  }
}
```

B. Boucles

Les boucles permettent de répéter un morceau de programme un nombre déterminé de fois.

On appelle un tour de boucle une **itération**

While

La boucle `while`, *"tant-que"* en français, est la boucle de base en C. Elle va répéter un bloc d'instruction tant qu'une condition est vraie.

Syntaxe

```c
while(/* condition */) {
    // bloc d'instruction
}
```

Sa syntaxe est très proche du `if`, si la condition est vraie alors le bloc d'instruction sera -exécuté. A la fin du bloc, la condition sera testée à nouveau. Si elle est encore vraie, le bloc sera rejoué à nouveau. Si il est faux, le programme continuera après le bloc.

Exemple

```c
int a = 0;
while(a < 10) {
    printf("%d", a);
    a++;
}
```

Ici on peut lire la boucle comme "tant que `a` est inférieur à 10, alors éxécuter le bloc"

Ce code affichera les valeurs de `i` entre 0 et 9

La boucle `while` est généralement utilisée quand une variable peut prendre des valeurs non régulières, mais que l'ont souhaite qu'elle respecte une condition

do...while

Il existe une variante du while qui permet d'executer au moins une fois le bloc avant de tester la condition:

```c
do {
    // bloc
```

```
} while (/* condition);
```

Attention au point-virgule après le `while` ici ! Il n'y en as pas dans le `while`

For

Sur un principe similaire, il existe la boucle `for`, "pour" en français. La différence est ce qui va se trouver entre les parenthèse, et au lieu de juste vérifier une condition, on pourra initialiser une variable, tester sa valeur à chaque itération, et modifier sa valeur à chaque itération si la condition est remplie.

Syntaxe

```c
for(/* initialisation */ ; /* condition */ ; /* itération */) {
  // bloc d'instruction
}
```

Exemple

```c
for(int i=0; i < 10; i++) {
  printf("%d", i);
}
printf("%d", i); // pas dans la boucle, i n'existe pas ici
```

Ce code affichera les valeurs de `i` entre 0 et 9

Ici on peut lire la boucle comme "nous utiliserons la variable entière `i` de valeur initiale 0, tant qu'elle est strictement inférieure à 10 alors exécuter le bloc, puis incrémenter de 1 à chaque itération"

La boucle se passe en trois phases:

- 1 - Initialisation
  - On déclare une variable `int i`
- 2 - Condition

- Si la condition `i < 10` est remplie, on éxécute le bloc d'instruction, sinon on sort de la boucle

- 3 - Fin de bloc
  - On exécute le code d'itération `i++` et on retourne a l'étape 2

Toujours pas de point-virgules autour du bloc for, en revanche les trois phases entre les parenthèses sont elles bien séparées par des points-virgules

La boucle `for` est généralement utilisée quand on veut répéter une opération un certain nombre de fois et de manière séquentielle sur une plage de valeurs.

Arrets de boucle

Il est possible d'arrêter une boucle ou de sauter le reste d'un bloc en dehors de la phase de test de la condition.

Ces instructions peuvent être utilisés dans n'importe quel type de boucle.

`break`

Ce mot-clé peut être utilisé dans une boucle pour l'arrêter complétement au moment où elle est appelée.

```c
for(int i=0; i < 10; i++) {
  if(i == 5) {
      break;
  }
  printf("%d", i);
}
```

Ce code affichera les valeurs de `i` entre 0 et 4

`continue`

Ce mot-clé peut-être utilisé pour sauter le reste du bloc en cours et recommencer au début de la boucle.

```c
for(int i=0; i < 10; i++) {
  if(i == 5) {
      continue;
  }
  printf("%d", i);
}
```

Ce code affichera les valeurs de `i` entre 0 et 9 mais n'affichera pas 5

Boucles infinies

Il est important de bien faire attention aux valeurs que peuvent prendre les variables dans la condition pour ne pas se retrouver dans une boucle infinie, avec une condition qui ne passe jamais à *false* et qui ne s'arrête jamais.

```c
while(1) {
    // boucle infinie
}

int a = 0;
while(a >= 0) {
    a++;
    // boucle infinie
}
```

C. Exemple complet

Voici un exemple pratique d'un programme qui demande l'âge d'une personne et qui affiche le résultat:

```c
#include <stdio.h>

int main(void)
{
```

```
    int age;

    printf("Quel âge avez-vous ? ");
    scanf("%d", &age);
    printf("Vous avez %d an(s)\n", age);

    if (age < 20)
    {
        printf("Vous êtes jeune\n");
    }
    else
    {
        printf("Vous êtes vieux\n");
    }

    for (int i = 0; i < age; i++)
    {
        printf("🕯️ ");
    }
    printf("\n");

    return 0;
}
```

🚧

Faire les exercices “Contrôles de flux”

VII. Les fonctions

Les fonctions sont des morceaux de programmes qu’on va pouvoir réutiliser. Si on doit faire la même chose à différents moments, on ne réécrira pas deux fois le meme code, mais on créera une fonction avec ce code, et c’est cette fonction qu’on utilisera.

Exemple

```c
// sans fonction
int r1 = a1 * 32 + 5;
```

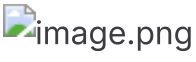
```c
int r2 = a2 * 32 + 5;

// avec fonction
int eq(int a) {
    return a * 32 + 5;
}
int r1 = eq(a1);
int r2 = eq(a2);
```

Une fonction est une suite d'opérations selon des paramètres variables qui retourne un résultat.

Comme pour les variables, il n'est pas possible de créer plusieurs fonctions du même nom, et de donner le même nom à une fonction et à une variable.

A. Définition et déclaration de fonctions

Chaque fonction a ce qu'on appelle un **prototype**. Ce prototype définira les **arguments** que prendront la fonction, et le **type** de valeur qu'elle **retournera**.



```c
float diviser(float a, float b) {
    float c = b / a;
    return c;
}

int main() {
    float z = 1.0;
    diviser(z, 2.0);
}
```

Une fonction est définie en dehors de tout bloc. On ne peut pas définir une fonction dans une fonction.

B. Passage de paramètres et valeurs de retour

Une fonction pourra recevoir **autant d'arguments** que souhaité, aussi appelés **paramètres**, et il est possible de ne pas en recevoir du tout.

```c
int fonctionSansArgument() {
}
int fonctionAvecUnArgument(int a) {
}
int fonctionAvecDeuxArguments(int a, int b) {
}
```

Une fonction doit avoir une valeur de **retour**, et **impérativement** retourner une valeur de ce **type**

```c
int fonctionInt() {
    return 1;
}
float fonctionFloat() {
    return 1.1;
}
char fonctionChar() {
    return 'a';
}
```

```c
int fonctionInt() {
    int a = 1;
    return a;
}
float fonctionFloat() {
    float a = 1.1;
    return a;
}
char fonctionChar() {
    char a = 'z';
```

```
    return a;
}
```

Lorsqu'on utilise des structures conditionnelles, on doit faire attention a retourner une valeur **quelque soit son parcours logique**

```c
int fonctionCondition(int a) {
    if(a>0) {
        printf("supérieur a 0");
        /*
        Dans cette branche on ne retourne rien, ce n'est pas normal
        */
    }
    else {
        printf("inferieur a 0");
        return 12;
    }
}
```

Une fonction peut ne pas retourner de valeur, dans ce cas le type de retour sera `void`, et on pourra appeler `return` sans valeur

```c
void fn(int a) {
    if(a<0) {
        return;
    }
    // ...
}
```

C. Portée des variables (locale, globale)

Toutes les variables créées dans une fonction seront locales à cette fonction, y compris ces arguments, comme dans tout nouveau bloc, et similaire au `for` pour les arguments

```c
int fn1() {
    int b = 2;
    printf("%d", b);
    if (b < 10) {
        int b = 2; // ne fonctionne pas car b existe déjà dans le bl
        printf("%d", b);
    }
}
int fn2() {
    printf("%d", b); // b n'existe pas ici
}

fn1();
fn2();
```

La fonction principale `main`

Il existe une fonction particulière, appelée `main` qui doit exister une seule fois, et c'est elle qui sera appelée au début du programme.

Son prototype habituel est:

```c
int main()
```

Elle ne prend pas d'arguments et elle retourne un entier.

La valeur qui est retournée servira a indiquer au terminal qui a lancé le programme si il s'est bien déroulé ou non.

Quand on appelle `return` dans le main, le programme s'arrete

Si on retourne `0`, on annonce que le programme s'est bien passé

Si on retourne un nombre positif on annonce qu'il y a eu un problème.

