

INTRODUCTION AU LANGAGE C

Guillaume Chanel



GÉNÉRALITÉS



UN BREF HISTORIQUE

Développé par Dennis Ritchie, Bell Labs, entre 1969 et 1973. Intimement lié au développement d'UNIX:

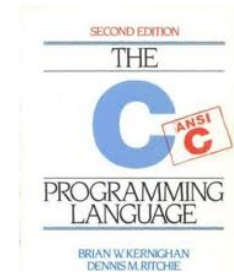
- C a été créé pour pallier aux limites des langages existants (BCPL, B) dans le développement d'UNIX.
- A partir de 1972 le noyau UNIX est développé en C



Ken Thompson (gauche), un des créateurs d'UNIX et Dennis Ritchie (droite)

Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, Inc., 1988.

Première édition: le standard "K&R-C"



NORME ANSI

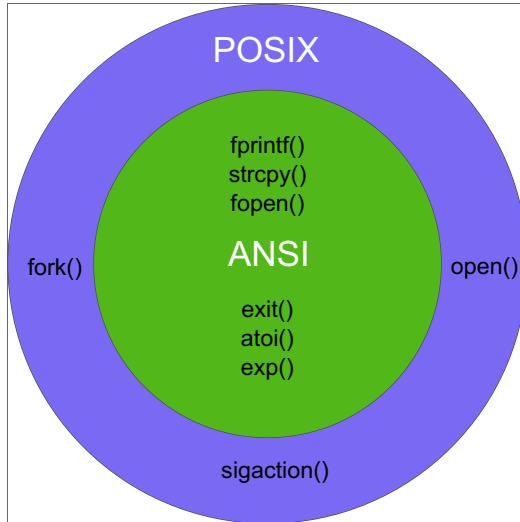
ANSI C: une norme qui garantie la portabilité:

- Définie les types manipulables et les convention d'utilisation;
- Définie les entêtes de fonctions et constantes de librairies standard;
- Exemples:
 - `math.h`: fonctions mathématiques courantes (`cos`, `exp`, `pow`, ...)
 - `stdio.h`: fonctions d'entrée/sortie du system (`printf`, `scanf`, `getc`, ...)
 - `stdlib.h`: allocation et libération de mémoire (`malloc`, `free`, ...), control du processus en cours (`exit`, ...), ...

NORME ANSI

Année	Nom(s)	Addition(s) principales
1989	C89	
1990	C90 ISO-9899:1990	Mineurs: C90 \approx C89
1999	C99 ISO-9899:1999	Gestion de nouveau types (complexes, booleen, ...), inline function, ...
2011	C11 ISO-9899:2011	Exécution de threads (default GCC \approx gnu11)

NORME ANSI



LE C AUJOURD'HUI

Beaucoup de code libre et ouvert est en C (maintenant en C++):

- GTK+, boîte à outils pour le développement d'IHM;
- Glib, boîte à outils variée (listes chaînée, arbres, timer ...);
- Qt, pour le développement d'applications et d'IHM multiplateformes;
- Boost, boîte à outil scientifique (traitement du signal, algèbre linéaire, ...).

Depuis sa création C est resté un langage populaire:

- très utilisé pour le code embarqué et le temps réel, de moins en moins utilisés pour les IHM;
- beaucoup de langages sont inspirés de C (C++, Java, C#, Objective C, PHP).

LES COMPILATEURS

Les principaux compilateurs C:

- **GCC**: nativement sous Linux et accessible sous Windows grâce à MinGW;
- Microsoft Visual Studio, Borland C: sous Windows avec IDE (Integrated Development Environment);
- TCC, LCC: compilateurs ANSI très simples (Linux + Windows).

CARACTÉRISTIQUES PRINCIPALES

Avantages

- Langage de bas niveau proche du langage machine et du système
 - code rapide et efficace
 - accès à la représentation interne des informations
 - sous UNIX /Linux le langage C est proche du système et permet de faire des appels au noyaux directement
- Langage de haut niveau
 - Indépendant de la machine (tant qu'il existe un compilateur C pour celle-ci)
 - Beaucoup de types de données sont disponibles (tableaux, structures,...)
 - La norme ANSI donne accès à des fonctions de plus haut niveau

CARACTÉRISTIQUES PRINCIPALES

Inconvénients

- Inadapté au développement occasionnel ou pour le test rapide d'algorithmes
- L'efficacité peut être au dépend de la compréhension → **Ne pas hésiter à mettre des commentaires clairs dans le code**
- Langage sans garde fou: indices des tableaux non contrôlés, il est possible de tenter un accès partout en mémoire → **programmation défensive:**
 - toujours se méfier du code que l'on croit correct, s'attendre au pire;
 - tester le résultat de tous les appels aux fonctions, penser à toutes les erreurs possibles.

UN PREMIER PROGRAMME EN C

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define PI 3.14159265359
5
6  double surface(float x) {
7      return x * x * PI;
8  }
9
10 /* La fonction main est toujours la première fonction du programmes à être
11 appelée lors que l'execution*/
12 int main(void) {
13     int input, i;
14
15     //la fonction printf affiche quelque chose
16     printf("Veuillez entrer une valeur: ");
17     scanf("%d", &input); //la fonction scanf lit l'entrée utilisateur
18     for(i=1; i < input; i++)    {
19         float per = surface(i);
20         printf("Le resultat pour %d est: %f\n", i, per);
21     }
```

MANIPULATIONS DE VARIABLES

LES DIRECTIVES DE PRÉ-COMPILATION (VERSION COURTE)

Les directives de pré-compilation:

- sont remplacées par leurs évaluation avant la compilation du programme;
- sont toujours précédée d'un '#'
- ne se terminent jamais par un ';' ;'

La directive include inclue les entêtes des fonctions d'une librairie:

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

La directive define permet de créer une sorte de "constante":

```
#define PI 3.14159265359
```

TYPAGE ET ÉCLARATION DE VARIABLES

Type	Base de déclaration
Entier	int
Virgule flottante	float double, un float tenant su plus d'octets and mémoire → plus de valeurs possibles
Caractère	char, en fait un petit entier
Vide	void, absence de valeur, utile pour les fonctions sans retour et sans paramètres ainsi que pour les pointeurs sur des types inconnus

TYPAGE ET ÉCLARATION DE VARIABLES

Il est possible de combiner ces types avec des mots clef:

Mot clef	Types acceptés	Effet
short	int	diminue la taille de l'entier en mémoire (moins de valeurs disponibles)
long	int, double	augmente la taille de la variable en mémoire (plus de valeurs disponibles)
unsigned	char, int	la valeur ne peut pas être négative → la valeur maximum du nombre augmente

```
float maVariable;  
int ceciEstUnEntier;  
unsigned int ceciEstUnEntierSuperieurAZero;  
char ceciEstUnCaractère;  
long int ceciEstUnEntierLong;
```

```
long long int ceciEstUnEntierTresLong; //C99  
long ceciEstUnEntierLong;  
double ceciEstUnDouble;  
long double ceciEstUnDoubleLong = 1.3421; //  
Declare + init.
```


BOOLÉEN

Attention ! Il n'y a pas de type booléen en C89/90. Depuis C99 on peut utiliser:

