Rappels sur le langage C et Python

Etienne Ménard

13 janvier 2019

Table des matières

Première partie

C

1 Ouvrir un fichier source, le compiler et l'exécuter

Ici, on va utiliser le logiciel `Geany` pour ouvrir nos fichiers, les compiler et les exécuter. Pour cela, il faut ouvrir `Geany`, puis ouvrir le fichier donné `TP1aComplete.c`.

Lorsqu'on a fini notre programme, on doit le compiler (il le sauvegardera en même temps) en cliquant sur la brique (ou presser `F9`. Pour lancer l'exécutable compilé, il faut cliquer sur les engrenages (*cf* figure 1, ou presser `F5`).

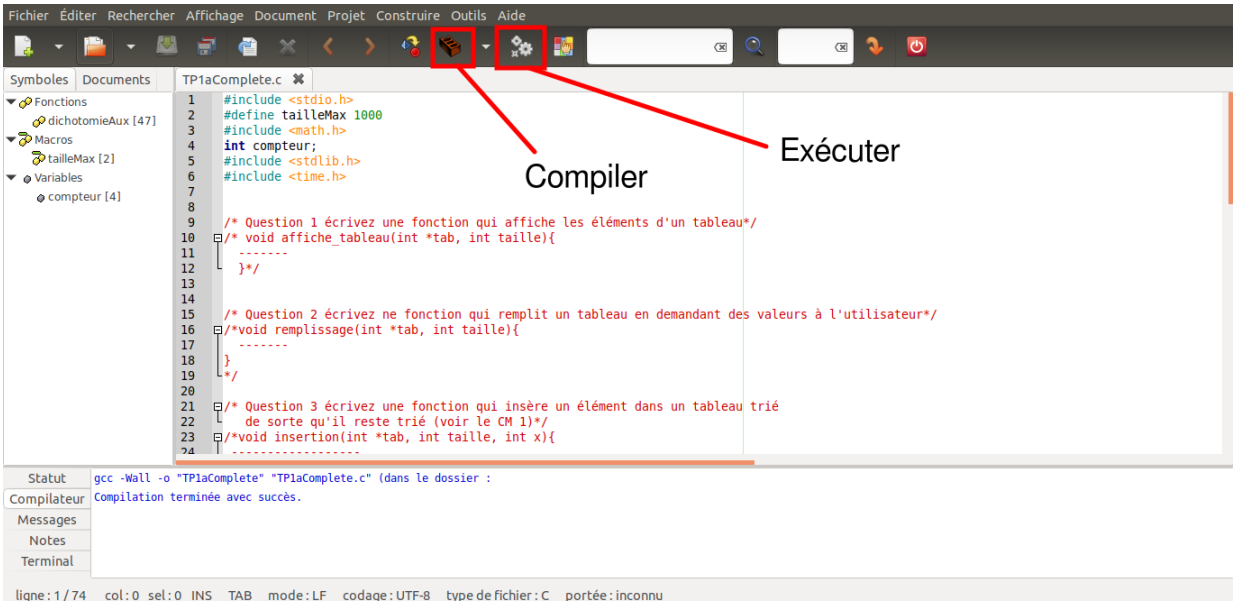

FIGURE 1 – Fenêtre de Geany

1.1 Compiler directement dans le terminal

Si l'on veut on peut aussi faire la compilation en direct dans le terminal. Pour cela on ouvre un terminal et on se place dans le répertoire du fichier source (si nécessaire en se déplaçant avec `cd`) et on tape la commande suivante :

```
gcc fichierSource.c -o NomDeLExécutableVoulu
```

Dans la plupart des cas l'exécutable aura le même nom que le fichier source. Pour exécuter le fichier il faut taper `nomDeLExecutable` ou `./nomDeLExecutable`.

1.2 Corriger le bug « Bash non trouvé »

Il est possible que lors du premier lancement d'un exécutable, Geany ouvre un terminal avec un message d'erreur ressemblant à « bash non trouvé » dans ce cas il faut aller dans `Éditer->Préférences`, onglet `Outils` et regarder la ligne `Terminal` et si il y a des guillemets, les enlever et tenter de relancer l'exécutable, *cf* figure 2.