```c
int main()
{
    int age;
    printf("Quel âge avez-vous ? ");
    scanf("%d", &age);

    if(age < 0) {
      return 1;
    }
    return 0;
}
```

Ces codes sont repéesentés par les constantes `EXIT_SUCCESS` et `EXIT_FAILURE` dans `<stdlib.h>`

Récursivité

Une fonction est dite récursive lorsqu'elle s'appelle elle même.

🚧

Faire les exercices "Fonctions"

VIII. Notions avancées

Tableaux

Les tableaux, aussi appelés vecteurs, sont des façons de stocker des **suites** de **variables** du **même type**.

Ils sont utile pour stocker des informations qui nécessitent plusieurs valeurs, comme des coordonnées, couleurs rgb, ..., ou emmagasiner plusieurs résultats de calcul.

Définir un tableau est similaire à définir une variable, sauf qu'on mentionnera sa taille entre crochets `[]`

```c
int a;      // definition d'une variable entière
int b[10];  // définition d'un tableau d'entiers de taille 10
```

Exemples d'utilisation:

Pour lire ou écrire une valeur dans le tableau, on utilisera les crochets egalement:

Les **index** d'un tableau commencent à 0. Le premier élement est à l'index 0, le deuxième à 1,...

```c
b[0] = 10;          // on affecte 10 au premier element du tableau

printf("%d", b[0]); // on lit le premier element du tableau
```

Lors de la declaration (uniquement), on peut affecter les valeurs en même temps en les mettant dans des accolades `{}` et séparé par des virgules `,` . Lorsqu'on affecte des valeurs lors de la définition, on n'est pas obligé d'indiquer la taille du tableau, elle sera déduite du nombre d'élements affectés.

```c
int c[3] = { 1, 2, 3 };
int d[] = { 1, 2, 3 };
```

Quand une fonction prend en paramètre un tableau, il est nécessaire d'indiquer également la taille du tableau a la fonction qui ne peut pas la deviner.

```c
#include <stdio.h>

void printTableau(int t[], int taille)
{
    for (int i = 0; i < taille; i++)
    {
        printf("t[%d] = %d\n", i, t[i]);
    }
    printf("\n");
```

```
}

int main(void)
{
    int t1[5] = {1, 2, 3, 4, 5}; // explicitement taille de 5
    int t2[] = {10, 11};         // implicitement taille de 2

    printf("t1[0]: %d\n", t1[0]); // on compte à  partir de 0
    printf("t1[4]: %d\n", t1[4]); // pour une taille de 5, le dernie

    t1[3] = 6;
    printf("t1[3]: %d\n", t1[3]);

    t3[0] = 100;

    printf("t1[0]: %d\n", t1[0]);
    printf("t2[0]: %d\n", t2[0]);
    printf("t3[0]: %d\n", t3[0]);

    printTableau(t1, 5);
    printTableau(t2, 5); // t2 est de taille 2 donc il y aura une er
}
```

Chaines de caractères

Les chaines de caractères sont simplement des suites de caractères. Le type `char` ne peut stocker qu'un caractère. Pour stocker une suite de charactère, on utilisera des tableau de `char`

Une chaine de caractère est écrite avec des double apostrophes `"`

Les chaines de caractères étant des tableaux, on ne peux affecter leur valeur qu'a l'initialisation. Après, on ne pourra qu'affecter un par un les caractères.

Une chaine de caractère doit terminer par le caractère spécial `\0`

Pour afficher une chaine de caractères avec `printf`, on utilise le code `%s`

Pour faire un `scanf` sur une chaine de caractère, on ne met pas le `&` comme pour les autres types

```c
#include <stdio.h>

void inverse(char str[], int len)
{
    for (int i = 0; i < len / 2; i++)
    {
        char tmp = str[i];
        str[i] = str[len - i - 1];
        str[len - i - 1] = tmp;
    }
}

int main(void)
{
    char c1[] = "Bonjour";      // Taille implicite
    char c2[255] = "Au revoir"; // Taille explicite

    printf("c1 = %s\n", c1);

    if (strcmp(c1, c2) == 0)
    {
        printf("Pareil\n");
    }
    else
    {
        printf("Pas pareil\n");
    }

    printf("c1[0] = %c\n", c1[0]);
    printf("c1[6] = %c\n", c1[6]);
    printf("c2[0] = %c\n", c2[0]);

    c1[0] = 'Z';
    printf("c1 = %s\n", c1);
    printf("c1[0] = %c\n", c1[0]);

    inverse(c2, strlen(c2));
```

```
    printf("c2 = %s\n", c2);

    scanf("%s", c2);


    return 0;
}
```

Il existe la librairie `<string.h>` qui donne des fonctions utilitaires très pratiques pour manipuler les chaines de caractères. Voir plus ici

---

Préprocesseur

Les commandes préprocesseur sont des morceaux de code qui sont interpretés avant la compilation. Elles sont toujours precedées d'un `#`

Le contenu du fichier sera alors modifié en consequence.

Instructions disponibles:

- `#include <library.h>` : inclut un fichier (header) dans le code
  - on écrit `<stdio.h>` pour une librairie externe
  - on écrit `"mylib.h"` pour un fichier local (chemin relatif au fichier)
- `#define PI 3.14` : déclare une constante. A chaque utilisation de cette constante, sa valeur sera directement remplacée dans le code par le préprocesseur

```c
#define PI 3.14

int i = PI * 2;

// Apres le preprocesseur, le code deviendra
int i = 3.14 * 2;
```

- `#ifdef` : si une constante est declarée, va avec `#ifndef`, `#endif` alors le code sera integré au fichier, sinon il sera effacé
  - Cela permet de ne pas déclarer plusieurs fois la meme chose. Si on inclus un header à plusieurs endroits, sans les `#ifdef`, les déclarations qui se trouvent à l'interieur se feront plusieurs fois et la compilation échouera

```c
/* On utilise des directives pour compiler
différemment des codes sous Windows et sous Linux */
#ifdef _WIN32
  printf ("Code pour Windows\n");
#else
  printf ("Code pour Linux\n");
#endif
```

Librairies standard utiles

- `stdlib.h` : codes d'érreurs...
- `stdio.h` : printf, scanf ...
- `math.h` : fonctions mathématiques
- `string.h` : manipulation de chaines de caractères

Organisation du code

Lorsqu'on commence à avoir beaucoup de fonctions, tout mettre dans un seul fichier peut rendre difficile la lecture du code.

On va alors diviser notre code en plusieurs fichiers

Pour que des fonctions soient accessibles entre les fichiers, il faut définir des headers, qui définissent le prototype des fonctions et structures de données.

Ces headers devront être inclus dans chacun des fichier qui utilisera ces fonctions.

