

Introduction au langage C

Guillaume Chanel - guillaume.chanel@hesge.ch

Florent Glück - florent.gluck@hesge.ch

May 21, 2025

ISC - HEPIA

Généralités

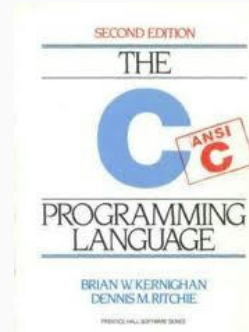
Un bref historique

Développé par Dennis Ritchie à Bell Labs entre 1969 et 1973 et est intimement lié au développement d'UNIX :

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



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



- Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", 2nd Edition, Prentice Hall, 1988
- Première édition: le standard "K&R-C "

Norme ANSI (1/3)

ANSI C est une norme qui garantie la **portabilité** :

- Défini les types manipulables et les conventions d'utilisation
- Défini les entêtes de fonctions et constantes de la librairies standard

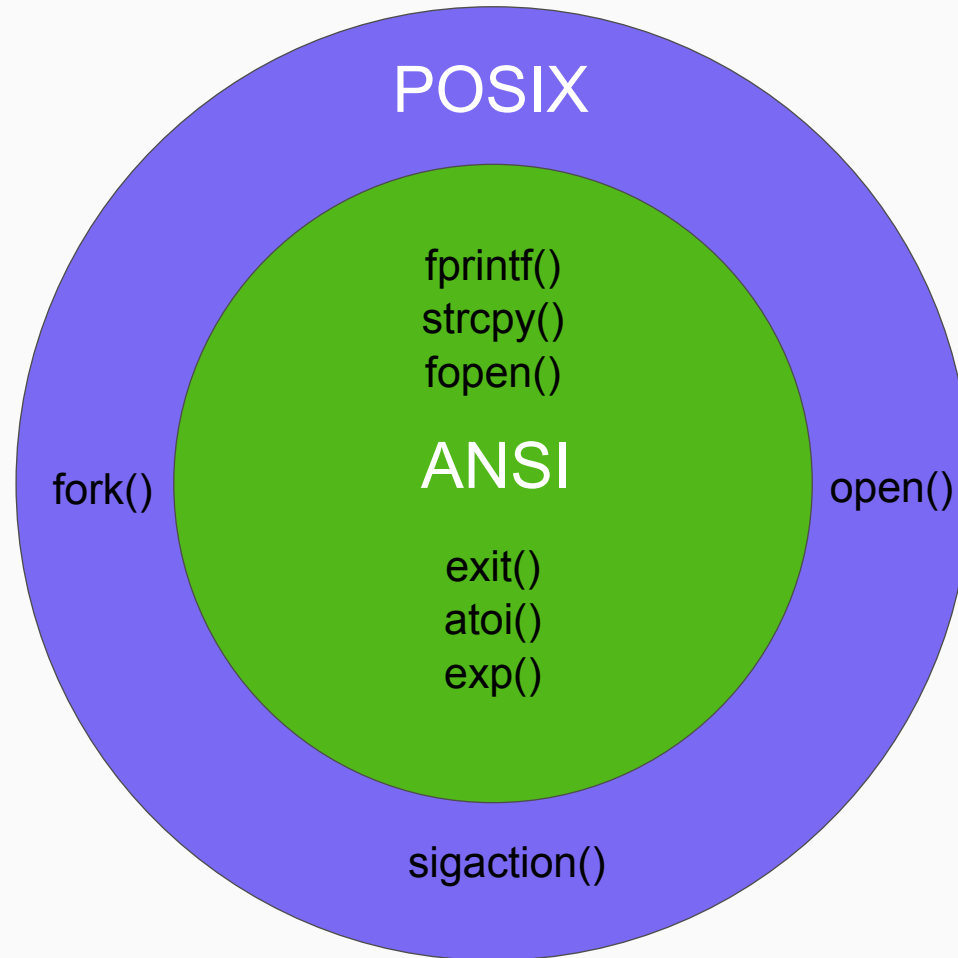
Exemples :

<code>math.h</code>	Fonctions mathématiques courantes (<code>cos</code> , <code>exp</code> , <code>pow</code> , etc.)
<code>stdio.h</code>	Fonctions d'entrée/sortie du système (<code>printf</code> , <code>scanf</code> , <code>getc</code> , etc.)
<code>stdlib.h</code>	Allocation et libération mémoire (<code>malloc</code> , <code>free</code> , etc.), contrôle de processus (<code>exit</code> , etc.), etc.

Norme ANSI (2/3)

Année	Nom(s)	Addition(s) principales
1989	C89	-
1990	C90 ISO-9899:1990	Mineurs : C90 \approx C89
1999	C99 ISO-9899:1999	Nouveaux types (complexes, booleen, etc.), fonctions inline, etc.
2011	C11 ISO-9899:2011	<i>Multi-threading</i> , unicode amélioré, etc.
2017	C17 ISO-9899:2018	Rien de significatif par rapport à C17, défaut de GCC est gnu17 (similaire à C17, mais pas identique)
2023	C23 ISO-9899:2024	Nouvelles fonctions lib standard, nouveaux types, mots-clés (true, false), directives préprocesseur, etc.

Norme ANSI (3/3)



Le C aujourd'hui

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

- Bibliothèques populaires :
 - GTK+ : interfaces graphiques (GUI)
 - Glib : listes chaînées, arbres, timer
 - Qt : GUI multi-plateformes
 - Boost : fonctions scientifiques, traitement signal, algèbre linéaire, etc.
 - SDL : rendu graphique, gestion souris, clavier, jeux, etc.

Depuis sa création, C est resté un langage populaire :

- Très utilisé dans l'embarqué et temps réel, applications système, de moins en moins utilisés pour les GUI
- Beaucoup de langages sont inspirés de C (C++, Go, Java, C#, Objective C, PHP)

Les principaux compilateurs C :

- **gcc** : natif sous Linux et utilisable sous Windows via MinGW
- **clang** : basé sur LLVM, natif sous Linux, en général messages d'erreurs plus clairs que gcc
- **Microsoft Visual Studio** : environnement de développement complet et intégré, uniquement pour Windows

Caractéristiques principales de C

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, C est proche du système et permet de réaliser des appels au noyaux directement
- Langage de **haut niveau** :
 - indépendant de la machine (tant qu'un compilateur C existe pour celle-ci)
 - beaucoup de types de données disponibles (tableaux, structures, etc.)
 - les normes ANSI et POSIX donnent accès à des fonctions de plus haut niveau

Caractéristiques principales

Inconvénients

- Inadapté au développement occasionnel ou pour le prototypage/test rapide d'algorithmes
- L'efficacité peut être au dépend de la compréhension → **commentez clairement les parties de code potentiellement peu compréhensibles !**
- Langage sans garde-fou : indices de tableaux non contrôlés, 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 programme buggé peut **sembler** fonctionner correctement !