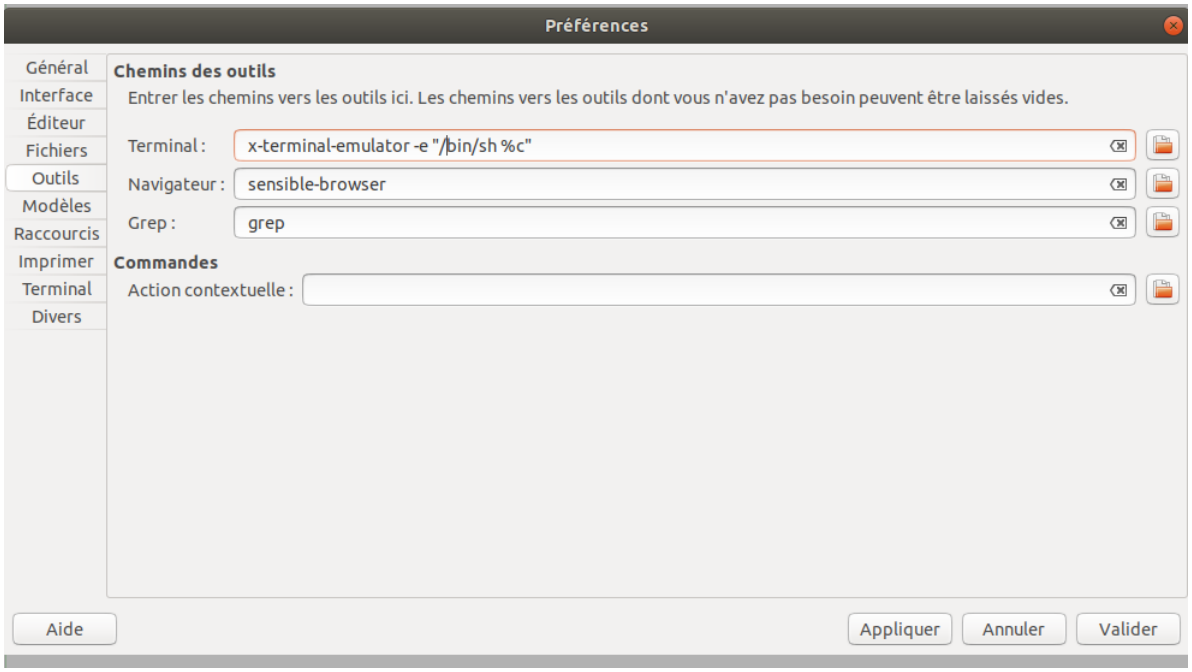

FIGURE 2 – Fenêtre de Geany

1.3 Corriger le bug « Référence indéfinie vers ... »

Il semble que Geany ait du mal à compiler lorsque l'on utilise une fonction de la bibliothèque `math.h`. Pour régler le souci, il faut aller dans `Construire -> Définir les commandes de construction` et dans le champ `Build`, rajouter `-lm` à la fin de la ligne pour obtenir l'instruction de la figure 3.

2 Les commentaires

Commenter son code se révèle très vite nécessaire, pour éviter de passer des heures à se remettre dedans quand on le reprend. C prévoit deux façons d'indiquer des commentaires. Le double slash `//` (`maj+:`) permet de dire que jusqu'au prochain retour à la ligne, tout ce qui suit est un commentaire. Si on veut faire un commentaire de plusieurs lignes ou au milieu de son code [1] on utilisera une balise ouvrante `/*` et une balise fermante `*/`.

3 Structure d'un fichier source C

Les fichiers sources de C comportent une extension en `.c`
Le fichier se compose de la façon indiquée à la figure 4

1. Faites pas ça, c'est bizarre comme idée.

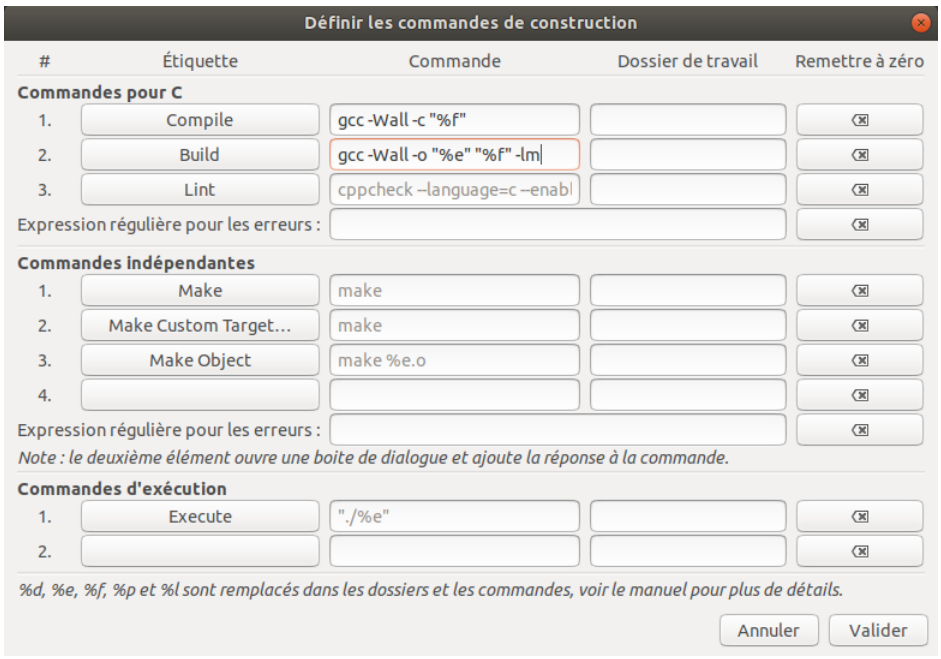

FIGURE 3 – Fenêtre de Geany

```c
//Appel des bibliothèques externes du type de la ligne suivante :
#include <stdio.h>

//Définition de variables de préprocesseur : dans tout le code qui suit,
//ce qui est écrit tailleMax sera remplacé par 1000
#define tailleMax 1000
//Définition de variables globales
int compteur;

//Définition des fonctions
void coucou(){
        printf("bonjour tout le monde");
}

int main(){
        //Définition de constantes, appels de fonctions etc
        void coucou()

        coucou()
        printf("bonsoir");
}
```

FIGURE 4 – Exemple de code C

C'est à dire qu'on définit tout d'abord les bibliothèques qu'on va utiliser (notamment celles pour afficher du texte, en saisir, celles pour réaliser des calculs mathématiques et celles pour gérer le temps) puis des variables de préprocesseurs (éventuellement), des variables globales (éventuellement), des fonctions, et enfin la fonction `main` qui sera celle qui sera exécutée par le programme.