```c
// lib.h
#ifndef LIB_H
#define LIB_H

int calculer(int);

#endif
```

```c
// lib.c
#include "lib.h"

int calculer(int a) {
    return a * 2;
}
```

```c
// main.c
#include <stdio.h>
#include <stdlib.h>
#include "lib.h"

int main(void)
{
    int resultat = calculer(2);

    return 0;
}
```

Compilation

Pour compiler un programme qui est séparé en plusieurs fichiers, on indique à gcc tous les fichiers .c à compiler. Exemple:

`gcc main.c lib.c -o main.exe`

ℹ️

Avant d'entamer ce cours, il est nécessaire de bien connaitre les bases du langage C, qui ne seront pas revues ici.

Vous retrouverez les pré-requis ici:

Langage C: Cours

Langage C: Cours avancé

Du C au C++

Le langage C++ est un successeur du Langage C auquel il apporte de nombreuses nouvelles choses qui vont permettre d'aller plus loin dans la complexité des programmes, notamment la programmation objet.

Il n'est pas exactement un remplaçant, car le C, plus léger et bas niveau, reste à privilégier dans certains cas d'usages tels que les systèmes embarqués.

Son compilateur principal reste **gcc**, et sa syntaxe reste compatible avec celle du C, mais de nouvelles façons d'écrire du code viennent apparaitre, qui faciliteront souvent la vie du développeur.

Nous retrouverons notamment sa bibliothèque standard `std` qui nous fournira beaucoup d'outils pour aider à la gestion de la memoire, des tableaux, pointeurs etc...

Enfin son caractère **objet** sera très utile pour construire des architectures logicielles propres, modulaires et maintenables.

La documentation officielle:

https://en.cppreference.com/w/

Compilation

Pour compiler nos programmes en C++ nous utiliserons la commande suivante:

```
g++ -Wall -std=c++17 code.cpp -o executable.exe
```

Décomposons:

- `g++` la version C++ de **gcc**
- `-Wall` afficher les “warnings”, avertissements sur problèmes dans le code
- `-std=c++17` pour utiliser la dernière version de C++
- `code.cpp` notre fichier texte de code que l’on souhaite compiler
- `-o executable.exe` la sortie (output, o) de la compilation (donc l’éxécutable) s’appellera `executable.exe`

Sous Max/Linux, ne pas rajouter `.exe` à l’executable

Pour executer:

```
./executable.exe
```

Versions

Il existe differentes versions de C++, et par défaut **gcc** utilise la version **97**

Les versions les plus recentes incluent les modifications (17 inclut 11, qui inclut 97)

Parfois en utilisant des exemples trouvé sur internet, vous verrez que votre code ne compile plus car ces exemples utiliseront probablement du code de nouvelles versions, pour laquelle la syntaxe a changé.

Pour changer la version de C++ utilisée, rajouter `-std=c++17` pour la version `2017` par exemple (il existe `c++97` `c++11` `c++17`)

Entrées/sorties

En C, pour afficher et demander des informations dans la console, nous utilisions `printf` et `scanf`.

Désormais nous utiliserons les entrées et sorties de la bibliothèque standard.

Il est nécessaire d'inclure `<iostream>`

Sorties

Voici comment fonctionne la sortie avec `std::cout`

```cpp
#include <iostream>

int main()
{
    // Un caractère peut être une lettre.
    std::cout << 'A' << std::endl;
    // printf("A\n");

    // Ou bien un chiffre.
    std::cout << 7 << std::endl;

    // Ou bien une chaine de caractère.
    std::cout << "Bonjour" << std::endl;

    // Ou bien une variable
        int temperature = 25;
    std::cout << temperature << std::endl;

    // On peut aussi les combiner
    std::cout << "Il fait " << temperature << " degrées" << std::end
    // printf("Il fait  %d degrées\n", temperature);

    return 0;
}
```

Pour sauter des lignes, nous utilisons `std::endl` (équivalent de `\n`)

Les doubles crochets `<<` sont des opérations de `stream` et permettent de gerer un flux de données entre des entités.

Et plus besoin de se rappeler les codes des types de variables avec `printf` !

Entrées

Pour les entrées nous utiliserons `std::cin` en changeant le sens des crochets `>>`

```cpp
#include <iostream>

int main()
{
    std::cout << "Entre ton age : " << std::endl;
    int age = 0;
    std::cin >> age;
    std::cout << "Tu as " << age << " ans.\n";

    return 0;
}
```

Pour les chaines de caractères, utiliser `std::getline`

```cpp
#include <iostream>

int main()
{
    std::string phrase;
    std::getline(std::cin, phrase);
    std::cout << phrase;

    return 0;
}
```

Gestions des erreurs des entrées

Avec le code suivant, si on tape une chaine de caractère au lieu d'un entier, on aura un problème pour taper son nom:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::cout << "Entre ton age : ";
    unsigned int age = 0;

    std::cin >> age;
    std::cout << "Tu as " << age << " ans.\n";
    std::cout << "Entre ton nom : ";
    std::string nom = "";
    std::cin >> nom;
    std::cout << "Tu t'appelles " << nom << ".\n";

    return 0;
}
```

La solution réside dans l'exemple suivant, qui va tester si l'entrée s'est bien passée, et remettre a zero `cin` autrement

```cpp
#include <iostream>
#include <limits>
#include <string>

template <typename T>
void input(T &var)
{
    while (!(std::cin >> var))
    {
        std::cout << "Il y a eu une erreur, recommencer:" << std::en
        std::cin.clear();
```

```
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
        }
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n
    }

    int main()
    {
        unsigned int age = 0;

        std::cout << "Entre ton age : ";
        input(age);
        std::cout << "Ton age " << age << std::endl;

        return 0;
    }
```

Types de variables

bool

Enfin nous avons officiellement un type de variable pour les booléens `true` et `false`.

```cpp
bool isReady = true;
```

Modifier const

Si une variable ne doit jamais changer, ont dit que c'est une constante. Pour forcer une variable à être constante, nous rajouterons le mot clé `const` avant *ou* après son type lors de sa definition.

```cpp
int const PI = 3.14;
const int PI = 3.14; // equivalent
```

```
PI = 6.2; // Plus possible
```

Les pointeurs

Les references

En C nous avions les pointeurs, en C++ est introduite la notion de *reference*.

Le concept est similaire, a la difference près qu'une reference s'utilise de la même manière qu'une variable normale, et quelle est liée à une autre variable.

cpp

```cpp
#include <iostream>
#include <string>

void fonctionParPointeur(int *pointeur)
{
    *pointeur = 0;
}
void fonctionParRef(int &reference)
{
    reference = 0;
}

int main()
{
    int variable = 24;
    // reference, sa valeur sera toujours identique à variable
    int & reference_variable = variable;

    fonctionParPointeur(&variable);
    std::cout << variable << std::endl;

    fonctionParRef(variable);
    std::cout << variable << std::endl;
    std::cout << reference_variable << std::endl; // même valeur que
```

```
    return 0;
}
```

On va souvent passer des arguments par reference aux fonctions, car cela evitera de devoir les copier (particulièrement pour les classes).

```cpp
#include <vector>
#include <iostream>


void print_container(const std::vector<int>& c)
{
    for (int i : c)
        std::cout << i << ' ';
    std::cout << '\n';
}

int main()
{
    std::vector<int> c = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    print_container(c);
}
```

On utilisera souvent le mot clé `const` avec les references.