```
#include <stdbool.h>
bool unBoolean = true // (en fait true==1 et false==0)
```

D'une manière générale:

- **toute valeur différente de 0 est considérée comme vraie**
- **0 ou NULL sont considérés comme faux**

TAILLE DES VARIABLES

La taille des variables dépend des plateformes et compilateurs. Pour connaître la taille d'une variable:

- Utiliser l'opérateur **sizeof(obj)**, obj peut être un type ou un nom de variable
- les fichiers header "limits.h" et "float.h" donne les limites des types:
 - CHAR_MIN: valeur minimum d'un char
 - UINT_MAX: valeur maximum d'un int
- en norme C99, "stdint.h" contient des types avec taille explicite:
 - int8_t, int16_t, int32_t
 - uint8_t, uint16_t, uint32_t

RÈGLES SUR LES VARIABLES

Les noms de variables doivent suivre les règles suivantes:

- doit commencer par une lettre ou un `_` ;
- peut comporter uniquement lettres, chiffres et `_` ;
- ne doit pas être un nom déjà existant (nom de fonction, `if`, `sizeof`, ...);
- il est fortement conseillé d'utiliser des noms de variable claire.

N'oubliez pas d'initialiser les valeurs des variables !

Portée d'une variable:

- Si elle est déclarée entre deux accolades {}, la variable ne sera accessible que entre ces accolades
- Si une variable est déclarée hors de toute accolade (hors de toute fonction) elle est alors globale et accessible de partout. **A éviter.**

OPÉRATEUR USUELS

Type	Opérateurs	Explication / Note	Exemple
Affectation	=		val = 5
Arithmétique	+ - * / %	Reste de la division entière	y = 3/4 * x + b 10 % 4 = 2 le reste de 10 / 4 est 2
Comparaison	< <= > >= == !=	Ne pas confondre = et ==	1 > 4 (retourne 0, faux) 7 != 4 (retourne !0, vrai)
Logique	&& !	ET OU NON	(y > 8)
Binaires	& ^ ~ << >>	ET OU XOR NON Décalage à gauche Décalage à droite avec conservation du signe	5 & 10 (retourne 0) 6 << 1 (retourne 12) 9 >> 2 (retourne 2)

QUESTION

Etant donnée la definition de fonction suivante:

```
function test_if_2(int a) {  
    if(a = 2)  
        return 1; //or true  
    else  
        return 0; //or false  
}
```

Que retourne l'appel suivant ?

```
test_if_2(3)
```



OPÉRATEURS USUELS

Il est possible de combiner les opérateurs arithmétiques et logiques avec l'affectation:

```
int y;  
y = 3;  
y += 4; //Equivalent à y = y + 4;  
x <<= 2; //Equivalent à x = x << 2;
```

Il existe un opérateur d'incrément / décrément (++ et --):

```
int x, i = 0;  
i++; //equivalent à i = i + 1  
i--; //equivalent à i = i - 1  
x = ++i; //x vaut 1 car l'incrément est fait avant l'affectation  
x = i++; //x vaut toujours 1 car l'incrément est fait après l'affectation
```

L'ordre de priorité des opérateur est:

- l'ordre habituel pour les opérateur arithmétiques;
- modifiable grâce au parenthèses ().

CONVERSION DE TYPES

Dans un calcul le type des variables détermine le type du résultat:

- si un calcul implique des variables de même type le résultat sera de ce type;
- si un calcul implique des variables de types différents le type « le plus général » est choisi.

```
1 int xInt = 3, resInt;  
2 float xFloat = 3, resFloat;  
3 char xChar = 3;  
4 resInt = xInt / 4; //resInt vaudra 0 car le calcul est entier  
5 resFloat = xInt / 4; //resFloat vaudra 0.00 car le calcul est entier  
6 resFloat = xInt / 4.; //resFloat vaudra 0.75 car 4. est un float  
7 resFloat = xFloat / 4; //resFloat vaudra 0.75 car xFloat est un float  
8 resInt = xFloat / 4; //resInt vaudra 0 (cas resInt est un entier)  
9 resInt = xChar + 2; //Conversion automatique et possible (resInt vaudra 5)
```

CASTING (MOULAGE)

Le casting permet de forcer la conversion de type

```
resFloat = (float) xInt / 4; // resFloat vaudra 0.75 car x est convertit en float
resFloat = (float) (xInt / 4); /* resFloat vaudra 0.00 car le calcul entier est fait
                                avant conversion */
resInt = (float) xInt/ 4; // resInt vaudra 0 car resInt est un entier
```


QUESTION

Quel est le problème avec les calculs suivants ?

```
1  int i; long int l; float f; char c; unsigned char uc;
2
3  c = -100;
4  uc = c;
5
6  c = 'i'; //code ascii = 239
7  uc = c;
8
9  l = 320254468;
10 f = 45879651324476.5;
11 i = l * f;
```

CONTRÔLE DU FLOT D'EXÉCUTION

STRUCTURE CONDITIONELLES

Les branchements `if`

```
int a, b, c;
...
if((a < 0) || !((b == 2) && (c != 4))) {
    printf("La condition est validée\n");
}
else {
    printf("La condition n'est PAS validée\n");
}
```

LES BOUCLES FOR

Les boucles for

```
//Calcul la somme des N premiers entiers
int value = 0;
for(int i=0; i < N; i++) {
    printf("Nouvelle itération de la boucle\n");
    value += i;
}
```

LES BOUCLES WHILE

La boucle while

```
//Calcul la somme des N premiers entiers
int value = 0;
int i = 0;
while(i < N) {
    value += i;
    printf("Nouvelle itération de la boucle: %d\n", i++);
}
```

La boucle do . . . while: le contenu est toujours exécuté au moins une fois

```
//Calcul la somme des N premiers entiers
int value = 0;
int i = 0;
do {
    value += i;
    printf("Nouvelle itération de la boucle: %d\n", ++i);
}
while(i < N);
```

INTERRUPTION DE BOUCLES

Il est possible d'interrompre le déroulement d'une boucle (for, while, do/while) en utilisant les mots clefs:

- `continue`: passe à l'itération suivante;
- `break`: sort de la boucle.

```
//Calcul la somme des nombre impairs allant de 0 à N
value = 0;
i = 0;
while(1) {
    if(i > N)
        break;
    if(i % 2 == 0) {
        i++;
        continue;
    }
    value += i;
    printf("Nouvelle itération de la boucle: %d\n", ++i);
}
```

Attention dans certains cas (cf. ci-dessus), l'utilisation de `break` et `continue` est à éviter car elle rend le code moins lisible.

LES FONCTIONS

Une fonction est toujours déclarée avant son utilisation:

```
typeValeurRetour nomDeLaFonction(type1 param1, ..., typeN paramN)
```

Le mot clef `return` indique un point d'arrêt de la fonction ainsi que la valeur que la fonction doit retourner.

```
float pourcentage(int valeur, int centPourcent) {  
    return ((float) valeur/centPourcent)*100;  
}
```

Une fonction peut ne rien retourner:

```
void pourcentage(int valeur, int centPourcent) {  
    printf("%f", ((float) valeur/centPourcent)*100);  
}
```

LES FONCTIONS

Il est possible de déclarer une fonction juste avec son entête:

- évite d'organiser les fonctions suivant la position de leurs appels;
- Sépare l'interface utilisateur de l'implémentation, un premier pas vers la création de librairies.

```
1 float pourcentage(int , int); //Déclaration de l'entête
2
3 int main(void) {
4     int a,b;
5     ...
6     //Utilisation de la fonction sans implémentation connue
7     pourcentage(a,b)
8     return 0;
9 }
10
11 float pourcentage(int valeur, int centPourcent){
12     return ((float) valeur/centPourcent)*100;
13 }
```


PASSAGE PAR ADRESSE

En C les paramètres sont passés par valeur (pas de modification des variables entrées).
Il n'y a pas de passage par référence, on a recours au passage par adresse:

- on déclare les paramètres de la fonction comme pointeurs et on les utilise comme tel avec le symbole *;
- lors de l'appel à la fonction on passe l'adresse de la variable, symbole &.