Premier programme en C

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

#define PI 3.14159265359

double surface(float x) {
    return x * x * PI;
}

// La fonction main est toujours la premiere fonction
// d'un programme C a etre appelee lors de l'execution
int main() {
    int input, i;
    // la fonction printf affiche quelque chose
    printf("Veuillez entrer une valeur: ");
    scanf("%d", &input); // la fonction scanf lit l'entree utilisateur
    for (i = 1; i < input; i++) {
        float per = surface(i);
        printf("Le resultat pour %d est: %f\n", i, per);
    }

    return 0;
}
```

Manipulations de variables

Les directives de pré-compilation (version courte)

Les directives de pré-compilation :

- Sont interprétées par le préprocesseur `cpp`
- Sont remplacées par leur évaluation **avant** la compilation du programme
- Sont toujours précédées du caractère `#`
- 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
```

Types et déclaration de variables (1/2)

Type	Base de déclaration
Entier	<code>int</code>
Virgule flottante simple	<code>float</code> , stocké sur 32 bits
Virgule flottante double	<code>double</code> , stocké sur 64 bits (plus de valeurs possibles)
Caractère	<code>char</code> , un petit entier (généralement 8 bits)
Vide	<code>void</code> , absence de valeur, utile pour les fonctions sans retour et sans paramètres ainsi que pour les pointeurs sur des types inconnus (programmation générique)

Types et déclaration de variables (2/2)

Il est possible de combiner ces types avec des mots-clés :

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

Types : exemples

```
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; // declaration + initialisation
```


Attention ! Il n'y a pas de type booléen en C89/90

- Depuis C99, on peut utiliser :

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

De 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 de l'architecture cible et du compilateur utilisé !

- Pour connaître la taille d'une variable :
 - utiliser l'opérateur `sizeof(t)` où `t` peut être un type ou un nom de variable
 - les fichiers *header* `limits.h` et `float.h` donnent les limites des types :
 - `CHAR_MIN` : valeur minimum d'un `char`
 - `UINT_MAX` : valeur maximum d'un `int`
 - en C99, `stdint.h` contient des types avec tailles explicites :
 - `int8_t`, `int16_t`, `int32_t`
 - `uint8_t`, `uint16_t`, `uint32_t`

Règles sur les variables : noms

Règles sur les noms de variables :

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



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

Règles sur les variables : portée

Portée d'une variable :

- Déclarée dans un bloc entre deux accolades `{ }`, une variable n'est accessible que dans ce bloc
- Déclarée hors de toute accolade (et hors de toute fonction), une variable est globale, donc accessible de partout → à éviter

Opérateurs 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)

Quiz

Étant donné la définition 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 opérateurs est :

- l'ordre habituel pour les opérateurs 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” sera choisi

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


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
```

Quiz

Quel est le problème avec les calculs suivants ?

```
int i; long int l; float f; char c; unsigned char uc;  
  
c = -100;  
uc = c;  
  
c = 'è'; // = 232 (code ascii étendu IBM Code page 437)  
uc = c;  
  
l = 320254468;  
f = 45879651324476.5;  
i = l * f;
```

Contrôle du flot d'exécution

Structure conditionnelles

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, 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, 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-clés :

- `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 (1/2)

Une fonction est toujours déclarée avant son utilisation :

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

Le mot-clé **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 (2/2)

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 modules et librairies

```
float pourcentage(int , int); // Déclaration de l'entête

int main() {
    int a,b;
    ...
    // Utilisation de la fonction sans implémentation connue
    pourcentage(a,b)
    return 0;
}

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

Passage par adresse

- En C les paramètres sont **toujours passés par valeur** (aussi appelé “par copie” car pas de modification des variables passées en argument)
- Il n’y a pas de passage par référence ; pour palier à ce manque, on recourt à passer l’adresse comme valeur :
 - 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 avec le symbole **&**

Passage par adresse : exemples

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

// Utilisation de la fonction

```
int main() {  
    int val = 30; ret;  
    if ( (ret = versPourcentage(&val, 100)) < 0 )  
        return ret;  
    else  
        return val;  
}
```

Les tableaux et chaînes 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 tableauDEntiers[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;
```

Attention !

Attention aux indices des tableaux !

Hors des **limites** d'un tableau, le comportement est **indéfini** :

- Au mieux, tout fonctionne correctement
 - ce qui ne signifie pas que le programme n'est pas buggé !
- Éventuellement un **Segmentation Fault**
- Au pire, la valeur d'une autre variable est modifiée et on peut obtenir des erreurs importantes

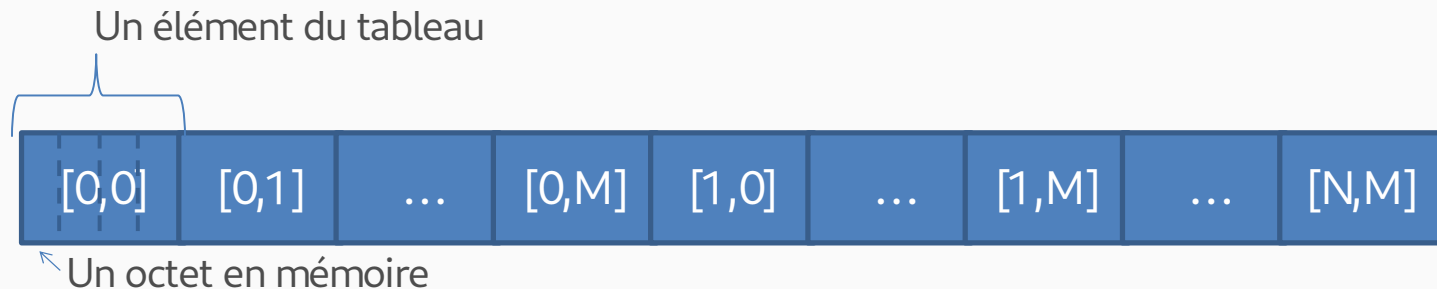
Tableau à N dimensions

```
// 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 !



Chaînes de caractères

- Le type chaîne de caractères (*string*) **n'existe pas 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 caractère `'\0'` est ignoré
 - si une chaîne de caractères ne contient pas de caractère `'\0'`, celle-ci est donc **sans fin** (dépassement de la capacité du tableau garantie)

Manipulation des chaînes

```
char machaine[6] = "Cool!"; // '\0' ajouté en fin de chaîne (pensez à réserver de la place)
char machaine2[] = "Cool!"; // Alloue automatiquement la bonne taille (avec le '\0' terminal)
printf("%s\n", machaine2); // Affiche la chaîne
machaine[1] = '\0'; // '' pour un caractère et "" pour une chaîne
printf("%s\n", machaine); // Affiche "C"
```

- Fonctions de manipulation de chaînes dans la librairie string (inclure `string.h`), e.g.:
 - `strlen(str)`, retourne la taille de la chaîne sans compter le `'\0'`
 - `strncpy(dest, src, n)`, copie `n` caractères de la chaîne `src` vers `dest`
 - `strncmp(str1, str2, n)`, retourne 0 si les deux chaînes sont identiques, parcourt au max `n` caractères de `str1` ou `str2`
 - `strncat(dest, src, n)`, ajoute `src` à la fin de `dest` (enlève le `'\0'`), copie au max `n` caractères de `src`
 - **ÉVITEZ** les versions des fonctions **sans 'n'**: `strcpy`, `strcmp`, `strcat`, etc.