Si il n'est n'est pas nécessaire de modifier un argument d'une fonction, on l'indiquera en tant que *constante*.

```cpp
// s est un pointeur modifiable vers une variable non modififable
const std::string & s;
// Equivalent
std::string const & s;

// s est un pointeur non modifiable vers une variable modifiable
std::string * const s;
```

```cpp
// s est un pointeur non modifiable vers une variable non modifiable
const std::string * const s;
```

```cpp
void fn(std::string& s)
{
        s = "hello"; // possible
}
void fn(const std::string& s)
{
        s = "hello"; // pas possible
}
```

Allocation de la mémoire

En nous C utilisions `malloc` pour allouer de la mémoire, et `free` pour la libérer. Désormais nous utiliseront `new` et `delete`

```cpp
int * entier = new int; // alloue un entier

delete entier;          // libère l'entier

int * tableau = new int[10]; // alloue un tableau de 10 entiers

delete [] tableau;    // Libère un tableau d'entier, ne pas oublier
```

Les conteneurs

Une notion appelée conteneurs fait son apparition, et servira a stocker des ensembles de variables.

En C, notre seule option était d'utiliser des tableaux, qui servaient à stocker des suites de variables du même type. On les écrivait comme cela:

```cpp
int tableau[5] = { 1, 2, 3, 4, 5 };
```

Désormais, nous pourrons utiliser les conteneurs de la librairie standard, dont particulièrement `std::array` , `std::vector` et `std::string`

Array

https://en.cppreference.com/w/cpp/container/array

`std::array` est l'équivalent des tableaux que nous avions en C, c'est à dire **statiques**.

Nous ne pourrons plus modifier sa taille, ajouter ou supprimer des éléments une fois qui aura été créé.

Il est nécessaire d'inclure `<array>`

- Déclarer un tableau statique

```cpp
#include <array>

int main()
{
        // valeurs modifiables
        std::array<int, 5> tableau = { 1, 2, 3, 4, 5 };

        // similaire à
        int tableau[5]

        // valeurs non modifiables
        std::array<int, 5> const tableau_constant = { 1, 2, 3, 4, 5

    return 0;
}
```

- Acceder aux elements

```cpp
// Comme en C
tableau[3] // lit le 4 element
```

- Modifier un element

```cpp
// Comme en C
tableau[3] = 12;
```

- Remplir le tableau

```cpp
tableau.fill(10);
```

- Connaitre la taille

```cpp
std::size(tableau)
```

- Copier un tableau

```cpp
std::array<int, 5> copie_du_tableau = tableau;
// On obtient deux tableaux distincts
// si on modifie l'un, l'autre ne sera pas modifié

// Different de
int t1[5];
int t2[5];
t2 = t1;  // ici on a deux pointeurs sur un seul tableau
// si on modifie l'un, l'autre sera modifié aussi
```

- Itérer sur le tableau

```cpp
// For classique
for (int i = 0; i < std::size(tableau); i++)
{
    std::cout << tableau[i] << std::endl;
```

```
}
// For Each
for (int const element : tableau)
{
    std::cout << element << std::endl;
}
```

String

https://en.cppreference.com/w/cpp/string/basic_string

Pour rappel, les chaines de caractères sont en fait des suites de caractères simples. En C, nous utilisions les tableaux de caractères `char []` ou `char *`, en C++ nous utiliserons `std::string`

Il est nécessaire d'inclure `<string>`

```cpp
#include <string>

int main()
{
        std::string phrase = "Voici une phrase normale.";

        phrase = "Voici une autre phrase normale.";

    return 0;
}
```

Leur fonctionnement de base est le meme que pour les Array, avec des possibilités en plus

- **Premier et dernier caractère**

```cpp
std::cout << "Première lettre : " << phrase.front() << std::endl;
std::cout << "Dernière lettre : " << phrase.back() << std::endl;
```

- **Vérifier qu'une chaîne est vide**

```cpp
std::cout << "Est ce vide ? " << std::empty(phrase) << std::endl;
```

- **Ajouter ou supprimer un caractère à la fin**

```cpp
phrase.pop_back();     // supprimer un caractère a la fin
phrase.push_back('.'); // ajouter un caractère a la fin
```

- **Supprimer tous les caractères**

```cpp
phrase.clear();
```

- **Comparer deux chaines**

```cpp
phrase.compare("test");
std::string p2 = "test";
phrase.compare(p2);
```

- **Découper une chaine**

```cpp
std::string phrase = "Hello world";
std::string s1 = phrase.substr(0, 5);  // = "Hello"
std::string s2 = phrase.substr(6, 11); // = "world"
```

Ajouter un `s` apres une chaine de caractère le force à être un `std::string`

Vector

https://en.cppreference.com/w/cpp/container/vector

Enfin, le nouveau type de tableau le plus interessant que nous allons retrouver en C++ est `std::vector`

Ce dernier nous permettra de créer des **tableaux dynamiques**, dont la taille pourra être modifiée. On pourra ajouter et supprimer des éléments à ce

tableau.

Notons ici que les valeurs stockées en mémoire sont **contiguës**.

Il est nécessaire d'inclure `<vector>`

```cpp
#include <string>
// N'oubliez pas cette ligne.
#include <vector>

int main()
{
    std::vector<int> tableau_de_int;
    std::vector<double> tableau_de_double;
    // Même avec des chaînes de caractères c'est possible.
    std::vector<std::string> tableau_de_string;

    return 0;
}
```

Le fonctionnement est encore similaire à Array, avec des choses en plus. Ce qui est faisable avec `array` est aussi faisable avec `vector`

- **Premier et dernier element**

```cpp
std::cout << "Premier : " << tableau_de_int.front() << std::endl;
std::cout << "Dernier : " << tableau_de_int.back() << std::endl;
```

- **Vérifier si un tableau est vide**

```cpp
std::cout << "Est-ce vide ? " << std::empty(tableau_de_int) << std::
```

- **Ajouter un element à la fin**

```cpp
tableau_de_int.push_back(36);
```

- **Supprimer le dernier élément**

```cpp
tableau_de_int.pop_back();
```

- **Assigner des valeurs**

```cpp
tableau_de_int.assign(10, 42); // on affecte 10 fois la valeur 42
```

Exemple d'utilisation

```cpp
std::vector<std::string> tableau_de_string;

tableau_de_string.push_back("Phrase 1"); // Ajout à la fin
tableau_de_string.push_back("Phrase 2"); // Ajout à la fin

// For normal
for(int i=0; i< std::size(tableau_de_string); i++) {
    std::cout << tableau_de_string[i] << std::endl;
}

// For Each
for(std::string s : tableau_de_string) {
    std::cout << s << std::endl;
}

tableau_de_string.pop_back(); // Suppression du dernier
tableau_de_string.clear(); // Suppression de tous les elements
```

Itérateurs

Les types de conteneurs que nous venons de voir sont particuliers car nous pouvons acceder à leurs élements par leur index numérique avec la notation `container[index] = element`

Certains types de conteneurs n'auront pas d'index composé d'entiers numériques successifs. Pour parcourir les éléments de ces conteneurs, il a

donc été inventé ce qu'on appelle les itérateurs.

