



Programmation Impérative 2

Rappels sur le langage C

C statique

Licence 2 Informatique

Antoine Spicher

`antoine.spicher@u-pec.fr`

Organisation



■ Cours

- ☐ C « statique »
- ☐ C « dynamique »
- ☐ E/S standard
- ☐ Programmation séparée, modulaire

■ Evaluation

- ☐ Examen (40% de la note finale)
- ☐ Projet
- ☐ Comptes rendus des TPs notés

Plan du cours : C statique



- Introduction
- Grammaire des programmes C
- Structures de contrôle
- Typage
- Variables et fonctions

Historique du langage C

■ Origine

- Dennis Ritchie en 1972
- Langage de programmation pour le développement du système UNIX
- Successeur du B, lui-même inspiré du langage BCPL (Thompson & Ritchie)



Kenneth Thompson

Dennis Ritchie
(source: Wikipedia)

■ Normes et standards

- 1989 : ANSI C
- 1999 : C99

■ Évolutions

- Future normalisation envisagée : C1X
- Nouveaux langages (successeurs : C++, D ; inspirations : Java, PHP, ...)

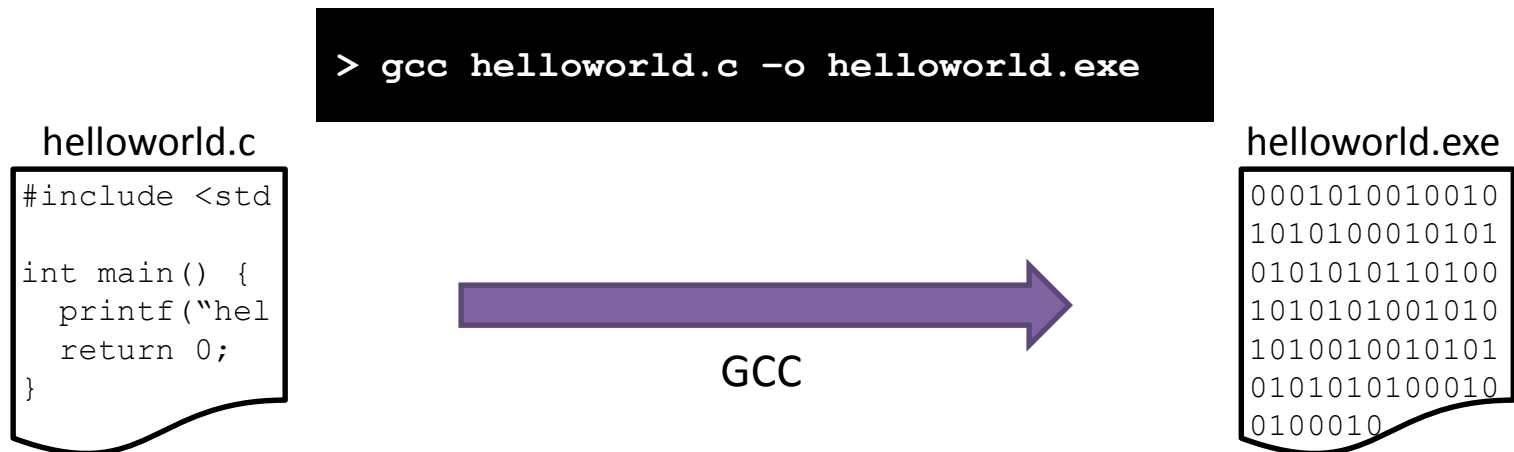
Compilateur GCC

■ Présentation

- ❑ Originellement, « *GNU C Compiler* », alternative *open source* de CC (UNIX)
- ❑ « *GNU Compiler Collection* », suite des compilateurs du projet GNU

■ Utilisation

- ❑ *Compilateur* : traducteur d'un texte/programme écrit dans un langage source vers un texte/programme écrit dans un langage destination
- ❑ Note cas : un programme en langage C vers un programme *exécutable* en langage machine (binaire)



Programmation impérative



■ Type de programmation

- Programmation fonctionnelle (e.g., Lisp, SML, OCaml, Haskell, F#, ...)
 - Haut niveau
 - Proche de la définition mathématique des calculs
 - Ce que fait un calcul
- Programmation impérative (e.g., C, C++, Java, PHP, FORTRAN, Pascal, ...)
 - Bas niveau
 - Proche du fonctionnement des ordinateurs
 - Comment effectuer un calcul

■ Programmation impérative

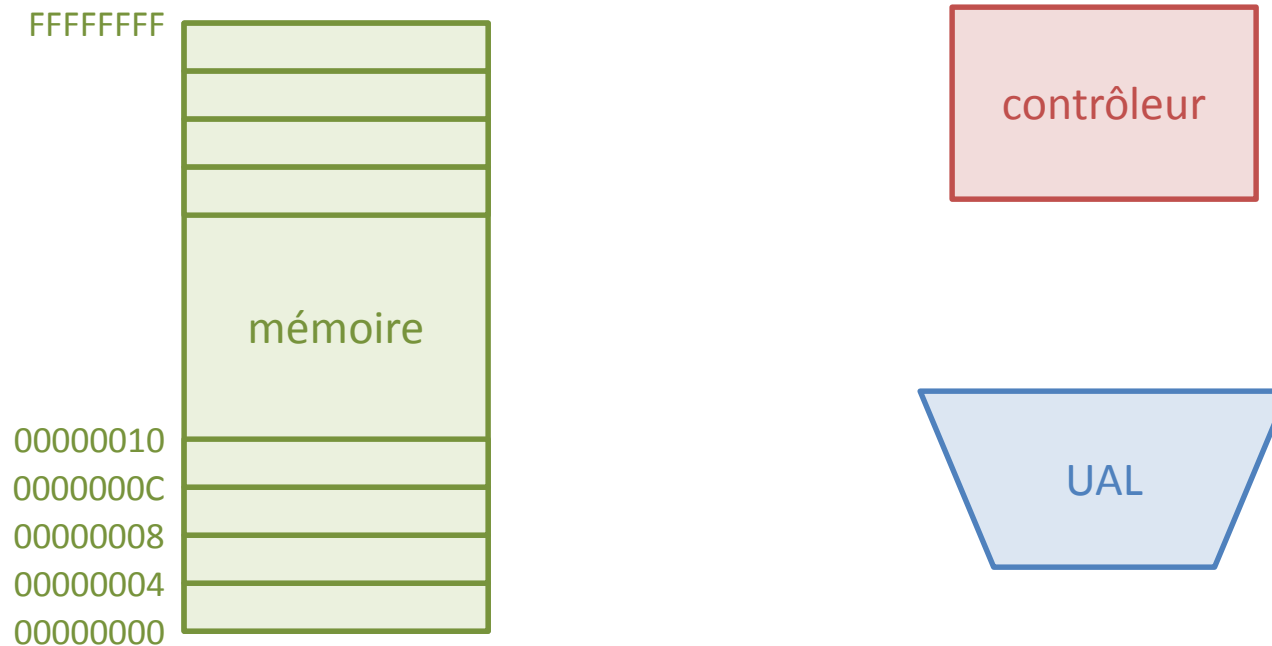
- Modèle de calcul : machine à états
 - état = mémoire, transition = instruction, exécution = états successifs de la mémoire
- Valeur, variable, adresse

