In [1]:

```
from common import utils
import random
u=utils("PDF") # PDF+SOL / PROF / PDF / INTER

# Conseil aux enseignants
# Si ce cours peut être fait en suivant le polycopié, il est conseillé
# de le présenter sous forme de démonstration interactive dans la première partie
# de la séance puis de faire programmer les étudiants en autonomie après.
```

```
Ce cours a été régénéré le 2018-12-06 16:09:14.292447. Mode sans corrigé. Mode statique.
```

Programmation en C

La programmation en langage C se fait de façon assez similaire au langage Python. Nous allons donc donner des éléments de syntaxe, et commenter les différences, plutôt que de donner tout à partir de rien.

Les bibliothèques

Si le langage Python permet d'adjoindre *dynamiquement* des bibliothèques en tant que langage interprété, le langage C utilise plutôt des *bibliothèques partagées*.

Si les bibliothèques n'ont pas besoin d'être disponibles au moment de l'écriture du code (puisque c'est dans la phase de *l'édition de liens* que la fabrication de l'exécutable se fait), il faut quand même que le compilateur connaisse des éléments sur les bibliothèques. Cela se fait par l'inclusion de *fichiers d'entête* (suffixe `.h`, habituellement rangées dans `/usr/include`) qui permettent de savoir *comment* utiliser la bibliothèque (à défaut de l'inclure *avec son code* tout de suite, ce qui est fait par les langages interprétés).

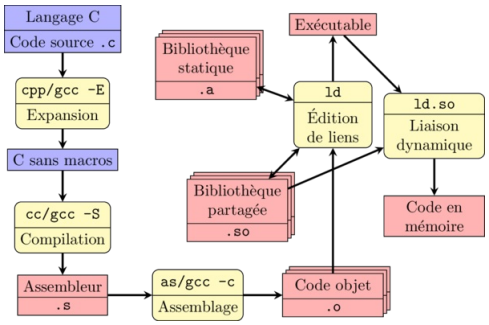
Cela se fait par des directives d'inclusion (traditionnellement au début du fichier):

```
#include <stdio.h>   /* Bibliothèque standard pour lire/écrire dans des fichiers */
#include <malloc.h>  /* Bibliothèque de gestion de mémoire */
```

On peut y voir un analogue avec les `import ...` au début des fichiers Python, mais la différence est qu'on n'a besoin que d'avoir les entêtes (la description en quelque sorte) des fonctions proposées par la bibliothèque.

In [2]:

Figure `compilation1*`.



Tout est fonction

Le reste du programme est une suite de déclarations de variables (comme vu dans les chapitres précédents) et de fonctions.

Contrairement aux scripts qui proposent de l'exécution tout au long du fichier (même si on peut y définir des fonctions), le fonctionnement du C est que même le « programme principal » est une fonction. Le système de compilation ajoute les éléments nécessaires pour que cette fonction soit appelée. La fonction appelée par défaut lors du lancement d'un exécutable s'appelle `main`.

De la même façon qu'on définit les types de données, on doit définir la façon dont la fonction interagit avec son environnement. La constante est que le C ne peut manipuler des données que s'il en connaît la taille, et donc, le type.

C'est ainsi que l'on doit donner pour chaque fonction ce qu'on appelle sa *signature*, c'est-à-dire le type des arguments de la fonction, et le type renvoyé par la fonction.

Donc une fonction se déclare comme ceci :

```
int addition(int x, int y) ...
int compte_signes(Texte t) ...
float racine_carree(float f) ...
struct paire solve_equation(float a,float b, float c) ...
```

Et après la déclaration de la fonction, on peut mettre le code de la fonction (il y a des cas où on ne fait que déclarer la signature et où oon met le code plus tard, mais dans les programmes simples que vous réaliserez, ce ne sera pas nécessaire).

Le code C qui fait des opérations (et qui doit donc faire partie d'une fonction) est forcément mis dans un *bloc de code*. Contrairement au Python, le C ne requiert pas l'indentation pour indiquer la structure de l'algorithme ; cette structure est donc indiqué par des blocs séparés entre accolades (il est possible de ne pas mettre de bloc si on a une seule instruction, mais c'est une pratique très dangereuse qui est déconseillée.

Ainsi, votre premier programme pourrait être :

```
#include <stdio.h>

/* la signature de la fonction main :
   int est renvoyé : c'est le code de retour du processus
   void comme argument est un type spécial qui ne désigne rien
   donc ici il n'y a pas d'argument à main */

int main(void) {
    printf("Hello world!"); // La fonction printf permet d'afficher un texte
    return(0);
}
```

Vous noterez au passage la syntaxe des commentaires.