Pour déclarer un itérateur, nous prendrons le type de son conteneur et ajouterons `::iterator`

Par exemple, `std::array<float, 5>::iterator`

Pour des conteneurs constants, utiliser `const_iterator`

Pour créer un itérateur au début du tableau, on utilise `std::begin()`

`std::end()` lui pointera non pas sur le dernier element, mais sur un element virtuel inexistant, **après** le dernier element

```c
std::vector<int> tableau = { -1, 28, 346, 1000 };
std::vector<int>::iterator debut_tableau = std::begin(tableau);

std::vector<float> const tableau_constant = { -1, 28, 346 } ;
std::vector<float>::const_iterator fin_tableau_constant = std::end(t
```

Pour acceder à un element pointé par un itérateur, il faut le **déreferencer**:

```cpp
std::cout << *debut_tableau << std::endl; // -1
```

Pour se déplacer, on incrémentera ou décrementera l'itérateur:

```cpp
std::cout << *debut_tableau << std::endl; // 28
debut_tableau++;
std::cout << *debut_tableau << std::endl; // 346
debut_tableau--;
std::cout << *debut_tableau << std::endl; // 28
debut_tableau += 2;
std::cout << *debut_tableau << std::endl; // 1000
```

Exemple dans une boucle:

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> tableau = { -1, 28, 346, 84 };

    for (std::vector<int>::iterator it = std::begin(tableau); it !=
    {
        std::cout << *it << std::endl;
    }

    return 0;
}
```

Utilisation des itérateurs dans les containers

Certains conteneurs tels que `std::vector` necessite en paramètres des itérateurs pour certaines de leurs fonction

- **Supprimer** un élément dans un tableau avec `erase`

```cpp
std::vector<int> nombres = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// suppression du quatrième element
nombres.erase(std::begin(nombres) + 3);
// suppression des elements entre le premier (inclus) et le dernier
nombres.erase(std::begin(nombres), std::begin(nombres) + 2);
```

Algorithms

```cpp
#include <algorithm>
```

- **compter** avec `count`

```cpp
std::string const phrase = "Exemple de phrase.";
// compter les 'e' dans toute la phrase
int const total_phrase { std::count(std::begin(phrase), std::end(phr
```

- **trouver** avec `find`

```cpp
std::string const phrase { "Exemple illustrant le tutoriel C++ de Ze

// On obtient un itérateur pointant sur le premier espace trouvé.
std::string::iterator iterateur_mot = std::find(std::begin(phrase),
// Si l'on n'avait rien trouvé, on aurait obtenu std::end(phrase) co
```

- **trier** avec `sort`

Fonctionne avec les nombres, caractères... Va modifier le container de départ pour le réordonner

```cpp
std::vector<double> constantes_mathematiques = { 2.71828, 3.1415, 1.
std::sort(std::begin(constantes_mathematiques), std::end(constantes_
```

- **inverser** l'ordre avec `reverse`

Va également modifier le container de départ

```cpp
std::list<int> nombres { 2, 3, 1, 7, -4 };
std::reverse(std::begin(nombres), std::end(nombres));
```

- **Etc** ... Il y en a beaucoup, allez lire les documentations ! Il existe certainement déjà un outil pour réaliser l'opération que vous souhaitez realiser...

Plus de containers

Maintenant que nous avons vu les bases des containeurs et les itérateurs, nous pouvons utiliser des types plus complexes.

https://en.cppreference.com/w/cpp/container

Sets

Les sets sont des conteneurs qui contient un ensemble de valeurs uniques. A voir comme un `vector` mais où l'on ne pourra pas mettre deux fois la même valeur.

```cpp
std::set<int> set = {1, 5, 3};

std::array<int, 7> arr = {1, 2, 2, 3, 3, 3, 4};
std::set<int> s2 = std::set(arr.begin(), arr.end());
// s2 == { 1, 2, 3, 4 };
```

Maps

Les maps sont des listes indexées par des Clés→Valeurs. Les clés sont uniques.

Imaginer un tableau à deux colonnes:

Clé	Valeur
CPU	10
GPU	15
RAM	20

```cpp
std::map<int, int> redirections;
redirections[80] = 8000; // ajouter/modifier un element
redirections[21] = 1021;
f.erase(21); // Supprimer un element
```

```cpp
std::map<std::string, int> computer;
computer["CPU"] = 10;
computer["GPU"] = 15;
computer["RAM"] = 20;
std::cout << computer["GPU"];  // Recuperation de valeur

// Initialisation
std::map<std::string, int> details{{"CPU", 10}, {"GPU", 15}, {"RAM",

// Iterer avec iterateurs
for (std::map<std::string, int>::const_iterator it = m.begin(); it !
    std::cout << it->first << " = " << it->second;
}

// Iterer avec For Each
// Chaque ligne du tableau est une paire
for (std::pair<std::string, int> paire : m)
{
    std::cout << "Clé : " << paire.first << std::endl;
    std::cout << "Valeur : " << paire.second << std::endl << std::en
}
```

Unordered

Ces deux types sont des versions appellées “ordonnées”, leur clés/valeurs sont triés par ordre croissant. Elles possèdent aussi une version “unordered”, qui est plus performante, mais dont l’ordre de lecture sera aléatoire.

```cpp
std::unordered_map<std::string, int> m;
std::unordered_set<int> s;
```

- Utilisez `std::map` lorsque vous avez besoin de maintenir les éléments triés ou lorsque vous avez besoin d'itérer sur les éléments dans un ordre spécifique.
- Utilisez `std::unordered_map` lorsque la performance est critique et que l'ordre des éléments n'a pas d'importance.

Polymorphisme

La surcharge de fonction

La surcharge de fonctions (ou surchage de méthodes) est un concept en C++ qui permet de définir plusieurs fonctions ayant le même nom mais des signatures différentes au sein d'une même portée. La signature d'une fonction inclut son nom, le nombre et le type de ses paramètres. La surcharge permet d'implémenter des fonctions qui accomplissent des tâches similaires mais avec des types ou nombres de paramètres différents.

Par exemple:

```cpp
int addition(int a, int b) {
    return a + b;
}

// Fonction pour ajouter deux nombres à virgule flottante
double addition(double a, double b) {
    return a + b;
}

// Fonction pour ajouter trois entiers
int addition(int a, int b, int c) {
    return a + b + c;
}

std::cout << addition(3, 4) << std::endl; // Appelle addition(int, i
std::cout << addition(3.5, 2.5) << std::endl; // Appelle addition(do
std::cout << addition(1, 2, 3) << std::endl; // Appelle addition(int
```

Au moment de l'appel à la fonction, le programme reconnaitra automatiquement, en fonction des paramètres donnés, la quelle appeler.

Les paramètres des prototypes des differentes fonction surchargées doivent être differentes pour qu'il puisse choisir.

L'exemple suivant ne fonctionne pas:

```cpp
int fonction(int a);
double fonction(int a); // Erreur : le type de retour seul ne peut p
```

La programmation Objet

Avant de plonger dans les concepts de base de la programmation orientée objet (POO) en C++, rappelons-nous que la POO est un paradigme de programmation qui utilise des "objets" pour modéliser des éléments du monde réel. Les objets sont des instances de classes, qui définissent leurs propriétés et comportements.

Classes et Objets

Les classes représente des élément de notre programme qui existeront en plusieurs exemplaires et qui auront chacune des valeurs et leur propre logique.

Les classes sont composées d'attributs (variables membres) ainsi que de méthodes (fonctions membre)

Exemples de classes

1. **Utilisateur**
   - **Attributs** : identifiant, nom, email, mot de passe (haché), rôle (admin, utilisateur, etc.)
   - **Méthodes** : authentifier(), changerMotDePasse(), afficherProfil()
2. **Session**
   - **Attributs** : idSession, utilisateur, heureDebut, heureFin, adresseIP
   - **Méthodes** : démarrer(), terminer(), estActive()
3. **Pare-feu**
   - **Attributs** : règles, état (activé/désactivé)
   - **Méthodes** : ajouterRègle(), supprimerRègle(), vérifierPaquet()

4. **Cryptographie**
   - **Attributs** : algorithme, clé
   - **Méthodes** : chiffrer(données), déchiffrer(données), générerClé()
5. **Réseau**
   - **Attributs** : adressesIP, sous-réseaux
   - **Méthodes** : scannerPorts(), analyserTrafic(), configurerRoute()

Définition d'une Classe

- **Syntaxe de base** :

```cpp
class NomDeClasse {
    // portée
  public:
      // Attributs (variables membres)
      int attribut;

      // Méthodes (fonctions membres)
      void methode();
};
```

- **Exemple** :

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

      void afficherDetails() {
          std::cout << "Marque: " << marque << ", Année: " << annee
      }
};
```

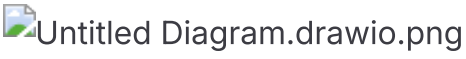
Création d'un Objet

Pour utiliser les classes, nous allons les **instancier** pour creer des **objets**.

