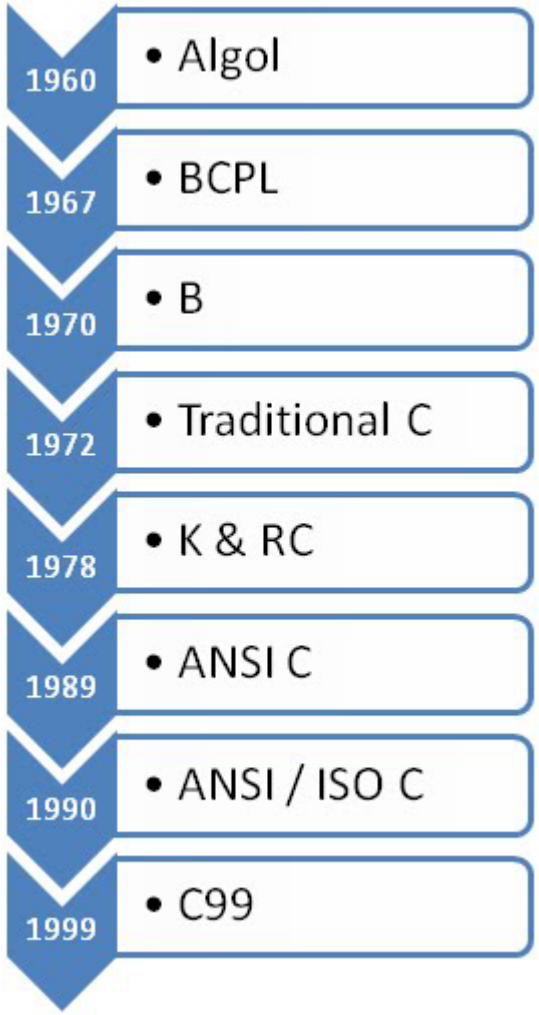
Chapitre 1
Introduction

Une brève histoire du C

1972 à aujourd'hui

- L'histoire du C est liée à celle d'Unix
- Les deux sont issus des Bell Labs
- Standardisé par l'ANSI en 1989
- La dernière norme est C17 (après C11)
- C17 corrige des problèmes de C11 mais n'introduit pas de réelles nouveautés



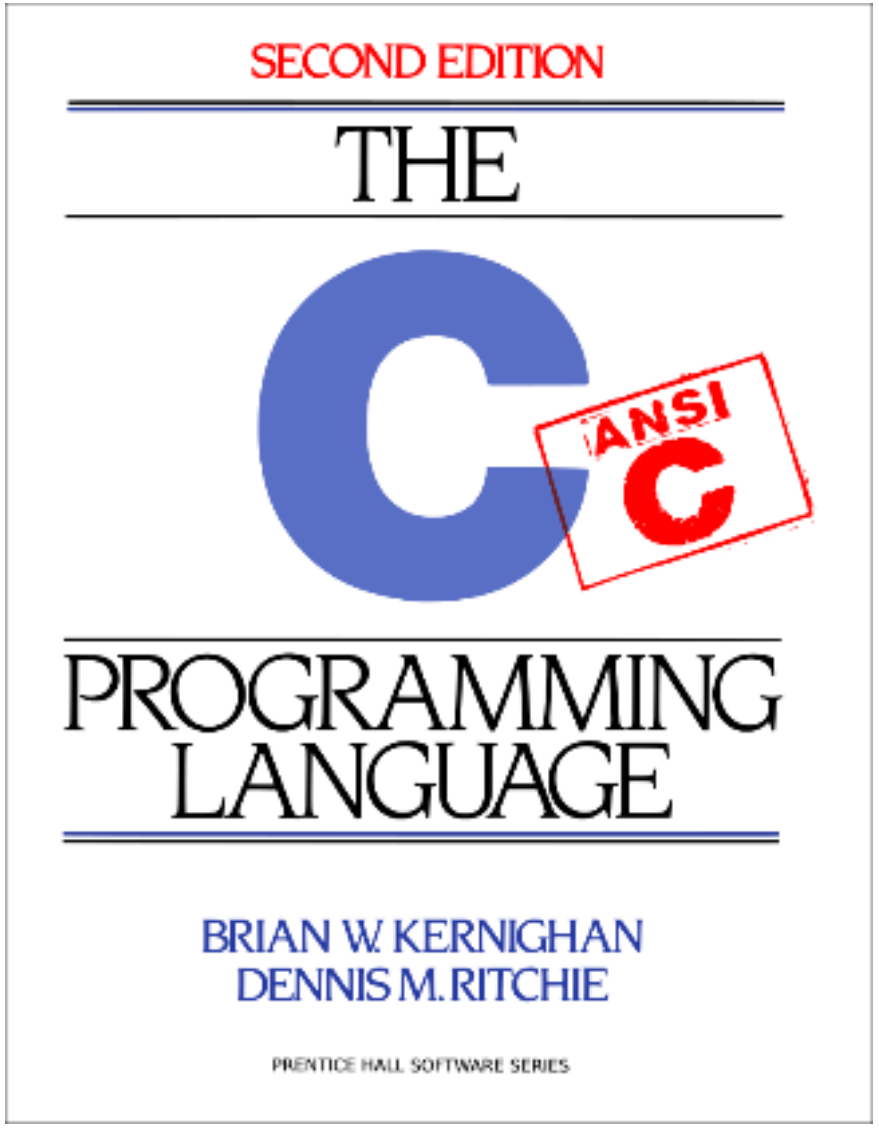
Pour plus de détails

Olve Maudal - *Histoire et esprit de C et C++*

https://vimeo.com/132192250

Les « inventeurs » du C

- Dennis M. Ritchie (1941 – 2011)
- Brian W. Kernighan (1942 –)



