Langage C

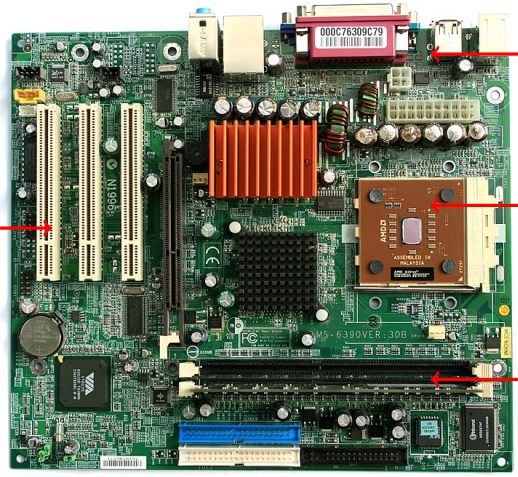
Nicolas Delestre

INSA
ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Sous le capot d'un PC



Ports (USB, HDMI, etc.)

Processeur

Bus

Mémoire vive

INSA ROUEN NORMANDIE

Processeur, microprocesseur

Définitions

- ≪ Un processeur est la partie d'un ordinateur qui exécute les instructions et traite les données des programmes. (Wikipédia)
- ≪ Un microprocesseur est un processeur dont tous les composants ont été suffisamment miniaturisés pour être regroupés dans un unique boîtier ≫ (Wikipédia)

Caractéristique d'un microprocesseur

- Le jeu d'instructions
- La complexité de son architecture (nb de cœurs, transitors)
- Le nombre de bits que le processeur peut traiter simultanément
- La vitesse de l'horloge

Un peu d'histoire

Évolutions des microprocesseurs

Année	Fabriquant	Nom	Nb instructions	Nb transistors	Fréquence	Nb bits
1971	Intel	4004	46	2 300	108 kHz	4/4
1974	Intel	8080	72	6 000	2 MHz	8/8
1979	Intel	8088	72	29 000	5 MHz	16/32
1979	Motorola	68000	82	68 000	2 à 12 MHz	16/32
1982	Intel	80286	99	134 000	6 à 16 MHz	16/16
1983	Acorn	ARM1	$\approx$ 50	25 000	$\leq$ 8 MHz	32/26
1985	Intel	80386	155	275 000	16 à 40	32/32
1992	IBM	PowerPC 601		2 800 000	50 à 120 MHz	32/32
1993	Intel	Pentium	168	3 100 000	60 à 233 MHz	32/64
1995	AMD	K5	168	4 300 000	75 à 100 MHz	32/64
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
2021	IBM	Power10		3 600 000 000	4 GHz	64
2022	Intel	i9 12900H	981	4 150 000 000	5 GHz	64
2022	Apple	M2 (ARMV8-5)	"232"	20 000 000 000	$\leq$ 3,4 GHz	64

Remarques

- Apple a souvent changé de microprocesseur pour ces Mac : 1984 Motorola, 1995 IBM, 2005 Intel, 2020 ARM
- Il est difficile de connaître le nombre de transistors dans les nouveaux microprocesseurs

Du code assembleur aux octets

Tout est octet

- Les programmes et les données sont représentés par des octets et stockés dans la mémoire vive
- Chaque instruction d'un microprocesseur est représentée par un ou plusieurs octets : codage
- Les octets représentants le programme se nomment le code objet
- Le langage assembleur est une représentation humainement compréhensible du code objet :
  - assembler : code assembleur → code objet
  - désassembler : code objet → code assembleur

Exemple pour x86

```
$ objdump -d helloworld.o

helloworld.o:     format de fichier elf64-x86-64


Déassemblage de la section .text :

0000000000000000 <main>:

   0: f3 0f 1e fa           endbr64
   4: 48 83 ec 08           sub    $0x8,%rsp
   8: 48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # f <main+0xf>
   f: e8 00 00 00 00        callq  14 <main+0x14>
  14: b8 00 00 00 00        mov    $0x0,%eax
  19: 48 83 c4 08           add    $0x8,%rsp
  1d: c3                    retq
```

Langage de plus haut niveau et compilation 1 / 3

Constats

- Le code assembleur est propre à chaque microprocesseur
- Il est très difficile d'écrire un code conséquent en assembleur

⇒ Nécessité de développer avec des langages de plus haut niveau qu'un programme va traduire en code assembleur / code objet

Compilation vs interprétation

- Traduction à la volée : langage interprété
- **Traduction complète du code : langage compilé**

```
# include <stdio.h>

int main(){
  printf("Hello world\n");
}
```



```
endbr64
sub    $0x8,%rsp
lea    0x0(%rip),%rdi
callq  14 <main+0x14>
mov    $0x0,%eax
add    $0x8,%rsp
retq
```



```
f3 0f 1e fa
48 83 ec 08
48 8d 3d 00 00 00 00
e8 00 00 00 00
b8 00 00 00 00
48 83 c4 08
c3
```

Langage de plus haut niveau et compilation 2 / 3

Gain en productivité

- Plus simple
- Abstraction
- Permet le changement de paradigme
- Des contrôles (sécurité) peuvent être ajoutés

⇒ Une instruction de haut niveau peut produire plusieurs instructions assembleur

Compilation = vérifications avant exécution

- Syntaxique : respect de la grammaire du langage
- Sémantique : respect du typage dans les opérations et les passages de paramètre

INSA ROUEN NORMANDIE

Langage de plus haut niveau et compilation 3 / 3

Portabilité

- Un code peut être compilé/interprété sur différents types de microprocesseurs (et de systèmes d'exploitation)

```c
# include <stdio.h>

int main(){
  printf("Hello world\n");
}
```

x86

```
endbr64
sub    $0x8,%rsp
lea    0x0(%rip),%rdi
callq  14 <main+0x14>
mov    $0x0,%eax
add    $0x8,%rsp
retq
```

→

```
f3 0f 1e fa
48 83 ec 08
48 8d 3d 00 00 00 00
e8 00 00 00 00
b8 00 00 00 00
48 83 c4 08
c3
```

ARM

```
stp    x29, x30, [sp, #-16]!
mov    x29, sp
adrp   x0, 0 <main>
add    x0, x0, #0x0
bl     0 <puts>
mov    w0, #0x0
ldp    x29, x30, [sp], #16
ret
```

→

```
a9 bf 7b fd
91 00 03 fd
90 00 00 00
91 00 00 00
94 00 00 00
52 80 00 00
a8 c1 7b fd
d6 5f 03 c0
```

Le langage C

Caractéristiques

- Langage compilé inventé en 1972 par Dennis Ritchie et Kenneth Thompson pour redévelopper le système d'exploitation UNIX
- Langage d'assez bas niveau parmi les langages de haut niveau
- Normalisé en 1989 (ANSI) et en 1990 (ISO, C90)
- Deux nouvelles versions de la norme ISO en 1999 (C99) et 2011 (C11)
- Langage du paradigme de la programmation structurée
- Langage à typage statique et typage faible
- Le cœur du C comprend peu d'instructions, mais il est accompagné de nombreuses bibliothèques standards

INSA ROUEN NORMANDIE

Le C un langage compilé

- Objectif : produire à partir d'un fichier `.c` (code source) un fichier `.o` (code objet)
- Le code source est composé de deux langages : macros (instructions commençant par un #) et code C

$$\text{code source avec macros} \xrightarrow{\textit{précompilation}} \text{code source sans macros} \xrightarrow{\textit{compilation}} \text{code objet}$$

- La `.o` n'est pas un exécutable !