Programmation impérative

■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire



Programmation impérative

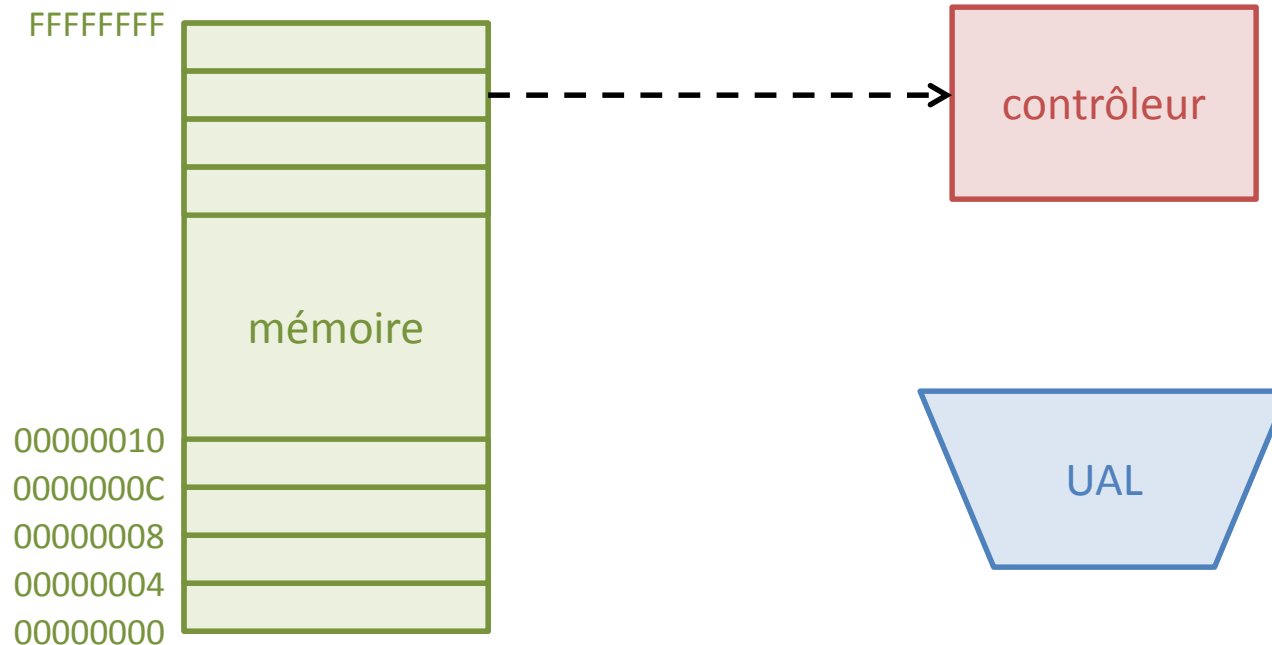
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction



Programmation impérative

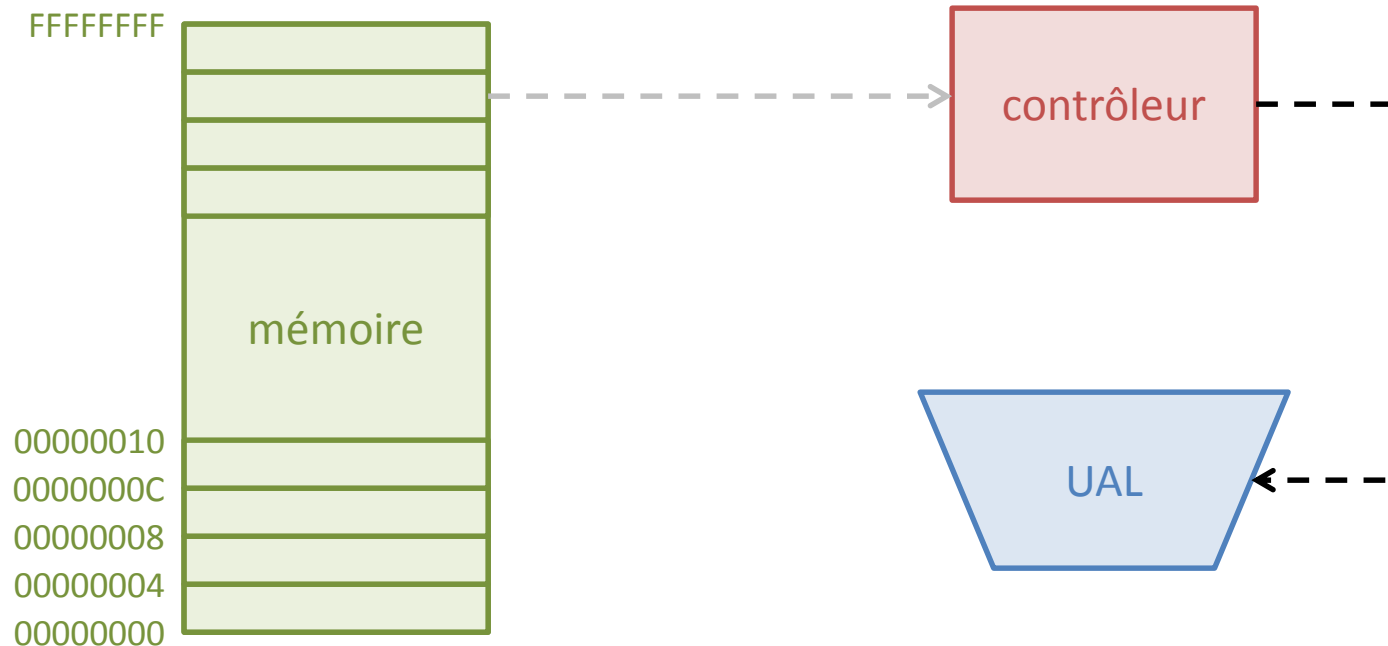
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction, *distribution des ordres*



Programmation impérative

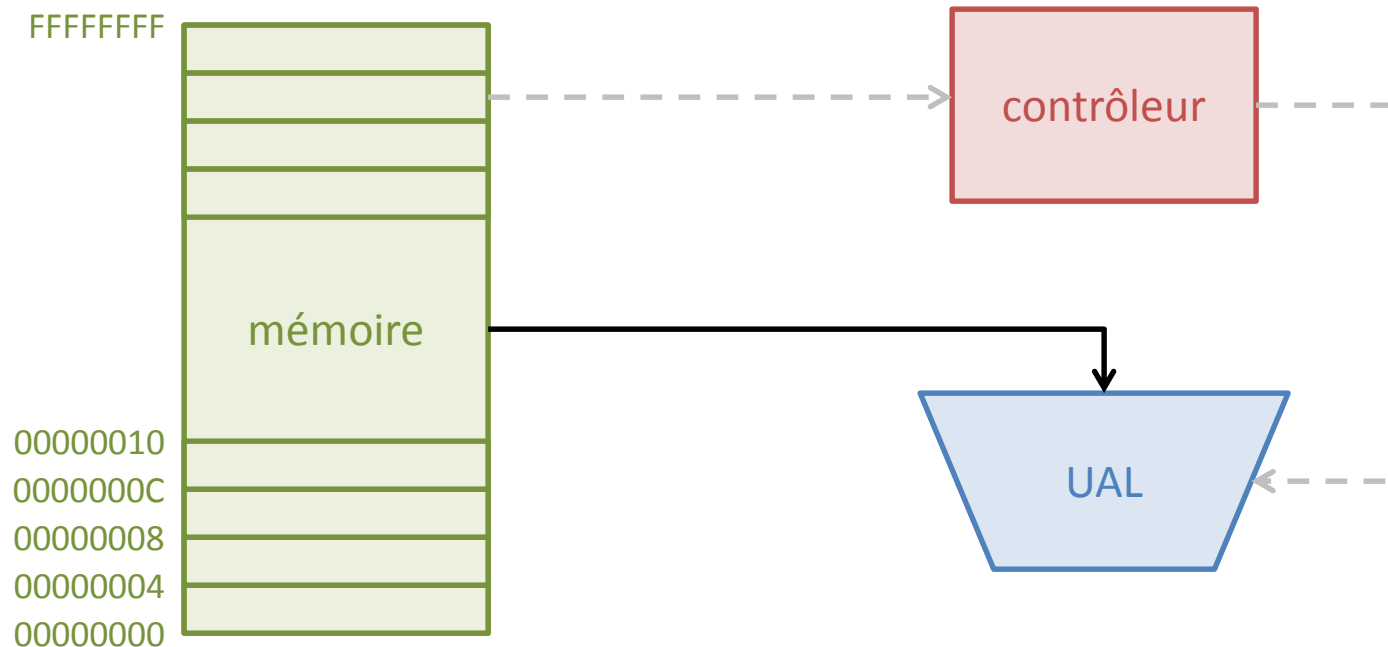
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction, distribution des ordres, *lecture des données nécessaires*