```
/
*****
*****
Calcul le pourcentage du nombre "valeur" par
rapport à "centPourcent"
IN:
    valeur: contient la valeur a mettre en
rapport pour le calcul
           du pourcentage. Une fois la
fonction exécutée valeur
           contient le pourcentage.
    centPourcent: valeur par rapport à laquelle
le pourcentage est calculé
OUT:
    retourne -1 si une valeur d'entrée n'est
pas valide, 0 sinon
*****
*****/
```

```
/* Utilisation de la fonction */
int main(void) {
    int val = 30; ret;
    if ( (ret = versPourcentage(&val, 100)) < 0
    )
        return ret;
    else
        return val;
}
```

```
int versPourcentage(float *valeur, float
centPourcent) {
if( (centPourcent > 0) && (*valeur >= 0) ) {
    *valeur = (*valeur / centPourcent)*100;
    return 0;
}
else
    return -1;
}
```

LES TABLEAUX ET CHAINES DE CARACTÈRES

DÉCLARATION DES TABLEAUX

Les tableaux sont:

- de type unique;
- indicés de 0 à N-1 pour un tableau de N éléments;
- organisés de manière contigüe en mémoire;
- statiques ou dynamiques (cf. pointeurs et C99).

Déclaration d'un tableau:

```
#define TAILLE_MAX 10  
int tableauEntiers[TAILLE_MAX];  
double tableauDeDoubles[TAILLE_MAX];
```

UTILISATION DES TABLEAUX

Affecter et lire les valeurs des tableaux:

```
// Initialization du tableau
for(int i=0; i<TAILLE_MAX; i++)
    tableauEntiers[i] = i+1;

// Alternative mais uniquement lors de la declaration
int tableauEntiers[TAILLE_MAX] = {1,2,3,4,5,6,7,8,9,10};

// Affichage du tableau
for(int i=0; i<0; i++)
    printf("Le tableau contient à l'indice %d la valeur %d", i, tableauEntiers[i]);
```

ATTENTION !

Attention aux indices des tableaux !

Hors des limites on attends un comportement indéfini:

- Au mieux tout fonctionne correctement.
- Eventuellement une erreur de segment.
- Au pire la valeur d'une autre variable est modifiée et on peut obtenir des erreurs importantes.

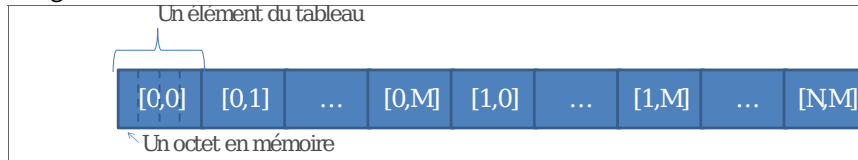
TABLEAU À N DIMENSIONS

```
//Lignes et colonnes sont arbitraires
#define NB_LIGNES 3
#define NB_COLONNES 5
#define NB_MORE 7

double tab2D[NB_LIGNES][NB_COLONNES];
float tab3D[NB_LIGNES][NB_COLONNES][NB_MORE];
float tab4D[NB_LIGNES][NB_COLONNES][NB_MORE][NB_MORE];

for(i=0;i<NB_LIGNES;i++)
    for(j=0;j<NB_COLONNES;j++)
        tab2D[i][j] = 0.0;
```

L'organisation en mémoire reste linéaire !



CHAINE DE CARACTÈRES

Il n'y a pas de type chaîne de caractère (string) en C, **pour cette raison on utilise des tableaux de caractères.**

Une chaîne de caractères se termine toujours par le caractère “\0”:

- il faut donc penser à réserver une place pour ce caractère dans le tableau;
- tout ce qui suit le “\0” est ignoré;
- si il n'y a pas de “\0” la chaîne de caractères est sans fin (donc plus que la taille du tableau).

MANIPULATIONS DES CHAINES

```
char maChaine[8] = "C cool!"; /* \0 est ajouter après la chaine (pensez a
                                reserver de la place) */
char maChaine2[] = "C cool!"; // Alloue automatiquement la bonne taille
printf("%s\n", maChaine2); //Affiche la chaine
maChaine[1] = '\0'; // ' ' pour un caractère et "" pour une chaine
printf("%s\n", maChaine); //Affiche "C"
```

Il existe des fonctions de manipulation de chaines dans la librairie string (inclure string.h), par exemple:

- strncpy(dest, src, n), copie n caractère de la chaine src vers dest;
- strlen(str), retourne la taille de la chaine sans compter '\0';
- strcmp(str1, str2), retourne 0 si les deux chaines sont identiques;
- strncat(dest, src, n), ajoute src à la fin de dest (enlève le '\0'), copie au maximum n bytes de src.

ENTRÉE / SORTIE CONSOLE

Pour afficher du texte sur stdout (console) on utilise la fonction printf:

```
printf(texteEtFormat, variable1, variable2, ...)
```

- **texteEtFormat**: une chaîne de caractère à afficher entre "" qui spécifie les positions et types des variables à afficher.
 - **variable1, variable2**, variable à afficher dans l'ordre de leur apparence dans **texteEtFormat**.
-

Pour lire du texte sur stdin (console) on utilise la fonction scanf:

```
scanf(texteEtFormat, *variable1, *variable2, ...)
```

- **texteEtFormat**: une chaîne de caractère comme pour printf. Attention taper du texte ici indique que ce texte doit être saisi par l'utilisateur et non pas que le texte sera affiché sur stdout.
 - ***variable1, *variable2**, adresse des variables où vont être rangées les valeurs entrées par l'utilisateur. Pour avoir l'adresse du variable on utilise le symbole &.
-

Les entête de fonction font partie du fichier `stdio.h`

ENTRÉES / SORTIE CONSOLE

liste des principaux spécificateur (man `printf` pour une liste complète):

Spécificateurs Affichage

%d, %i	integer, char
%f	float, double
%c	unsigned char afficher sous forme de caractère
%u	unsigned integer, char
%s	string (chaines de caractères)
%lS	précède le spécifieur S par une indication de long (e.g. %ld)
%Pd, %P.Sf	P indique la taille minimum du champ à afficher S indique le nombre de chiffres significatifs après la virgule

ENTRÉES / SORTIE CONSOLE

On utilise les combinaisons de symboles suivantes pour les caractères spéciaux:

Symboles Affichage

<code>\n</code>	saut de ligne
<code>\r</code>	retour à la ligne
<code>\t</code>	tabulation
<code>\\</code>	backslash
<code>\'</code> <code>\"</code>	simple ou double quote
<code>\0</code>	NUL character, utilisé pour indiquer la fin d'une chaîne de caractère

ENTRÉES / SORTIE CONSOLE

Autres fonctions intéressantes pour les entrées sorties:

```
int getchar( void )
```

- attends une entrée clavier STDIN
- retourne le code du caractère tapé (sans écrire sur STDIN);

```
int putchar( int car )
```

- écrit le caractère en paramètre sur STDOUT;
- retourne le caractère si pas d'erreur, EOF sinon.

```
char* fgets ( char* string, int size, stdin ) //!!!
```

- Ne pas utiliser « gets »
- lit une chaîne de caractère sur STDIN de taille maximum size et la place dans “string”;
- ≡ • stdin est en fait un pointeur sur FILE;

- retourne NULL en cas d'erreur.