Une classe n'est pas utilisable en soit. Nous allons creer des variables du type de la classe, et ces variables seront les objets.

- **Instance d'une classe** :

```cpp
Voiture maVoiture;
maVoiture.marque = "Toyota";
maVoiture.annee = 2022;
maVoiture.afficherDetails();
```



Difference entre structures et classes

- Avec structures:

```cpp
#include <iostream>
#include <string>

struct Personne {
    std::string nom;
    int age;
};

// Méthode pour définir le nom
void setNom(struct Personne& personne,  const std::string& nom) {
    personne.nom = nom;
}

// Méthode pour obtenir le nom
std::string getNom(struct Personne& personne) {
    return personne.nom;
}

// Méthode pour afficher les informations de la personne
void afficherInfos(struct Personne& personne) const {
    std::cout << "Nom: " << nom << ", Age: " << age << std::endl;
```

```
}

int main() {
    Personne personne1;
    setNom(personne1, "Alice");
    personne1.age = 12;
    afficherInfos(personne1);

    Personne personne2;
    setNom(personne1, "Bob");
    personne2.age = 21;
    afficherInfos(personne2);

    return 0;
}
```

- Avec les classes:

cpp

```cpp
#include <iostream>
#include <string>

class Personne
{
public:
    std::string nom;
    int age;

    // Méthode pour définir le nom
    void setNom(const std::string &n)
    {
        this->nom = n;
    }

    // Méthode pour obtenir le nom
    std::string getNom()
    {
        return this->nom;
    }
```

```
    // Méthode pour afficher les informations de la personne
    void afficherInfos()
    {
        std::cout << "Nom: " << this->nom << ", Age: " << this->age
    }
};

int main()
{
    Personne personne1;
    personne1.setNom("Alice");
    personne1.age = 32;
    personne1.afficherInfos();

    Personne personne2;
    personne1.setNom("Bob");
    personne1.age = 12;
    personne1.afficherInfos();

    return 0;
}
```

cpp

```
class Personne
{
public:
    std::string nom;
    int age;

    // Méthode pour définir le nom
    void setNom(const std::string &n);
};

void Personne::setNom(const std::string &n)
{
    this->nom = n;
}
```

Classes et pointeurs

Il est possible d'avoir des pointeurs vers des instances de classes (=objets), et dans ce cas nous accederons à ses membres avec le symbole `->` plutot que le point `.` de la même manière que pour les structures.

```cpp
Classe objet;
Classe *pointeurObjet = &objet;

std::cout << objet.membre << std::endl;
std::cout << pointeurObjet->membre << std::endl;
```

Le mot-clé this

Afin d'acceder au membres d'une classe à l'intérieur même d'une classe, on utilisera le pointeur spécial `this` qui pointe vers l'instance actuelle de l'objet.

```cpp
class Classe {
public:
    int var;

    int get() {
        return this->var;
    }
    int set(int var) {
        this->var = var;
    }
};
```

Constructeurs et Destructeurs

Constructeurs

Définition :

- Un constructeur est une méthode spéciale d'une classe qui est automatiquement appelée lorsqu'un objet de cette classe est créé. Il initialise l'objet nouvellement créé.
- On s'en servira principalement pour initialiser les variables membre de la classe, allouer la mémoire nécessaire et initialiser le contexte

Caractéristiques :

- **Nom** : Le constructeur porte le même nom que la classe.
- **Pas de type de retour** : Contrairement aux autres méthodes, les constructeurs n'ont pas de type de retour, même pas `void`.
- **Automatique** : Il est appelé automatiquement lors de l'instanciation de la classe.

Surcharge :

- Les constructeurs peuvent être surchargés, c'est-à-dire que plusieurs constructeurs peuvent être définis dans une même classe, chacun ayant une signature différente (différents paramètres).

Types de Constructeurs :

- **Constructeur par défaut** : Un constructeur sans paramètres. Si aucun constructeur n'est défini, le compilateur en génère un par défaut.
- **Constructeur paramétré** : Un constructeur qui prend des arguments pour initialiser les attributs de la classe avec des valeurs spécifiques.
- **Constructeur de copie** : Un constructeur qui initialise un objet en le copiant à partir d'un autre objet de la même classe. Sa signature prend un argument qui est une référence constante à un objet de la même classe.

```cpp
class MaClasse {
  public:
      MaClasse() {} // Constructeur par defaut
      MaClasse(int a, float b) {} // Constructeur parametré
      MaClasse(const MaClasse & objet) {} // Constructeur de copie
};
```

Initialisation des Membres :

- Les constructeurs peuvent utiliser une liste d'initialisation des membres pour initialiser les attributs avant que le corps du constructeur ne soit exécuté. Cela est souvent plus efficace et nécessaire pour les membres constants ou les références.

```cpp
class MaClasse {
  public:
      int nombre;

    MaClasse(): nombre(0) {}
};
```

- exemple avec Voiture

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

            // Constructeur par defaut
      Voiture() {
          marque = "";
          annee = 0;
      }
      // Constructeur parametré
      Voiture(std::string m, int a) {
          marque = m;
          annee = a;
      }
      // Equivalent à
      Voiture(std::string m, int a): marque(m), annee(a) {}

      // Constructeur de copie
      Voiture(const Voiture& v): marque(v.marque), annee(v.annee) {}
};

int main() {
```

```
    Voiture v1 = Voiture("Toyota", 2001);
}
```

Destructeurs

Définition :

- Un destructeur est une méthode spéciale d'une classe qui est automatiquement appelée lorsqu'un objet de cette classe est détruit. Il nettoie les ressources allouées par l'objet avant que celui-ci ne soit retiré de la mémoire.

Caractéristiques :

- **Nom** : Le destructeur porte le même nom que la classe, précédé d'un tilde (`~`).
- **Pas de type de retour** : Comme le constructeur, le destructeur n'a pas de type de retour.
- **Automatique** : Il est appelé automatiquement lorsque l'objet sort de son scope ou est explicitement détruit.

Unique :

- Une classe ne peut avoir qu'un seul destructeur. Contrairement aux constructeurs, les destructeurs ne peuvent pas être surchargés.

Libération des Ressources :

- Le rôle principal du destructeur est de libérer les ressources que l'objet a acquises durant sa durée de vie, comme la mémoire dynamique, les descripteurs de fichiers, ou les connexions réseau.

Ordre d'Appel :

- Les destructeurs sont appelés dans l'ordre inverse de la création des objets. Pour les objets membres d'une classe, leurs destructeurs sont appelés après celui de la classe enveloppante.

Destructeur virtuel :

- Si une classe est destinée à être dérivée, il est souvent nécessaire de déclarer son destructeur comme `virtual` pour assurer que le destructeur de la classe dérivée est appelé lorsque l'objet est détruit via un pointeur de la classe de base.
- **Syntaxe** :

```cpp
class NomDeClasse {
  public:
      ~NomDeClasse() {
          // Corps du destructeur
      }
  };
```

- **Exemple** :

```cpp
class Voiture {
  public:
      std::string marque;
      int annee;

      ~Voiture() {
          std::cout << "Destruction de la voiture " << marque << std
      }
};

int main()
{
    Voiture *v1 = new Voiture();
    delete v1; // suppression de l'objet alloué, appel du destructeu

    if (true)
    { // Nouveau bloc
        Voiture v2;
    } // on quitte le bloc, v2 disparait, destructeur appelé

    Voiture v3;
    // Programme terminé, destructeur appelé
}
```

- Exemple avec allocation de mémoire:

```cpp
class AutoFree {
   public:
       int *tableau;

       AutoFree(int size) {
          tableau = new int [size];
       }
       ~AutoFree() {
          delete [] tableau;
       }
 };

 int main() {
     Autofree a = Autofree(100);
 } // la mémoire est automatiquement liberée à la fin du programme
```

Modificateurs de méthodes

Méthodes statiques

Les méthodes statiques sont des fonction membres d'une classe qui ne sont pas liées à une instance de celle ci. Elles permettent d'effectuer des opérations qui ne dépendent pas de l'état d'un objet

```cpp
class MaClasse {
   public:
       static int staticMethod() {
           return 0;
       }
 };

 int main() {
```

```
    int a = MaClasse::staticMethod();
}
```

Méthodes constantes

On peut rajouter le modifier const apres le prototype d'une methode pour indiquer que celle ci ne modifiera pas les variables membres de la classe.