Programmation impérative

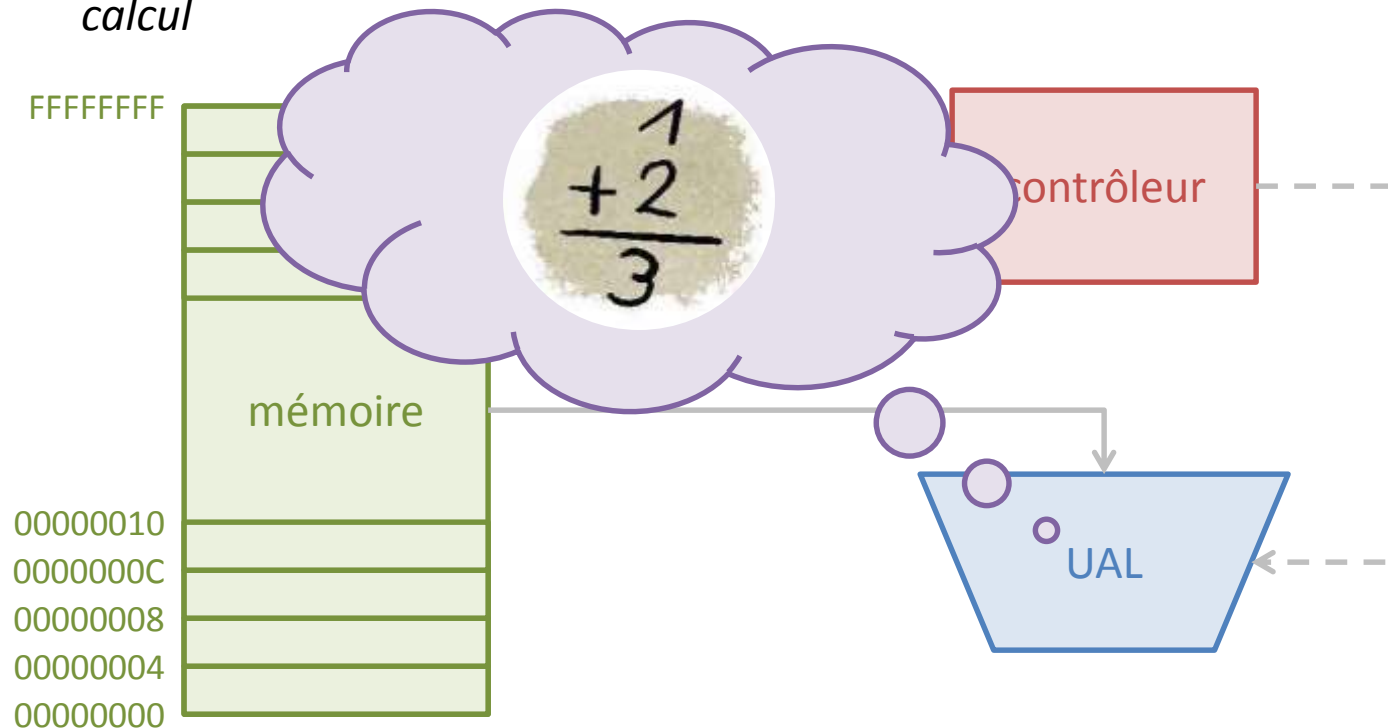
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction, distribution des ordres, lecture des données nécessaires, *calcul*



Programmation impérative

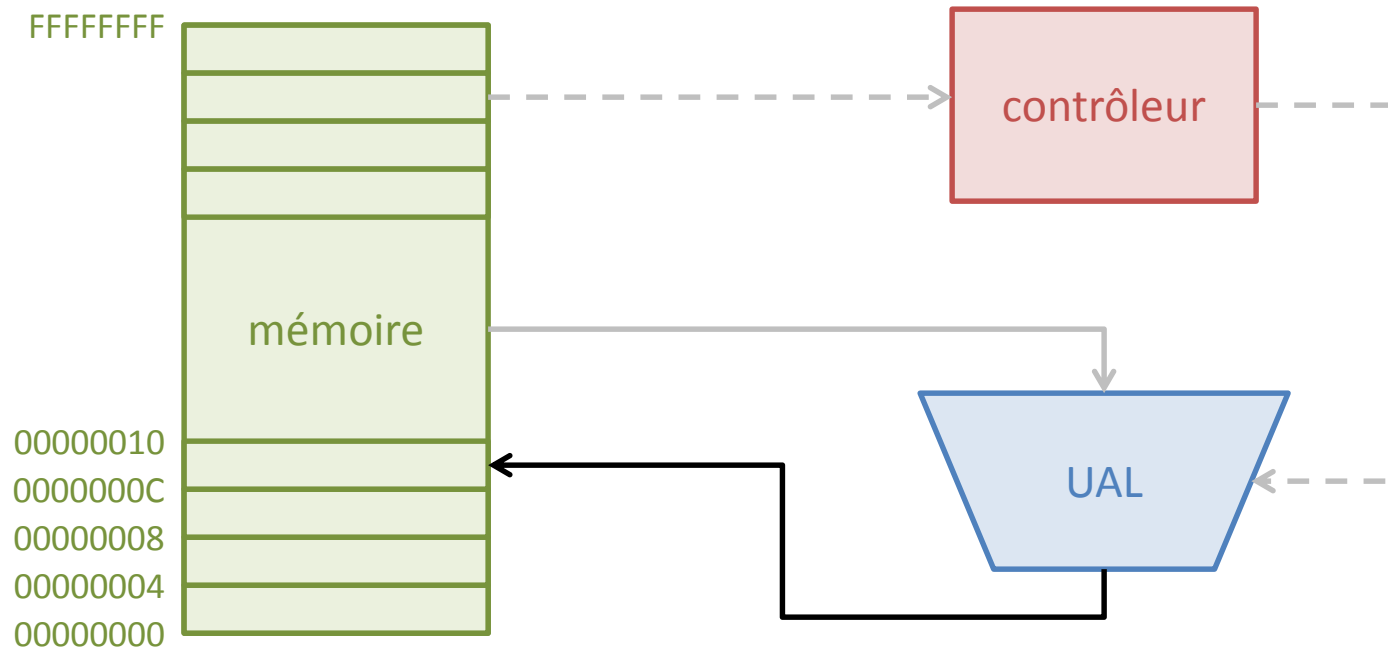
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction, distribution des ordres, lecture des données nécessaires, calcul, *enregistrement du résultat*