Plan du chapitre 1

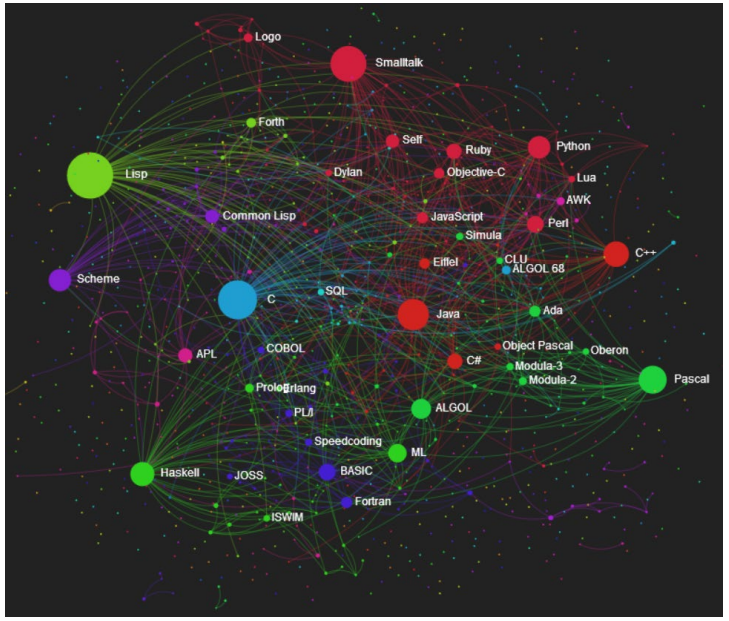
1. Motivation du C et comparaison avec C++ [5-14]
2. Le préprocesseur [15-35]
3. Les types de base [36-44]
4. Les opérateurs [45-57]
5. Les entrées / sorties avec <stdio.h> [58-76]
6. Résumé [77-81]

1. Motivations du C et comparaison avec C++

Pourquoi apprendre le C aujourd'hui ?

- Langage avec un long passé qui ne disparaîtra pas
- Langage disponible sur beaucoup de plateformes.
  « S'il existe un compilateur, ce sera pour du C »
- Beaucoup de librairies et d'outils sont disponibles
- D'autres langages appellent des librairies C
- OS écrits surtout en C (et assembleur)

❖ C vous laisse le contrôle !



Influence du C sur C++, Java…

Sources : http://exploring-data.com/vis/programming-languages-influence-network

Avantages du C

Le C est généralement utilisé pour obtenir un contrôle particulier de la machine et/ou des bonnes performances

- Systèmes embarqués
- Hautes performances recherchées
- Systèmes d'exploitation (noyaux)
- Accès matériel (pilotes de périphériques, …)
- Sous-systèmes critiques : fichiers, réseaux, vidéo…
- Compilation des langages de programmation

Caractéristiques essentielles de C

- Langage de programmation procédurale, de bas niveau, mais plus facile à utiliser que l'assembleur
- Les exécutables sont très compacts et presque aussi rapide que l'assembleur
- Le programmeur dispose d'un grand contrôle sur l'exécution
  - *… en supposant que vous savez ce que vous faites !!*
- Peu de structures de données et d'algorithmes sont fournis dans la bibliothèque standard du C

➢ L'efficacité est plus grande, le risque de bugs aussi !

Comparaison de C et C++

Bonnes nouvelles

- Le C++ est basé sur le C, vous connaissez donc déjà un peu le C !

  Types de base, sélections, itérations, fonctions …
- Tous les opérateurs du langage C sont valides en C++
- Les expressions sont les mêmes en C++ et C

… mais !

- C++ et C sont semblables mais pas identiques !
- De **nombreux** mots-clés du C++ n'existent pas en C

  `class`, `namespace`, `reference (&)`, `try/catch`, `template`, `vector`, `stream`, `string`, ...
- … ou parfois ont des sens différents en C et C++ …
- Les surcharges de fonctions ou d'opérateurs ne sont pas possibles