```
int puts (const char* string)
```

- écrit la chaîne de caractère "string" sur STDOUT;
- retourne EOF en cas d'erreur.

EXEMPLE D'ENTRÉES / SORTIES

```
#include <stdio.h>
/* La fonction main est toujours la première fonction du programme à être
appelée lors que l'exécution*/
int main(void)
{
    float inputFloat;
    char inputChar;
    int nbCorrespondance;

    //Entrée de l'utilisateur pour un float et affichage du
    //float de différentes manières
    printf("Veuillez entrer un float:\t");
    scanf("%f", &inputFloat);
    printf("Affichage de l'entree sous forme de float: %f\n", inputFloat);
    printf("Idem avec 2 digits apres la virgule: %.2f\n", inputFloat);
    printf("Idem avec au moins 6 caractères: %6.2f\n", inputFloat);
    printf("Notation scientifique: %e\n", inputFloat);
    printf("Affichage de l'entree sous forme d'entier %d\n", inputFloat);

    //Boucle tant que l'utilisateur n'appuye pas sur 'enter' uniquement
    do
```


QUESTION

Dans quel cas ce code provoque des erreurs?

```
#include <stdio.h>
#define TAILLE_MAX 3

void affiche_chaine(char*, int);

int main(void)
{
    int valeur1 = 1;
    char chaine[TAILLE_MAX] = {'a', 'b', '\0'};
    int valeur2 = 2;

    // Affiche l'état de la mémoire pour les variable ci-dessus
    printf("Valeur1:\tAdresse: %x\tValeur:%d\n", &valeur1, valeur1);
    affiche_chaine(chaine, TAILLE_MAX);
    printf("Valeur2:\tAdresse: %x\tValeur:%d\n", &valeur2, valeur2);

    // NE JAMAIS UTILISER GETS
    . . . . .
}
```

EXÉCUTION DE PROGRAMME

LA FONCTION MAIN

La fonction main est le point d'entrée du programme et a l'entête suivante:

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

Arguments:

- argc: nombre d'arguments passés au programme lors de son appel;
- argv: un tableau de chaînes de caractères contenant les arguments;

La fonction main retourne un code entier indiquant généralement:

- 0: le programme s'est terminé avec succès;
- !=0: le programme a rencontré une erreur.

La fonction `exit (int)` permet également de terminer un programme à tout moment en renvoyant le code en paramètre (`stdlib.h`).

Dans le shell on peut taper "echo \$?" pour avoir ce code de retour.

EXEMPLE DE FONCTION MAIN

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    int i, sum = 0;
    if(argc != 0)
        printf("Le nom du programme est: %s\n", argv[0]);
    if(argc > 1)  {
        //Sum the inputs
        for(i=1;i < argc; i++) {
            //Affiche le paramètre traité
            printf("Param %d: %s\n", i, argv[i]);
            //Convert the parameter to a number and sum it
            sum += atoi(argv[i]);
        }
    }
    return sum;
}
```

VARIABLES D'ENVIRONEMENTS

La variable globale `environ` est un tableau de chaîne de caractère permettant d'accéder aux variables d'environnement. Cette variable n'est pas utilisée directement, on préférera utiliser:

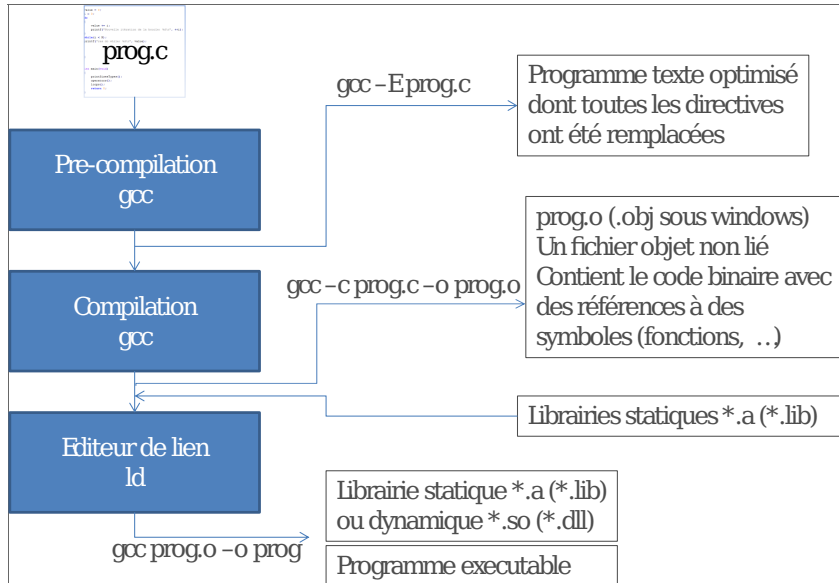
```
char* getenv (const char *name)
```

- retourne la valeur de la variable d'environnement `name` sous forme de chaîne de caractère (NULL si pas défini);
- attention, la chaîne retournée ne doit pas être modifiée et peut être changée par des appels successifs à `getenv`.

```
int putenv (char *string)
```

- retourne 0 en cas de succès (une autre valeur sinon);
- si `string` est de la forme `variable=va` cette définition est ajoutée à l'environnement. Si la forme de `string` est `variable` la variable est supprimée de l'environnement;
- ≡ • attention, si `string` est modifiée ultérieurement, l'environnement le sera également.

COMPILER ET LIER UN PROGRAMME



COMPILER UN PROGRAMME

gcc appelle automatiquement ld, **pour compiler et lier un programme il suffit donc d'utiliser gcc:**

```
$ gcc prog.c -o prog
```

Les options principales de gcc:

Option	Effet
-Wall	Affiche tous les warning possibles
-I dir	Inclue le répertoire dir pour la recherche de .h
-g	Génère les informations symboliques pour debugage

COMPILER UN PROGRAMME

Lorsque gcc trouve une option qu'il ne connaît pas il passe cette option et les suivantes à ld.

Il faut toujours mettre les options ld à la fin:

Option Effet

-L dir	Inclue le répertoire dir pour la recherche de librairie statiques
--------	---

-l nom	Inclue la librairie statique libnom (ne pas mettre le lib)
--------	--

Les libraires portent le nom libnom et ce trouvent généralement dans:

```
/usr/lib  
/usr/lib64
```

Par exemple la librairie svn s'appelle "libsvn" et pour la lier on utilise:

```
$ gcc monClientSVN.c -o clientsvn -l svn
```

DEBUGGER UN PROGRAMME

Si une erreur intervient lors de l'exécution d'un programme le système génère souvent un **coredump**. Le coredump est une image de la mémoire (et donc de **l'état**) **du processus lors de son arrêt**.

On peut consulter ces informations en utilisant le debugger gdb (GNU debugger):

```
$ gdb monexecutable -c core
```

On peut alors voir:

- la ligne de code responsable de l'erreur;
- la valeur des variables, des arguments des fonctions, de la pile d'exécution des fonctions, ...

Il est bien sûr préférable que l'exécutable contienne des informations symboliques de débogage (option -g).