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

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

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

Entrée / sortie console (2/4)

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

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

- `texteEtFormat` : une chaîne de caractères comme pour `printf`
 - Attention : taper du texte ici indique que ce texte doit être saisi pas l'utilisateur et non pas que le texte sera affiché sur stdout
- `*variable1`, `*variable2` : adresses des variables où seront stockées les valeurs entrées par l'utilisateur
 - pour obtenir l'adresse d'une variable, on utilise le symbole `&`

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

Entrée / sortie console (3/4)

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

Entrée / sortie console (4/4)

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

Symboles	Affichage
\n	saut de ligne
\r	retour à la ligne
\t	tabulation
\\	backslash
\' \"	simple ou double quote
\0	NUL character, utilisé pour indiquer la fin d'une chaîne de caractères

fonctions intéressantes pour les entrées sorties (1/2)

```
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 argument sur STDOUT
- retourne le caractère si pas d'erreur, EOF sinon

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

- écrit la chaine de caractères string sur STDOUT
- retourne EOF en cas d'erreur

fonctions intéressantes pour les entrées sorties (2/2)

- N'utilisez **jamais** la fonction **gets** !
- Utilisez **fgets** à la place :

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

- lit une chaîne de caractères 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

Exemple d'entrées / sorties

```
#include <stdio.h>

int main() { // à l'exécution, main est toujours la première fonction du programme à être exécutée
    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);

    do { // Boucle tant que l'utilisateur n'appuie pas sur 'enter'
        // Entrée de l'utilisateur pour un caractère avec effacement du buffer d'entrée clavier
        printf("Veuillez entrer un caractere: ");
        while ((inputChar = getchar()) != '\n' && inputChar != EOF);
        inputChar = getchar(); // possibilité d'utiliser scanf("%c", &inputChar)
        // Si le caractère est valide, l'afficher sous forme d'entier non signé
        if (inputChar != '\n')
            printf("Le code ASCII de %c est %u\n", inputChar, inputChar);
    } while(inputChar != '\n');

    return 0;
}
```

Quiz: pourquoi ce code est-il boggé ?

```
// Compiler avec: gcc -O0 -fno-stack-protector
#include <stdio.h>
#define TAILLE_MAX    3
void affiche_chaine(char *s, int len) {
    for (int i = 0; i < len; i++) {
        printf("Chaine[%d]:\tAdresse: 0x%x\tValeur: %d\n", i, s+i, *(s+i));
    }
}
int main() {
    int val1 = 1;
    char chaine[TAILLE_MAX] = {'a', 'b', '\0'};
    int val2 = 2;

    // Affiche l'état de la mémoire pour les variable ci-dessus
    printf("Valeur1:\tAdresse: %x\tValeur:%d\n", &val1, val1);
    affiche_chaine(chaine, TAILLE_MAX);
    printf("Valeur2:\tAdresse: %x\tValeur:%d\n", &val2, val2);
    gets(chaine);    // NE JAMAIS UTILISER GETS !

    // Affiche l'état de la mémoire pour les variable ci-dessus
    printf("Valeur1:\tAdresse: %x\tValeur:%d\n", &val1, val1);
    affiche_chaine(chaine, TAILLE_MAX);
    printf("Valeur2:\tAdresse: %x\tValeur:%d\n", &val2, val2);
    return 0;
}
```

Exécution de programme

La fonction main

- La fonction `main` est le **point d'entrée** du programme
- Elle possède l'entête suivante :

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

- `argc` : nombre d'arguments passés au programme
- `argv` : un tableau de chaînes de caractères contenant les arguments

La valeur de l'entier retourné indique si le programme s'est bien déroulé :

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

La fonction `exit`

La fonction `exit` termine le programme courant en renvoyant le code spécifié en paramètre :

```
void exit(int status);
```

- Dans un shell, `$?` contient le code de retour de la dernière commande exécutée

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'environnements

- La variable globale `environ` est un tableau de chaînes de caractères permettant d'accéder aux variables d'environnement
- Manipuler `environ` est déconseillé, on préférera obtenir la valeur d'une variable d'environnement avec :

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

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

Ajouter / modifier une variable d'environnement

Pour ajouter ou modifier une variable d'environnement :

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

- Retourne 0 en case de succès
- si `string` est de la forme `variable=value` et que `variable` :
 - n'existe pas, alors elle est ajoutée à l'environnement
 - existe, alors sa valeur est mise à jour

Attention : si `string` est modifiée ultérieurement, l'environnement le sera également !

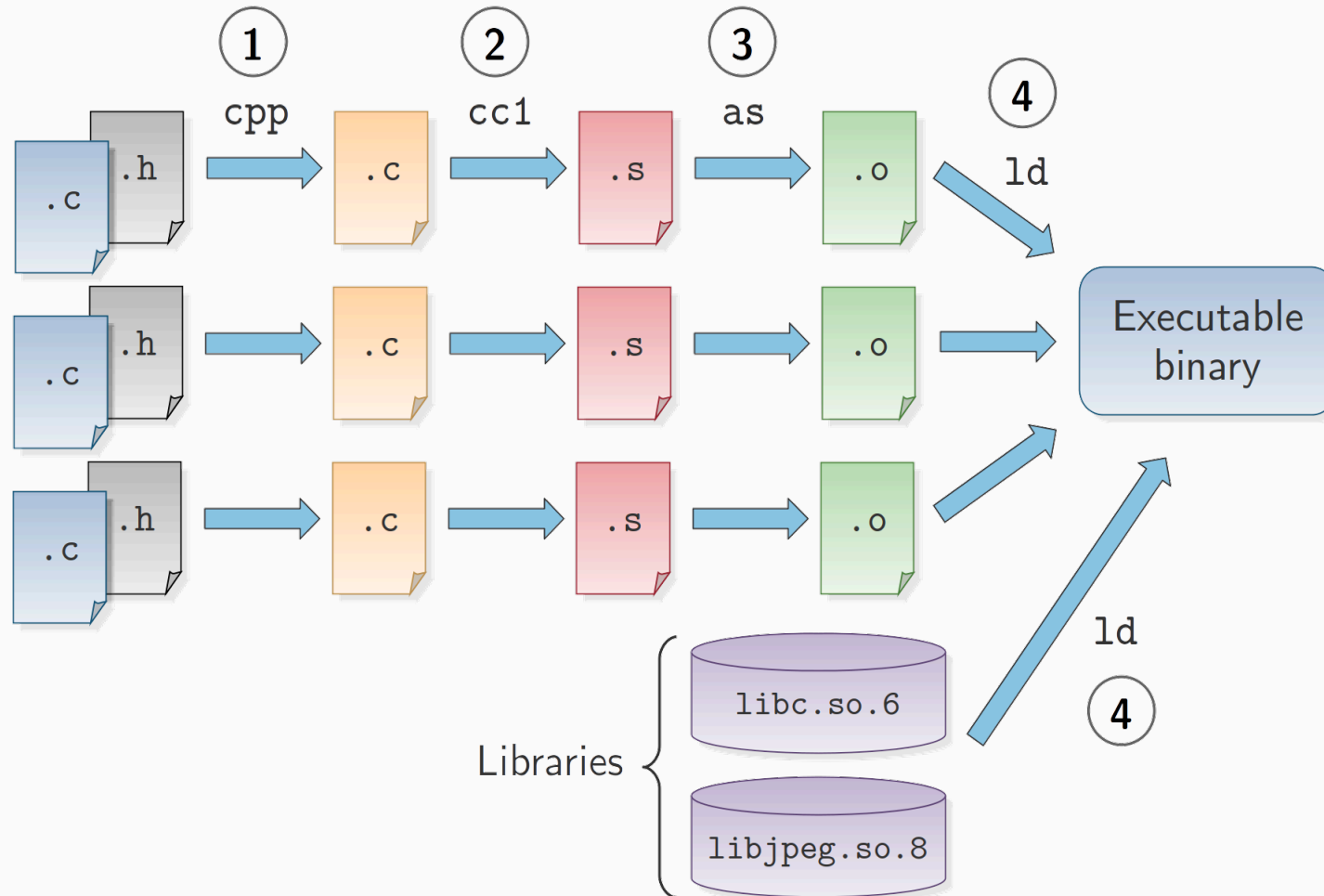
Génération d'un exécutable