Sources : https://en.wikipedia.org/wiki/Compatibility_of_C_and_C++

« *Hello world* » en C

.cpp => .c

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

int main(void)
{
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```

Fichier .h

void obligatoire

Pas de cout << mais printf

Différences entre C et C++

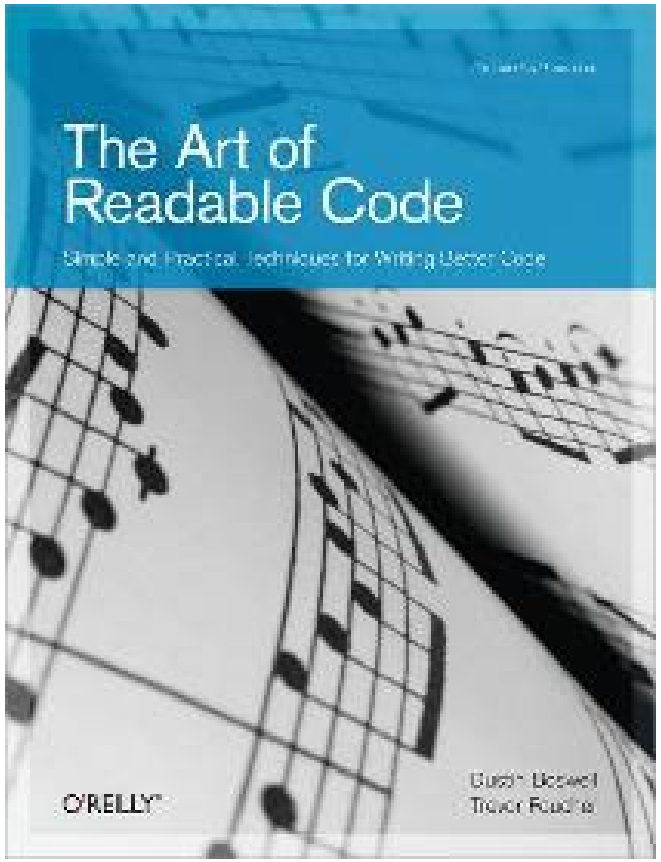
Eléments	C	C++
Compilateur	gcc : C uniquement	g++ : C et C++
Paradigme	Procédural uniquement	Procédural et orienté objet
Entrées / Sorties	Typiquement `scanf` et `printf`	Flux `cin`, `cout` et `cerr` avec les opérateurs de flux `<<` et `>>`
Surcharge	Non	Oui pour les opérateurs et fonctions
Fonction membre	Non, même dans les structures	Oui dans les structures et les classes
Passage de paramètres	Par copie et par adresse	Par copie, adresse ou référence
Gestion dynamique de la mémoire	Allocation : `malloc()` et `calloc()` Libération : `free()`	Allocation : `new` Libération : `delete`
Gestion d'exceptions	Non	Oui avec les blocs `try/catch`
Généricité	Non	Oui
Espace de nommage (namespace)	Non	Oui

Avantages du C

- Rapide à l'exécution, économe en mémoire
- Portable : souvent sur des plateformes « rares », le seul outil disponible est un compilateur C
  - Plus facile que de devoir écrire en assembleur…
  - Le code C standard pourra compiler partout
- Puissant : langage simple mais tout de même expressif
- Flexible : comme l'atteste son usage très répandu
  - utilisé même pour de grands projets

Points faibles du C

- Les programmes en C peuvent :
  - générer des erreurs d'exécution
  - parfois être difficiles à comprendre
  - être difficiles à modifier / mettre à jour
  - ❖ *Dépend beaucoup du programmeur…*



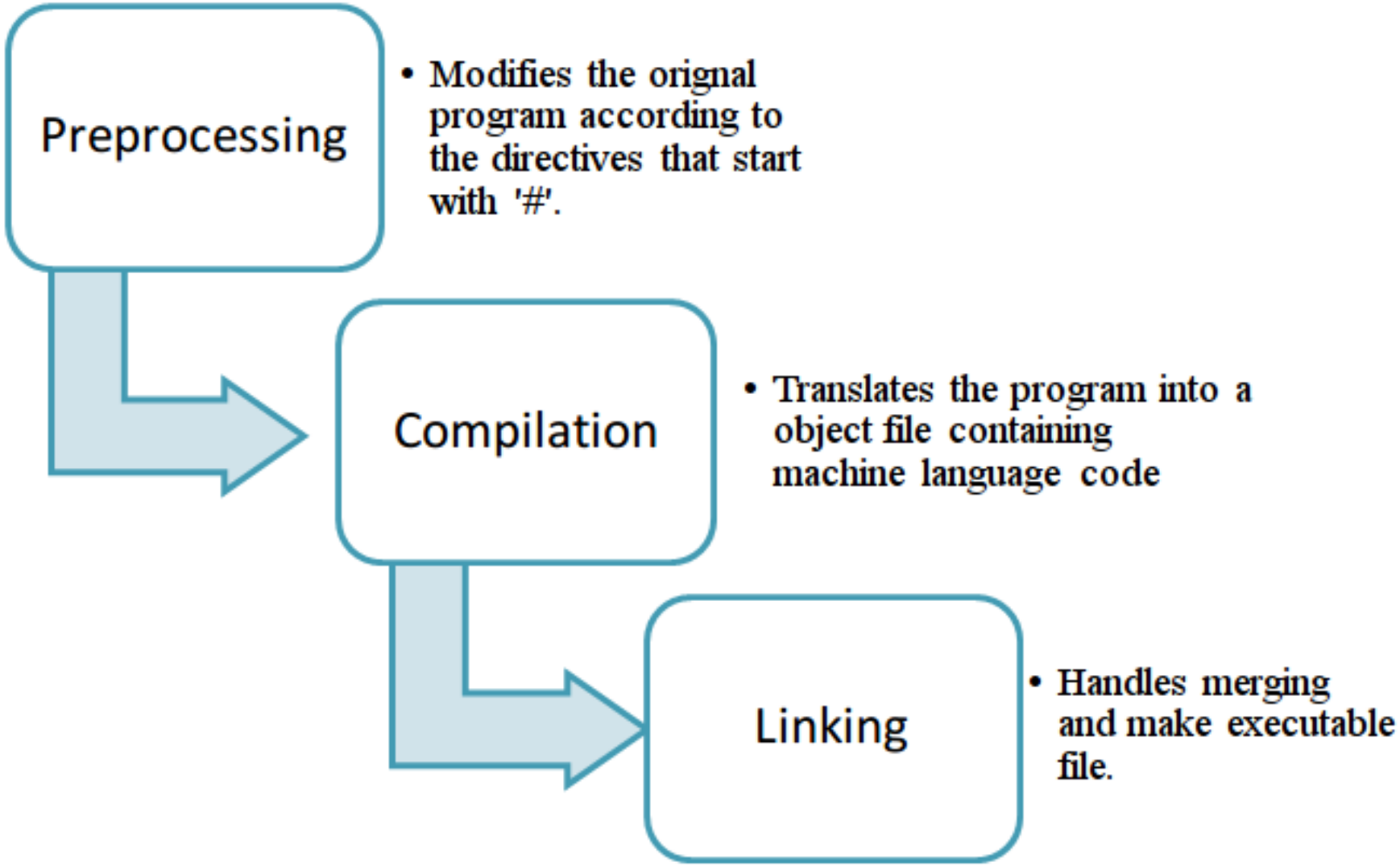
- *The International Obfuscated C Code Contest* ☺
  - Flight simulator 1998 http://www.ioccc.org/1998/banks.c
  - Best use of a flag 2000 http://www.ioccc.org/2000/anderson.c