DEBUGGER UN PROGRAMME

Il est possible d'utiliser gdb pour debugger un programme directement:

```
$ gdb monProg
```

Une fois dans le debugger les commandes suivantes sont utiles (help):

- `run`: lance le programme qui poursuit son exécution jusqu'à une erreur ou la fin;
- `list`: liste les 10 lignes de codes autour du point actuel;
- `break param`: positionne un breakpoint à la ligne ou la fonction `param1`;
- `clear param`: supprime un breakpoint;
- `cont`: continue l'exécution du code;
- `step`: exécute la prochaine ligne de code;
- `print param`: affiche la valeur courante de la variable `param`;
- `info param`: information sur beaucoup de choses (`info locals`, `info`)

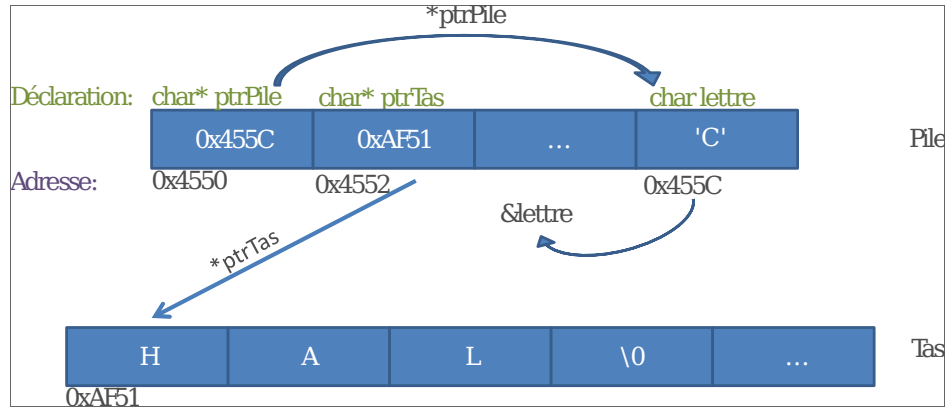


- `quit`: quitte le débogage.

LES POINTEURS



CONCEPT



EQUIVALENCE POINTEURS ↔ TABLEAUX

Un tableau (e.g. chaîne de caractère) est représentable par un pointeur et une taille.

```
int main(int argc, char* argv[]) {
    char *chainePtr;
    char chaineTab[] = "Je suis bien content !";
    long long int *intPtr, i;
    long long int intTab[TAILLE_TAB];

    //Initialisation du tableau d'entiers
    for(i=0;i<TAILLE_TAB;i++)
        intTab[i] = i+1;

    //Mettre les pointeurs sur les tableaux
    intPtr = intTab; //equivalent à intPtr = &intTab[0];
    chainePtr = chaineTab;

    //Affichage des equivalences
    printf("Adresse chainPtr: %s, contenu chaineTab: %s\n", chainePtr, chaineTab);
    printf("Adresse chainePtr: %x, adresse chaineTab: %x\n", chainePtr, chaineTab);
    printf("Contenu chainPtr: %s, contenu chaineTab: %s\n", chainePtr, chaineTab);

    for(i=0;i<TAILLE_TAB;i++)
        printf("%ld = %ld\n", *(intPtr+i), intTab[i]);
}
```

QUESTION

Quelle est l'équivalence `tab[i][j]` sous forme de pointeurs ?

Quelle est la différence de taille entre

```
sizeof(char*)
```

et

```
sizeof(long int*)
```


LES POINTEURS POUR LES CHAINES DE CARACTÈRES

En fait toutes les entêtes de fonctions utilisant des chaines de caractère utilisent `char*`.
Mais attention il reste des différences:

```
//Exemple de difference:
printf("Taille pointeur: %d, Taille tableau : %d\n", sizeof(intPtr),
sizeof(intTab));
//Output: Taille Pointeur: 4, Taille tableau: 160 (20*8, 8 taille d'un long long
int)
```

`const` permet de s'assurer qu'une chaine de caractères (ou toute valeur pointée) ne soit pas modifiée par la fonction:

```
int afficheChaine(const char *tab)
{
    while(*tab) //Equivalent à while(*tab != '\0')
        putchar(*(tab++));
    puts("\n");
    /*tab = 'a'; //Erreur à la compilation !
}
```

ALLOCATION DYNAMIQUE

L'allocation dynamique se fait **sur le tas**:

- permet d'allouer de la mémoire sans savoir à l'avance quelle est la quantité nécessaire exacte;
- La taille de la mémoire allouée dépend donc de l'exécution du programme (i.e. de l'utilisateur et du système).
- cette allocation se fait dans une zone de la mémoire du processus appelée «tas».

Des fonctions sont disponibles dans `stdlib` pour effectuer l'allocation de la mémoire:

- `malloc(nbOctets)`: retourne un pointeur sur une zone allouée de `nbOctets` ou `NULL` en cas d'erreur d'allocation;
- `calloc(nbElements, nbOctets)`: retourne un pointeur sur une zone allouée de `nbElements` de `nbOctets` chacun initialisés à 0. Retourne `NULL` en cas d'erreur d'allocation;
- `free(ptr)`: libère l'espace mémoire pointé par `ptr`. **Ne remet pas `ptr` à `NULL` !**

EXEMPLE D'ALLOCATION DYNAMIQUE

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    float *dynFloat = NULL;
    char *buffer = NULL;
    int taille;
    printf("Entrer la taille desiree: "); scanf("%d", &taille);

    //malloc et calloc retournent void*, il est donc nécessaire de
    //faire un cast dans le bon type du pointeur
    dynFloat = malloc(taille*sizeof(float));
    buffer = calloc(taille, sizeof(char));

    //Erreur lors de l'allocation
    if((dynFloat == NULL) || (buffer == NULL)) {
        printf("Erreur: impossible d'allouer la memoire necessaire.\n");
        free(buffer); free(dynFloat); //Ne fait rien si == NULL
        return 1; //ou exit(1)
    }

    *buffer = 'H'; *(buffer+1) = 'A'; *(buffer+2) = 'I';
```

STRUCTURES DE DONNÉES



DÉFINITION DE NOUVEAUX TYPES

Il est possible de définir de nouveaux type de variable grâce au mot clé `typedef`.

Cela permet de:

- augmenter la lisibilité du code;
- garantir une opacité du point de vu de l'utilisateur (il n'a pas besoin de savoir ce qui se cache derrière le type définit);
- changer le type d'une variable sans avoir à changer toutes les entêtes de fonctions.

```
typedef unsigned char bool; //definition d'un type booleen

typedef int number; //peut etre facilement remplacé par un float
number traiteLesDonnees(number x); //Fonction qui ne changera pas (à priori)

// TYPES OPAQUES
typedef /*something*/ time_t; //Definition officielle de time_t (time.h)
//It is almost universally expected to be an integral value representing
// the number of seconds elapsed since 00:00 hours, Jan 1, 1970 UTC.

// Definition du type FILE utilisé (mais pas déclaré) dans stdio.h
struct _IO_FILE;
```



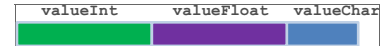
```
typedef struct _IO_FILE FILE;
```

DÉFINITION DE STRUCTURES

Une structure de donnée est une collection de données hétérogènes:

- chaque champ peut avoir un type différent;
- chaque champ à son propre espace mémoire réservé.

Représentation mémoire



Déclaration d'une variable

```
struct {  
    int valueInt;  
    float valueFloat;  
    char valueChar;  
} maVariable;
```

Attention: le mot clef struct ne définit pas un type !

QUESTION

Quel type de structures de données complexe les `struct` permettent de construire ?



UTILISATION DE STRUCTURES (DÉFINITIONS)

Une structure est définie et appelée de la manière suivante:

```
//Déclaration de la structure
struct personne {
    char* nom;
    char* prenom;
    int age;
};

struct personne unePersonne; // une instance de la structure;
```

Pour plus de facilité on a souvent recours aux définitions de types:

```
typedef struct el { //el est necessaire pour faire les declaration de pointeurs
    struct el *suivant;
    void* contenu;
} element_t; //A partir de cette définition on peu déclarer 'element_t unElement'

typedef element_t* listeChaine;
```

UTILISATION DE STRUCTURES (API PERSONNE)