Programmation impérative

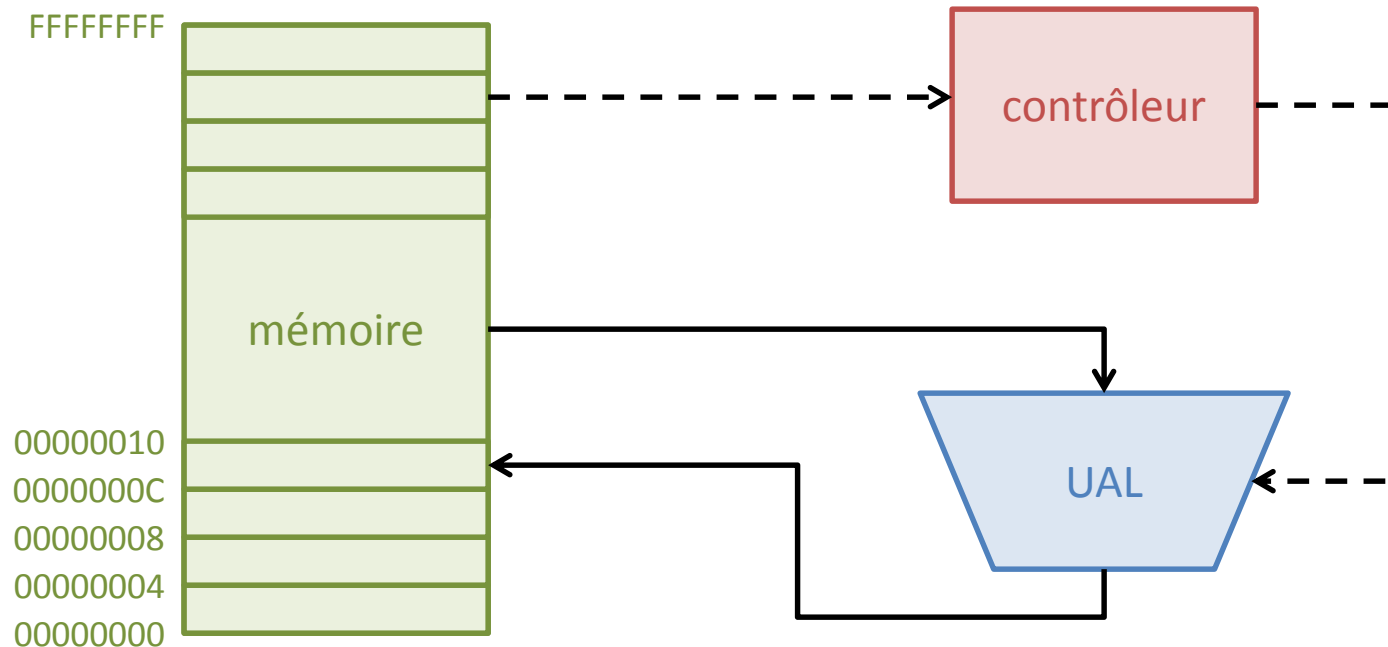
■ Schématisation du modèle de calcul

□ Architecture des ordinateurs

Unité arithmétique et logique (UAL), contrôleur, mémoire

□ Étapes du calcul

Lecture de l'instruction, distribution des ordres, lecture des données nécessaires, calcul, enregistrement du résultat



Plan du cours : C statique



- Introduction
- Grammaire des programmes C
- Structures de contrôle
- Typage
- Variables et fonctions

Grammaire des programmes C

■ Éléments de base sur les grammaires

- Outil mathématique pour la description de la syntaxe d'un langage
- Cf. les UEs « Langage Formel » et « Compilation » du L3
- Éléments syntaxiques

- *Terminaux* : les mots que vous pouvez écrire (e.g., `'if'`)

- *Non-terminaux* : les sous-parties du langage (e.g., `<instruction>`)

- *Construction syntaxique* :

- Le choix `... | ...` (se lit « soit ..., soit ... »)

`<chiffre_paire> ::= '0' | '2' | '4' | '6' | '8'`

`<chiffre> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'`

- La séquence `(...)*` (se lit « une suite de ... »)

`<entier_naturel> ::= (<chiffre>)*`

- L'option `[...]` (se lit « avec possiblement ... »)

`<entier_relatif> ::= ['-'] <entier_naturel>`

Grammaire des programmes C

■ Sous-partie du langage

`<bloc>` ::= `\{ ' [<déclarations>] <séquence> \} '`

`<séquence>` ::= `<instruction>*`

`<instruction>` ::= `<expression> \; ' | <bloc> | ... à compléter ...`

`<expression>` ::= `<expression> <op> <expression>`
| `(\~ ' | _ ' | \! ') <expression>`
| `\ (' <expression> \) '`
| `<nom_de_fonction> \ (' <arguments> \) '`
| `<constante> | <variable> | <affectation>`

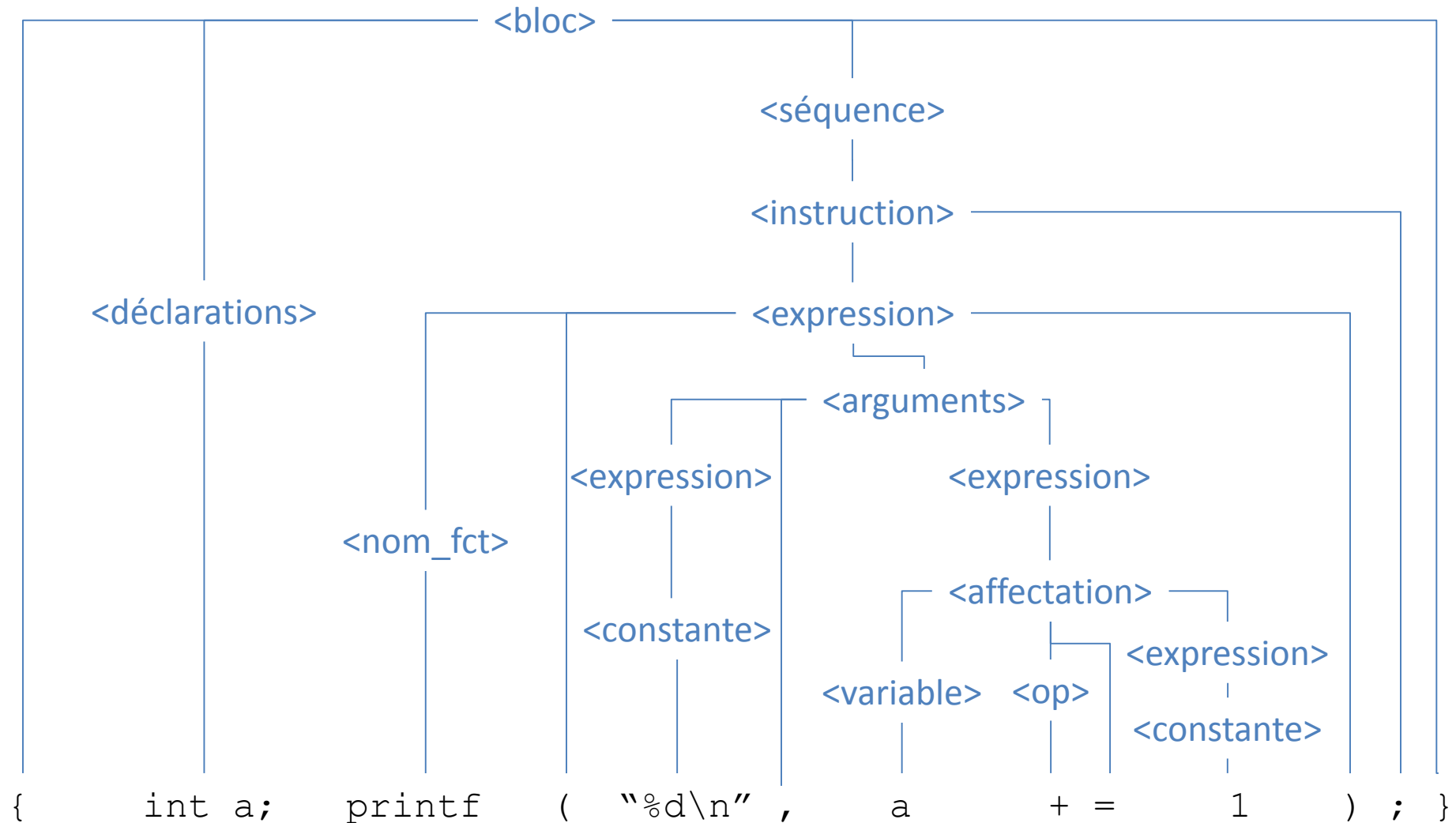
`<arguments>` ::= `[<expression> (\, ' <expression>)*]`

`<affectation>` ::= `<variable> [<op>] \= ' <expression>`
| `<variable> (\++ ' | \-- ')`
| `(\++ ' | \-- ') <variable>`

Les non-terminaux non-définis sont en italique.