Plusieurs compilateurs

- **GCC (GNU Compiler Collection)**
- intel
- Clang
- Microsoft Visual C++

`https://en.wikipedia.org/wiki/List_of_compilers#C_compilers`

GCC

Compilation d'un fichier C

```
$ ls
helloworld.c
$ gcc -c helloworld.c
$ ls
helloworld.c helloworld.o
```

Options à utiliser

- `-Wall` pour afficher tous les *warnings*
- `-pedantic` pour n'accepter que du code C ISO

```
$ gcc -c -pedantic -Wall helloworld.c
$ ls
helloworld.c helloworld.o
```

L'édition des liens

Constats sur l'exemple

- Le `.o` n'est pas un exécutable !
- Le cœur du langage C propose peu de fonctionnalités
- Le code correspondant à l'affichage d'une chaine de caractères (code de `printf`) n'est pas inclus dans le `.o`
- Il faut donc **lier** l'appel du sous programme `printf` du code objet de `helloworld.o` avec le code de ce sous programme, code qui se trouve dans une bibliothèque

$$\Rightarrow \text{L'édition des liens}$$

code source avec macros $\xrightarrow{\textit{précompilation}}$ code source sans macros $\xrightarrow{\textit{compilation}}$ code objet $\xrightarrow{\textit{édition des liens}}$ exécutable

Deux types de bibliothèques

Bibliothèque statique (`.a` sous Unix, `.lib` sous Windows)

- Le code objet de la bibliothèque est ajouté au code objet du programme et le lien entre l'appel du sous programme et le corps du sous programme est réalisé à l'édition des liens
- Avantage : chaque programme à sa version d'une bibliothèque
- Inconvénients : exécutable plus volumineux et une même version d'une bibliothèque peut être présente plusieurs fois en mémoire

Bibliothèque dynamique (`.so` sous Unix, `.dll` sous Windows)

- Le code objet de la bibliothèque est chargé en mémoire de l'ordinateur au lancement du programme (si ce n'est déjà fait) et le lien entre l'appel du sous programme et le corps du sous programme est réalisé à l'exécution
- Avantage : une bibliothèque utilisée par plusieurs programmes n'est présente qu'une seule fois en mémoire
- Inconvénients : gestion de plusieurs versions d'une bibliothèque et problème si la bibliothèque n'est pas pré installée

INSA ROUEN NORMANDIE

L'édition des liens avec gcc sous UNIX

Les bibliothèques sous UNIX

- Le nom du fichier commence par `lib`
- L'extension du fichier est `.a` ou `.so`

L'édition des liens avec gcc

- option `-l` suivi du nom de la bibliothèque (sans `lib`, sans l'extension)
- le code objet
- option `-o` pour préciser le nom de l'exécutable (par défaut `a.out`)

Création et exécution du programme helloworld

```
$ ls
helloworld.c helloworld.o
$ gcc -o helloworld -lc helloworld.o
```

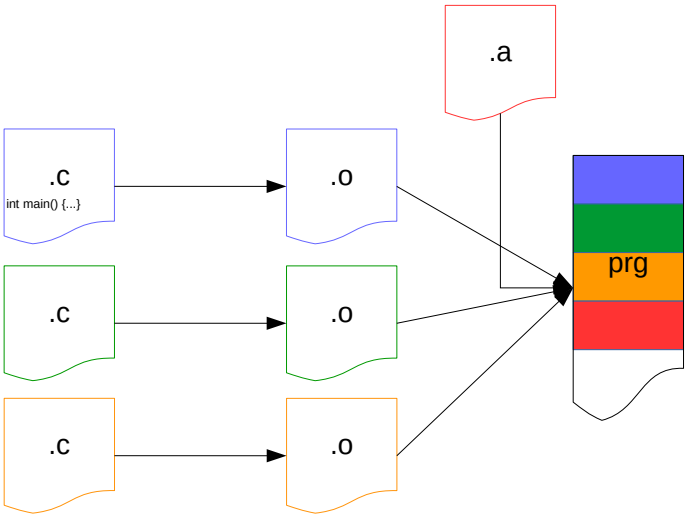
```
$ ls
helloworld helloworld.c helloworld.o
$ ./helloworld
Hello world
$
```

Principales bibliothèques standards

Fichiers	Entêtes	Fonctionalités
libc.so	assert.h	macro `assert` pour activer des préconditions en mode debug
	errno.h	codes d'erreur "renvoyés" par les fonctions standards
	limits.h	constantes bornes min et max des `int`
	stdargs.h	pour créer des fonctions à arité variable
	stdlib.h	transtypage, nombres aléatoires, allocations de mémoire, ...
	stdio.h	gestion des entrées/sorties
	...	...
libm.so	complex.h	manipulation des nombres complexes
	math.h	fonctions mathématiques (trigo, exp, etc.)

INSA ROUEN NORMANDIE

Synthèse



Quelques remarques

- Sous UNIX, la libc est par défaut incluse au système (donc optionnelle)
- Si le programme n'utilise aucun module/bibliothèque (outre `libc`), on peut compiler et linker directement en utilisant gcc suivi du fichier C (avec optionnellement l'option -o), par exemple :
  - *gcc -o helloworld helloworld.c*
- Il est fortement conseillé d'ajouter l'option `-Wall` pour avoir tous les *warnings*
- L'option `-pedantic` permet de s'assurer que le code C est ISO

INSA ROUEN NORMANDIE

Les types simples

- Le C propose les types simples suivants :
  - int [a], float, double, char [b]
- On peut donc suivre les règles de traduction suivantes :

a. On peut lui adjoindre les qualificatifs `short`, `long` ou `long long` (sur architecture 64bits)
b. On peut leur adjoindre les qualificatifs `signed` ou `unsigned`

Algo. ⇒ C

Algorithmique	C
Entier,Naturel	int, long, short
Réel	float, double
Caractère	char
Booléen	int (1=VRAI, 0=FAUX)
Chaîne de caractères	*Voir les tableaux et pointeurs*