```
struct personne createPersonne(const char* nom, const char* prenom, int age) {
    int length;
    struct personne pers;

    length = strlen(nom) + 1; //+1 pour '\0'
    pers.nom = calloc(length, sizeof(char));
    pers.prenom = calloc(strlen(prenom) + 1, sizeof(char));
    if((pers.nom == NULL) || (pers.prenom == NULL)) {
        printf("Erreur allocation mémoire\n");
        exit(1);
    }
    strcpy(pers.nom, nom);
    strcpy(pers.prenom, prenom);
    pers.age = age;
    return pers;
}
```

UTILISATION DE STRUCTURES (API LISTES)

```
listeChaine initListeChaineVide() {  
    return NULL;  
}  
  
void addListe(listeChaine* liste, void* contenu) {  
    //Creation d'un element et ajout dans la liste  
    element_t *unEl = (element_t*) malloc(sizeof(element_t));  
    unEl->contenu = contenu; //(*unEl).contenu se transforme en unEl->contenu  
    unEl->suivant = *liste;  
    *liste = unEl;  
}  
  
void* removeListe(listeChaine *liste) {  
    listeChaine tmp;  
    void* ret;  
  
    ret = (*liste)->contenu;  
    tmp = *liste;
```

UTILISATION DE STRUCTURES (PROGRAMME)

```
int main(void)
{
    struct personne *ptPersonne;
    listeChaine maListe = initListeChaineVide();

    //Une première personne ajoutée
    unePersonne = createPersonne("Rodepeter", "Jessica", 36);
    affichePersonne(unePersonne);
    addListe(&maListe, &unePersonne);

    //Une seconde personne ajoutée
    if((ptPersonne = malloc(sizeof(struct personne))) == NULL)
    {
        printf("Erreur mémoire\n");
        return 1;
    }
    *ptPersonne = createPersonne("Page", "Marc", 8);
    affichePersonne(*ptPersonne);
    addListe(&maListe, ptPersonne);
}
```

ENUMÉRATIONS

Les énumérations permettent de créer des types:

- dont les valeurs sont discrètes, qualitative et éventuellement ordonnées;
- en fait ces valeurs sont représentées par des entiers.

Exemples de déclarations:

```
enum typeMedaille { bronze, argent, or };  
typedef enum { false, true } booleen;  
typedef enum { lundi = 1, mardi, mercredi, jeudi, vendredi, samedi, dimanche }  
jours;  
typedef enum { micro = 2, mineur = 4, leger = 5, modere = 6, major = 7, important =  
9,  
                devastateur = 10} richter;
```

Exemples d'utilisations:

```
jours aujourd'hui;  
aujourd'hui = mardi;  
if(aujourd'hui != mardi)  
    printf("Que faites-vous la ?\n");  
else  
    printf("Merci d'etre la.\n");
```

ENUMÉRATIONS

```
enum typeMedaille maMedaille = or;
switch(maMedaille) //Séparer les different cas que peut prendre une variable
{
    case bronze:
        printf("Peux mieux faire !\n");
        break; //A ne pas oublier
    case argent:
        printf("Pas mal !\n");
        break;
    case or:
        printf("Le controle est par la...\n");
        break;
    default:
        printf("Erreur: type de medaille non connu\n");
}
```

Attention les énumération ne sont PAS fortement typées:

```
//Toutes ces affectations sont autorisées !!!
//Mais à éviter !
int tremblementFukushima = 9; //Déclaré comme entier au lieux de richter
richter tremblement = 3; //La valeur 3 n'est pas dans la liste
booléen estValideMaisFaux = mardi; //mardi = 2 -> n'est pas dans la liste
```

UNIONS

Comme les structures, une union a plusieurs champs mais dans le même espace mémoire:

- l'adresse de chaque champs est la même;
- l'union a la taille du plus grand des champ.

Structures



```
typedef struct {  
    int valueInt;  
    float valueFloat;  
    char valueChar;  
} u;
```

Unions



```
typedef union {  
    int valueInt;  
    float  
valueFloat;  
    char  
valueChar;  
} s;
```


UNIONS

Utilisé pour:

- minimiser l'espace mémoire nécessaire à un programme;
- la simulation du polymorphisme avec la création de type « variant »;
- **la conversion de types (conversion sauvage non contrôlée par le compilateur).**

```
#include <stdio.h>
#include <stdint.h>
typedef union {
    int16_t valueInt;
    struct {
        unsigned char msb; //most significant byte
        unsigned char lsb; //least significant byte
    } bytes;
} convert;

int main(void) {
    convert value;
    printf("Entrer un entier: ");
    scanf("%d", &value);
    printf("Valeur entree: %d\n", value.valueInt);
    printf("Valeur entree (hexadecimal): %x\n", value);
```



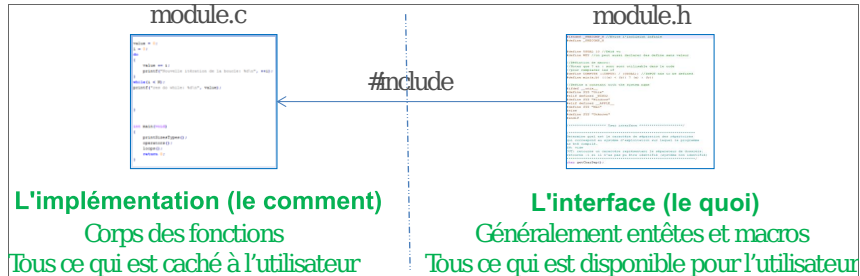
```
    printf("MSB: h%02x, LSB h%02x\n", value.bytes.lsb, value.bytes.msb);  
}
```

ORGANISATION D'UN PROGRAMME



MODULES

Un programme complexe est divisé en modules (.c + .h)

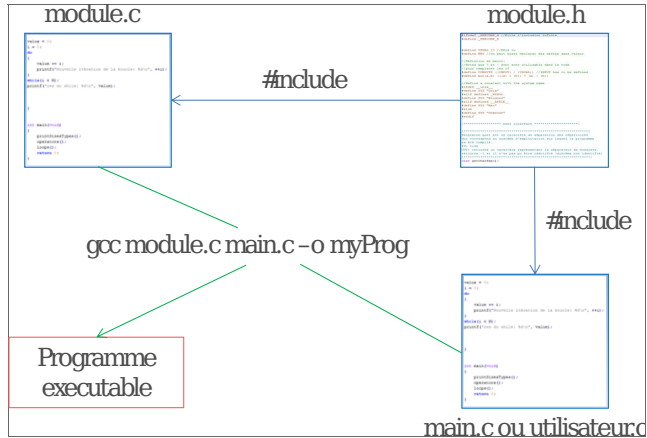


Cela permet notamment de:

- créer une forme d'encapsulation qui facilite le débogage et les mises à jour;
- réduire les temps de compilation en ne compilant que les modules modifiés;
- répartir le travail entre programmeurs;
- favoriser le partage de code (bibliothèques).

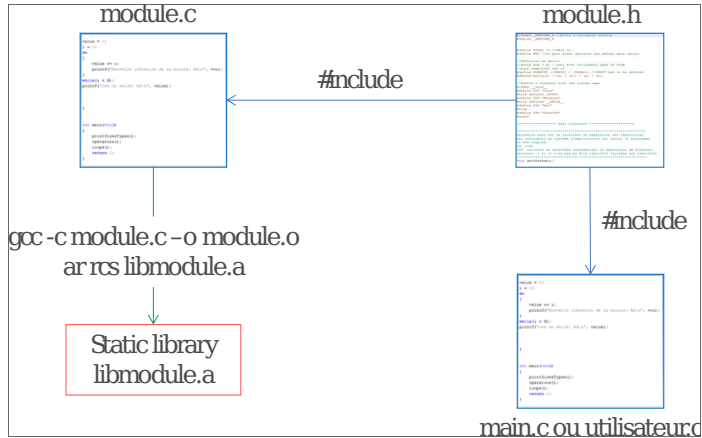
COMPILATION DE MODULES

On peut utiliser un module en le compilant avec son programme principal:



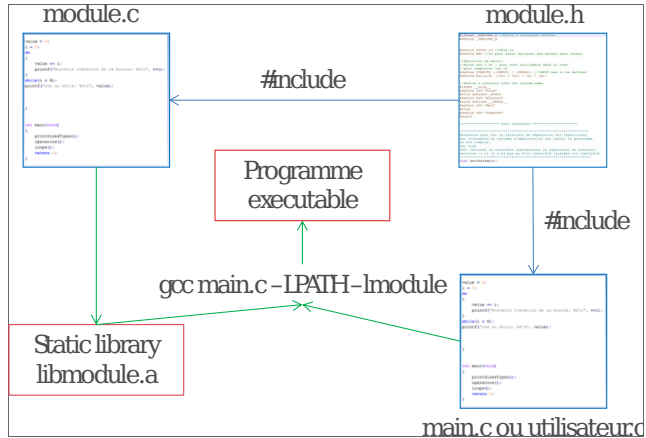
COMPILATION DE LIBRAIRIE

On peut utiliser un module en créant une librairie (statique dans cet exemple):



LIER UNE LIBRAIRIE

Pour utiliser la librairie il faut la lier:



TYPES OPAQUES

Les types opaques sont des structures de données (i.e. typedef) qui ne sont **pas définies dans l'interface** (i.e. fichier header).

Cela permet:

- cacher les détails l'implémentation -> abstraction;
- de modifier la structure de donnée sans modifier le comportement du code utilisateur.

ListeChaine.c