```
gcc hello.c -o hello
```

Générer `hello` ci-dessus à partir de `hello.c` implique les 4 étapes **implicites** :

1. `gcc` exécute `cpp`, le **préprocesseur**, qui effectue de la substitution de texte et génère du code C (texte) → *substitution*
2. `gcc` exécute `cc1`, le **compilateur C**, qui compile le code C en code assembleur (texte) → *compilation*
3. `gcc` exécute `as`, l'**assembleur**, qui compile le code assembleur en code **objet** (code machine avec des références à des symboles : variables, fonctions, etc.) → *compilation*
4. `gcc` exécute `ld`, l'**éditeur de liens (*linker*)**, qui lie le code objet avec les librairies (et potentiellement d'autres codes objets) pour produire le binaire exécutable final → *édition des liens (*linking*)*

Génération d'un exécutable



Générer un programme exécutable (1/2)

- `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
<code>-Wall</code>	Affiche tous les <i>warning</i> possibles
<code>-I dir</code>	Inclue le répertoire <code>dir</code> pour la recherche de fichiers <code>.h</code>
<code>-g</code>	Génère les informations symboliques pour le débogage

Générer un programme exécutable (2/2)

Exemple : générer l'exécutable `clientsvn` en le liant à la librairie `svn` :

```
gcc monClientSVN.c -o clientsvn -lsvn
```

Veillez à toujours placer les options `ld` à la fin de la ligne de compilation

Option	Effet
--------	-------

<code>-Ldir</code>	Inclue le répertoire <code>dir</code> pour la recherche de librairies
--------------------	---

<code>-lnom</code>	Inclue la librairie <code>libnom</code> (ne jamais indiquer le préfixe <code>lib</code>)
--------------------	---

Les libraires portent le nom `libnom` et se trouvent généralement dans :

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

Débogger un programme

Un programme, compilé avec les symboles de déboggage (option `-g`), peut être déboggué à l'aide de `gdb`, le débogger GNU :

```
gdb monProg
```

Voici quelques commande utiles de `gdb` (*help*):

<code>run</code>	exécute le programme jusqu'à un crash ou sa terminaison naturelle
<code>list</code>	liste les 10 lignes de code autour du point actuel
<code>break p</code>	positionne un <i>breakpoint</i> à la ligne ou la fonction <code>p</code>
<code>clear p</code>	supprime un breakpoint
<code>cont</code>	continue l'exécution du code
<code>step</code>	exécute la prochaine ligne de code
<code>print p</code>	affiche la valeur courante de la variable <code>p</code>
<code>info p</code>	information sur beaucoup de choses (<code>info locals</code> , <code>info</code>)
<code>quit</code>	quitte <code>gdb</code>

Débogger un programme ayant crashé

- Lorsque qu'un programme *crash* dû à un bug, l'OS peut générer un *coredump*
- Un *coredump* est une image de la mémoire du processus (donc son état) au moment du crash
- On peut débogger un fichier *coredump* avec **gdb** :

```
gdb monexecutable -c core
```

Grâce à **gdb** on peut observer :

- la ligne de code responsable du *crash*
- les valeurs des variables, des arguments des fonctions, de la pile, etc.

Les pointeurs

Concept

```
char lettre = 'C';  
char *ptrPile;  
char *ptrTas;  
ptrPile = &lettre;  
ptrTas = malloc(1024);  
strcpy(ptrTas, "HAL")
```

Déclaration

Valeur
(contenu)

Adresse

Valeur
(contenu)

Adresse

char
*ptrPile

char
*ptrTas

ptrTas

lettre

0x455C

0xAF51

...

'C'

0x4550

0x4554

0x455C

'H'

'A'

'L'

'\0'

...

0xAF51

Equivalence pointeurs ↔ tableaux

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

```
#define TAILLE_TAB 20
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]);
}
```

Différence pointeurs ↔ tableaux

Attention, il reste des différences entre pointeurs et tableaux :

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

Quiz 1

1. Si `intTab` du code précédent est passé en argument à une fonction, dans la fonction quelle sera la taille de :

```
sizeof(intTab)
```

2. Soit le code :

```
int *t = malloc(64);
```

- comment écrire `t[7] = 42;` sous forme de pointeur ?
 - comment écrire `int n = t[0];` sous forme de pointeur ?
3. Comment peut-on écrire l'expression `p[i]` sous forme de pointeur ?
4. Quelle est l'équivalence de `tab[i][j]` sous forme de pointeur ?

Quiz 2

Soit le code :

```
int n = 4;  
int *t = &n;
```

1. Que représente `&n` ?
2. Que représente `t` ?
3. Que représente `*t` ?
4. Que représente `&t` ?
5. Quelle est la valeur de `t` ?
6. Quelle est la valeur de `*t` ?
7. Quelle est la valeur de `&t` ?
8. Quelle est la valeur de `*(&t)` ?

Quiz 3

1. Soit le code :

```
char t[] = {1,0,0,0,2,0,0,0,9,10,11,12,13,14,15,16,17,0,0,0};  
char *a = t;  
int *b = (int *)t;  
void main() {  
    printf("%d %d %d %d %d %d", a[0], b[0], a[1], b[1], a[4], b[4]);  
}
```

- Sans exécuter ce programme, quelles seront les valeurs affichées ?

2. Quelle est la différence de taille entre :

```
sizeof(char *)
```

et :

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

Allocation dynamique : principe

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 est réalisée dans une zone de la mémoire du processus appelée **tas** (*heap*)

Allocation dynamique : comment ?

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

- `malloc(size)` : retourne un pointeur sur une zone de mémoire allouée de taille `size` en bytes
 - retourne NULL en cas d'échec
- `calloc(count, size)` : retourne un pointeur sur une zone allouée de `count` éléments, chacun de taille `size` (en bytes), le tout initialisé à 0
 - retourne NULL en cas d'échec
- `free(ptr)` : libère l'espace mémoire pointé par `ptr`
 - attention : ne remet pas `ptr` à NULL !

Pointeurs et const (1/2)

man strcmp :

NAME

strcmp, strncmp - compare two strings

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <string.h>
```

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char s1[.n], const char s2[.n], size_t n);
```

Pourquoi le mot-clé `const` devant la déclaration `char *` ?

Pointeurs et const (2/2)

Quelle sont les différences entre :

```
int n = 42;  
const int *p = &n;           // (1)  
int const *p = &n;           // (2)  
int *const p = &n;           // (3)  
const int *const p = &n;     // (4)
```

Pointeurs et const (2/2)

Quelle sont les différences entre :

```
int n = 42;  
const int *p = &n;           // (1)  
int const *p = &n;           // (2)  
int *const p = &n;           // (3)  
const int *const p = &n;     // (4)
```

1. La valeur pointée par `p`, `*p` est constante et ne peut pas être modifiée : p.ex. `*p = 7` provoquera une erreur de compilation. Par contre, `p` peut-être modifié : p.ex. `p = &v` ou `p += 1`

Pointeurs et const (2/2)

Quelle sont les différences entre :