3.1 Structure de la fonction `main`

La fonction `main` sera composée de la façon indiquée à la figure 5.
On appelle prototype de fonction, la première ligne de la fonction. Elle comporte :
— Le type de la sortie,
— Le nom de la fonction,

```c
int main(){
        // Le prototype des fonctions qu'on utilisera
        void coucou();
        int carre(int );
        // Définition de variables
        int nombre;
        int nombre2=2;
        /* Appel des fonctions, à partir d'ici le programme commence
        vraiment à faire quelque chose */
        nombre=carre(nombre2)/* ici on calcule le carré de 2 et on envoie
        donc 4 à la variable nombre */

}
```

FIGURE 5 – Exemple de main

Ici, on va utiliser le logiciel Geany pour ouvrir nos fichiers, les compiler et les exécuter. Pour cela, il faut ouvrir Geany, puis ouvrir le fichier donné TP1aComplete.c.

— Le type des paramètres (mais pas nécessairement le nom des arguments, le compilateur les ignorera quoi qu'il arrive).

Indiquer le prototype de la fonction dans le main permet au compilateur de savoir ce que fera la fonction et quel type auront ses arguments. Si la fonction est écrite avant le `main` dans le fichier, c'est facultatif, mais si elle est écrite après et que son prototype n'est pas dans le `main`, la compilation plantera, la fonction étant inconnue au moment de son usage. Par sécurité, la bonne pratique est donc de recopier le prototype dans le `main` dès que l'on définit une fonction [2]. Ici on a la fonction `coucou` précédente, on indique que sa sortie est de type `void` (pas de sortie), que son nom est `coucou` et le type de ses arguments (ici aucun). On suppose aussi qu'on a créé une fonction `carre` qui prend un entier et renvoie le carré de cet entier. On a donc `int` le type entier de la sortie, `carre`, le nom de la fonction et `int` le type de l'argument (le nombre qu'on va mettre au carré).

Ensuite on déclare des variables qu'on va utiliser, soit on les déclare juste (`int nombre` : on déclare une variable de type entier, qui s'appelle `nombre`) soit on les déclare et on leur affecte une valeur (`int nombre2=2`, pareil qu'avant sauf que nombre2 a maintenant une valeur). Il est souvent plus prudent d'affecter les variables dès leur création pour éviter des erreurs. Si une variable n'est pas affectée mais qu'on veut la lire, on risque de lire la valeur stockée précédemment dans la case mémoire, ce qui risque fortement de faire planter le programme.

Enfin, on écrit l'enchaînement des fonctions qu'on veut, on peut définir d'autres variables etc...

3.2 La structure d'une fonction

La définition d'une fonction commence par le type de sa sortie, `void` si elle ne contient pas de sortie (par exemple une fonction qui afficherait le contenu d'un tableau), le type correspondant sinon. Puis on a le nom de la fonction, et entre parenthèses, le nom des arguments, précédés de leur type. Ensuite on place les prototypes des fonctions qui seront appelées par celle-ci (sauf elle-même, si jamais on a du récursif), on déclare les variables et on met la liste des instructions. Enfin, si nécessaire, on finit par des `return` pour renvoyer une valeur. Chaque instruction doit se finir par un point-virgule (source d'énormément d'erreur de compilation).

4 Les boucles en C

4.1 La boucle `while`

La boucle `while` va se répéter tant que la condition indiquée reste vraie. Si on l'a mal rédigée, elle risque de donner naissance à une boucle infinie, il faut donc être rigoureux. Sa syntaxe est indiquée à la figure 6

4.2 La boucle `do ... while`

Elle fait la même chose que la boucle `while` si ce n'est que la condition n'est évaluée qu'en fin de boucle. Cela veut dire qu'elle s'exécutera toujours au moins une fois. Sa syntaxe est indiquée à la figure 7

2. Dans notre cadre, dans le cas de projets plus ambitieux, on écrira la fonction dans un autre fichier c, son prototype dans un fichier h et on incluera tout ça au fichier `main.c`. Mais ici c'est au delà du cadre de ce cours.

```
while(/*condition*/){
        //liste d'instructions
}
```

FIGURE 6 – Exemple de boucle `while`

```
do{//ici pas de condition
        //liste d'instructions
}while (condition) ; //Attention à ne pas oublier le point-virgule
```

FIGURE 7 – Exemple de boucle `do ... while`

4.3 La boucle `for`

La boucle `for` n'est en fait qu'une façon condensée d'écrire la boucle `while` mais là où, avec `while`, on doit incrémenter le curseur à un moment de la boucle, ici on le précise dès le début, c'est plus clair. Sa syntaxe est indiquée à la figure 8. On définit d'abord le compteur à sa valeur initiale, on donne la condition d'arrêt (qui dépendra souvent du compteur) et le pas de l'algorithme (ici `compteur++` est un raccourci pour `compteur=compteur+1`). Ici le pas est de 1 mais on pourrait tout à fait en spécifier un autre. Attention initialisation, fin et incrémentation sont séparés par des points-virgules.

```
for (compteur=0 ; compteur <25 ; compteur++){
//liste d'instructions
}
```

FIGURE 8 – Exemple de boucle `for`

5 Les types

C a un certain nombre de types définis par défaut. Les plus utiles sont les suivants : `int` pour représenter des entiers (jusqu'à 32 000 environ), `double` pour représenter des réels.