Si l'on veut faire un programme un peu plus compliqué, on pourra par exemple faire la somme des éléments d'un tableau :

```
#include <stdio.h>
int main(void) {
    int somme = 0;
    int tab[6] = { 12, 34, -78, 3, 22, 45 };
    int i = 0;

    while (i<6) {
        somme = somme + tab[i];
        i = i + 1;
    }

    printf("La somme des %d nombres est %d\n",i,somme);
    return(0);
}
```

Notez bien l'usage de printf, qui permet d'afficher du texte *variable* : dans la chaîne de caractères qui lui est donnée en premier argument, les symboles % suivis d'une lettre sont remplacées par la valeur qui correspond dans les arguments suivants. Le `%d` dit d'afficher les octets qu'il trouve comme un entier (la fonction `printf` est un spéciale parce qu'elle a une signature variable). D'autres symboles permettent d'afficher les valeurs d'autres façons : `%x` affiche en hexadécimal, `%g` affiche un double, `%s` affiche une autre chaîne de caractères... et `%%` affiche le caractère `%`. Le manuel de la fonction `printf` vous donnera tous les détails (trop de détails, probablement).

Compiler un programme simple

Si compiler peut se faire en petites étapes, il est rare de s'arrêter dans les étapes intermédiaires. Tant que votre fichier contient la fonction `main` et toutes les fonctions dont il a besoin (hormis celles de la bibliothèque standard du C, dont fait partie `printf`), la commande de compilation est on ne peut plus simple :

```
gcc -o hello hello.c
#      ----- -------
#        |     \______ fichier source
#         \____________ fichier exécutable désiré
```

L'exécutable est fabriqué avec le chemin indiqué et peut être lancé par le même nom :

```
./hello # Les exécutables ne sont pas cherchés par défaut dans le répertoire courant
```

In [3]:

```
%%sh
/bin/echo '#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
    return(0);
}' > hello.c
gcc -o hello hello.c
./hello
ls -l hello hello.c
rm ./hello ./hello.c
```

```
Hello world!
-rwxr-xr-x 1 jcdubacq users 16608 déc.   6 16:09 hello
-rw-r--r-- 1 jcdubacq users    87 déc.   6 16:09 hello.c
```

In [4]:

Activité : Votre premier programme

Tapez votre premier programme en C. Compilez-le et exécutez-le. Ce programme doit simplement afficher...
`Hello, world!`

Des programmes plus structurés

Les structures importantes de contrôle existent en C : tests, boucles, fonctions... voici un exemple un peu plus complexe de programme.

```
#include <stdio.h>

int tab[6]={1,2,3,4,5,6}; /* Cette variable est globale, elle est visible de partout */

void print_tab(void) {
    int i;
    /* Itération en C : quatre parties
       initialisation
       test d'entrée dans la boucle
       conclusion de la boucle
       et derrière : le bloc de code à exécuter dans la boucle
     */
    for ( i=0 ; i<6 ; i=i+1 ) {
        printf("%d ",tab[i]);
    }
    printf("\n");
}

int doubleEntier(int a) {
    return (a+a);
}

int main(void) {
    /* On fait les déclarations de variable locales au début de la fonction */
    int i;
    int somme=0;
    somme = 0;
    for (i=0;i<6;i++) {
        somme = somme + tab[i];
    }
    while (somme < 100) {
        print_tab(); // Pas d'arguments
        somme = 0;
        for (i=0;i<6;i++) {
            tab[i] = doubleEntier(tab[i]); // Argument
            somme = somme + tab[i];
        }
    }
    printf("C'est bon !")
    print_tab();
    return(0);
}
```

Il y a beaucoup de choses à voir dans ce programme (qui n'a absolument aucun intérêt sinon).

In [5]:

Activité : Syntaxe du C

Repérez dans le programme ci-dessus la syntaxe pour les fonctions, pour les boucles (deux syntaxes), les commentaires (deux syntaxes), l'affectation de variable (normales, tableaux), etc.

Accessoirement, que fait ce programme ?

Votre proposition > _____

In [6]:

Activité : Détection d'erreurs à la compilation

Copiez ce listing dans un fichier `somme.c` et essayez de le compiler avec `gcc -o somme somme.c`. Corrigez les erreurs jusqu'à ce qu'il n'y en ait plus.

Votre proposition > _____

Voir le programme tourner

Les programmes, une fois compilées, sont exécutées directement par le processeur. Toutefois, le langage C prévoit une possibilité de surveiller l'exécution d'un programme, et d'enregistrer un lien entre le code source d'une part, et le code machine d'autre part. Une fois ce lien créé (et stocké sous forme de texte non utilisé par le processeur), il est possible de lier le résultat de l'exécution au code source.