```
int n = 42;  
const int *p = &n;           // (1)  
int const *p = &n;           // (2)  
int *const p = &n;           // (3)  
const int *const p = &n;     // (4)
```

1. La valeur pointée par `p`, `*p` est constante et ne peut pas être modifiée : p.ex. `*p = 7` provoquera une erreur de compilation. Par contre, `p` peut-être modifié : p.ex. `p = &v` ou `p += 1`
2. Une autre manière d'écrire (1), donc équivalent à 1.

Pointeurs et const (2/2)

Quelle sont les différences entre :

```
int n = 42;  
const int *p = &n;           // (1)  
int const *p = &n;           // (2)  
int *const p = &n;           // (3)  
const int *const p = &n;     // (4)
```

1. La valeur pointée par `p`, `*p` est constante et ne peut pas être modifiée : p.ex. `*p = 7` provoquera une erreur de compilation. Par contre, `p` peut-être modifié : p.ex. `p = &v` ou `p += 1`
2. Une autre manière d'écrire (1), donc équivalent à 1.
3. Le pointeur `p` est constant et ne peut pas être modifié, donc p.ex. `p += 1` provoquera une erreur de compilation. Par contre, la valeur pointée par `p` peut être modifiée ; p.ex. : `*p = 10`

Pointeurs et const (2/2)

Quelle sont les différences entre :

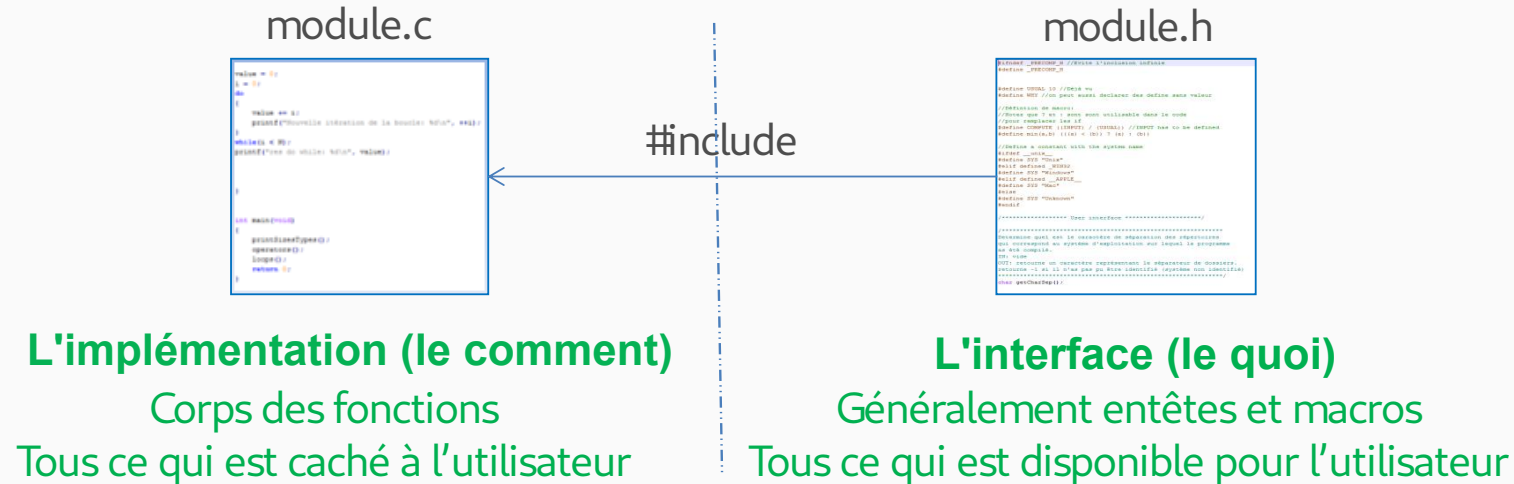
```
int n = 42;  
const int *p = &n;           // (1)  
int const *p = &n;           // (2)  
int *const p = &n;           // (3)  
const int *const p = &n;     // (4)
```