Les variables

On déclare en C une variable en la précédant de son type et en la suivant optionnellement d'une initialisation

```
int a;
float x=1.0;
char c;
```

- Une variable globale est définie en dehors de toute fonction
  - Sa portée est le fichier où elle est définie à partir de sa définition (pas au dessus)
- Une variable locale est définie à l'intérieur d'une fonction
  - Sa portée est uniquement la fonction où elle a été définie

INSA ROUEN NORMANDIE

Définir un type alias

typedef

- Transforme la déclaration d'une variable en déclaration d'un nouveau type
- Syntaxe :
  - **typedef** declaration

```
typedef int Entier;
```

Les énumérés

enum

- Permet de définir un ensemble de valeurs constantes associées à un type
- Syntaxe

  enum [nom] {id1 [=val1], id2 [=val2], ...}

- enum crée un type dont le nom commence par enum
- L'utilisation de enum est souvent couplé avec typedef

typedef enum {LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMAMCHE} Semaine;
Semaine unJour = LUNDI;

INSA ROUEN NORMANDIE

Les constantes 1 / 2

Constantes entières

Pour traduire les constantes entières, le C propose trois notations :

1. La notation décimale, en base dix, par exemple 379
2. La notation octale, en base huit, qui doit commencer par un zéro par exemple 0573
3. Le notation hexadécimale, en base seize, qui doit commencer par un zéro suivi d'un x (ou X), par exemple 0X17B (ou 0x17B, 0X17b, 0x17b)

Constantes caractères

Les constantes caractères sont entourées de quote, par exemple 'a'

- Certains caractères (les caractères non imprimables, avec un code ASCII inférieur à 32, le \, le ') ne sont utilisables qu'en préfixant par un \ :
  - le code ASCII du caractère (exprimé en octal), par exemple '\001'
  - un caractère tel que '\n', '\t', '\\', '\"'

Les constantes 2 / 2

Constantes flotantes

Pour les constantes flotantes, on utilise le "." pour marquer la virgule et le caractère "e" suivi d'un nombre entier, ici a, pour représenter 10^a, par exemple :

- 2. , .3, 2e4, 2.3e4

Constantes chaîne de caractères

Les constantes chaîne de caractères doivent être entourées de guillemets, par exemple "une chaine"

- Par défaut le compilateur ajoute à la fin de la chaîne de caractères '\0', qui est le marqueur de fin de chaîne

- Il n'y a pas véritablement de constante nommée en C, on utilise commande `#define` du préprocesseur
- Toutefois il y a le qualificatif `const` (pour variable ou paramètre formel)

Les opérateurs 1 / 2

Algo. ⇒ C

On traduit les opérateurs en respectant les règles suivantes :

Algorithmique	C
$=, \neq$	==, !=
$<, \leq, >, \geq$	<, <=, >, >=
et, ou, non	&&, ||, !
+, -, *, /	+, -, *, /
div, mod	/, %

Attention

En C l'affectation est une opération (opérateur =)

Les opérateurs 2 / 2

Opérateurs ++ et −−

Les opérateurs unaires ++ et −− sont des opérateurs particuliers qui peuvent avoir jusqu'à deux effets de bord :

- En dehors de toute affectation, elle incrémente l'opérande associée, par exemple
  - `i++` et `++i` sont équivalents à `i=i+1`
- Lorsqu'ils sont utilisés dans une affectation, tout dépend de la position de l'opérateur par rapport à l'opérande, par exemple :
  - `j=i++` est équivalent à `j=i;i=i+1;`
  - `j=++i` est équivalent à `i=i+1;j=i;`

INSA ROUEN NORMANDIE

Les instructions

Instructions simples

Elles finissent toujours par un ' ;'

Exemple

a=a+1;

Instructions composées

Les instructions composées qui permettent de considérer une succession d'instructions comme étant une seule instruction.
Elles commencent par "{" et finissent par "}"

Exemple

{a=a+1;b=b+2;}

Les conditionnelles 1 / 4

if

L'instruction `si...alors...sinon` est traduite par l'instruction `if`, qui a la syntaxe suivante :

if (condition)
  // instruction (s) du if
[**else**
  // instruction (s) du else
]

Exemple

```
if (a<10)
  a=a+1;
else
  a=a+2;
```

Les conditionnelles 2 / 4

Attention

Le `else` dépend toujours du `if` le plus proche

Exemple

```
if (a>b)
if (c<d)
u=v;
else
i=j;
```

Le mieux étant toutefois de clarifier cette ambiguité

```
if (a>b)
  if (c<d)
    u=v;
  else
    i=j;
```

```
if (a>b) {
  if (c<d)
    u=v;
} else {
  i=j;
}
```

Les conditionnelles 3 / 4

switch...case

L'instruction cas..où est traduite par l'instruction switch, qui a la syntaxe suivante :

```
switch ( leSelecteur ) {
  case cas1:
    // instruction (s) du cas 1
    break;
  case cas2:
    // instruction (s) du cas 2
    break;
  ...
  default:
    // instruction (s) du default
}
```

Les conditionnelles 4 / 4

Exemple

```
switch(choix) {
    case 't' : printf("vous voulez un triangle\n"); break;
    case 'c' : printf("vous voulez un carre\n"); break;
    case 'r' : printf("vous voulez un rectangle\n"); break;
    default : printf("erreur. recommencez !\n");
}
```

INSA ROUEN NORMANDIE

Les itérations 1 / 3

for

L'instruction Pour est traduite par l'instruction for, qui a la syntaxe suivante :

```
for( initialisation ;
     condition_d_arret ;
     operation_effectuée_à_chaque_itération )
  instruction ;
```

Exemple

```
for(i=0;i<10;i++)
  printf("%d\n", i);
```

Attention

Contrairement à l'algortihmique le for est une itération indéterministe

Les itérations 2 / 3

while

L'instruction Tant...que est traduite par l'instruction while, qui a la syntaxe suivante :

while(condition)
    instruction ;

Exemple

```
float g,d,m;
g=1;
d=x;
while (d-g>epsilon) {
  m=(d+g)/2.0;
  if (m*m>x)
    d=m;
  else
    g=m;
}
```

Les itérations 3 / 3

do..while

L'instruction répéter est traduite par l'instruction do..while, qui a la syntaxe suivante :

do
    instruction ;
while(condition);

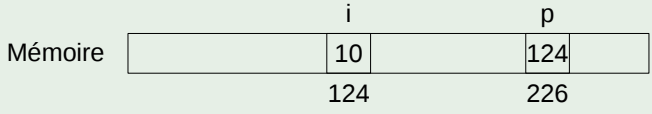
Exemple

```
char temp[MAX];
int motDePasseValide = 0;
int nbEssais = 0;
do {
  printf("Mot de passe : ");
  scanf("%s", temp);
  motDePasseValide = hash(temp) == hashMotPassseRef;
  nbEssais++;
} while (!motDePasseValide && nbEssais<nbEssaisMax) ;
```