Grammaire des programmes C

■ Exemple



Plan du cours : C statique



- Introduction
- Grammaire des programmes C
- Structures de contrôle
- Typage
- Variables et fonctions

Structures de contrôle



■ Définition

- « Instructions permettant le *contrôle de l'ordre l'exécution d'autres instructions* »
- Origine dans les ordinogrammes
 - Représentation graphique du fonctionnement d'un programme
 - Programme = réseau ferroviaire ; instruction = voie ; contrôle = aiguillage
- Deux types de contrôle
 - Structures conditionnelles : choix entre plusieurs possibilité
 - Structures itératives : répétition d'un même traitement

Structures de contrôle

■ Structure conditionnelle « if ... then ... else ... »

□ Syntaxe

`<instruction>` ::= ... | `<if>` | ... à compléter ...

`<if>` ::= `'if'` `'(' <expression> ')'` `<instruction>` [`'else'` `<instruction>`]

□ Sémantique

- *Si* la condition (expression) est vraie *alors* effectuer la première instruction, *sinon* effectuer la seconde instruction
- La branche « sinon » est optionnelle ; en cas d'absence, rien n'est fait

□ Exemple

```
int x, abs_x;  
x = ...  
  
if (x >= 0)  
    abs_x = x;  
else  
    abs_x = -x;
```



```
int x, abs_x;  
x = ...  
  
abs_x = x;  
if (x < 0)  
    abs_x = -x;
```

Structures de contrôle

■ Structure conditionnelle « switch ... case ... »

□ Syntaxe

`<instruction> ::= ... | <switch> | ... à compléter ...`

`<switch> ::= 'switch' '(' <expression> ')' '{' <case>* [<default>] '}'`

`<case> ::= 'case' <constante> ':' <sequence>`

`<default> ::= 'default' ':' <sequence>`

□ Sémantique

- *Choix multiple de traitements* suivant la valeur d'une expression
- **Attention** : les traitements ne sont pas exclusifs (à la différence du if)
Le traitement débute à partir de la première valeur égale ; tous les traitements suivants sont également effectués.
- On peut interrompre la suite des traitements à l'aide du mot clé `break`

Structures de contrôle

■ Structure conditionnelle « switch ... case ... »

□ Exemple 1 : calcul du cardinal d'un jour de l'année

```
int day, month, year, n;
scanf("%d %d %d", &year, &month, &day);
n = day;
switch (month) {
    case 12: n += 30;
    case 11: n += 31;
    case 10: n += 30;
    case 9: n += 31;
    case 8: n += 31;
    case 7: n += 30;
    case 6: n += 31;
    case 5: n += 30;
    case 4: n += 31;
    case 3: if (year%4==0) n += 29; else n += 28;
    case 2: n += 31;
}
printf("%d\n", n);
```

Structures de contrôle

■ Structure conditionnelle « switch ... case ... »

□ Exemple 2 : schéma d'utilisation du `break`

```
int choice;
scanf("menu choice 1).. 2).. 3).. ? %d", &choice);

switch (choice) {
    case 1: ... /* first submenu */; break;
    case 2: ... /* second submenu */; break;
    case 3: ... /* third submenu */; break;
    default: printf("Submenu %d does not exist\n", choice);
}
```

Structures de contrôle

■ Structure itérative « while ... »

□ Syntaxe

`<instruction>` ::= ... | `<while>` | ... à compléter ...

`<while>` ::= `'while'` `'('` `<expression>` `)'` `<instruction>`

□ Sémantique

- *Tant que* la condition (expression) est vérifiée, effectuer le traitement (instruction)
- Le traitement est également appelé le *corps* de la boucle

Structures de contrôle

■ Structure itérative « while ... »

□ Exemple : calcul de $s = \sum_{i=0}^k i \leq n < \sum_{i=0}^{k+1} i$

```
int s, i, n, k;
scanf("%d", &n);
s = i = 1;

while (s <= n) {
    i += 1;
    s += i;
}

s = s-i;
k = i-1;
printf("s = %d, k = %d\n", s, k);
```

Structures de contrôle

■ Structure itérative « do ... while ... »

□ Syntaxe

`<instruction>` ::= ... | `<do_while>` | ... à compléter ...

`<do_while>` ::= `'do'` `<instruction>` `'while'` `'('` `<expression>` `)'`

□ Sémantique

- *Tant que* la comportement équivalent au while, à la différence près qu'il effectue le traitement du corps au moins une fois avant d'évaluer la condition

```
do
  instruction
while (condition)
```



```
instruction
while (condition)
  instruction
```

Structures de contrôle


■ Structure itérative « do ... while ... »

□ Exemple : calcul de $s = \sum_{i=0}^k i \leq n < \sum_{i=0}^{k+1} i$

```
int s, i, n, k;
scanf("%d", &n);
s = i = 0;

do {
    i += 1;
    s += i;
} while (s <= n)

s = s-i;
k = i-1;
printf("s = %d, k = %d\n", s, k);
```



Structures de contrôle

■ Structure itérative « for ... »

□ Syntaxe

`<instruction> ::= ... | <for>`

`<for> ::= 'for' '(' '<expr.>' ';' '<expr.>' ';' '<expr.>' ')' '<instruction>`

□ Sémantique

- Initialise la boucle avec la première expression (*initialisation*)
- Effectue l'instruction (traitement) *tant que* la deuxième expression (*condition*) est vraie
- À chaque fin de traitement, évalue la troisième expression (*incrément*)
- Pratique pour les traitements itératifs avec compteur

```
for (e1; e2; e3)
    instruction
```



```
e1;
while (e2) {
    instruction
    e3;
}
```

Structures de contrôle

■ Structure itérative « for ... »

□ Exemple : calcul de $s = \sum_{i=0}^k i \leq n < \sum_{i=0}^{k+1} i$

```
int s, i, n, k;  
scanf("%d", &n);  
i = 1;  
  
for (s=1; s<=n; s+=i) i++;  
  
s = s-i;  
k = i-1;  
printf("s = %d, k = %d\n", s, k);
```

Plan du cours : C statique

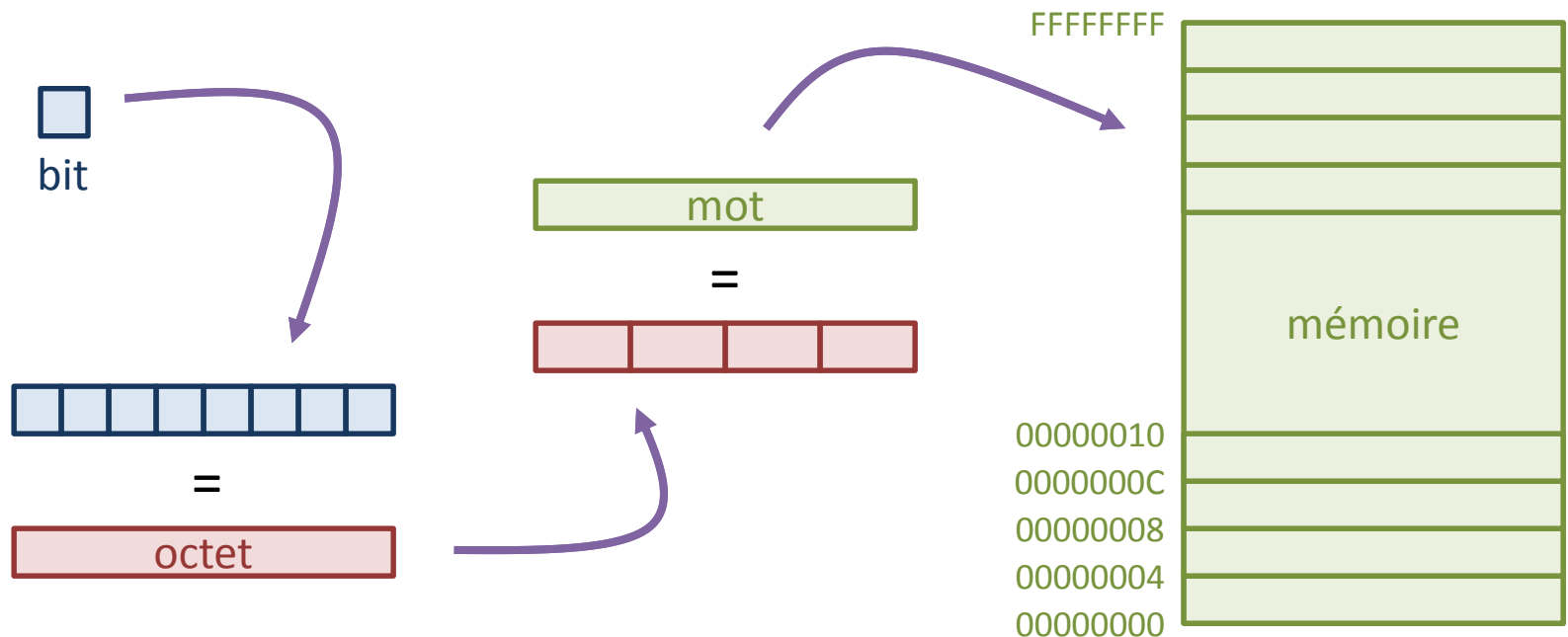


- Introduction
- Grammaire des programmes C
- Structures de contrôle
- Typage
- Variables et fonctions