1. La valeur pointée par `p`, `*p` est constante et ne peut pas être modifiée : p.ex. `*p = 7` provoquera une erreur de compilation. Par contre, `p` peut-être modifié : p.ex. `p = &v` ou `p += 1`
2. Une autre manière d'écrire (1), donc équivalent à 1.
3. Le pointeur `p` est constant et ne peut pas être modifié, donc p.ex. `p += 1` provoquera une erreur de compilation. Par contre, la valeur pointée par `p` peut être modifiée ; p.ex. : `*p = 10`
4. Le pointeur `p` ainsi que la valeur pointée par `p`, `*p` sont tous deux constants ! Donc, ni l'un ni l'autre ne peuvent être modifiés

Organisation d'un programme

Modules

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

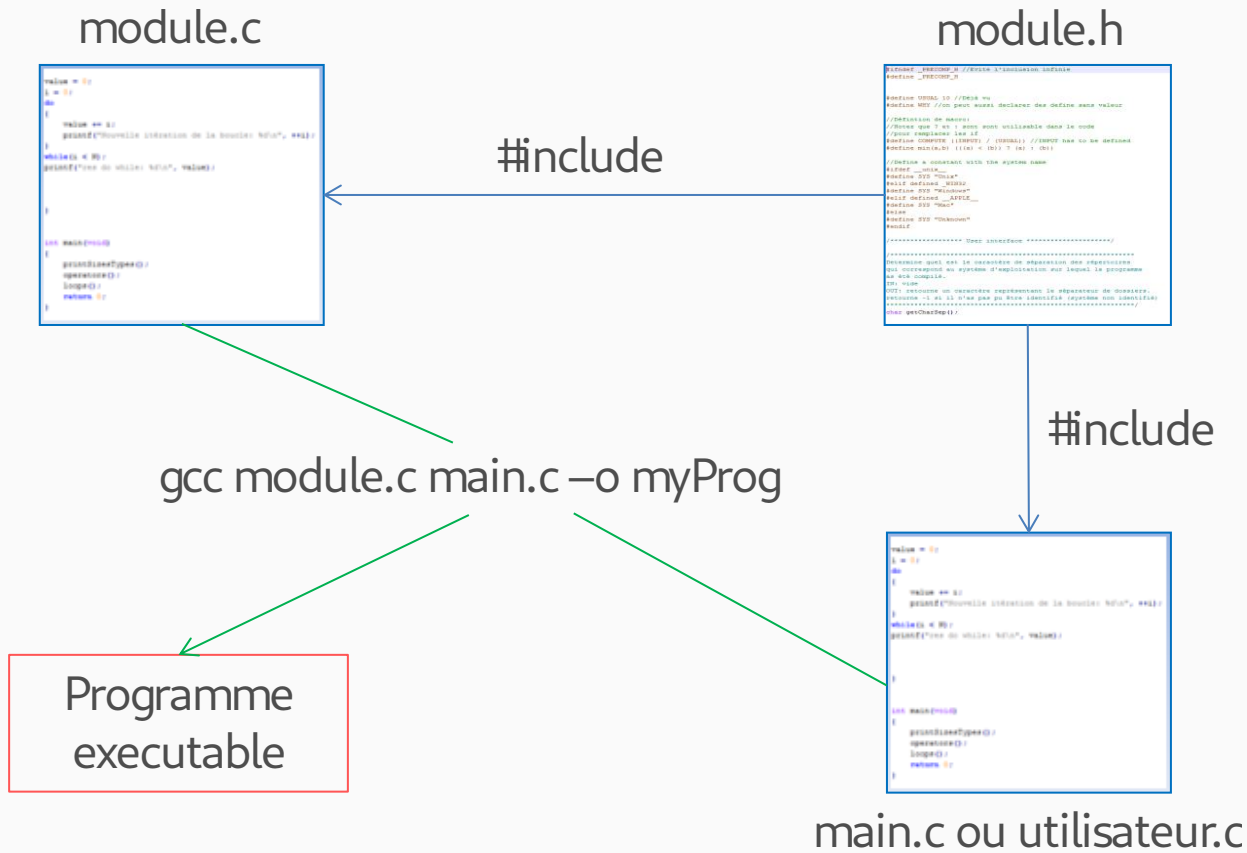


Cela permet notamment de :

- Créer une forme d'encapsulation qui facilite le déboguage 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)

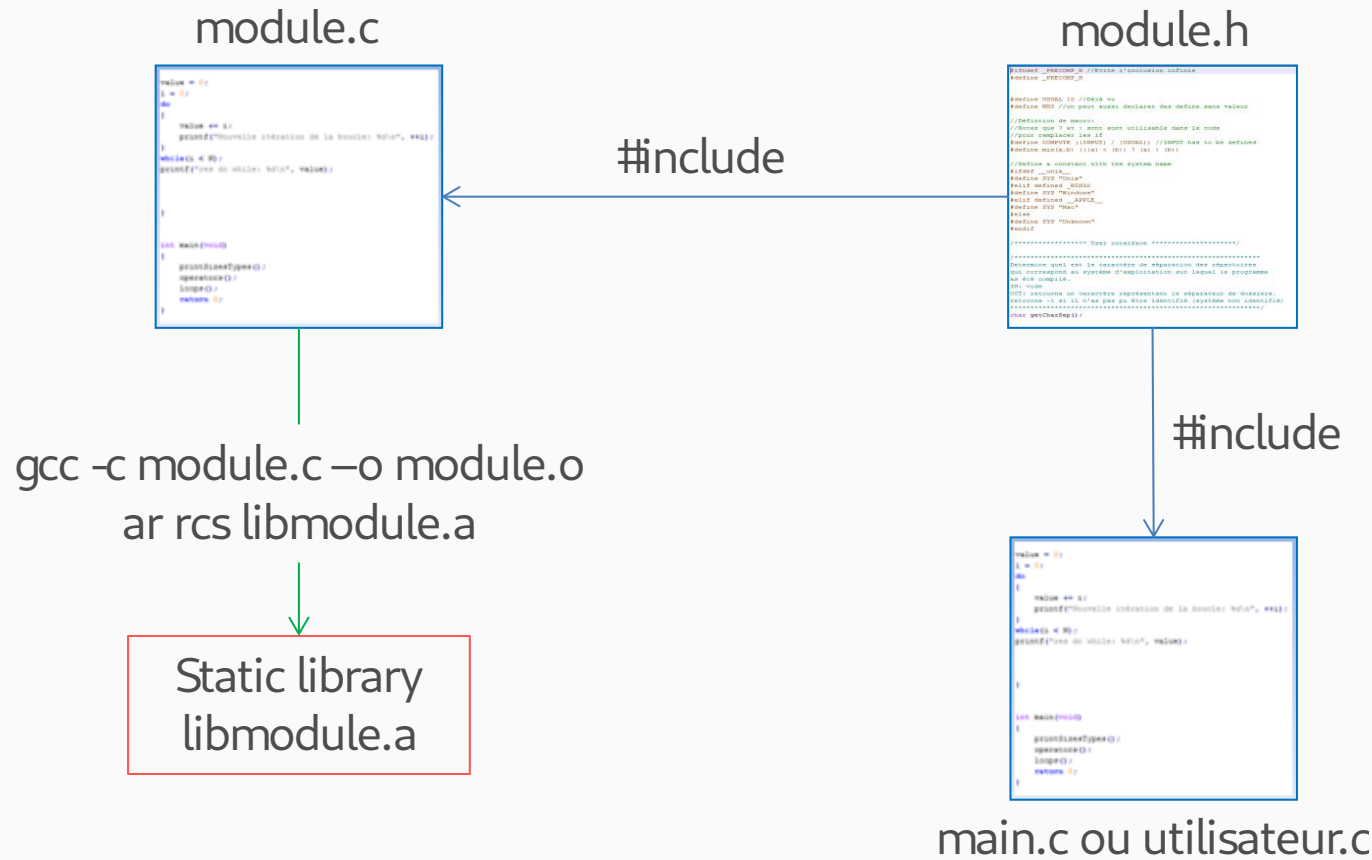
Compilation de modules

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



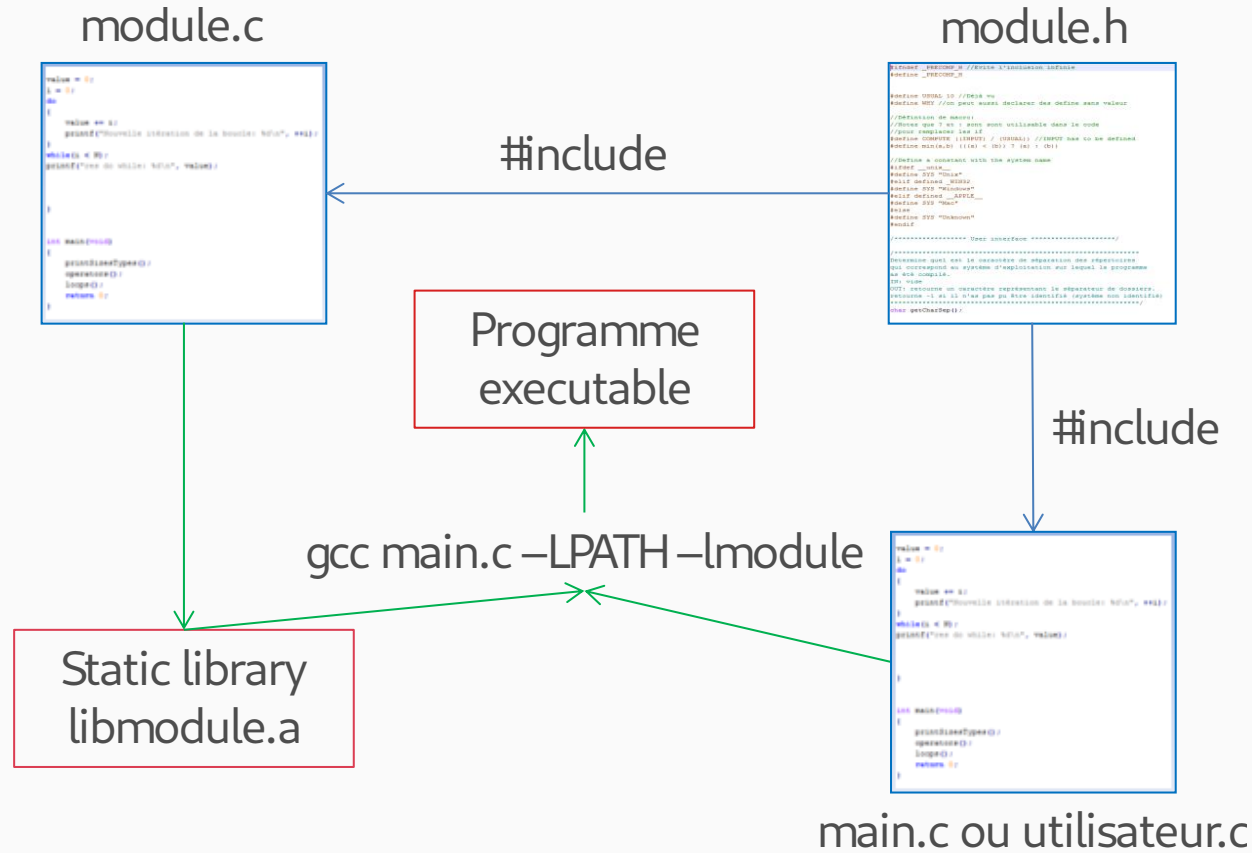
Compilation de librairie

On peut créer une librairie (statique dans cet exemple) et l'utiliser comme module :



Utiliser une librairie

Pour qu'un programme utilise une librairie, il faut la **lier** (*linking*) à celui-ci :



Type opaque

- Un type opaque est une structure de données (i.e. `typedef`) qui n'est pas définie dans l'interface (i.e. fichier **header**)
- Permet de :
 - cacher les détails l'implémentation → abstraction
 - modifier la structure de données sans modifier le comportement du code l'utilisant

listChaine.c