Les pointeurs 1 / 3

- Lorsque l'on déclare une variable, par exemple un entier `i`, l'ordinateur réserve un espace mémoire pour y stocker la valeur de `i`
- L'emplacement de cet espace dans la mémoire est nommé adresse
- Un pointeur est une variable qui permet de stocker une adresse

Par exemple si on declare une variable entière `i` (initialisée à 10) et que l'on déclare un pointeur `p` dans lequel on range l'adresse de i (on dit que `p` pointe sur `i`), on a par exemple le schéma suivant :



Les pointeurs 2 / 3

On déclare une variable de type pointeur en préfixant le nom de la variable par le caractère *

```
int i,j;
int *p;
```

Opérateurs * et &

Il existe deux opérateurs permettant d'utiliser les pointeurs :

1. & : permet d'obtenir l'adresse d'une variable, permettant donc à un pointeur de pointer sur une variable
2. * : permet de déréférencer un pointeur, c'est-à-dire d'accéder à la valeur de la variable pointée

```
p=&i; // p pointe sur i
*p=*p+2; // ajoute 2 a i
j=*p; // met la valeur de i dans j (donc 12)
```

Les pointeurs 3 / 3

- Par défaut lorsque l'on déclare un pointeur, on ne sait pas sur quoi il pointe
- Comme toute variable, il faut l'initialiser
  - On peut dire qu'un pointeur ne pointe sur rien en lui affectant la valeur NULL

```
int i;
int *p1,*p2;
p1=&i;
p2=NULL;
```

INSA ROUEN NORMANDIE

Les fonctions 1 / 2

On déclare une fonction en respectant la syntaxe suivante :

```
typeRetour nomFonction(type1 param1, type2 param2, ...) {
  // variables locales

  // instructions avec au moins
  // une fois l'instruction return
}
```

Par exemple

```
int plusUn(int a){
  return a+1;
}
```

INSA ROUEN NORMANDIE

Les fonctions 2 / 2

- Il n'y a pas de procédure en C
  - Pour traduire une procédure, on crée une fonction qui ne retourne pas de valeur, on utilise alors le type `void` comme type de retour
- Tous les passages de paramètres en C sont des passages de paramètres en entrée (on parle de passage par valeur)
  - Lorsque l'on veut traduire des passages de paramètres en sortie ou en entrée/sortie on utilise les pointeurs (on parle de passage de paramètres par adresse)
    - On passe le pointeur sur la variable en lieu et place de la variable

INSA ROUEN NORMANDIE

Implanter un passage de paramètre en entrée

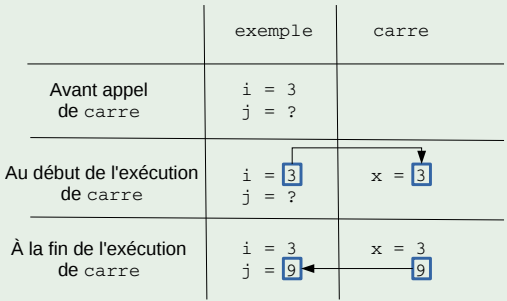
...par un passage de paramètre par valeur

On copie la valeur du paramètre effectif dans le paramètre formel :

```
int carre (int x){
  return (x*x);
}

void exemple (){
  int i = 3;
  int j;

  j = carre (i);
   printf ("%d\n",j);
}
```



Implanter un passage de paramètre en entrée/sortie ou en sortie

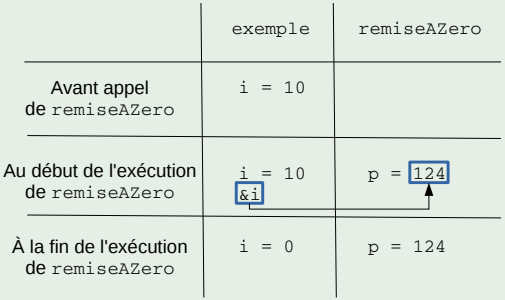
...par un passage de paramètre par adresse

On copie la valeur d'un pointeur sur le paramètre effectif dans le paramètre formel :

```
void remiseAZero(int *p) {
   *p=0;
}

void exemple2() {
   int  i=10;

   remiseAZero(&i);
    printf ("%d\n",i);
}
```



Sortie standard 1 / 2

printf

- L'instruction `printf` (de la bibliothèque `stdio.h`) permet d'afficher des informations à l'écran
- Syntaxe :

  printf ("chaîne de caractères" [, variables])

  - Si des variables suivent la chaîne de caractères, cette dernière doit spécifier comment présenter ces variables :
    - %d pour les entiers (int, short, long)
    - %f pour les réels (float, double)
    - %s pour les chaînes de caractères
    - %c pour les caractères
  - La chaîne de caractères peut contenir des caractères spéciaux :
    - \n pour le retour chariot
    - \t pour les tabulations

Sortie standard 2 / 2

- Par exemple :

```
int i=1;
float x=2.0;
printf("Bonjour\n");
printf("i = %d\n",i);
printf("i = %d, x = %f\n",i,x);
```

- ...affiche :

```
Bonjour
i = 1
i = 1, x=2.0
```

Entrée standard 1 / 2

scanf

- L'instruction `scanf` (de la bibliothèque `stdio.h`) permet à l'utilisateur de saisir des informations au clavier
- Syntaxe :

  scanf("chaîne de formatage",pointeur var1, ...)

  La chaîne de formatage spécifie les caractères et le type des données attendues, par exemple :
  - %d pour les entiers (int, short, long)
  - %f pour les réels (float, double)
  - %s pour les chaînes de caractères
    - l'espace, la tabulation, le retour chariot sont considéré comme un séparateur de chaîne
    - préférez dans ce cas `gets` (voire `fgets`)
  - %c pour les caractères

Entrée standard 2 / 2

Par exemple

```
int i;
float x;
scanf("%d%f",&i,&x);
```

Les tableaux 1 / 2

On déclare une variable de type tableau en ajoutant après l'identificateur la taille de chaque dimension (entre [et])

```
int tab1[10]; // un tableau de 10 entiers (une dimension)
float tab2[2][5]; // un tableau de 10 réels (deux dimensions)
```

- Le premier indice d'une dimension est toujours 0
- C'est vous qui déterminez la sémantique de chaque dimension

INSA ROUEN NORMANDIE

Les tableaux 2 / 2

Une fois déclaré, un élément d'un tableau s'utilise comme une variable classique

```
int t[10]; // un tableau de 10 entiers (une dimension)
int i;
for(i=0;i<10;i++)
  scanf("%d",&t[i]);
```

- Un tableau est considéré comme un pointeur constant
- Lors d'un appel de fonction, le passage de paramètre pour un tableau est un passage de paramètre par adresse
- Dans la signature d'une fonction, la dimension d'un tableau en tant que paramètre formel est facultative

INSA ROUEN NORMANDIE

Les chaînes de caractères

- Il n'y a pas de type chaîne de caractères en C
- Souvent les tableaux de caractères sont utilisés pour les représenter

```
char nom[30];
 printf ("Votre nom: ");
scanf("%s",nom);
```

Si n caractères sont saisis $n+1$ éléments du tableau sont utilisés ('\0').