Comment bien faire

- Adopter des règles de rédaction cohérentes
- Compiler avec des options strictes de compilation pour améliorer la fiabilité
- Utiliser des librairies existantes et répandues
- Éviter les astuces et les codes trop complexes
- Respecter la norme
- Tester !!!

2. Le préprocesseur

Création d'un exécutable (rappel)



Sources : https://www.cplusplus.com/articles/2v07M4Gy

Attention !

- Les erreurs causées par le préprocesseur peuvent être difficiles à localiser et à corriger (localisées dans un code généré automatiquement)
- Principales difficultés
  - effets de bord imprévus (on y reviendra)
  - pas de numéros de ligne pour les erreurs
  - difficile d'insérer des tests ou des affichages
- Pour conserver le fichier après l'action du préprocesseur

```
gcc -E fichier.c -o fichier.i
```

Les directives du préprocesseur

- Les directives commencent par le symbole #
- Elles servent à
  - inclure un fichier externe
  - définir une constante
  - définir une macro
  - rendre la compilation conditionnelle

- *Note* : le préprocesseur fait plus qu'exécuter des directives
  - efface tous les commentaires (→ 1 espace)
  - supprime les espaces inutiles (débuts de ligne, indentation, …)

Rappel : #include

- #include insère à l'endroit de la commande le contenu d'un fichier (en général des librairies)

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

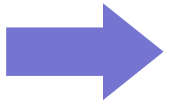
- Sémantique des délimiteurs
  - <nom> : chercher le fichier dans les dossiers système
    - afficher les dossiers connus : `gcc -v foo.c -Wl, --verbose`
    - on peut également ajouter d'autres dossier avec le flag `-I dossier`
  - "nom" : chercher dans le dossier courant
  - librairies C avec .h (sans extension en C++)

Définition d'une macro avec #define

- `#define` crée une macro
  qui est l'association entre un identificateur et sa définition, typiquement un texte de remplacement ou plus simplement une valeur
- Le préprocesseur répond à une directive #define en stockant l'identificateur de la macro et sa définition
- Lorsque la macro est utilisée, le préprocesseur développe la macro en remplaçant son identificateur par sa définition

```
#define CSTE 20

int main(void) {
   int i = CSTE;
   ...
}
```



```
int main(void) {
   int i = 20;
   ...
}
```

Syntaxe des directives `#define`

- Une macro est formée de « mots » ou *tokens*, séparés par des espaces ou tabulations (peu importe le nombre)
- Une macro se termine au premier retour à la ligne, mais
- … un « \ » indique qu'elle se poursuit sur la ligne suivante (améliore la lisibilité)
- On peut utiliser des commentaires avec // (tout le reste de la ligne est un commentaire)
- Lorsque toutes les valeurs sont connues, les expressions peuvent être pré-calculées

```
#define DISK_CAPACITY (SIDES *              \
                       TRACKS_PER_SIDE *    \
                       SECTORS_PER_TRACK * \
                       BYTES_PER_SECTOR)
```

Avantages de `#define` en C

- Rendre les programmes plus faciles à lire
  - permet de nommer une expression, lui donne un sens
- Rendre les programmes plus faciles à modifier
  - la valeur d'une constante est mise à jour à un seul endroit
- Éviter les incohérences et les erreurs en centralisant les définitions
  - garder *PI* identique partout (et non 3.1416 parfois)

```
#define PI 3.14159265358979323846
double  p = 2.0 * PI * r;
```

Pas de «;»
… sinon ?