Conjugué avec l'utilisation d'un debugger qui permet de faire tourner en mode pas-à-pas un programme (quelconque), il devient possible de comprendre pourquoi un programme ne fait pas ce qui lui est demandé.

Le site `pythontutor.com` propose une interface au-dessus de divers langages pour ce faire. La partie dédiée au langage C utilise justement ces possibilités, comme montrée dans cet exemple (https://goo.gl/z3uBDR).

Pour pouvoir faire cette surveillance, il est indispensable de d'abord surcharger l'exécutable de commentaires qui ne serviront qu'au lien. Cela se fait avec l'option `-g` du compilateur, puis en lançant le débogueur :

```
gcc -g -o somme fichiers/somme.c
gdb ./somme
# ...affichage d'information de démarrage
(gdb)
```

Une fois le débogueur lancé, il attend des instructions. À ce stade, l'exécutable est chargé, mais il ne tourne pas. Le débogueur permettra de démarrer le programme, et il s'arrêtera là où on lui a dit de s'arrêter (un *point d'arrêt*). De là, on peut faire avancer instruction par instruction ou faire continuer normalement jusqu'au prochain point d'arrêt ou jusqu'à la fin de la fonction courante. Le mode d'emploi raccourci est disponible dans la **GDB cheat sheet (https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf)**, mais voici ce dont vous aurez besoin :

- **break *nomdefonction*** pour poser un point d'arrêt (on peut aussi mettre un numéro de ligne)
- **run** pour démarrer le programme
- **display *expression*** pour afficher entre chaque étape le résultat d'une expression.
- **print *expression*** pour la même chose, mais ponctuellement
- **next** permet de passer à la ligne suivante dans le code source
- **step** permet de passer à la ligne suivante, sauf s'il y a un appel de fonction, auquel cas on *rentre* dans la fonction
- **finish** permet de reprendre normalement jusqu'à la fin de la fonction courante
- **continue** permet de reprendre normalement jusqu'à la fin du programme... (ou jusqu'au prochain point d'arrêt).

In [7]:

```
u.activite("Déboguage")
u.mark("Compilez votre programme en mode déboguage, et regardez-le s'exécuter")
u.mark("Vous pourrez utiliser la séquence suivante une fois dans `gdb` :\n\n    break main\n    run\n    display tab\n    display somme\n    display i\n    next\n    next\n    ...(beaucoup de fois next qui peut s'abréger j
uste n)\n    continue\n\nDe temps en temps, comparez l'effet de `step` et `next`.")
```

Activité : Déboguage

Compilez votre programme en mode déboguage, et regardez-le s'exécuter

Vous pourrez utiliser la séquence suivante une fois dans gdb :

```
break main
run
display tab
display somme
display i
next
next
...(beaucoup de fois next qui peut s'abréger juste n)
continue
```

De temps en temps, comparez l'effet de `step` et `next`.

Les macros

Il est possible d'avoir des expressions pré-programmées qui permettent de factoriser du code ou des valeurs, sans pour autant en faire des fonctions. Si leur utilité peut paraître assez marginale pour le code, elles sont encore largement utilisées pour définir des « constantes » qui sont fixées.

Dans notre exemple, le 6 intervient à plusieurs endroits. Il serait possible de le remplacer partout d'un seul coup. Ceci se fait par des *macros*, qui dans le cas suivant permet de passer de :

```
#include <stdio.h>
int tab[6]={1,2,3,4,5,6}; /* Cette variable est globale, elle est visible de partout */
int main(void) {
    /* On fait les déclarations de variable locales au début de la fonction */
    int i;
    int somme=0;
    somme = 0;
    for (i=0;i<6;i++) {
        somme = somme + tab[i];
    }
    printf("Somme : %d\n",somme);
    return(0);
}
```

à

```
#include <stdio.h>
#define NB_TAB 6

int tab[NB_TAB]={1,2,3,4,5,6}; /* Cette variable est globale, elle est visible de partout */
int main(void) {
    /* On fait les déclarations de variable locales au début de la fonction */
    int i;
    int somme=0;
    somme = 0;
    for (i=0;i<NB_TAB;i++) {
        somme = somme + tab[i];
    }
    printf("Somme : %d\n",somme);
    return(0);
}
```

In [8]:

Activité : Faites votre propre programme

À vous d'écrire votre propre programme ! Faites un programme qui fait la somme d'un tableau, trouve aussi le maximum et le minimum.

Si le résultat n'est pas celui qui est attendu, ou même dans le cas contraire, regardez le fonctionnement de votre programme avec le débogueur.