INSA ROUEN NORMANDIE

Les structures

- On traduit une structure en utilisant le mot clé `struct`
- Syntaxe :

```
struct [Nom] {
  type1 attr1 ;
  ...
}
```

```
struct Personne {
  char nom[30];
  char prenom[30];
}
```

Attention

- `struct` crée un nouveau type dont le nom commence par *struct*
- L'utilisation de `struct` est souvent couplé avec `typedef`

Les unions

- Équivalent aux structures sauf que l'espace mémoire réservé est le maximum des espaces mémoires des attributs (alors que pour les structures c'est l'addition) [a]
- Un seul champ est donc interprétable à un moment donné
- Syntaxe :

```
union [Nom] {
  type1 attr1;
  ...
}
```

a. à un alignement près...

```
union Nombre {
  int unEntier;
  float unReel;
  Complexe unComplexe;
}
```

Les fichiers

Le type FILE*

- Pas de concept de fichiers texte ou d'enregistrement : ce sont des fichiers binaires
- Ouverture : `FILE *fopen(const char *path, const char *mode);`
  - mode : "r", "r+", "w", "w+", "a", "a+"
  - "b" peut être ajouté pour la compatibilité C89
- Fermeture : `int fclose(FILE *fp)`
- Fonction de lecture : `fscanf`, `fgetc`, `fgets`, `fread`, ...
- Fonction d'écriture : `fprintf`, `fputc`, `fputs`, `fwrite`, ...
- Fonction gestion curseur : `fseek`, `ftell`, `rewind`, `feof`, ...
- Pointeurs de fichier prédéfinis : `stdin`, `stdout`, `stderr`

INSA ROUEN NORMANDIE

Retour sur la fonction `main`

En fait la signature de la fonction `main` est `int main(int argc, char **argv)`, tel que :

- `argc` est le nombre de paramètres
- `argv` est un tableau de chaîne de caractères

`argv[0]` est le nom du programme

Exemple : args.c* [a]

a. http://www.ai.univ-paris8.fr/~jalb/langimp/main.html

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

int main(int argc, char **argv) {
  int i;
  printf("argc: %d\n", argc);
  for (i= 0; i < argc; i++)
    printf("argv[%d]: %s\n", i, argv[i]);
  return EXIT_SUCCESS;
}
```

Pointeurs de fonction

- Les fonctions peuvent être considérées comme des variables, elles possèdent :
  - un identifiant
  - un type (type des paramètres et type de retour)
- Elles peuvent être un argument d'un appel de fonction

Exemple 1 / 3

Tri d'un tableau d'entiers dans un ordre croissant puis décroissant

```
void triABulle(int t[], int (*comparer)(int,int), int nbElements) {
  int i,j,estTrie;
  estTrie=0;
  i=0;
  while ((i<nbElements) && (!estTrie)) {
    estTrie=1;
    for (j=nbElements-1;j>0;j--) {
      if (comparer(t[j-1],t[j])>0) {
        echanger(&t[j-1],&t[j]);
        estTrie=0;
      }
    }
    i++;
  }
}
```

Exemple 2 / 3

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

void trierEtAfficherTableauDEntiers (int (*comparer) (int, int)) {
  int i;
  int tabEntier [4];

  for (i=0;i<4;i++) {
    tabEntier[i]=random() % 10;
  }
  afficherTabEntier (tabEntier,4);
  triABulle (tabEntier,comparer,4);
  afficherTabEntier (tabEntier,4);
}
```

Exemple 3 / 3

```
int plusGrand(int a, int b) {
  return a<b;
}

int plusPetit (int a, int b) {
  return a>b;
}

int main() {
   trierEtAfficherTableauDEntiers ( plusPetit );
   trierEtAfficherTableauDEntiers (&plusGrand);
}
```

```
3 6 7 5
3 5 6 7
3 5 6 2
6 5 3 2
```

Les bibliothèques 1 / 3

- Le C est un langage modulaire :
  - Le cœur du C ne contient que le strict minimum
  - Les extensions sont des bibliothèques
- Lorsque l'on écrit un programme en C :
  - On réutilise des bibliothèques
  - On crée des modules et quelques fois des bibliothèques
  - Le programme principal est court et utilise ces modules et ces bibliothèques

Comment créer une bibliothèque ou un module ?

- Créer un `.h` qui contient :
  - la définition des types disponibles
  - la signature des fonctions disponibles
- Créer un `.c` qui contient :
  - la définition de types et/ou fonctions privés
  - la définition des fonctions déclarées dans le `.h`

Les bibliothèques 2 / 3

Comment rendre disponible une bibliothèque ?

- Compiler le(s) `.c` : obtention de `.o` (code objet)
- Inclure le(s) `.o` dans une bibliothèque :
  - statique : `.a` (`.lib` sous Windows)
    ex : `ar -r libtest.a test1.o test2.o ...`
  - dynamique : `.so` (`.dll` sous Windows)
    ex : `gcc -o libtest.so -shared test1.o test2.o ...`
- Rendre disponible le(s) `.h` et le(s) `.a` ou `.so`

Attention

- Par abus de langage, on appelle bibliothèque aussi bien le `.h` que le `.a` (ou `.so`)
- Puisque une bibliothèque (`.a` ou `.so`) peut être la concaténation de plusieurs `.o`, plusieurs `.h` peuvent être associés à une bibliothèque

Les bibliothèques 3 / 3

Comment utiliser une bibliothèque ?

- Inclure le `.h` dans les fichiers (`.h` ou `.c`) l'utilisant
- À la compilation du projet le `.h` doit être disponible
  - le référencer en spécifiant dans quel répertoire il se trouve (option `-I` de gcc)
- À l'édition des liens, le(s) `.a` (`.so`) doit(doivent) être disponible(s)

INSA ROUEN NORMANDIE

Problème des inclusions multiples 1 / 2

entierA.h

```
#include "entierB.h"
typedef int entierA;
entierA entierBEnentierA(entierB);
```

entierB.h

```
#include "entierA.h"
typedef int entierB;
entierB entierAEnentierB(entierA);
```

Problème

L'interprétation de `entierA.h` par le préprocesseur va inclure `entierB.h`, qui va inclure `entierA.h`, qui va inclure `entierB.h`, ...