Présentation du typage

■ Définition

- Au niveau de l'architecture, l'ordinateur manipule uniquement des *mots* composés de 0 et 1. Le typage permet de faire une abstraction sur ces mots suivant l'utilisation que l'on souhaite en faire. Autrement dit, le typage permet de représenter/manipuler différentes sortes de valeurs, toutes représentées de la même façon en interne.



Présentation du typage

■ Définition

- Au niveau de l'architecture, l'ordinateur manipule uniquement des *mots* composés de 0 et 1. Le typage permet de faire une abstraction sur ces mots suivant l'utilisation que l'on souhaite en faire. Autrement dit, le typage permet de représenter/manipuler différentes sortes de valeurs, toutes représentées de la même façon en interne.
- Le langage C fournit
 - Des types simples : valeurs numériques (entières et flottantes), les caractères
 - Des moyens pour la définition de nouvelles structures de données
 - Énumération
 - Structure (enregistrement)
 - Union
 - L'alias
 - Des tableaux

Types simples : valeurs numériques entières

■ Différents types d'entiers selon l'usage

- Occupation en mémoire (nombre d'octets utilisé)
- Signé ou non-signé

type	sizeof	valeur min	valeur max
char	1	-128	127
unsigned char	1	0	255
short	2	-32768	32767
unsigned short	2	0	65535
int	4	-2147483648	2147483647
unsigned int	4	0	4294967295
long	4	-2147483648	2147483647
unsigned long	4	0	4294967295
long long	8	-9223372036854775808	9223372036854775807
unsigned long long	8	0	18446744073709551615

Attention, les valeurs de ce tableau dépendent de votre machine.

Types simples : valeurs numériques entières

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

int main() {
    printf("Size of char: %d\n", sizeof(char));
    printf("Signed char min/max: %d/%d\n", CHAR_MIN, CHAR_MAX );
    printf("Unsigned char min/max: %u/%u\n", 0, UCHAR_MAX );
    printf("Size of short: %d\n", sizeof(short));
    printf("Signed short min/max: %d/%d\n", SHRT_MIN, SHRT_MAX );
    printf("Unsigned short min/max: %u/%u\n", 0, USHRT_MAX );
    printf("Size of int: %d\n", sizeof(int));
    printf("Signed int min/max: %d/%d\n", INT_MIN, INT_MAX );
    printf("Unsigned int min/max: %u/%u\n", 0, UINT_MAX );
    printf("Size of long: %d\n", sizeof(long));
    printf("Signed long min/max: %ld/%ld\n", LONG_MIN, LONG_MAX );
    printf("Unsigned long min/max: %lu/%lu\n", 0, ULONG_MAX );
    printf("Size of long long: %d\n", sizeof(long long));
    printf("Signed long long min/max: %lld/%lld\n", LLONG_MIN, LLONG_MAX );
    printf("Unsigned long long min/max: %llu/%llu\n", 0, ULLONG_MAX );
    return 0;
}
```

Types simples : valeurs numériques entières

■ Utilisation des entiers comme suite de bits

□ Décalage à gauche et à droite : $\langle op \rangle ::= \langle '<<' \mid '>>' \rangle$

$147 \gg 1$
 : 147 \longrightarrow : 73

$147 \ll 4$
 : 147 \longrightarrow : 48

□ Complément à 1 (conversion des 0 en 1 et des 1 en 0) : \sim

~ 147
 : 147 \longrightarrow : 108

□ Opérations booléennes binaires : $\langle op \rangle ::= \dots \mid \langle '&' \mid '|' \mid '^' \rangle \mid \dots$

 : 147

et logique $\&$

 : 204

=

 : 128

 : 147

ou logique $|$

 : 204

=

 : 223

 : 147

ou exclusif \wedge

 : 204

=

 : 95

Types simples : caractères

■ Type char

- Utilisé dans deux contextes différents
 - Représentation des « petits » entiers (entre 0 et 255)
 - Représentation des caractères textuels, notamment pour les caractères alphanumériques et de ponctuation
 - Les caractères sont dénotés entre apostrophes (e.g., '!', '0', 'g', '\n', '\0, ...)
- La table ASCII : bijection entre entiers et caractères (encodage)
 - Les chiffres (exercice : conversion d'une chaîne de caractères en nombre)
 $\text{'0'} \rightarrow 48, \text{'1'} \rightarrow 49, \dots, \text{'9'} \rightarrow 57$
 - Les lettres (exercice : implanter l'ordre lexicographique)
 $\text{'A'} \rightarrow 65, \dots, \text{'Z'} \rightarrow 90 \qquad \text{'a'} \rightarrow 97, \dots, \text{'z'} \rightarrow 122$
 - La ponctuation
 $\text{'\t'} \rightarrow 9, \text{'!'} \rightarrow 33, \text{'.'} \rightarrow 46, \text{'\;'} \rightarrow 59, \dots$
 - Les caractères spéciaux
 $\text{'\0'} \text{ (null)} \rightarrow 0, \text{'\n'} \rightarrow 10, \text{'\r'} \rightarrow 13$

Types simples : valeurs numériques flottantes

■ Différents types de flottants selon l'usage

- Occupation en mémoire (nombre d'octets utilisé)
- Précision de la représentation (en base 10)

type	sizeof	précision	valeur min	valeur max
float	4	6	1.1755e-38	3.4028e+38
double	8	15	2.2251e-308	1.7977e+308
long double	12	18	3.3621e-4932	1.1897e+4932

Attention, les valeurs de ce tableau dépendent de votre machine.

■ Arithmétique sur les flottants

- Normalisation IEEE 754 (ici en 32 bits)



- Valeur = signe * 1, mantisse * 2^(exposant - 127)
- Représentation des infinis, des « *not a number* » (NaN)

Types simples : valeurs numériques flottantes

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

int main() {
    printf("Size of float: %d\n", sizeof(float));
    printf("Min/Max value of a float: %.5g/%.5g\n", FLT_MIN, FLT_MAX);
    printf("Precision of a float: %d digits\n\n", FLT_DIG);

    printf("Size of double: %d\n", sizeof(double));
    printf("Min/Max value of a double: %.5g/%.5g\n", DBL_MIN, DBL_MAX);
    printf("Precision of a double: %d digits\n\n", DBL_DIG);

    printf("Size of long double: %d\n", sizeof(long double));
    printf("Min/Max value of a long double: %.5Lg/%.5Lg\n", LDBL_MIN,
                                                    LDBL_MAX);
    printf("Precision of a long double: %d digits\n\n", LDBL_DIG);

    return 0;
}
```

Types simples : valeurs numériques flottantes

■ Quelques opérations sur les flottants

□ Utilisation de la librairie mathématique standard de C

- Dans le code : `#include <math.h>`
- Pour compiler : `gcc helloworld.c -o helloworld.exe -lm`

□ Constantes

- `M_E M_LOG2E M_LOG10E M_LN2 M_LN10`
- `M_PI M_TWOPI M_PI_2 M_PI_4 M_3PI_4 M_SQRTPI M_1_PI M_2_PI`
- `M_SQRT2 M_SQRT1_2 M_2_SQRTPI M_SQRT3`
- ...