5.1 Les tableaux

Un tableau est une succession de variables du même type avec des adresses qui se suivent. Un tableau est toujours de taille fixée et on ne peut pas la modifier [3].

Pour définir un tableau on doit indiquer son type, son nom et sa taille, par exemple `int tableau[5];` pour un tableau d'entiers comportant 5 cases. On rappelle que la première case du tableau est l'indice 0 et la dernière est `taille-1`.

Pour accéder à la i^{ème} case du tableau, on tape `tableau[i]`. En fait, quand on crée un tableau, ce que renvoie le système d'exploitation c'est un pointeur vers la première case. Et en fait écrire `tableau[i]` revient à dire « je veux accéder à la case `tableau+i` ». Et c'est pour ça que la première est 0.

Comme dans le cas où on confond `*` et `&` entre pointeur et variable, si on veut accéder à la case `tableau[taille]` c'est à dire la première case mémoire après la fin du tableau, le programme va être coupé par le système d'exploitation parce qu'on essaie d'accéder à une case qui ne nous appartient pas.

Comme un tableau est défini par un pointeur, lorsqu'on utilisera un tableau dans une fonction, le type qu'on indiquera sera `int *` (pour un tableau d'entier évidemment). Par exemple :

```
void lectureTableau(int *tableau)
```

5.2 Les structures

On peut définir un nouveau type à partir de types déjà existants : on appellera cela une *structure*. On peut le voir comme un tableau un peu particulier : on a des cases mémoires regroupées ensemble auxquelles on peut

3. Sauf par allocation dynamique mais on le verra plus tard.

accéder directement sauf qu'ici elles peuvent être de différents types là où toutes les cases d'un tableau devaient avoir le même type. Pour définir une structure on utilisera le mot-clé `struct` et on précisera entre accolades l'ensemble des types des cases de cette structure et on leur donnera un nom. Enfin on n'oubliera pas de mettre un point-virgule à la fin de l'accolade. Par exemple, à la figure 9, on définit un `Noeud` d'une liste chaînée. A chaque fois qu'on aura besoin d'indiquer le type d'une structure qu'on a créé, on n'oubliera pas [4] le mot-clé `struct`, par exemple si on veut créer un nœud appelé `nd` : `struct Noeud nd;`.

```
struct Noeud{
int valeur;
struct Noeud * suivant;
};
```

FIGURE 9 – Exemple de structure

Pour accéder à l'un des champs d'une structure, on tape le nom de la structure puis un point et enfin le nom du champ. Par exemple si on veut affecter des valeurs aux différents champs d'une structure qu'on vient de définir, on pourra taper le code de la figure 10.

```
struct Personne{
int age;
int departement;
int taille;
};
struct Personne Jean;
Jean.age=40;
Jean.departement=14
Jean.taille=175
```

FIGURE 10 – Accès à un champ d'une structure

6 Les pointeurs

C (contrairement à Python) est fait pour manipuler des pointeurs. On rappelle que le pointeur d'une variable est en fait l'adresse mémoire à laquelle est stockée cette variable.

6.1 Les opérateurs `&` et `*`

Si on a défini une variable, par exemple `age`, on peut obtenir son adresse avec le caractère esperluette : `&age`.

Pour créer un pointeur sur une variable, on va utiliser le caractère étoile `*` de la façon suivante. Par exemple si on veut créer un pointeur `p` vers une variable de type `int`, on notera `int *p`. Pour donner une valeur par défaut à un pointeur, on n'utilisera pas le 0 ou le 1 comme pour une variable de type `int` mais le mot clé `NULL`.

Si on veut que notre pointeur contienne l'adresse mémoire d'une autre variable, nous allons pouvoir utiliser ce qu'on a vu précédemment pour obtenir le code de la figure 11.

```
int compteur=10;
int *pointeurSurCompteur;
pointeurSurCompteur = &compteur;
```

FIGURE 11 – Récupération de l'adresse mémoire d'une autre variable

Ici la première ligne crée la variable, la deuxième définit un pointeur `int *pointeurSurCompteur` c'est à dire « définit un pointeur sur une variable de type `int` ». Ensuite, à la troisième ligne, on affecte à ce pointeur la valeur `&compteur` qui est l'adresse de la variable `compteur`.

En fait on peut résumer en disant que :

— `&variable` donne l'adresse de `variable`,

— `*pointeur` donne la valeur de la variable dont l'adresse est stockée par `pointeur`.