Problème des inclusions multiples 2 / 2

entierA.h

```
1 #ifndef __ENTIER_A__
2 #define __ENTIER_A__
3 typedef int entierA;
4 #include "entierB.h"
5 entierA entierBEnentierA(entierB);
6 #endif
```

entierB.h

```
1 #ifndef __ENTIER_B__
2 #define __ENTIER_B__
3 typedef int entierB;
4 #include "entierA.h"
5 entierB entierAEnentierB(entierA);
6 #endif
```

Macro définition (macro-fonction) 1 / 4

- Possibilité de créer des fonctions *inline*
- Syntaxe : `#define identifiant(arguments) code`
  - Les parenthèses suivent aussitôt l'identifiant
  - Les paramètres sont séparés par des virgules avec aucun type de spécifié
  - Le code peut être sur plusieurs lignes (utilisation de \ en fin de ligne)
- Objectifs :
  - Avoir de la modularité sans appel de fonction : gain de performance, non utilisation de la pile puisque le code est copié/collé avec les paramètres effectifs qui remplacent les paramètres formels
  - Possibilité d'une sorte de surcharge, même identifiant de macro avec un nombre de paramètres formels différents (un *warning* est produit)

INSA ROUEN NORMANDIE

Macro définition (macro-fonction) 2 / 4

Code source

```
#include <stdlib.h>
#include <stdio.h>
#define ADD(x,y) ((x)+(y))

int main(int argc, char **argv) {
  int milieu, gauche, droite;

  scanf("%d%d",&gauche,&droite);
  milieu = ADD(gauche,droite);
  printf("%d\n", milieu);

  return EXIT_SUCCESS;
}
```

Code produit (extrait)

```
int main(int argc, char **argv) {
  int milieu, gauche, droite;

  scanf("%d%d",&gauche,&droite);
  milieu = ((gauche)+(droite));
  printf("%d\n", milieu);

  return 0;
}
```

Macro définition (macro-fonction) 3 / 4

Des problèmes peuvent apparaître [a]

a. https ://wiki.deimos.fr/Mieux_connaître_et_utiliser_le_préprocesseur_du_langage_C.html

- Utilisation multiple du paramètre formel
- Les conditionnels et itérations avec des macros sur plusieurs lignes

INSA ROUEN NORMANDIE

Macro définition (macro-fonction) 4 / 4

Les conditionnels et itérations avec des macros sur plusieurs lignes

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

#define ADD(a, b, c) \
   printf ("%d + %d ", a, b); \
  c = a + b; \
   printf ("= %d\n", c);

int main(int argc, char *argv[]) {
  int i = 0;
  int r = 0;
  while (argv[i++])
    ADD(r, 1, r);
  return EXIT_SUCCESS;
}
```

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

#define ADD(a, b, c) \
   printf ("%d + %d ", a, b); \
  c = a + b; \
   printf ("= %d\n", c);

int main(int argc, char *argv[]) {
  int i = 0;
  int r = 0;
  while (argv[i++]) {
    ADD(r, 1, r);
  };
  return EXIT_SUCCESS;
}
```

```
$ ./a.out 1 2 3
0 + 1 0 + 1 0 + 1 0 + 1 = 1
```

```
$ ./a.out 1 2 3
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
```

Mot clé `static`

...devant la déclaration d'une variable locale

Permet de conserver l'état de la variable entre deux exécutions d'une fonction
La variable n'est alors initialisée qu'une seule fois

```
void foo() {
   static int compteurDAppels=0;
   compteurDAppels++;
    ...
}
```

Mot clé `static`

...devant la déclaration d'une variable globale ou d'une fonction

- Cela permet de spécifier que la variable globale ou la fonction est accessible uniquement depuis le fichier où elle est déclarée
- Cela permet d'avoir des variables globales ou des fonctions ayant des même noms dans des fichiers différents

INSA ROUEN NORMANDIE

Mot clé `extern` [a]

a. http://www.lri.fr/~aze/page_c/aide_c/

- Indique que l'emplacement réel et la valeur initiale d'une variable, ou du corps d'un sous-programme sont définis ailleurs, généralement dans un autre fichier source.
- Cela permet à plusieurs fichiers de se partager une variable ou un sous-programme.
- Pour un prototype de fonction, le mot réservé extern est facultatif.

INSA ROUEN NORMANDIE

Arithmétique sur les pointeurs

- On peut appliquer les opérateurs $+$, $-$, $++$ et $--$ à des variables du type pointeurs

```
int t [10]={1,2,3,4,1,7,2,4,9,60};
int *p;

p=t;
p=p+3;
*p=0;
p--;
*p=10;
```

Allocation statique et dynamique

- Lorsque l'on déclare une variable locale, le compilateur ajoute des instructions dans le code généré réservant de l'espace mémoire dans la pile lorsque la fonction de la variable sera exécutée
  - C'est ce que l'on appelle l'allocation statique
  - Il ajoute aussi du code libérant cet espace mémoire qui sera exécuté lorsque l'on sortira de la fonction
- On aimerait quelquefois pouvoir déterminer la taille de l'espace mémoire à réserver non pas à la compilation, mais à l'exécution
  - C'est ce que l'on appelle l'allocation dynamique
  - Elle se fait dans le tas
  - On réserve l'espace mémoire à l'aide de la fonction `malloc` (et consoeurs) et on libère cet espace à l'aide de la fonction `free`

INSA ROUEN NORMANDIE

Exemple

```
void trierEtAfficherTableauDEntiers (int (*comparer) (int, int), int nb) {
  int i;
  int* tabEntier;

  tabEntier=(int*)malloc(nb*sizeof(int));
  for (i=0;i<nb;i++) {
    tabEntier[i]=random() % nb;
  }
  afficherTabEntier (tabEntier,nb);
  triABulle (tabEntier,comparer,nb);
  afficherTabEntier (tabEntier,nb);
  free (tabEntier);
}

int main() {
  int nb;
  printf ("nb entier");
  scanf("%d",&nb);
  trierEtAfficherTableauDEntiers ( plusPetit ,nb);
  trierEtAfficherTableauDEntiers (plusGrand,nb);
```

Chaînes de caractères 1 / 8

Questions

- Lorsque l'on demande à l'utilisateur de saisir une chaîne de caractères, combien doit on réserver d'espace ?
- Lorsque l'on veut écrire un fonction qui doit « calculer » une chaîne de caractères, qui réserve l'espace mémoire, l'appelant ou l'appelé ?

Réservées où ? comment ?