- Les déclarations de "constantes" restent recommandées `const int CSTE = 20;`

Macro avec paramètres

- Macro définie par `identificateur(x1, x2, ..., xn)`
- À chaque invocation de la macro par `identificateur(y1, y2, ..., yn)` le préprocesseur remplace le nom par la définition et dans celle-ci `x1` par `y1`, `x2` par `y2`, etc…

```
#define min(a,b) ((a) < (b) ? (a) : (b))
int main(void) {
   return min(3, 5);
}
```

- NB : pas d'espace entre le nom et le(s) paramètre(s) formel(s) de la macro

```
#define min(a,b) ((a) < (b) ? (a) : (b))
```

- Même rôle que les fonctions *inline* en plus «générique»

Attention aux effets de bord !

- Penser au résultat réel du remplacement

```
#define DOUBLE(a) a * 2
#define min(a, b) ((a) < (b) ? (a) : (b))

int main(void) {
   int i = 0;
   int j = 2;
   int a = DOUBLE(i + j); // a = i + j * 2 => priorité des opérateurs ?
   int b = min(i++, j++); // b = (i++) < (j++) ? (i++) : (j++) => i++ 2x
   return EXIT_SUCCESS;
}
```

- La priorité des opérateurs peut être gérée par l'ajout de parenthèses

```
#define DOUBLE(a) ((a) * 2)
int a = DOUBLE(i + j); // a = (i + j) * 2
```

L'opérateur # dans les macros

- Dans une macro, l'opérateur # peut précéder l'un des paramètres de la macro : *stringizing / stringification*
- Il indique que le paramètre doit apparaître sous forme de chaîne ("...") lorsque le corps de la macro sera remplacé
- Il est souvent combiné avec des chaînes constantes, sachant que juxtaposer des chaînes revient à les concaténer

```
#define VIEW_VAR(a) printf(#a " = %d\n", (a))

int main(void) {
   int i = 5;
   VIEW_VAR(i);
}
```

```
i = 5
```

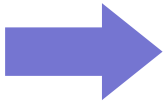
Pratique pour afficher le nom de la variable

L'opérateur ## dans les macros

- ## : concaténation de deux *tokens* dans une macro
  - pour des paramètres ou des constantes littérales

```
#define MK_ID(n) i##n

int main(void) {
   int MK_ID(1), MK_ID(2);
   // ...
}
```



```
int main(void) {
   int i1, i2;
   // ...
}
```

Généricité avec

- Permet d'obtenir un effet semblable aux templates en C++
- Par exemple, définir le `template` d'une fonction `somme` que l'on pourra instancier par appels explicites
  - le paramètre de la macro joue le rôle de `typename`

```
#define GENERIC_SOMME(type)                    \
type somme_##type(type lhs, type rhs) {  \
    return lhs + rhs;                          \
}
```

```
// avant main
GENERIC_SOMME(int)
GENERIC_SOMME(float)
```

Pas de ;

```
int   somme_int  (int   lhs, int   rhs) { return lhs + rhs; }
float somme_float(float lhs, float rhs) { return lhs + rhs; }
```

```
int i = 2;
int a = somme_int(i++, 3 * 4); // => a = 14
```

- Permet d'éliminer les effets de bord

do {…} while (0) dans les macros

- Si on veut écrire une macro plus longue et l'utiliser comme fonction, on l'encapsule souvent dans un bloc **do {…} while (0)**
- Cette technique permet de garantir que la macro sera correctement développée dans tous les contextes

```
#define PROCESS(a) do { \
/* do something */      \
} while (0)

int main(void) {
   int a;
   PROCESS(a);
   return EXIT_SUCCESS;
}
```

Détails https://stackoverflow.com/questions/154136/why-use-apparently-meaningless-do-while-and-if-else-statements-in-macros

Propriétés des macros avec #define

- Les macros ne suivent pas les règles normales de portée telle que étudiées pour les identificateurs
- Une macro définie reste normalement en vigueur jusqu'à la fin du fichier dans lequel elle apparaît
- Si une macro est redéfinie
  - Le compilateur signale un warning
  - Le préprocesseur ne conserve que la dernière version
- Les définitions de macros peuvent être supprimées par la directive `#undef`

```
#define MSG "Hello World\n"
printf(MSG);

#define MSG "Bonjour à tous\n"
printf(MSG);   // => macro redefined

#define MSG
printf(MSG);   // => Too few argument

#undef MSG
printf(MSG);   // => Undeclared identifier
```

Quelques macros standard

- Les macros `__DATE__` et `__TIME__` identifient quand un programme a été compilé en dernier
- Les macros `__LINE__` and `__FILE__` identifient des endroits du code
- `__STDC__VERSION__` fournit un moyen de vérifier quelle version de la norme C est appliquée par le compilateur.
  - pour C99, elle vaut 199901L
  - pour C11, elle vaut 201112L
  - Pour C17, elle vaut 201710L
- L'identificateur `__func__` est une chaîne qui contient le nom de la fonction qui est en cours d'exécution