4. On pourrait s'en passer en utilisant une instruction `typedef` mais c'est hors du périmètre de ce TP.

Ou, encore résumé dans la table 1. Ici la variable de la ligne 89 456 est un pointeur (on va l'appeler `p`) sur la variable de la ligne 47 856 (on va l'appeler `somme`). Donc, finalement, on a `p`=47 856, `somme`=175, `&somme`=47 856, `&p`=89 456 et `*p`=175. Par contre si on essaie de taper `*somme`, c'est comme si on essayait d'accéder à la case mémoire 175. Et dans ce cas, le système d'exploitation va couper le programme puisque ce dernier essaie d'accéder à une case mémoire auquel il n'a pas le droit (parce qu'il est très probable que la case 175 a été allouée par un autre programme).

Adresse	Valeur stockée	Type
0	'a'	Caractère
1	42	Entier
2	6.02×10^{23}	Double
⋮	⋮	⋮
47 856	175	Entier
⋮	⋮	⋮
89 456	-> 47 856	Pointeur sur entier

TABLE 1 – Exemple d'occupation de la mémoire

Ici, on va utiliser le logiciel Geany pour ouvrir nos fichiers, les compiler et les exécuter. Pour cela, il faut ouvrir Geany, puis ouvrir le fichier donné TP1aComplete.c.

Il ne faut pas confondre les deux significations de `*` : lorsqu'on déclare une variable, ça sert à indiquer un type pointeur, lorsqu'on veut accéder à une variable, on accède à la valeur de la variable pointée par le pointeur.

6.2 L'opérateur `->`

Si on définit une structure contenant plusieurs champs (par exemple la structure `Noeud` des listes chaînées contient les champs `valeur` et `suivant` cf figure 9), on va avoir deux façons d'accéder aux champs à partir d'un pointeur sur la structure.

Fondamentalement, ce qu'on veut c'est accéder à l'un des deux champs d'une variable de type nœud. Si on a une telle variable, on peut accéder à chacun de ses champs par l'opérateur `.` comme dans le code de la figure 10. Sauf qu'ici on a rarement directement une variable du type voulu mais plutôt un pointeur sur cette variable. Dans ce cas on utilise l'opérateur `*`, si `p` est un pointeur sur un nœud, on accède à l'attribut `valeur` de la variable pointée en tapant `(*p).valeur`. En effet, on commence par accéder à la valeur pointée puis on accède à l'attribut `valeur`. Comme c'est long, on a définit un raccourci pour cette expression, la flèche `->` formée du tiret [5] et du chevron fermant [6]. Avec ce raccourci on peut noter `p->valeur` comme en pseudo-code.

7 Les fonctions indispensables en C

7.1 Afficher du texte : `printf`

Pour afficher une chaîne de caractères, on utilise la fonction `printf`. La syntaxe est la suivante :

```
printf("coucou")
```

Une chaîne de caractères étant définie entre guillemets.

Une des grandes utilités de `printf` est aussi d'afficher le contenu des variables. Pour cela il faut utiliser un caractère spécial et lui indiquer par quelle variable le remplacer. Par exemple le code de la figure 12 va afficher `Bonjour, j'ai 19 ans`.

```
int age=19;
printf("Bonjour, j'ai %d ans", age);
```

FIGURE 12 – Exemple d'utilisation de `printf`

Les différents codes que l'on doit utiliser suivant le type de la fonction sont résumés dans la table 2Votre branche est en retard sur 'origin/master' de 1 commit

5. touche 6 en minuscule.
6. MAJ+touche chevrons

Type	Symbole
int	%d
double	%f
pointeur	%p
chaîne de caractères	%s
caractère	%c
entier héxadécimal	%x

TABLE 2 – Table des différents paramètres à transmettre à `printf`
https ://www.google.fr/

On peut aussi faire afficher plusieurs variables, il suffit de les faire se suivre, séparées par une virgule et dans l'ordre où elles apparaissent dans la chaîne de caractères. Par exemple :

```
printf("Bonjour, nous sommes à %s et il fait %d °C\n",ville,temperature);
```

pourra provoquer l'affichage de `Bonjour, nous sommes à Caen et il fait 35 °C`.

Le symbole `\n` insère un retour à la ligne.

7.2 Récupérer une saisie : `scanf`

La fonction `scanf` va arrêter l'exécution du programme et demander à l'utilisateur de saisir une valeur. Une fois cette valeur saisie, elle la convertira en un type et la stockera dans une variable. Mais pour stocker dans cette variable, elle n'aura pas besoin du nom de la variable, mais de son adresse mémoire. Ainsi, le morceau de code de la figure 13 va faire la chose suivante :

Votre branche est en retard sur 'origin/master' de 1 commit

```
int age=19;
printf("j'ai %d ans\n",age);
printf("Quel est ton âge ?");
scanf("%d",&age);
printf("tu as %d ans",age);
```

FIGURE 13 – Exemple d'utilisation de `scanf`
Lorsqu'on a fini notre programme, on doit le compiler (il le sauvegardera en même temps) en cliquant sur la briqu

1. On initialise une variable `age` à 19
2. On affiche `"J'ai 19 ans"` en lisant la valeur de la variable
3. On affiche un texte invitant à la saisie
4. On récupère la saisie en la considérant comme un entier (%d) et on le stocke dans l'emplacement mémoire désigné par l'adresse `&age` (c'est à dire qu'on remplace la valeur de `age`)
5. On affiche le nouveau contenu de la variable `age`, qui a pu être modifié par notre saisie.