- À l'aide d'un tableau de caractères. C'est donc une allocation statique, qui n'a d'existence que dans le fonction où elle est faite
- À l'aide d'une allocation dynamique. Il doit donc y avoir une désallocation quelque part

INSA ROUEN NORMANDIE

Chaînes de caractères 2 / 8

Premier exemple

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

#define MAX 100

char* saisieQuiNeMarchePas() {
  char buffer[MAX];
  gets(buffer);
  return buffer;
}

int main(int argc, char **argv) {
   printf("%s\n",saisieQuiNeMarchePas());
  return EXIT_SUCCESS;
}
```

Compilation

```
$ gcc -c -Wall -pedantic testChaine1.c
testChaine1.c: In function 'saisieQuiNeMarchePas':
testChaine1.c:9:3: attention: cette fonction retourne l'adresse d'une
variable locale [enabled by default]
```

Chaînes de caractères 3 / 8

Exécution

```
$ ./testChaine1
un petit test qui risque de ne pas marcher
un petit test q
$
```

Chaînes de caractères 4 / 8

Deuxième exemple

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

#define MAX 10

void saisieQuiDevraitMarcher(char buffer []) {
  gets( buffer );
}

int main(int argc, char **argv) {
  char buffer [MAX];
  saisieQuiDevraitMarcher( buffer );
  printf ("%s\n",buffer);
  return EXIT_SUCCESS;
}
```

Compilation

```
$ gcc -c -Wall -pedantic testChaine2.c
$
```

Chaînes de caractères 5 / 8

Exécution

```
$ ./testChaine2
ca marche
ca marche
$ ./testChaine2
ca ne marche plus du tout
ca ne marche plus du tout
*** stack smashing detected ***: ./testChaine2 terminated
======= Backtrace: =========
/lib/x86_64-linux-gnu/libc.so.6(__fortify_fail+0x37)[0x7fe72ed77007]
/lib/x86_64-linux-gnu/libc.so.6(__fortify_fail+0x0)[0x7fe72ed76fd0]
...
```

INSA ROUEN NORMANDIE

Chaînes de caractères 6 / 8

Troisième exemple

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

#define MAX 10

void saisieQuiMarche(char buffer []) {
   fgets( buffer ,MAX,stdin);
}

int main(int argc, char **argv) {
  char buffer [MAX];
  saisieQuiMarche( buffer );
   printf ("%s\n",buffer);
  return EXIT_SUCCESS;
}
```

Compilation

```
$ gcc -c -Wall -pedantic testChaine3.c
$
```

Chaînes de caractères 7 / 8

Exécution

```
$  ./testChaine3
un petit test qui marche tres bien
un petit
```

Chaînes de caractères 8 / 8

Quatrième exemple

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

#define MAX 10

char* saisieQuiMarche() {
  char* buffer=(char*)malloc(sizeof(char)*MAX);
  fgets(buffer,MAX,stdin);
  return buffer;
}

int main(int argc, char **argv) {
  char* ch;
  ch=saisieQuiMarche();
  printf("%s\n",ch);
  free(ch);
  return EXIT_SUCCESS;
}
```

ROUEN NORMANDIE

La bibliothèque standard

- Le C standard est livré avec des bibliothèques standards, dont les fichiers d'en-tête sont :
  - **assert.h**
  - ctypes.h
  - **errno.h**
  - float.h
  - limits.h
  - locale.h
  - math.h
  - setjmp.h
  - signal.h
  - stdarg.h
  - stddef.h
  - stdio.h
  - stdlib.h
  - string.h
  - time.h

INSA ROUEN NORMANDIE

Erreurs : `errno.h`

Gestion des erreurs

- Lorsqu'une fonction ne retourne pas naturellement de valeur (`void`) ou qu'elle doit retourner un pointeur, et qu'elle peut produire des erreurs, c'est la valeur de retour qui indique si une erreur s'est produite :
  - `int` à la place de `void` et une valeur différente de 0 est retourné ;
  - `NULL` si c'est un pointeur qui est retournée
- Dans les autres cas, c'est la variable globale `errno` qui est utilisée

INSA ROUEN NORMANDIE

Préconditions : assert.h 1 / 2

Gestion des préconditions

- On peut faire apparaître des préconditions en C via l'utilisation de la bibliothèque assert qui propose la macro
  `assert(expression)`
- « Si la macro `NDEBUG` est définie au moment de la dernière inclusion de `<assert.h>`, la macro `assert()` ne génère aucun code et ne fait rien. Sinon, la macro `assert()` affiche un message d'erreur sur la sortie d'erreur et termine l'exécution du programme en cours en appelant `abort()` si l'expression est fausse (égale à zéro).
  Le but de cette macro est d'aider le programmeur à trouver des bogues dans son application. Le message ‘‘`assertion failed in file foo.c, function do_bar(), line 1287`’’ n'est d'aucune aide pour l' utilisateur. »[a]

a. *man*

Préconditions : assert.h 2 / 2

Un exemple de code : *assert.c*

```
#include <stdio.h>

/* #define NDEBUG
    décommenter pour annuler les tests
    de précondition
*/
#include <assert.h>

int foo(int a) {
  assert(a>0);
  return a;
}

int main() {
  int i;
  scanf("%d",&i);
  printf("%d\n",foo(i));
}
```

Un exemple d'exécution

```
$ ./assert
-10
assert: assert.c:10: foo: Assertion 'a>0' failed.
Abandon (core dumped)
```

Compiler un projet C 1 / 3

Organisation d'un projet C

- Un projet C est composé de plusieurs répertoires :

src	les `.c` (avec possibilité de sous répertoires), c'est aussi dans ce répertoire que sont mis les `.o`
include	les `.h`
src/tests	les sources des tests unitaires
lib	les bibliothèques du projet (`.a`, `.so`)
bin	le(s) exécutable(s)
tests	les tests unitaires
doc	la documentation technique générée (souvent `doc/html` et `doc/pdf`)

Un exemple : programme de tri par minimum successif

- Composé de :
  - `src/main.c`
  - `include/echanger.h`, `src/echanger.c`
  - `include/triParMinSuccessif.h`, `src/triParMinSuccessif.c`
- Objectifs :
  - `lib/libechanger.a`
  - `bin/tri`

Compiler un projet C 2 / 3

Comment compiler ce projet ? utilisation d'un script bash. . .

```
1 #!/bin/bash
2 gcc -o src/echanger.o -Wall -pedantic -c -Iinclude src/echanger.c
3 ar -r lib/libechanger.a src/echanger.o
4 gcc -o src/triParMinSuccessif.o -Wall -pedantic -c -Iinclude src/triParMinSuccessif.c
5 gcc -o src/main.o -Wall -pedantic -c -Iinclude src/main.c
6 gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger
```