```
#include "listeChaine.h"

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

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

listChaine.h

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

listeChaine initListeChaineVide();

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

Fonction déclarée static

Le mot-clé `static` permet de :

- Déclarer une fonction comme **locale** (aussi appelée **privée**) à un module (i.e. elle n'est pas utilisable dans un autre module)
- Rendre le nom de cette fonction utilisable dans d'autres modules

main.c

```
#include <stdio.h>

void show() { // PAS DE CONFLIT !
    printf("Main show\n");
}

void main() {
    show();
    interface();
}
```

interface.c

```
#include <stdio.h>

static void show() { //PAS DE CONFLIT !
    printf("Interface show\n");
}

void interface() {
    show();
}
```

Variable déclarée `static`

Un variable **globale** déclarée `static` signifie que :

- La variable est **globale**, mais sa visibilité est **privée** au module courant
 - elle n'est pas utilisable (ni visible) dans un autre module
-

Un variable **locale** déclarée `static` signifie que :

- La variable possède une durée de vie **globale**
 - sa valeur **persiste** à travers les appels à la fonction
 - sa valeur **persiste** durant toute l'exécution du programme
- La variable est uniquement accessible dans la fonction où elle est déclarée

Variable déclarée static : exemple

```
int counter;           // Accessible dans d'autres modules
static int counter_priv; // Seulement accessible dans ce module

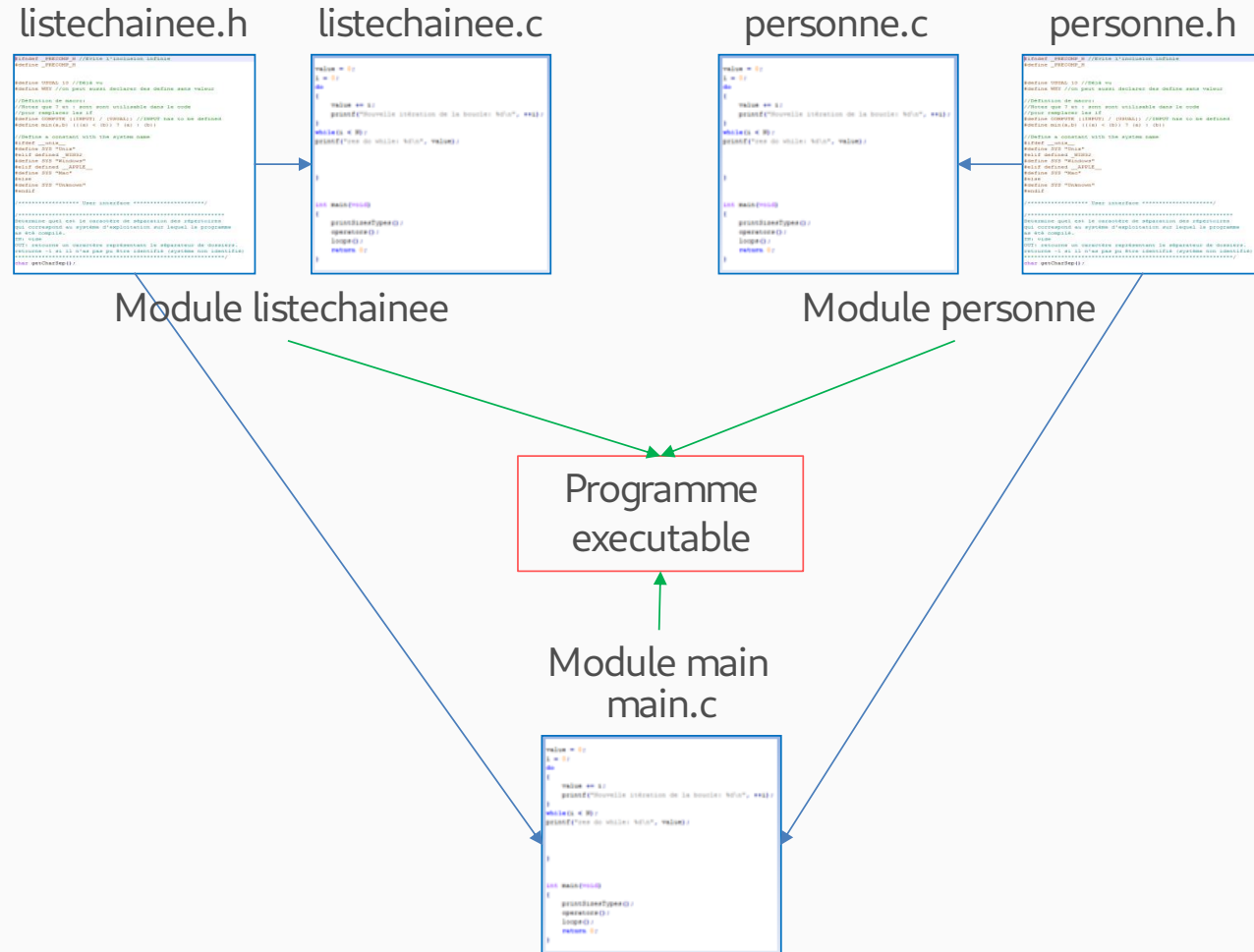
void f() {
    static int x = 0;    // Accessible dans f uniquement, mais durée de vie globale
    int y = 0;
    print("%d %d / ", x, y);
    x++;
    y++;
}

int main() {
    for (int i = 0; i < 4; i++) {
        f();
    }
    printf("\n");
    return 0;
}
```