7.3 Les fonctions `ceil` et `floor`

La fonction `floor` prend en entrée un nombre (entier, ou décimal) et renvoie le premier entier inférieur (ou égal). `ceil` fait la même chose mais renvoie le premier entier supérieur (ou égal)

7.4 Les fonctions `malloc` et `free` : l'allocation dynamique

Lorsqu'on veut créer un tableau, jusqu'à présent, la seule solution qu'on ait est de définir dans une variable de préprocesseur une taille suffisamment grande pour que le tableau ne soit jamais plein et ensuite ne remplir que les premières cases. Ce n'est pas une gestion très optimisée de la mémoire, mais comme la taille d'un tableau ne peut être définie par une variable en C, c'est la seule solution simple qu'on ait.

Néanmoins il existe une solution technique : dire à l'ordinateur de nous réserver la place mémoire suffisante et récupérer le pointeur sur cette case mémoire. Pour cela on utilisera la fonction `malloc` (*Memory ALLOCation*)

qui prend en entrée un entier [7] et renvoie un pointeur sur l'emplacement réservé. L'entier est la nombre d'octets qu'il faudra réserver. Comme cette quantité peut dépendre du compilateur et du système d'exploitation, le mieux est d'utiliser une formulation « portable » avec l'instruction `sizeof` qui, à un type, associe la place mémoire en octets qu'il prend. Ainsi avec tout ceci, si on veut réserver un tableau d'entiers de `taille` cases, on entrera [8] la commande de la figure 14. On a alors créé un tableau de `taille` cases. Son utilisation sera alors la même que pour un tableau classique.

```
int taille=25;Votre branche est en retard sur 'origin/master' de 1 commit
        Lorsqu'on a fini notre programme, on doit le compiler (il le sauvegardera en même temps) en cli
int *pointeurSurTableau;
pointeurSurTableau=malloc(taille*sizeof(int));
```

FIGURE 14 – Exemple d'allocation dynamique

Une fois qu'on n'a plus besoin de la variable allouée dynamiquement, on peut libérer la place en mémoire à l'aide de l'instruction `free` qui s'utilise tout simplement comme ceci : `free(pointeurSurTableau);`.

7.5 La fonction `rand`

La fonction `rand` ne prend pas de paramètre en entrée et retourne en sortie un nombre pseudo-aléatoire. C'est à dire que chacun des nombres tirés par cette fonction est équiprobable mais par contre, sans plus de précautions que ça, si on fait un programme qui ne fait qu'appeler `rand` et l'afficher, le résultat sera toujours le même. En fait c'est comme si on avait une liste contenant tous les nombres de 0 à une certaine limite dans le désordre et qu'on tirait toujours le premier. Pour que ce soit un peu plus aléatoire, il faut qu'on fixe d'où on commence à tirer le nombre. Pour cela on va lier ça à l'heure actuelle [9]. La fonction qui permet de placer ce point de départ est `srand`. Et l'instant actuel se récupère avec la fonction `time(NULL)` (de la bibliothèque `time.h`). Il n'est nécessaire d'appeler `srand` qu'une fois.

Enfin, comme `rand` renvoie un nombre entre 0 et une borne qui varie d'une implémentation à l'autre, pour avoir une valeur aléatoire entre 0 et notre borne `Max`, on va utiliser le modulo. Comme un nombre modulo `n` est compris entre 0 et `n`-1, finalement on aura un code semblable à celui de la figure 15. On va alors avoir l'affichage d'un nombre dans l'intervalle $[[0, \texttt{borne}]]$.

```
int borne=25;//on définit la borneVotre branche est en retard sur 'origin/master' de 1 commit
srand(time(NULL));//on initialise l'aléatoire
int nombreAuHasard=rand()%(borne+1);//on tire un nombre au hasard
// et on le prend directement entre les bonnes bornes par modulo
printf("%d",nombreAuHasard);// on l'affiche
        Lorsqu'on a fini notre programme, on doit le compiler (il le sauvegardera en même temps) en cli
```

FIGURE 15 – Exemple d'utilisation de `rand`

7.6 Les fonctions `min` et `max`

Elles n'existent pas en C, il faut les écrire soi-même.

7.7 La fonction `log`

La fonction `log` de la bibliothèque `math.h` prend en entrée un `double` et renvoie un `double`. Elle calcule le logarithme népérien du nombre. On rappelle que $\log_2(x) = \frac{\ln(x)}{\ln(2)}$.

8 Les connecteurs logiques

Si on veut faire un test un peu élaboré, on peut avoir besoin de tester si une condition ou un autre est vraie, si deux conditions sont vraies en même temps, si la négation d'une condition est vraie. Là où Python a opté

7. En fait une variable de type `size_t` mais ici la différence n'est pas importante.
8. Théoriquement on est censé vérifier que l'allocation a réussi mais ici ce serait rajouter de la complexité inutile.
9. En fait, le *timestamp*, c'est à dire le nombre de secondes entre le 1er janvier 1970 à minuit et maintenant.

pour des mots du langage naturel (`AND,OR,NOT`), C utilise des symboles : `&&` pour le ET logique, `||` pour le OU logique (`alt gr+6` sur un clavier azerty Windows), `!` pour la négation. On recommande bien sûr d'entourer chaque proposition de parenthèses pour que le compilateur n'ait pas de difficulté à interpréter les expressions.