INSA ROUEN NORMANDIE

Compiler un projet C 3 / 3

Objectifs du makefile

Faciliter :

- la compilation, avec vérification des dépendances (si utilisation d'`autoconf` et `automake`) et ne compiler que ce qui est nécessaire
- l'installation
- la désinstallation

utilisation classique

```
./configure
make
make install
```

- le fichier `./configure` est généré par des utilitaires (`autoconf` et `automake`)
- les environnements de développement génèrent automatiquement ces fichiers

Écriture d'un makefile à la main 1 / 7

Pour plus d'information `http://gl.developpez.com/tutoriel/outil/makefile/`

Structure du fichier

[`déclaration de variables`]
`Règles`

Règles

- Permet de décomposer une action en plusieurs actions
- Structure :

```
cible: dependance
    actions
```

- Quelques cibles prédéfinies : `all`, `install`, `clean`

ROUEN NORMANDIE

Écriture d'un makefile à la main 2 / 7

Un premier makefile

```
1  all : bin/tri
2
3  bin/tri : lib/libechanger.a src/triParMinSuccessif.o src/main.o
4          gcc -o bin/tri src/main.o src/triParMinSuccessif.o -Llib -lechanger
5
6  lib/libechanger.a : src/echanger.o
7          ar -r lib/libechanger.a src/echanger.o
8
9  src/echanger.o : src/echanger.c
10         gcc -o src/echanger.o -c -Wall -pedantic -Iinclude src/echanger.c
11
12 src/triParMinSuccessif.o : src/triParMinSuccessif.c
13         gcc -o src/triParMinSuccessif.o -c -Wall -pedantic -Iinclude src/triParMinSuccessif.c
14
15 src/main.o : src/main.c
16         gcc -o src/main.o -c -Wall -pedantic -Iinclude src/main.c
17
18 clean :
19         rm -f bin/*
20         rm -f src/*.o
21         rm -f lib/*.a
```

INSA ROUEN NORMANDIE

Écriture d'un makefile à la main 3 / 7

Un makefile avec variables

```
SRCDIR=src
LIBDIR=lib
BINDIR=bin
INCLUDEDIR=include
CC = gcc
AR = ar
CFLAGS=-Wall -pedantic -I$(INCLUDEDIR)
LDFLAGS=-L$(LIBDIR)
EXEC=tri

all : $(BINDIR)/$(EXEC)

$(BINDIR)/tri : $(LIBDIR)/libechanger.a $(LIBDIR)/triParMinSuccessif.o $(LIBDIR)/main.o
	$(CC) $(LDFLAGS) -o $(BINDIR)/$(EXEC) $(LIBDIR)/main.o $(LIBDIR)/triParMinSuccessif.o -lechanger

$(LIBDIR)/libechanger.a : $(SRCDIR)/echanger.o
	$(AR) -r $(LIBDIR)/libechanger.a $(SRCDIR)/echanger.o

$(SRCDIR)/echanger.o : $(SRCDIR)/echanger.c
	$(CC) -o $(SRCDIR)/echanger.o -c $(CFLAGS) $(SRCDIR)/echanger.c

$(SRCDIR)/triParMinSuccessif.o : $(SRCDIR)/triParMinSuccessif.c
	$(CC) -o $(SRCDIR)/triParMinSuccessif.o -c $(CFLAGS) $(SRCDIR)/triParMinSuccessif.c

$(SRCDIR)/main.o : $(SRCDIR)/main.c
	$(CC) -o $(SRCDIR)/main.o -c $(CFLAGS) $(SRCDIR)/main.c

clean :
	rm $(BINDIR)/*
	rm $(SRCDIR)/*.o
```

Écriture d'un makefile à la main 4 / 7

Quelques variables internes

- `$@` Le nom de la cible de la règle
- `$<` Le nom de la première dépendance de la règle
- `$^` La liste des dépendances de la règle
- `$?` La liste des dépendances plus récentes que la cible
- `$*` Le nom du fichier sans suffixe

INSA ROUEN NORMANDIE

Écriture d'un makefile à la main 5 / 7

Un makefile avec variables

```
SRCDIR=src
LIBDIR=lib
BINDIR=bin
INCLUDEDIR=include
CC = gcc
AR = ar
CFLAGS=-Wall -pedantic -I$(INCLUDEDIR)
LDFLAGS= -L$(LIBDIR)
EXEC=tri

all : $(BINDIR)/$(EXEC)

$(BINDIR)/tri : $(LIBDIR)/libechanger.a $(SRCDIR)/triParMinSuccessif.o $(SRCDIR)/main.o
	$(CC) $(LDFLAGS) -o $@ $(SRCDIR)/triParMinSuccessif.o $(SRCDIR)/main.o -lechanger

$(LIBDIR)/libechanger.a : $(SRCDIR)/echanger.o
	$(AR) -R $@ $^

$(SRCDIR)/echanger.o : $(SRCDIR)/echanger.c
	$(CC) -o $@ -c $< $(CFLAGS)

$(SRCDIR)/triParMinSuccessif.o : $(SRCDIR)/triParMinSuccessif.c
	$(CC) -o $@ -c $< $(CFLAGS)

$(SRCDIR)/main.o : $(SRCDIR)/main.c
	$(CC) -o $@ -c $< $(CFLAGS)

clean :
	rm $(BINDIR)/*
	rm $(SRCDIR)/*.o
```

Écriture d'un makefile à la main 6 / 7

Règles d'inférence

- Possibilité de généraliser des règles

```
%.o: %.c
    commandes
```

Écriture d'un makefile à la main 7 / 7

Un makefile avec variables

```
1  SRCDIR=src
2  LIBDIR=lib
3  BINDIR=bin
4  INCLUDEDIR=include
5  CC = gcc
6  AR = ar
7  CFLAGS=-Wall -pedantic -I$(INCLUDEDIR)
8  LDFLAGS=-L$(LIBDIR)
9  EXEC=tri
10
11 all : $(BINDIR)/$(EXEC)
12
13 $(BINDIR)/tri : $(LIBDIR)/libechanger.a $(SRCDIR)/triParMinSuccessif.o $(SRCDIR)/main.o
14         $(CC) $(LDFLAGS) -o $@ $(SRCDIR)/triParMinSuccessif.o $(SRCDIR)/main.o -lechanger
15
16 $(LIBDIR)/lib%.a : $(SRCDIR)/%.o
17         $(AR) -r $@ $^
18
19 $(SRCDIR)/%.o : $(SRCDIR)/%.c
20         $(CC) -o $@ -c $< $(CFLAGS)
21
22 clean :
23         rm $(BINDIR)/*
24         rm $(SRCDIR)/*.o
25         rm $(LIBDIR)/*.a
```