```cpp
class MaClasse {
   public:
        int a;
      int getter() const {
        return a;
      }
      int setter(int a) {
        this->a = a;
      }
};
```

Encapsulation, héritage et polymorphisme

Encapsulation

L'encapsulation consiste à protèger certaines variables ou fonctions membres d'un classe et restreindre leur usage en dehors d'elle même.

Il existe differentes categories d'accès, qui autoriseront ou nous l'accès aux membres sur les objets:

- `public` : Les membres seront accessibles de partout
- `private` : Les membres ne seront accessible que depuis la classe même
- `protected` : Les membres seront accessibles que dans la classes et classes heritées

Afin de rendre disponible l'information ou être en mesure de modifier les membres protégées, nous créerons ce que l'on appele des **getters** et **setters**

```cpp
class Voiture {
  private:
      std::string marque;
      int annee;

  public:
      Voiture(std::string &m, int a): marque(m), annee(a) {}

      int getAnnee() { // getter
          return annee;
      }

      void setAnnee(int a) { // setter
           if(annee > 0) {
              annee = a;
          } else {
              std::cout << "Erreur annee < 0" << std::endl;
            }
      }

      int getMarquee() { // getter
          return annee;
      }

      void setMarque(std::string const &m) { // setter
          marque = m;
      }

};

int main() {
    Voiture v1 = Voiture("Toyota", 2001);
    v1.setAnnee(2010);
    std::cout << v1.getAnnee() << std::endl;
}
```

Heritage

L'héritage est un principe fondamental de la programmation orientée objet (POO) qui permet de créer de nouvelles classes à partir de classes existantes. Cette capacité de dériver une classe de base pour créer une classe dérivée permet de réutiliser du code, de simplifier la maintenance et de promouvoir la modularité.

- **Classe de Base (ou Superclasse)** : La classe dont les propriétés et méthodes sont héritées.
- **Classe Dérivée (ou Sous-classe)** : La classe qui hérite des propriétés et méthodes de la classe de base.

```cpp
class ClasseDeBase {
    public:
    // Membres de la classe de base
};

class ClasseDerivee : public ClasseDeBase {
    public:
    // Membres supplémentaires de la classe dérivée
};
```

Exemple

```cpp
// Classe de base
class Vehicule {
public:
    std::string marque;
    int annee;

    Vehicule(std::string const &m, int a) : marque(m), annee(a) {}

    void afficherDetails() {
        std::cout << "Marque: " << marque << ", Année: " << annee <<
    }
};
```

```
// Classe derivée
class Voiture : public Vehicule {
public:
        // rajout de nouvelles variables de classe
    int nombreDePortes;

    // on n'oublie pas le constructeur de base
        Voiture(std::string m, int a, int portes) : Vehicule(m, a),
        }

    void afficherDetails() {
        Vehicule::afficherDetails(); // Appel de la methode heritée
        std::cout << "Nombre de portes: " << nombreDePortes << std::
    }
};
```

Exemple d'héritage de classes

1. **Classe de Base : Compte**
   - **Attributs** : identifiant, nom, email
   - **Méthodes** : afficherProfil(), modifierEmail()
     - **Classe Dérivée : CompteAdministrateur**
       - **Attributs supplémentaires** : permissions
       - **Méthodes supplémentaires** : gérerUtilisateurs(), afficherLogs()
     - **Classe Dérivée : CompteUtilisateur**
       - **Attributs supplémentaires** : historiqueConnexions
       - **Méthodes supplémentaires** : afficherHistoriqueConnexions()
2. **Classe de Base : SystèmeDeFichier**
   - **Attributs** : cheminRacine, espaceLibre
   - **Méthodes** : créerFichier(), supprimerFichier()
     - **Classe Dérivée : SystèmeDeFichierSécurisé**
       - **Attributs supplémentaires** : niveauChiffrement

- **Méthodes supplémentaires** : chiffrerFichier(), déchiffrerFichier()

3. **Classe de Base : Transaction**

- **Attributs** : montant, date, idTransaction
- **Méthodes** : validerTransaction(), annulerTransaction()

Classe Dérivée : TransactionBancaire

- **Attributs supplémentaires** : numéroCompteBancaire
- **Méthodes supplémentaires** : vérifierSolde(), appliquerFrais()

Classe Dérivée : TransactionCryptographique

- **Attributs supplémentaires** : adresseWallet
- **Méthodes supplémentaires** : vérifierSignature(), confirmerTransaction()

4. **Classe de Base : BaseDeDonnees**

- **Attributs** : urlConnection, connection
- **Méthodes** : connecter(), deconnecter(), insert(), delete()
  - **Classe Dérivée : BaseDeDonneesSQL**
    - **Attributs supplémentaires** : tables
    - **Méthodes supplémentaires** : select(), outerJoin(), innerJoin()
  - **Classe Dérivée : BaseDeDonneesNoSQL**
    - **Attributs supplémentaires** : collections
    - **Méthodes supplémentaires** : filter()

5. **Classe de Base : ChiffrementSymetrique**

- **Attributs** : message
- **Méthodes** : chiffrer(), dechiffrer()

Classe Dérivée : ChiffrementSymetriqueAES

- **Attributs supplémentaires** : keyLength
- **Méthodes supplémentaires** : setKeyLength()

Classe Dérivée : ChiffrementSymetriqueBlowFish

- **Attributs supplémentaires** : pary, sbox
- **Méthodes supplémentaires** :

Polymorphisme

Le polymorphisme est l'un des piliers fondamentaux de la programmation orientée objet (POO). En C++, il permet aux objets de différentes classes dérivées d'être traités comme des objets de la classe de base, facilitant ainsi l'écriture de code plus flexible et extensible. Il existe principalement deux types de polymorphisme en C++ : le polymorphisme statique (ou de compilation) et le polymorphisme dynamique (ou d'exécution).

Statique

Le polymorphisme statique est celui que nous avons vu plus haut pour les surcharges de fonctions. Il est possible de declarer plusieurs méthodes dans une classe qui ont le même nom mais différents arguments.

Le polymorphisme statique est résolu au moment de la compilation. Le choix est fait en fonction des arguments utilisés.