Entrées et sorties

La bibliothèque standard du C (`libc`) comprend entre autres une série de fonctions qui permettent (facilement) de lire et d'écrire dans un fichier. La première (`printf`) permet d'écrire suivant un formatage (nous l'avons vue plus haut), la deuxième (`scanf`) permet de lire des données, là-aussi selon un formatage.

```
#include <stdio.h>
#define NB_TAB 6

int tab[NB_TAB]={}; /* Cette variable est globale, elle est visible de partout */
int main(void) {
    /* On fait les déclarations de variable locales au début de la fonction */
    int i;
    int somme=0;
    somme = 0;
    for (i=0;i<NB_TAB;i++) {
        printf("Quel est la donnée numéro %d : ",i)
        scanf("%d",&(tab[i]));
    }
    for (i=0;i<NB_TAB;i++) {
        somme = somme + tab[i];
    }
    printf("Somme : %d\n",somme);
    return(0);
}
```

Si le code ressemble fortement à ce que nous avons déjà vu, il y a quelque chose qui paraît étrange ici:

```
scanf("%d",&(tab[i])); /* Et pourquoi pas scanf("%d",tab[i]) ? */
```

Pour expliquer ceci, il faut revenir un peu sur l'appel de fonctions:

L'appel de fonctions

Donnons nous une fonction simple :

```
int chtulhu(int a, double b, Complexe c) {
a=a+1;
b=b*2;
c.re = sqrt(c.re*c.re + c.im*c.im);
c.im = 0;
return(3);
}
```

Si on fait l'appel de `chtulhu(x,y,z)`, que se passe-t-il pour les valeurs de x, y et z ?

La réponse est que ni x, ni y, ni z ne sont modifées par l'appel de la fonction. En C, l'appel de fonction ne se fait que par *copie* des variables. Une fonction qui est appelée ne peut *jamais* modifier les variables qui lui sont passées.

En revanche, si au lieu de passer à une fonction une *variable* (c'est-à-dire qu'on recopie la valeur contenue dans la variable), on passe l'endroit où est stockée la variable, alors la copie permet autant que l'original de modifier la variable.

Si j'ai dans mon carnet les coordonnées d'un livre dans une bibliothèque (par exemple, son rayonnage et son numéro), et que je copie ces coordonnées sur un bout de papier, le bout de papier permet tout autant que le texte de mon carnet de retrouver les livres (et, si nécessaire, de gribouiller dessus).

On procède de la même façon en C pour laisser une fonction modifier une variable qui appartient à une autre fonction. Pour cela, on utilise deux opérateurs & et *.

Par exemple :

```
#include <stdio.h>

void doubleEntier(int *a) {
    *a = *a + *a;
}

int main(void) {
    int x=2;
    printf("%d\n",x);
    doubleEntier(&x);
    printf("%d\n",x);
}
```

La fonction `doubleEntier` reçoit non pas la copie de la variable x, mais l'*adresse* de la variable x, c'est-à-dire l'endroit de la mémoire où est stockée x (par exemple, 8000, et pas la valeur 2). C'est d'ailleurs précisé dans la *signature* de `doubleEntier` : a est un pointeur sur entier, et pas un entier lui-même.

Et lorsqu'on est à l'intérieur de la fonction `doubleEntier`, on n'utilise plus a (qui est une adresse) mais *a qui désigne le valeur qui est pointée par a. Le * est très polysémique en C, il faut faire très attention à son utilisation (et il est recommandé d'utiliser plutôt trop de parenthèses que pas assez).

Et donc `scanf("%d",&(tab[i]))` est bien un appel de fonction à qui on passe l'adresse à laquelle sera stockée le *i*-ème élément du tableau `tab`. Du coup, `scanf` peut remplir sa fonction qui est de modifier `tab[i]`.

In [10]:

```
u.activite("Tableau saisi à la volée")
u.mark("Modifiez votre programme pour que le tableau soit rempli dans un premier temps, puis ensuite qu'on l'examine pour trouver par exemple le maximum.")
u.activite("Travail optionnel : tri à bulles (par permutation)")
u.mark("Faites un tableau (de taille fixe) qui contient des nombres, triez ce tableau par permutation (algorithme du tri à bulles), puis affichez le tableau trié.")
```

Activité : Tableau saisi à la volée

Modifiez votre programme pour que le tableau soit rempli dans un premier temps, puis ensuite qu'on l'examine pour trouver par exemple le maximum.

Activité : Travail optionnel : tri à bulles (par permutation)

Faites un tableau (de taille fixe) qui contient des nombres, triez ce tableau par permutation (algorithme du tri à bulles), puis affichez le tableau trié.