```
#include "listeChaine.h"

/*Déclaration dans le .c pour une
utilisation privée */
typedef struct el {
    struct el *suivant;
    void *contenu;
} element_t;

listeChaine initListeChaineVide()
{
    return NULL;
}
```

ListeChaine.h

```
// Anciennement typedef element_t* listeChaine;
typedef struct el* listeChaine;

listeChaine initListeChaineVide();
void addListe(listeChaine* liste, void*
contenu);
void* removeListe(listeChaine *liste);
```


FONCTIONS `static`

Le mot clef `static` permet de:

- s'assurer que une fonction sera locale à un module (i.e. elle ne pourra pas être utilisée dans un autre module);
- Libérer le nom de cette fonction pour d'autres modules.

main.c

```
1  #include <stdio.h>
2
3  void show() { // PAS DE
    CONFLIT !
4      printf("Main show\n");
5  }
6
7  void main() {
8      show();
9      interface();
10 }
```

interface.c

```
1  #include <stdio.h>
2
3  static void show() { //PAS
    DE CONFLIT !
4      printf("Interface
    show\n");
5  }
6  void interface() {
7      show();
8  }
```

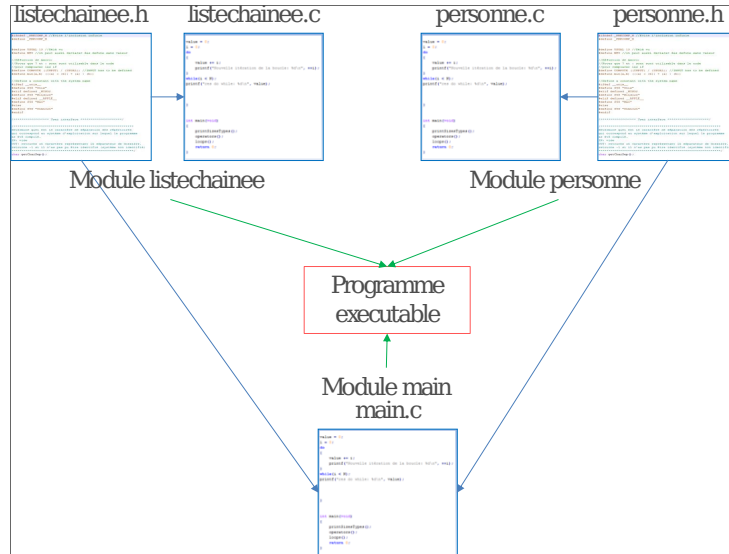


Attention le mot clef `static` est aussi utilisé pour des variables mais il peut avoir un sens différent

dans ce cas.



EXEMPLE D'ORGANISATION D'UN PROGRAMME

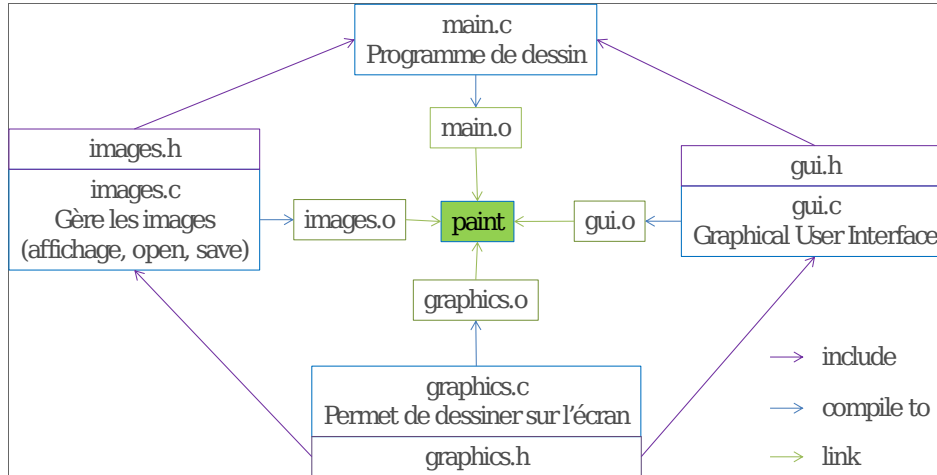


MAKEFILES



COMPLEXITÉ DE COMPILE MODULAIRE

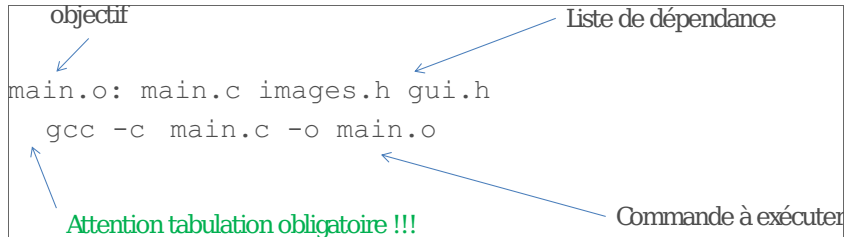
Lorsque l'on a plusieurs modules qui dépendent les uns des autres il devient difficile de savoir quel module il faut recompiler après une modification.



CRÉATION D'UN MAKEFILE

Un makefile est un fichier texte qui contient des objectifs de compilation:

- chaque objectif représente (en général) un fichier .o ou le programme;
- chaque objectif est associé à:
 - une liste de dépendances (.o, .c, .h);
 - une commande à exécuter **si une des dépendances à une date plus récente que l'objectif.**



The diagram shows a Makefile rule with annotations. The rule is:

```
main.o: main.c images.h gui.h
gcc -c main.c -o main.o
```

Annotations with arrows pointing to the rule:

- objectif** points to `main.o`.
- Liste de dépendance** points to the list of dependencies `main.c images.h gui.h`.
- Commande à exécuter** points to the command `gcc -c main.c -o main.o`.
- Attention tabulation obligatoire !!!** points to the first tab character before the command.

EXEMPLE DE MAKEFILE

```
paint: main.o images.o gui.o graphics.o
    gcc main.o images.o gui.o graphics.o -o paint

main.o: main.c images.h gui.h
    gcc -c main.c -o main.o

images.o: images.c images.h graphics.h
    gcc -c images.c -o images.o

gui.o: gui.c gui.h graphics.h
    gcc -c gui.c -o gui.o

graphics.o: graphics.c graphics.h
    gcc -c graphics.c -o graphics.o
```

UTILISATION D'UN MAKEFILE

Pour compiler il suffit de taper:

```
$ make objectif
```

Pour compiler le premier objectif:

```
$ make
```



VARIABLES

On peut également ajouter des noms de variable pour effectuer des changements facilement ou générer le makefile automatiquement:

```
VARIABLE = value
```

Pour utiliser le contenu de la variable:

```
$(VARIABLE) ou ${VARIABLE}
```

On peut aussi définir des variables lors de l'exécution de la commande make:

```
$ make VARIABLE=value objectif
```

Dans ce cas on utilise `?` si l'on souhaite remplacer la valeur de la variable dans le makefile par celle donnée en paramètre par l'utilisateur:

```
VARIABLE ?= value
```

AMÉLIORATION D'UN MAKEFILE (1)

```
CC = gcc
OBJS = main.o images.o gui.o graphics.o
CFLAGS = -g -Wall -c
LFLAGS = -L /home/me/blas/openblas/lib -l openblas

paint: $(OBJS)
    $(CC) $(OBJS) -o paint $(LFLAGS)

main.o: main.c images.h gui.h
    $(CC) $(CFLAGS) paint.c -o main.o

images.o: images.c images.h graphics.h
    $(CC) $(CFLAGS) images.c -o images.o

gui.o: gui.c gui.h graphics.h
    $(CC) $(CFLAGS) gui.c -o gui.o

graphics.o: graphics.c graphics.h
    $(CC) $(CFLAGS) graphics.c -o graphics.o
```

BRANCHEMENTS CONDITIONNELS

Les branchements permettent de définir des variables ou des commandes différentes suivant une condition.

Par exemple:

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
    $(CC) -o foo $(objects) $(libs_for_gcc)
else
    $(CC) -o foo $(objects) $(normal_libs)
endif
```

source: [tutorialpoint](#)

On peut aussi utiliser `ifneq`.

AMÉLIORATION D'UN MAKEFILE (2)

```
CC = gcc
OBJS = main.o images.o gui.o graphics.o
CFLAGS = -g -Wall -c
BLASLIB ?= openblas

ifeq ($(BLASLIB), openblas)
    LFLAGS = -L /home/me/blas/openblas/lib -l openblas
else ifeq ($(BLASLIB), cblas)
    LFLAGS = -L /home/me/blas/cblas/lib -l cblas
else
    $(error unknown library $(BLASLIB))
endif

paint: $(OBJS)
    $(CC) $(OBJS) -o paint $(LFLAGS)

...
```

OBJECTIFS PHONY (BIDON)

Certain objectifs ne sont pas associés à des fichiers

```
.PHONY = clean install  # pas obligatoire mais evite le test d'existence de  
fichiers
```

```
clean:
```

```
    rm ./*.o ./paint
```

```
install:
```

```
    cp ./paint /usr/local/bin
```

QUESTIONS

Quelles sont les actions usuelles pour compiler les sources d'un programme téléchargé sous Unix ?

Pourquoi la commande «clean» précédente peut être dangereuse si mal utilisée ?

Refaire un graph de compilation et le makefile à partir de l'exemple listechainee et personnes.

DIRECTIVES DE PRECOMPILATION

INCLUDE

Inclure les entêtes (.h) de fonctions des librairies ou des fichier C :

- pour les entêtes de librairies standard (emplacements prédéfinis):

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

- pour les entêtes utilisateur (même emplacement que le fichier courant puis emplacements prédéfinis):

```
#include "listeChaine.h"
```

MACROS

On peut définir des macros et constantes de la manière suivante:

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

#define USUAL 10; //Déjà vu
#define WHY //on peut aussi déclarer des define sans valeur

//Définition de macro:
//Notez que ? et : sont utilisables dans le code
//pour remplacer les if
#define COMPUTE (INPUT / USUAL) //INPUT has to be defined
#define MIN(a,b) ((a) < (b)) ? (a) : (b))

int main()
{
    char string[] = STRING; //STRING doit être défini lors de la compilation
    printf("defined STRING: %s\n", string);
    printf("defined VALUE: %d\n", INPUT); //INPUT doit être défini lors de la compilation
    printf("Le plus petit nombre est: %d\n", MIN(INPUT, 5));
}
```



Lors de la compilation:

```
$ gcc precomp.c -o precomp -D STRING=\"Je suis bien content\" -D INPUT=5
```



QUESTION

Y a-t-il des cas ou une macro ne sera pas dans un .h ?



DIRECTIVES PRÉ-DÉFINIES

Des constantes sont prédéfinies par la plupart des compilateurs pour chaque fichier source:

- `__DATE__` string indiquant le jour du dernier passage du pré-compilateur;
- `__TIME__`, string indiquant la date du dernier passage du pré-compilateur;

Utile pour la gestion d'erreurs:

- `__LINE__`, numéro de ligne du fichier;
- `__FILE__`, nom du fichier .

Utile pour compiler du code différent suivant la plateforme:

- `_WIN32`, `_WIN64`, `__unix__`, `__APPLE__`, `__MACH__` définissent le type de système d'exploitation

COMPILATION CONDITIONELLE

La compilation conditionelle:

- Effectue une pré-compilation différente suivant les paramètres de pré-compilation;
- Compile un code différent suivant les paramètres de pré-compilation;

Utile pour:

- La compilation sur des plateformes différentes;
- L'inclusion infinie de .h;
- pour la création de constantes suivant les options de compilation;

```
#if TAILLES==0
#define TAILLE_TAMPON 512
#elif TAILLES==1
#define TAILLE_TAMPON 1024
#else
#define TAILLE_TAMPON 2048
#endif
```

```
#ifndef _win64
//Quelque chose pour les plateformes 64 bit
windows
#elif defined _win32
//Quelque chose pour les plateformes windows
#else
//Quelque chose pour les autres
#endif
```


COMPILATION CONDITIONELLE

precomp.h

```
#ifndef _PRECOMP_H //Evite l'inclusion infinie
#define _PRECOMP_H

//Define a constant with the system name
#ifdef __unix__
#define SYS "Unix"
#elif defined _WIN32
#define SYS "Windows"
#elif defined __APPLE__
#define SYS "Mac"
#else
#define SYS "Unknown"
#endif

/***** User interface*****/

/*****
Détermine quel est le caractère de séparation des répertoires
*****/
```

COMPILATION CONDITIONELLE

precomp.c

```
/******  
Détermine quel est le caractère de séparation des répertoires  
qui correspond au système d'exploitation sur lequel le programme  
as été compilé.  
IN: vide  
OUT: retourne un caractère représentant le séparateur de dossiers.  
retourne -1 si il n'as pas pu être identifié (système non identifié)  
*****/  
char getCharSep()  
{  
#ifdef __unix__  
    return '/';  
#elif defined _WIN32  
    return '\\';  
#elif defined __APPLE__  
    return '/';  
#else  
    return -1;  
#endif  
}
```