Exemple de traçage

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

#define TRACE() \
printf("Trace point in %s:%d %s\n", __FILE__, __LINE__, __func__)

void f(const char* msg) {
   printf(msg);
   TRACE();
}

int main(void) {                // output
   f("Oops\n");                 // Oops
   return EXIT_SUCCESS;         // Trace point in /PRG2/main.c:9 f
}
```

Directives de compilation conditionnelle

- Une partie conditionnelle commence par les directives `#if`, `#ifdef` ou `#ifndef`
- … et se termine par `#endif`
- Le code ainsi délimité n'est compilé que si la condition est vraie
- Les conditions doivent être évaluables statiquement (càd à la compilation)

```
#include <stdio.h>
#define VRAI 1

int main(void) {
   #if 1
   printf("1\n");      // 1
   #endif

   #ifdef VRAI // ou #if defined(VRAI)
   printf("VRAI\n");  // VRAI
   #endif

   #ifndef FAUX
   printf("FAUX\n");  // FAUX
   #endif
}
```

Directives de compilation conditionnelle

- Les directives conditionnelles `#if`, `#ifdef` ou `#ifndef` peuvent naturellement être complétées par `#else` ou `#elif`
- Les directives conditionnelles peuvent également être imbriquées

```
#include <stdio.h>
#define TEMPERATURE 10

int main(void) {

   #ifdef TEMPERATURE
      #if TEMPERATURE < 0
         printf("froid");
      #elif TEMPERATURE < 40
         printf("tiede"); // tiede
      #else
         printf("chaud");
      #endif
   #else
      printf("non defini");
   #endif
}
```

Directives de compilation conditionnelle

- Il peut être utile de tester non seulement l'existence d'une macro mais d'évaluer son contenu
- Les expressions peuvent utiliser les opérateurs de comparaison
  < <= == != >= >
- … et les opérateurs logiques && ||

```
// messages_EN.h
#define MSG "Hello"
```

```
// messages_FR.h
#define MSG "Bonjour"
```

```
#define VERSION   3
#define EN        1
#define FR        2
#define LANG     EN

#if !defined(VERSION) || VERSION < 3
   #error Version 3 required
#elif LANG == EN
   #include "messages_EN.h"
#elif LANG == FR
   #include "messages_FR.h"
#endif

printf(MSG);
```

Directives de compilation conditionnelle

En complément

- `#error` stoppe le processus de compilation
- La directive `#pragma` permet de donner certaines instructions au compilateur
- Le prédicat `assert()` dans le header <assert.h> permet de vérifier des conditions requises mais à l'exécution du code comme vu en PRG1

```
#if defined(WIN32)
   #error Win32 is unsupported
#elif defined(MAC_OS)
   ...
#elif defined(LINUX)
   ...
#else
   #error No operating system
#endif
```

3. Les types de base

Les types de base (comme en C++)

- Types entiers : normaux, longs, courts – signés ou non
  - `sizeof(short) <= sizeof(int) <= sizeof(long)`
  - le *header* `<limits.h>` définit des macros qui retournent les valeurs les plus petites/grandes de chaque type entier
  - C99 fournit deux types entiers standard supplémentaires : `long long int` et `unsigned long long int` (au moins 64 bits)
- Types flottants : `float`, `double`, et `long double`
- Caractères : `char` (ASCII et ISO-LATIN-1 sur 8 bits)
- `_Bool` (C99) ou `bool` dans `<stdbool.h>`

Conversion de types

- Pour effectuer une opération arithmétique, les opérandes doivent généralement avoir la même taille (en bits) et être stockés de la même manière
- Les compilateurs peuvent modifier les types de certains opérandes afin que le processeur puisse évaluer l'expression (*promotion numérique ou ajustement de type*)
- Les promotions de type en C convertissent en int (ou unsigned int) tout entier de taille inférieure à int (ou unsigned int)
- Les conversions explicites en C se font par l'opérateur de *cast* `(int)3.14`
  La forme fonctionnelle n'existe pas en C

Depuis C99 : `stdint.h`

- Le header `stdint.h` définit (grâce à `typedef`) des types entiers occupant un nombre précis de bits

- Types entiers standard les plus répandus
  - `signed` `int8_t, int16_t, int32_t, int64_t`
  - `unsigned` `uint8_t, uint16_t, uint32_t, uint64_t`

- Types entiers rapides (*fast types*) : garantissent un minimum de bits, mais le stockage sous-jacent peut être plus grand, si c'est plus efficace pour la plateforme
  - `signed` `int_fast8_t, int_fast16_t, int_fast32_t, int_fast64_t`
  - `unsigned` `uint_fast8_t, uint_fast16_t, uint_fast32_t, uint_fast64_t`

Types `stdint.h` dépendant du système

- Les types pointeurs sur un entier
  - `intptr_t` et `uintptr_t`
- Dépend du système : adressage de la mémoire
  - systèmes 32 bits : `intptr_t` coïncide avec `int32_t`
  - systèmes 64 bits : `intptr_t` coincïde avec `int64_t`
- Le type `ptrdiff_t` caractérise les soustractions de pointeurs (*pointer offsets*)

Opérateur sizeof

- La valeur de l'expression `sizeof(typename)`

  est un entier non signé représentant le nombre d'octets requis pour stocker une valeur de type `typename`

- L'opérateur `sizeof` peut également être appliqué aux constantes, aux variables et aux expressions en général