```cpp
#include <iostream>

class Calculateur {
public:
    int ajouter(int a, int b) {
        return a + b;
    }

    double ajouter(double a, double b) {
        return a + b;
    }
};

int main() {
    Calculateur calc;
    std::cout << "Addition de deux entiers : " << calc.ajouter(3, 4)
    std::cout << "Addition de deux doubles : " << calc.ajouter(3.5,
```

```
    return 0;
}
```

Dynamique

Le polymorphisme dynamique est résolu au moment de l'exécution et est généralement implémenté à l'aide de pointeurs ou de références à des classes de base. Il repose sur l'utilisation de fonctions virtuelles.

Une fonction virtuelle est une fonction membre qui peut être redéfinie dans une classe dérivée. Pour qu'une fonction soit virtuelle, il faut rajouter le mot clé `virtual` lors de sa definition.

Lorsqu'une fonction virtuelle est appelée sur un objet via un pointeur ou une référence à la classe de base, la version de la fonction qui est exécutée est déterminée par le type de l'objet réel, et non par le type du pointeur ou de la référence.

Attention, si on place un objet de classe herité dans une variable de type de la classe mère (sans pointeur), les methodes de la classe mère seront appelés

```cpp
#include <iostream>

class Animal
{
public:
    virtual void parler() const
    {
        std::cout << "L'animal fait un bruit." << std::endl;
    }
};

class Chien : public Animal
{
public:
    void parler() const
    {
        std::cout << "Le chien aboie." << std::endl;
```

```
    }
};

class Chat : public Animal
{
public:
    void parler() const
    {
        std::cout << "Le chat miaule." << std::endl;
    }
};

int main()
{
    Animal chienA = Chien();
    Animal *chienB = new Chien();
    Animal *chat = new Chat();

    chienA.parler();    // Animal
    chienB->parler();   // chien
    chat->parler();     // chat

    return 0;
}
```

Classes abstraites

Une classe abstraite est une classe qui ne peut pas être instanciée et qui est destinée à être une classe de base pour d'autres classes. Elle contient au moins une méthode virtuelle pure. On ne pourra pas l'instancier

Méthode Virtuelle Pure : Une méthode virtuelle pure est déclarée en assignant `0` à la déclaration de la méthode virtuelle dans la classe de base.

cpp

```
#include <iostream>

class Forme {
public:
```

```
    virtual void dessiner() const = 0; // Méthode virtuelle pure
};

class Cercle : public Forme {
public:
    void dessiner() const {
        std::cout << "Dessiner un cercle." << std::endl;
    }
};

class Rectangle : public Forme {
public:
    void dessiner() const {
        std::cout << "Dessiner un rectangle." << std::endl;
    }
};

void afficherForme(const Forme &f) {
    f.dessiner();
}

int main() {
    Cercle cercle;
    Rectangle rectangle;

    afficherForme(cercle);     // Affiche : Dessiner un cercle.
    afficherForme(rectangle);  // Affiche : Dessiner un rectangle.

        Forme f; // Erreur de compilation

    return 0;
}
```

Separation du code

Afin d'éviter d'avoir des fichiers de code trop gros, il est habituel de séparer le code en plusieurs fichiers. Les classes seront en general écrite chacune dans un fichier à part.

Pour pouvoir utiliser des classes, fonctions ou autre déclarées dans d'autres fichiers, il faudra créer des fichiers de header et les inclure là où on les utilise

Contenus des fichiers

cpp

```cpp
// Voiture.hpp

#ifndef _VOITURE_H
#define _VOITURE_H

#include <string>

class Voiture
{
private:
    std::string marque;

public:
    Voiture(const Voiture &v);
    Voiture(std::string m, int a);
    ~Voiture();

    std::string getMarque();
    void setMarque(const std::string &m);
};

#endif
```

cpp

```cpp
// Voiture.cpp

#include <iostream>
#include <string>
#include "Voiture.hpp"

Voiture::Voiture(const Voiture &v) : marque(v.marque), annee(v.annee
{
}
Voiture::Voiture(std::string m, int a)
```

```
{
    marque = m;
    annee = a;
}

Voiture::~Voiture()
{
    std::cout << "Destruction de la voiture " << marque << std::endl
}

std::string Voiture::getMarque()
{
    return marque;
}

void Voiture::setMarque(const std::string &m)
{
    marque = m;
}
```

```c
// main.cpp
#include "Voiture.hpp"

int main()
{
    Voiture v1 = Voiture("Toyota", 2001);
    v2->setMarque("Renault");
}
```

Compilation

Pour compiler un programme qui contient plusieurs fichiers, il suffit d'ajouter tous les fichiers .cpp au compilateur:

`g++ -o executable main.cpp class.cpp`

Makefile

Cet outil permet de faciliter la compilation.

Voici un exemple pour le C++:

```makefile
OUT = main

SRC = *.cpp
CFLAGS = -O -Wall -std=c++17
CC = g++
OBJ = $(SRC:.cpp = .o)

$(OUT): $(OBJ)
    $(CC) $(CFLAGS) -o $(OUT) $(OBJ)

clean:
    rm -f $(OUT) *.o
```

Cette configuration compilera tous les fichiers .cpp dans le dossier courant et créera un executable qui s'appellera `main` Pour compiler, lancer la commande `make`

Exemple de projet tout prêt

makefiles.zip

Compiler et executer avec la commande `make run`

Les flux

Concatenation

Lorsqu'on a besoin de remplir des chaines de caractères avec des valeurs de variables, on peut utiliser `std::streamstring`

```cpp
#include <sstream>

int a = 5;
float b = 12.5;
std::string s = "coucou";

std::stringstream ss;
ss << "a: " << a << " b: " << b << " s: " << s;

std::string final = ss.str(); // -> "a: 5 b: 12.5 s: coucou"
```

Exemple de serialisation/deserialisation

```cpp
    std::stringstream ss;
    std::string a = "bonjour";
    std::string b = "aurevoir";
    ss << a << ':' << b << std::endl;

    std::string concat = ss.str();

    size_t pos = concat.find(':');
    std::string aa = concat.substr(0, pos);   // == bonjour
    std::string bb = concat.substr(pos + 1);  // == aurevoir
```

Fichiers

Pour lire et écrire dans des fichiers, nous utiliserons `std::ifstream` et `std::ofstream`

Lecture

```cpp
#include <iostream>
#include <fstream>
```

```cpp
#include <string>

int main() {
    std::ifstream file("example.txt"); // ouverture de fichier en le
    if (!file.is_open()) {             // verification qu'il s'est b
        std::cerr << "Erreur lors de l'ouverture du fichier." << std
        return 1;
    }

    std::string line;
    while (std::getline(file, line)) {  // lecture ligne par ligne
        std::cout << line << std::endl;
    }

    file.seekg(0, std::ios::beg);       // deplacement du curseur au

    char c;
    while (file.get(c)) {  // lecture lettre par lettre
        std::cout << c;
    }

    file.close();                       // Fermeture du fichier
    return 0;
}
```

Ecriture

```cpp
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("output.txt");  // ouverture de fichier en ec
    if (!file.is_open()) {
        std::cerr << "Erreur lors de l'ouverture du fichier." << std
        return 1;
    }

    file << "Ceci est une ligne de texte." << std::endl; // ecriture
    file << "Ceci est une autre ligne de texte." << std::endl;
```

```
    file.close();
    return 0;
}
```

Attention, par défaut ici on va écraser le contenu du fichier. Tout le contenu precedent sera perdu.

Si on veut rajouter du contenu à la fin, on écrira:

```cpp
std::ofstream file("output.txt", std::ios::app); // append
```

Divers

assert

Si l'on souhaite garantir une condition pendant l'execution de notre programme, on peut utiliser assert. Si la condition est fausse, le programme va s'arreter.

```cpp
#include <cassert>

int a = 5;
int b = 6;
assert(a < b);
```

auto

`auto` n'est pas vraiment un type de variable, mais un mot clé pour que le compilateur décide automatiquement du type de la variable, si on l'initialise au moment de sa déclaration et que son type n'est **pas ambigu**

Nécessaire d'utiliser > C++11 : `g++ -std=c++11`

```cpp
// ok
int varInt = 5;
```

```
auto varAuto = varInt;

// pas ok
auto varAuto2;
if(a < b) {
    varAuto2 = 5.5;
} else {
    varAuto2 = true;
}
```

Enum

Surcharge d’opérateurs