Deuxième partie
Python

1 Ouvrir un fichier source, l'interpréter avec `Geany`

Ici, on va utiliser le logiciel Geany pour ouvrir nos fichiers, les compiler et les exécuter. Pour cela, il faut ouvrir `Geany`, puis ouvrir le fichier donné `TP2aComplete.py`.

Lorsque l'on a fini notre programme on doit l'interpréter [10]. Pour cela, il faut cliquer sur les engrenages (cf figure 1, ou presser `F5`). Il ne faut pas oublier de l'enregistrer avant, pour prendre en compte les modifications apportées (`CTRL+S`).

On peut aussi utiliser `IDLE`, qui permet d'avoir une session interactive (on peut saisir des commandes) et pas juste exécuter le fichier.

2 Faire utiliser Python 2 ou Python 3

Python a deux versions qui évoluent plus ou moins en simultané même si la version 2 devrait finir par s'éteindre. En attendant, le code qu'on veut exécuter peut devoir nécessiter Python 2 ou Python 3. Sous Linux Python 2 est appelé avec la commande `python` et Python 3 avec `python3`.

Pour cela il faut reprendre la démarche de la section 1.3 mais en ayant ouvert un fichier Python et aux lignes `Compile` et `Execute`, remplacer `python` par `python3` si on a du code écrit en Python 3 et inversement si on a du code écrit en Python 2.

3 Les commentaires

En python pour commenter sur une ligne, comme le faisait `//`, il faut utiliser le croisillon : `#` (`AltGr+3`). On peut aussi faire des commentaires sur plusieurs lignes en utilisant les triples guillemets comme dans la figure 16.

```
def factorielle(n)
"""
   Permet de calculer de façon récursive la factorielle d'un entier
"""
if n==0:
   return(1)#Pour finir la récursivité
return(n*factorielle(n-1))
#Ligne vide après cette fonction
```

FIGURE 16 – Exemple de commentaires

3.1 Les docstrings

De plus, en python si on insère un commentaire avec les trois guillemets juste après la définition d'une fonction (comme ici), l'aide comprendra qu'il s'agit d'une description de la fonction (une *docstring*) qu'on pourra appeler avec `help(nomDeLaFonction)`.

4 Structure d'un fichier python

Dans un fichier python, contrairement au C, on n'aura pas besoin d'une fonction `main` pour dire ce qui sera exécuté. Tout ce qui est dans le fichier et qui n'est pas à l'intérieur d'une fonction sera exécuté. On peut mêler dans le même fichier la déclaration des fonctions et leurs appels. Par cohérence avec le C il est préférable de définir toutes les fonctions avant la partie du code à exécuter.

10. Contrairement au C, le Python n'est pas un langage compilé, mais interprété.

4.1 L'indentation

Par contre, contrairement au C, en python l'indentation ne sert pas qu'à comprendre le code plus facilement, elle est utilisée par l'interpréteur pour se repérer dans l'imbrication des boucles. Les normes de codage demandent qu'on utilise une indentation de 4 espaces mais certains préfèrent utiliser des tabulations, l'important étant de rester cohérent du début à la fin du document. Si l'indentation est incorrecte, lors de l'interprétation on aura le message : `IndentationError: unindent does not match any outer indentation level` ou un message équivalent. On peut la corriger en faisant afficher à Geany les endroits où ça pose souci en cochant les deux options `Affichage->afficher les espaces` et `Affichage->afficher les guides d'indentation`.

4.2 La structure d'une fonction

La déclaration d'une fonction commence toujours par le mot-clé `def` suivi du nom de la fonction et de ses arguments. On finit cette ligne par deux points `:` et on écrit ensuite le reste de la fonction (en commençant par la docstring *cf* section 3.1). Ainsi on pourra avoir un code semblable à celui de la figure 16.

La fonction se finit par une ligne vide.

Contrairement au C, on peut attribuer des valeurs par défaut aux derniers arguments de la fonction en tapant `nom_de_l'argument=valeur_par_défaut` par exemple `factorielle(n=0)` renverra la factorielle de `n` si on lui précise la valeur de `n` et 0! sinon.

5 Les boucles en python

5.1 La boucle `while`

Elle fonctionne comme en C, sa syntaxe est néanmoins légèrement différente *cf* figure 17.

```
while(condition):
        #liste d'instructions
        #fin de la boucle while
#Suite des instructions au même niveau d'indentation que while
```

FIGURE 17 – Exemple de boucle `while`

5.2 La boucle `do ... while`

Elle n'existe pas en Python.