```
int i = 2;
unsigned long int nbreOctets;
nbreOctets = sizeof(2);     // => 4 (expression)
nbreOctets = sizeof(i);     // => 4 (variable)
nbreOctets = sizeof(int);   // => 4 (type)
```

volatile

- le qualificatif `volatile` informe le compilateur que la valeur d'une variable peut changer pour des raisons externes
  - Interface avec du matériel qui utilise des registres d'entrée / sortie
  - Utilisation de variables qui peuvent être modifiées en dehors du contrôle de flux du programme

```
volatile uint8_t* pReg = (uint8_t*)0x1717;

// Wait for register to become non-zero
while (*pReg == 0) {}

// Continue ...
```

- sans `volatile` le compilateur va optimiser le code généré de sorte à ne lire qu'une seule fois la valeur à 0x1717 (*pReg) ce qui donnera une boucle infinie

Différence entre C et C++ pour const

- `const` est le même mot-clé en C et C++, avec deux différences
- les objets `const` doivent être initialisés en C++ mais peuvent être laissés non initialisés en C
- `const` en C ne peut pas être utilisé pour construire des *expressions constantes*
- Solution : constante littérale, `enum` ou un #define

```
#include <stdio.h>

int main(void) {
   int i = 2;
   const int C = 2;
   switch(i) {
     case C : printf("Hello");
              break;
     default: printf("World");
   }
}
```

Ne compile pas !

Classes de stockage des variables

Storage classes

Specifiers	Lifetime	Scope	Default initializer
`auto`	Block (stack)	Block	Uninitialized
`register`	Block (stack or CPU register)	Block	Uninitialized
`static`	Program	Block or compilation unit	Zero
`extern`	Program	Global (entire program)	Zero
(none)[1]	Dynamic (heap)		Uninitialized (initialized to `0` if using `calloc()`)

[1] Allocated and deallocated using the `malloc()` and `free()` library functions.

Sources

- https://en.wikipedia.org/wiki/C_syntax#Storage_class_specifiers
- https://courses.cs.washington.edu/courses/cse374/16wi/lectures/memory_model.jpg
- http://www.cs.uleth.ca/~holzmann/C/system/memorylayout.pdf

4. Les opérateurs

Les opérateurs (rappel)

Le C propose les opérateurs (comme en C++)

- arithmétiques binaires `+ - * / %`
  et unaires `+ -`
- d'affectation simple `=`
  et composée `+= -= *= /= %=`
- de pré et post incré-/décrémentation `++ --`
- de comparaison `< <= == != >= >`
- logiques `&& ||`
  NB `and` et `or` n'existent pas en C

Priorité des opérateurs
https://en.cppreference.com/w/c/language/operator_precedence

Precedence	Operator	Description	Associativity
1	++ -- ( ) [ ] . -> (*type*) { *list* }	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	++ -- + - ! ~ (*type*) * & sizeof _Alignof	Prefix increment and decrement[note 1] Unary plus and minus Logical NOT and bitwise NOT Cast Indirection (dereference) Address-of Size-of[note 2] Alignment requirement(C11)	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	= += -= *= /= %= <<= >>= &= ^= |=	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Opérateur &

- L'opérateur & a deux significations en C/C++
  - AND bit à bit dans les expressions
  - fournit l'adresse d'une variable

```
// opérateur bit à bit
unsigned a = 3u, b = 7u, c;
c = a & b;     // => 3 (détaillé plus tard)

// adresse mémoire
int  i   = 0;
int* ptr = &i; // ptr contient l'adresse de i
```

- Il n'y a pas d'ambiguïté : l'opérateur bit à bit est binaire, l'autre est unaire

Opérateur *

- L'opérateur * a trois significations en C/C++
  - opérateur de multiplication (opérateur binaire)
  - déclaration d'une variable de type pointeur
  - accès à une valeur stockée à une adresse spécifique « *déréférencement* »

```
int  i   = 3 * 7;   // multiplication
int* ptr = &i;      // déclaration
int  a   = *ptr;    // déréférencement
```

- Il n'y a pas d'ambiguïté puisqu'il est binaire, unaire, ou qu'il apparaît dans une déclaration de variable
- Les opérateurs * et & seront utilisés pour le passage de paramètres par adresse dans le cadre des fonctions

Les opérateurs de manipulation de bits

- Pour effectuer des opérations au niveau des bits d'information
  - programmation système : compilateurs, OS, etc.
  - cryptographie, traitements graphiques, …
  - ou lorsque la vitesse et l'espace mémoire sont des paramètres essentiels
- Le langage C fournit six opérateurs de manipulation de bits (*bitwise operators*)
  - fonctionnent sur des données entières en binaire
  - opérateurs logiques `&` `|` `^` `~`
  - décalage à gauche `<<` et à droite `>>`

Les opérateurs de décalage

- Le résultat de `i << j` est la valeur de *i* en binaire avec les bits décalés vers la gauche de *j* positions
  - les bits qui débordent à gauche sont perdus
  - pour chaque bit décalé, on ajoute un 0 à droite