□ Fonctions

- `fmax fmin floor ceil ...` (et leurs homologues sur les float)
- `cos sin tan atan ...` (et leurs homologues sur les float)
- `exp log sqrt pow ...` (et leurs homologues sur les float)
- ...

Types simples : valeurs numériques

■ Opérateurs arithmétiques

- Les quatre opérations de base : `<op> ::= ... | '+' | '-' | '*' | '/' | ...`
- Le modulo (reste par la division euclidienne) : `<op> ::= ... | '%'`

Ne fonctionne que pour les valeurs entières

- Les opérateurs arithmétiques d'affectation
 - Ils *effectuent une opération et affectent une variable*
 - La suite des opérateurs `<op> '='` (e.g., `+='`, `*=`, ...)

```
int x = 0;  
x += 10;
```



```
int x = 0;  
x = x + 10;
```

- Les opérateurs d'incrément `++` et `--`

```
void main() {  
    int a1, a2, b1, b2;  
    b1 = b2 = 2; a1 = a2 = 1;  
    b1 += (++a1); printf("a1=%d, b1=%d\n", a1, b1); /* a1=2, b1=4 */  
    b2 += (a2++); printf("a2=%d, b2=%d\n", a2, b2); /* a2=2, b2=3 */  
}
```


Types simples : valeurs booléennes

■ Il n'existe pas de type booléen (vrai/faux) en C

Les valeurs numériques servent de valeurs booléenne (par convention)

- 0 vaut *faux*
- Tout entier non-nul est évalué à *vrai*

■ Les expressions conditionnelles

- Les opérateurs *relationnels* : `'=='` | `'!='` | `'<'` | `'>'` | `'<='` | `'>='`
- Les opérateurs *logiques* : `'&&'` | `'||'` | `'!'`

■ Tables de vérité

&&	0	1
0	0	0
1	0	1

	0	1
0	0	1
1	1	1

!	
0	1
1	0

- Évaluation *non-strict* (second membre non-évalué si ce n'est pas nécessaire)

```
int num = ..., den = ..., res;  
div = (den != 0) && ((res = num / div) || 1);;
```

Types simples : conversion de types

■ Conversion d'une valeur d'un type vers un autre

□ Conversion *automatique* sur les types simples

- Entre 2 valeurs flottantes : conversion vers la représentation la plus grande

`float` → `double` → `long double`

- Entre une valeur flottante et une valeur entière : conversion en flottant

- Entre 2 valeurs entières : conversion vers la représentation la plus grande

`char` → `short` → `int` → `long` → `long long`

□ Conversion explicite : le *cast*

- `<expression> ::= ... | '(' <type> ')' <expression>`

- Exemples

□ Calcul de l'arrondi à l'entier le plus proche

```
int round; float real = ...;
round = (int)real + (real - ((int)real) >= 0.5);
```

□ Calcul du ratio d'une image

```
float ratio; int width = ..., height = ...;
ratio = (float)width / (float)height;
```

Types dérivés

■ Comment représenter avec les types simples ?

- Un dictionnaire
- Une liste d'entiers (dans le sens de SML)
- Un jeu de carte
- ...

■ Réponse : définir des structures de données

- Au même niveau que la définition des fonctions
- Nouveau non-terminal

`<type_dec> ::= <enum> | <struct> | <union> | <alias>`

■ Trois sortes de types dérivés

- Les *énumérations*
- Les *enregistrements*
- Les *unions*

Types dérivés : énumération

■ Utilisation

Représenter un ensemble fini de symboles

■ Syntaxe

```
<enum> ::= 'enum' <name> '{' <enum_field> '}'
```

```
<enum_field> ::= (<name> ['=' <entier>] ',')* <name> ['=' <entier>]
```

■ Représentation mémoire

- Chaque symbole est associé à un entier

■ Exemple : structure de données pour un jeu de tarot

- Représentation des cartes d'atouts
- Représentation des couleurs
- Représentation des valeurs



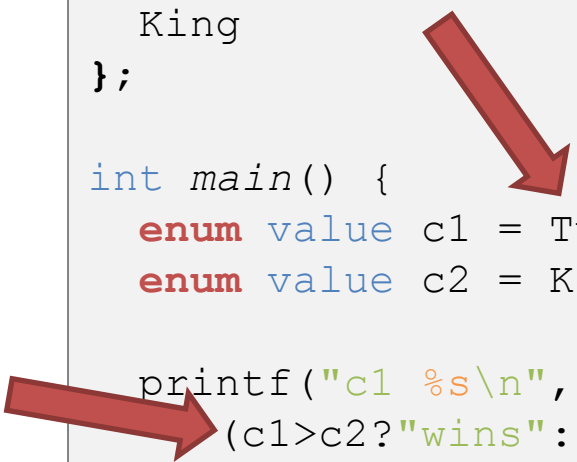
Types dérivés : énumération

■ Exemple : structure de données pour un jeu de tarot

```
enum trump {  
    Fool = 0,  
    OneOfTrump = 1,  
    TwoOfTrump = 2,  
    ...  
    TwentyOneOfTrump = 21  
};
```

```
enum suit {  
    Spades,  
    Hearts,  
    Diamonds,  
    Clubs  
};
```

```
enum value {  
    Ace = 1,  
    Two,  
    ...  
    Ten,  
    Jack,  
    Knight,  
    Queen,  
    King  
};  
  
int main() {  
    enum value c1 = Two;  
    enum value c2 = King;  
  
    printf("c1 %s\n",  
           (c1>c2?"wins":"looses"));  
} /* c1 looses */
```



Types dérivés : énumération

■ Exemple : structure de données pour un jeu de tarot

```
void print_trump(enum trump c) {  
    switch (c) {  
        case Fool: printf("fool"); break;  
        default:   printf("%d of trump", c);  
    }  
};
```

Types dérivés : enregistrement

■ Utilisation

Produit cartésien de différentes valeurs (type produit)

■ Syntaxe

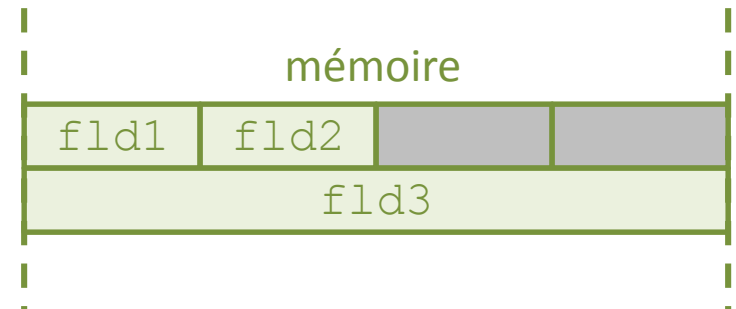
`<struct> ::= 'struct' <name> '{' <struct_field> '}'`

`<struct_field> ::= (<type> <name> ';')*`

■ Représentation mémoire

- Concaténation des différentes valeurs en mémoire
- Attention à l'alignement !

```
struct record {  
    char fld1;  
    char fld2;  
    int  fld3;  
};
```



`sizeof(struct record) = 8` et non 6 (1+1+4)

Types dérivés : enregistrement

■ Exemple : structure de données pour un jeu de tarot

```
void print_card(struct card c) {  
    char* color;  
    switch (c.color) {  
        case Spades : color = "spades"; break;  
        case Hearts  : color = "hearts"; break;  
        case Diamonds: color = "diamonds"; break;  
        case Clubs   : color = "clubs";  
    }  
    switch (c.rank) {  
        case Ace:      printf("ace of %s", color); break;  
        case King:     printf("king of %s", color); break;  
        case Queen:    printf("queen of %s", color); break;  
        case Knight:   printf("knight of %s", color); break;  
        case Jack:     printf("jack of %s", color); break;  
        default:       printf("%d of %s", c.rank, color);  
    }  
};
```

```
struct card {  
    enum suit color;  
    enum value rank;  
};
```