5.3 La boucle `for`

Sur le principe elle fonctionne comme en C mais elle a certaines possibilités qui la rendent très puissante.

```
for (i in range(16)):
        #liste d'instructions
        #fin du for
#suite des instructions au même niveau d'indentation que for
```

FIGURE 18 – Exemple de boucle `for`

En fait au lieu d'avoir un compteur qui va évoluer et s'arrêter selon un certain test comme en C, en python on va demander à ce qu'on ait une variable qui appartienne à une certaine suite. Ici c'est `i` qui appartient à la suite définie par `range` (*cf* section 8.1), souvent utilisé avec `len` (*cf* section 8.2). Mais si on voulait par exemple parcourir les différents éléments de la liste `L`, on pourrait taper `for i in L:`. Sauf que à ce moment là le `i` n'est plus un entier mais directement l'élément de la liste (et donc possiblement une autre liste, un entier, une chaîne de caractères etc.).

```
L=[1,2,'a',[1,2,4]]#Une liste qui contient deux entiers, un caractère et une liste
for (i in L):
        print(L)#Ici i est directement l'élément de la liste

for i in range(len(L)):
        print(L[i])#Ici i est un indice
```

FIGURE 19 – Exemple de boucle `for`

6 Les types

Python se charge lui-même d'attribuer le bon type à chaque variable en fonction de la valeur qu'on lui affecte. Si jamais il y a un souci on peut néanmoins contrôler le type de la variable avec l'instruction `type` qui prend en paramètre l'objet dont on veut connaître le type. On peut aussi forcer la conversion en un certain type avec des fonctions spécifiques. Par exemple `int` pour convertir en entier, `float` pour convertir en réel, `str` pour convertir en chaîne de caractère, `list` pour convertir en liste etc. A chaque fois c'est le nom du type.

6.1 Les listes

Python a une gestion très puissante des listes. Pour définir une liste, il suffit d'entrer entre crochets et séparés par des virgules, les différents éléments de la liste. Par exemple `premier=[2,3,5,7,11]` pour la liste des premiers nombres premiers.

On accède au différents éléments de la liste en précisant leurs indices entre crochets. Par exemple `premier[2]` va valoir 5 (les listes commencent avec un indice 0). On peut aussi donner un indice à partir de la droite avec un nombre négatif `premier[-1]` vaut 11, `premier[-3]`, 5. Attention, dans ce cas là, on commence la numérotation à 1.

6.1.1 Le problème de la copie de liste

Comme dans le TD, au niveau de la mémoire, Python gère les liste avec des pointeurs, c'est à dire que lorsqu'on définit une liste, on récupère un pointeur sur la liste. Si on tape le code de la figure 20 on voit qu'on a copié une liste, modifié sa copie mais que ça a quand même modifié la liste initiale. Pour éviter cela on peut forcer python à recopier toute la liste en utilisant la méthode `copy`. C'est à dire que après le nom de la variable on rajoute un point et `copy()`. Ceci renverra une copie physiquement distincte de la liste originale. Attention, dans le cas de listes imbriquées dans des listes on aura quand même des soucis et il faudra utiliser `deepcopy` de la bibliothèque `copy`.

```
L=[1,2,3,4,5]
M=L
M[2]=23
print(M)#[1,2,23,4,5]
print(L)#[1,2,23,4,5]

L=[1,2,3,4,5]
M=L.copy()
M[2]=23
print(M)#[1,2,23,4,5]
print(L)#[1,2,3,4,5]
```

FIGURE 20 – Exemple de boucle `for`

6.2 « Slicer » une liste

On peut facilement découper une liste en python. Pour avoir les termes entre les indices `i` (inclus) et `j` (exclu) il faut indiquer `L[i:j]`. Si on veut partir du début ou aller jusqu'à la fin, on peut ne pas écrire l'indice en question. Enfin on peut aussi utiliser les indices négatifs. Par exemple `L[:-1]` renverra la liste privée de son dernier terme.

7 Les pointeurs

Python est fait pour cacher le plus possible l'utilisation de pointeurs.

8 Les fonctions indispensables en Python

8.1 Créer une suite : `range`

`range` est une fonction qui va retourner une suite. Sa version la plus simple est de lui donner un seul entier `n` et `range(n)` renverra la suite $(0, \dots, n-1)$. Sa borne de fin est donc exclue de la liste. On peut aussi lui préciser la valeur de départ (inclue) et le pas. Par exemple `range(i,j,2)` renverra la suite $(i, i+2, \dots, j-2, j)$ (en supposant que `j` et `i` sont de même parité). On peut donc lui préciser : la valeur de fin seule (il commencera à 0 et ira de 1 en 1), la valeur de début et la valeur de fin (il ira de 1 en 1) ou la valeur de début, de fin et le pas, dans cet ordre.

Attention cependant `range` ne renvoie pas une liste mais un type `range`. Pour obtenir une liste à partir de `range` il faut faire `list(range(0,78,2))` qui renverra la liste des pairs de 0 à 76 (car 78 est exclu).

8.2 Longueur d'une liste : `len`

`len` prend en entrée une liste et renvoie la longueur de cette liste. Comme les listes sont numérotées à partir de 0 en python, que `range` commence à 0 et finit à $n-1$, on peut très facile passer d'une boucle sur les élément d'une liste (`for i in L`) à une boucle sur les indices d'une liste (`for i in range(len(L))`).

9 Les connecteurs logiques

Python possède les opérateurs de comparaison habituels (`==,>,<,>=,<=,!=`) et pour faire des tests logiques il utilise les mots-clés : `and, or, not`. On peut aussi lui rajouter l'instruction `in`, très utilisée dans les boucles `for` qui permet de tester par exemple si un élément appartient à une liste.