Affichage :

```
0 0 / 1 0 / 2 0 / 3 0 /
```

Exemple d'organisation d'un programme

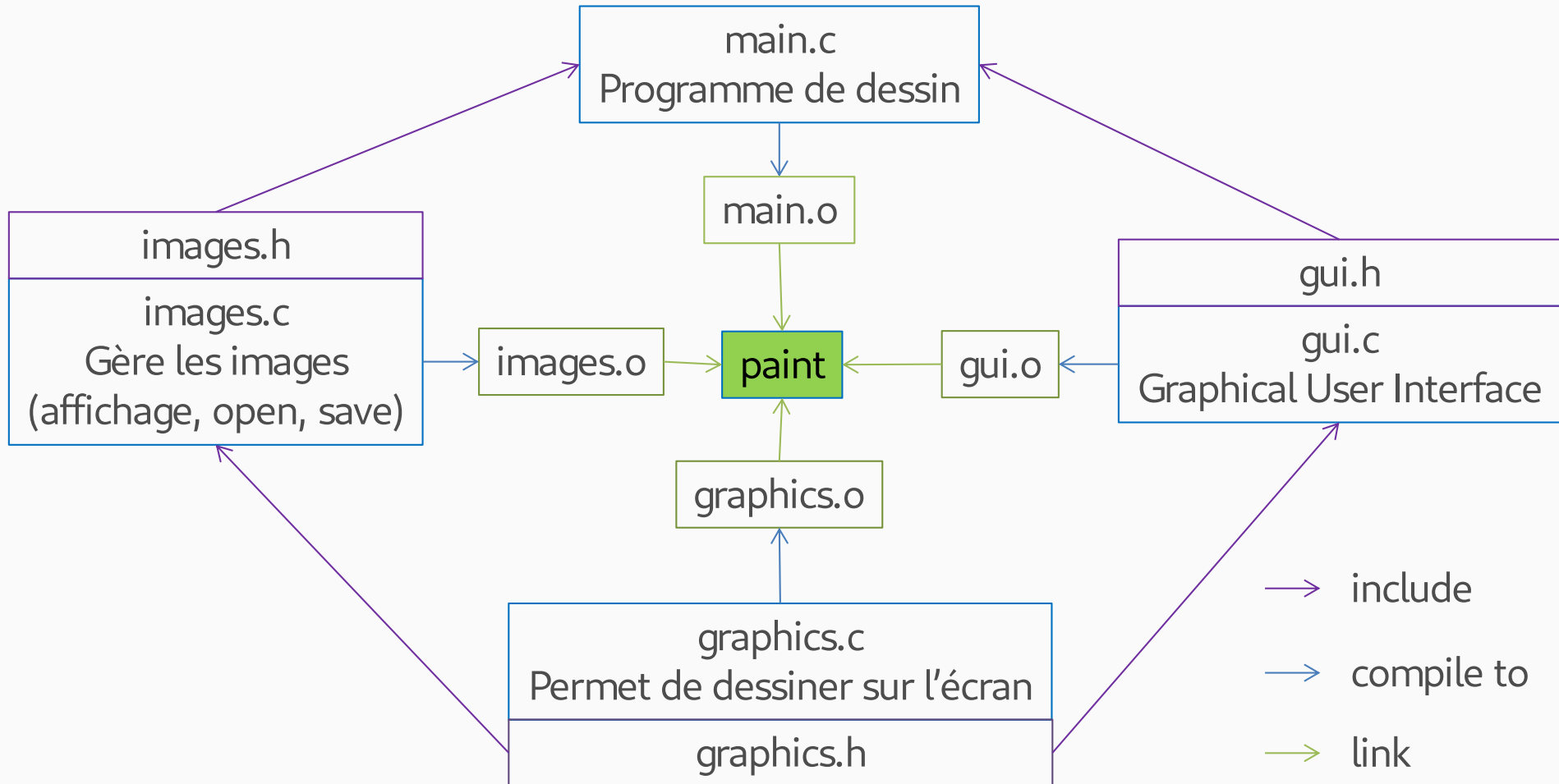


Makefiles

Complexité de compilation 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

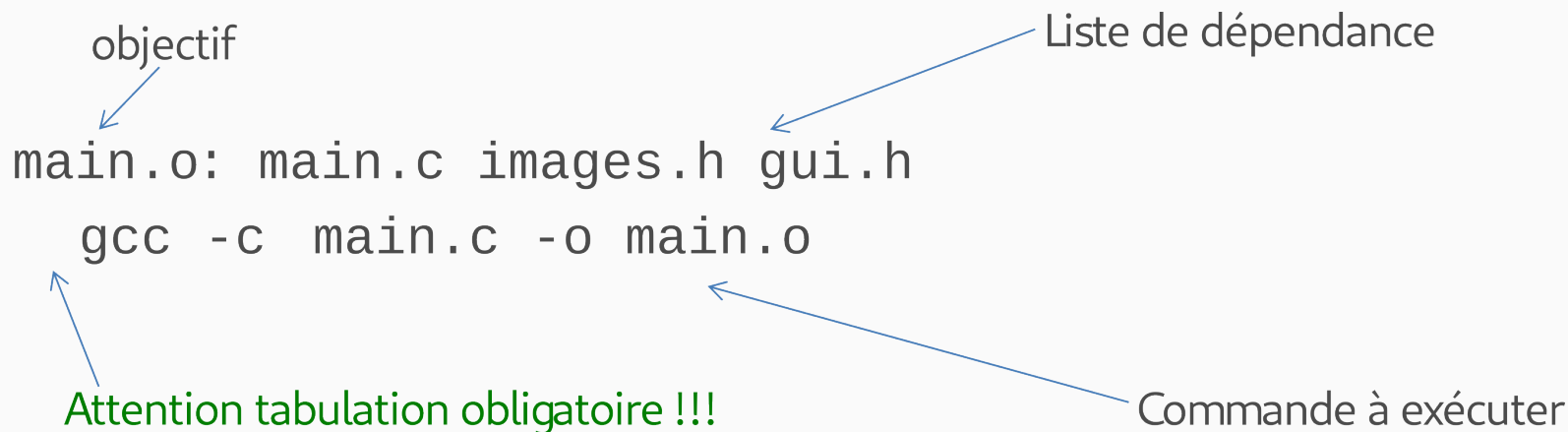
Complexité de compilation modulaire



Création d'un Makefile

Un Makefile est un fichier texte qui contient des **cibles** :

- Chaque cible représente (en général) un fichier .o ou le programme final à générer
- Chaque cible est associée à :
 - une liste de dépendances (.o, .c, .h)
 - une commande à exécuter si une des dépendances est plus récente que la cible



The diagram shows a Makefile entry with three annotations. The entry is:
main.o: main.c images.h gui.h
gcc -c main.c -o main.o
Annotations:
- 'objectif' points to 'main.o' in the first line.
- 'Liste de dépendance' points to 'main.c images.h gui.h' in the first line.
- 'Attention tabulation obligatoire !!!' points to the first tab character before 'gcc' in the second line.
- 'Commande à exécuter' points to 'gcc -c main.c -o main.o' in the second line.

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

objectif

Liste de dépendance

Attention tabulation obligatoire !!!

Commande à exécuter

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

- L'outil `make` permet de lire un Makefile et de créer la cible spécifiée par l'utilisateur
- Si aucune cible n'est spécifiée, alors la première est créée
- Pour créer la première cible :

```
make
```

- Pour créer la cible xyz :

```
make xyz
```

Variables

On peut ajouter des noms de variables pour effectuer des changements facilement :

```
VARIABLE = value
```

Pour utiliser le contenu d'une 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 à `make` :

```
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 conditionels

- 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 :

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

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 cible `clean` précédente peut être dangereuse si mal utilisée ?
- Refaire un graph de compilation et le Makefile à partir de l'exemple `listechainee` et `personnes`