Types dérivés : union

■ Utilisation

Alternative entre différents types de valeurs (type somme)

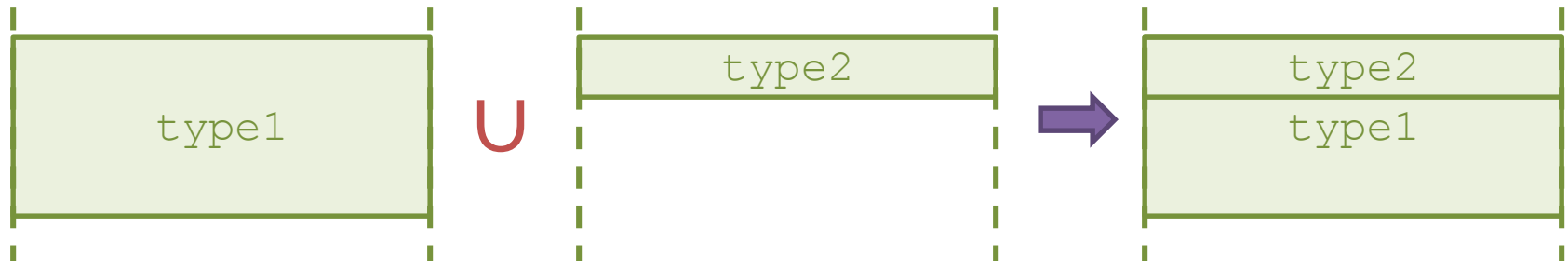
■ Syntaxe

```
<union> ::= 'union' <name> '{' <union_field> '}'
```

```
<union_field> ::= (<type> <name> ';' )*
```

■ Représentation mémoire

- ❑ Superposition des différentes possibilités
- ❑ La place mémoire prise est la taille maximale des alternatives



```
sizeof(union type12) == max(sizeof(type1), sizeof(type2))
```


Types dérivés : union

■ Exemple : structure de données pour un jeu de tarot

```
union tarot_kind {  
    enum trump trump;  
    struct card usual;  
};
```

```
struct tarot_card {  
    int kind;  
    union tarot_kind card;  
};
```

```
void print_tarot_card(struct tarot_card c) {  
    switch (c.kind) {  
        case 1: print_card(c.card.usual); break;  
        case 2: print_trump(c.card.trump); break;  
        default: printf("Unknown kind of card\n");  
    }  
};
```



Alias de type

■ Simplifier les écritures

Éviter l'utilisation permanente des mots clés **enum**, **struct**, **union**

■ Syntaxe

`<alias> ::= 'typedef' (<type> | <type_dec>) <name>`

■ Représentation mémoire

□ Il s'agit juste d'un renommage

■ Exemple

```
typedef struct tarot_card_st {  
    int          kind;  
    union tarot_kind card;  
} tarot_card;
```


Les tableaux

- Globalement, en-dehors du contexte de ce cours...
- En se restreignant au langage C statique

Les tableaux sont de taille fixe

```
typedef tarot_card deck[78];
```

```
void init_deck(deck d) {  
    int r, c, t, i = 0;  
  
    for(c = Spades; c <= Clubs; c++)  
        for(r = Ace; r <= King; r++, i++) {  
            d[i].kind = 1;  
            d[i].card.usual.color = c;  
            d[i].card.usual.rank = r;  
        }  
    for(t = Fool; t <= TwentyOneOfTrump; t++, i++) {  
        d[i].kind = 2;  
        d[i].card.trump = t;  
    }  
}
```



Plan du cours : C statique



- Introduction
- Grammaire des programmes C
- Structures de contrôle
- Typage
- Variables et fonctions

Variables locales

■ Définition des variables

- Effectuer dans la partie <déclarations> d'un <bloc>
- Grammaire associée

```
<bloc> ::= '{' [<déclarations>] <séquence> '}'  
<déclarations> ::= (<déclaration> ';' )*  
<déclaration> ::= ['static'] <type> <variable> (',' <variable>)*  
<type> ::= 'int' | 'float' | 'double' | etc.  
          | ['enum' | 'struct' | 'union'] <name>  
<variable> ::= <name> ['[' <entier> ']]'
```

■ Portée d'une variable

- Une variable est définie pour le bloc courant et tous ses sous-blocs
- La définition d'une variable peut être cachée par une redéfinition dans un sous-bloc

Variables locales

■ Exemple

- Comprenez ce code et donnez la valeur des variables lors de l'exécution

```
{ /* bloc niveau 0 */
  int a, x;
  x = 0; a = x + 1;
  { /* bloc niveau 1 */
    int x, y;
    y = 5; x = 1; a = x + 1;
  }
  { /* bloc niveau 1 */
    int x, b;
    x = 2; a = x + 1; b = y + 2;
    { /* bloc niveau 2 */
      int z;
      z = 5; x = z + 1;
    }
  }
  a = x + 1;
}
```

Variables locales

■ Variables statiques

- ❑ Faire en sorte de se rappeler la valeur d'une variable entre plusieurs visites du bloc
- ❑ La portée de la variable reste la même (*i.e.*, sa valeur est indéfinie en dehors du bloc) ; la seule différence est que sa valeur n'est pas réinitialisée
- ❑ Exemple : nombre de fois qu'une fonction est appelée

```
int get_counter() {  
    static int cntr = 0;  
    return cntr++;  
}  
  
int main() {  
    printf("%d ", get_counter());  
    printf("%d ", get_counter());  
    printf("%d ", get_counter());  
    printf("%d\n", get_counter());  
} /* 0 1 2 3 */
```


Fonctions

■ Définition d'une fonction

- Une fonction peut être déclarée dans n'importe quelle *<déclarations>*
En générale, elles sont déclarées au plus niveau pour que leur portée atteigne tous les sous-blocs (notamment toutes les fonctions déclarées ensuite)
- Une fonction peut être soit *définie*, soit uniquement *déclarée* (*prototypée*)
Voir le cours sur la compilation séparée
- Les *arguments* sont vus comme des *variables locales* du corps
- La fonction est définie dans son propre corps (appel *récuratif*)
- L'instruction *return* permet de spécifier la valeur retournée
Elle doit vérifier le type retour de la fonction
- Grammaire

```
<déclaration> ::= ... | <type> <name> `(` <args_dec> `)` `(` ';' | <bloc> )  
  <args_dec> ::= [ <arg> ( ',' <arg> )* ]  
    <arg> ::= <type> <variable>  
  <instruction> ::= ... | `return' <expression> `;'`
```

Fonctions

■ Appel à une fonction

- Grammaire (déjà vue plus haut)

`<expression> ::= ... | <name> ' (' <arguments> ') ' '`

`<arguments> ::= [<expression> (' , ' <expression>) *]`

- Stratégie d'appel par valeur

- Évaluation des arguments

L'ordre d'évaluation n'est pas défini (e.g., `printf("%d,%d\n", ++i, i--);`)

- Copie des valeurs des arguments dans la pile d'exécution
- Appel du code de la fonction
- Dépilement des arguments
- Lecture du résultat

- Principale conséquence

Une modification de la valeur d'un argument est locale à la fonction

Fonctions

■ Exemples

- Fonction pour incrémenter une variable de 6 (**mauvaise solution**)

```
void plus_six(int n) {  
    n += 6;  
}
```

```
void main() {  
    int x = 3; plus_six(x);  
    printf("x = %d\n", x);  
} /* x = 3 */
```

- Fonction pour incrémenter une variable de 6 (**bonne solution**)

```
void plus_six(int *n) {  
    *n += 6;  
}
```

```
void main() {  
    int x = 3; plus_six(&x);  
    printf("x = %d\n", x);  
} /* x = 9 */
```

- L'exception : les tableaux ! (qui, en fait, n'en est pas une...)

```
void plus_six(int t[1]) {  
    t[0] += 6;  
}
```

```
void main() {  
    int x[1]; x[0] = 3; plus_six(x);  
    printf("x = %d\n", x[0]);  
} /* x = 9 */
```