```
uint8_t i = (uint8_t)7;
i = i << 1;   // i devient 14
```

0	0	0	0	0	1	1	1	<<

- Un décalage d'une position à gauche revient à faire une multiplication de la valeur par 2

```
i = i << 1;   // fois 2
i = i << 2;   // fois 4
i = i << 3;   // fois 8
i = i << n;   // fois 2^n
```

Les opérateurs de décalage

- Le résultat de `i >> j` est la valeur de *i en binaire* avec les bits décalés vers la droite de *j* positions
  - les bits qui débordent à droite sont perdus
  - Le ou les bits insérés à gauche dépendent du compilateur (0 ou valeur du signe)

```
uint8_t i = (uint8_t)15;
i = i >> 1;   // i devient 7
```

>>	0	0	0	0	1	1	1	1

- Un décalage d'une position à droite revient à faire une division entière de la valeur par 2

```
i = i >> 1;   // div. par 2
i = i >> 2;   // div. par 4
i = i >> 3;   // div. par 8
i = i >> n;   // div. par 2^n
```

Les opérateurs logiques bit à bit

- Le complément ou inversion « ~ », inverse tous les bits de la valeur entière

```
uint8_t i = (uint8_t)14;
i = ~i;   // i devient 241
```

compl « ~ »	
0	1
1	0

0	0	0	0	1	1	1	0

Les opérateurs logiques bit à bit

- Les opérateurs logiques bit à bit

1er nombre	0	0	1	1
2ème nombre	0	1	0	1

and « & »	0	0	0	1
or « | »	0	1	1	1
xor « ^ »	0	1	1	0

```
uint8_t i = (uint8_t)14,  // 0000 1110
        j = (uint8_t)246; // 1111 0110

// opérateurs bit à bit
uint8_t et = i & j, // 0000 0110 =>   6
        ou = i | j, // 1111 1110 => 254
        ex = i ^ j; // 1111 1000 => 248

// opérateurs logiques (rappel)
bool a;
        a  = !i;      // => false
        a  = !i || j; // => true
        a  = !i && j; // => false
```

Les opérateurs d'affectation composée

- Tout comme les opérateurs d'affectation composée
  `+=  -=  *=  /=  %=`

- le C / C++ propose des opérateurs similaires pour les opérateurs de manipulation bit à bit `&=  |=  ^=  <<=  >>=`

```
uint8_t i = (uint8_t)14,  // 0000 1110
        j = (uint8_t)246; // 1111 0110

        i &= j;  // 0000 0110 =>   6
i = 14; i |= j;  // 1111 1110 => 254
i = 14; i ^= j;  // 1111 1000 => 248

        i >>= 3; // 0001 1111 =>  31
        i <<= 2; // 0111 1100 => 124
```

Attention aux priorités des opérateurs

- Eviter de mettre des parenthèses inutiles qui alourdissent le code

```
(i & (~j)) | k
```

→

```
i & ~j | k
```

```
i ^ (j & (~k))
```

→

```
i ^ j & ~k
```

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10	|	Bitwise OR (inclusive or)	
11	&&	Logical AND	
12	||	Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
	&= ^= |=	Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Usages fréquents (1 bit)

```
// Déterminer la valeur d'un bit spécifique
unsigned bit      =   3u;
unsigned valeur   = 125u;                          // 0111 1101
unsigned shift    = valeur >> bit;                 // 0000 1111
unsigned checkBit = (valeur >> bit) & 1u;          // 0000 0001
```

```
// Mettre un bit à 0
unsigned   bit    =   2u;
unsigned valeur   = 125u;                          // 0111 1101
unsigned mask     = 1u << bit;                     // 0000 0100
unsigned bitTo0   = valeur & ~mask;                // 0111 1001
// Mettre un bit à 1
unsigned bitTo1 = bitTo0 | mask;                   // 0111 1101
// Mettre un bit à X (X à remplacer par 0 ou 1)
unsigned valeur = (valeur & ~(1u << bit)) | (X << bit);
          //      \    efface le bit    /   \ bit à X /
```

Usages fréquents (mask)

```
// effacer un ou plusieurs bits
unsigned valeur     = 125u;                                  // 0111 1101
unsigned mask       =  15u; // ou 0xF                        // 0000 1111
unsigned resetBits  = valeur & ~mask;                        // 0111 0000
```

```
// setter un ou plusieurs bits
unsigned valeur     = 125u;                                  // 0111 1101
unsigned mask       =  15u;                                  // 0000 1111
unsigned setBits    = valeur | mask;                         // 0111 1111
```

```
// vérifier si certains bits sont tous à 1
unsigned valeur     = 125u;                                  // 0111 1101
unsigned mask       =  41u;                                  // 0010 1001
unsigned checkBits  = (valeur & mask) == mask;               // 0000 0001
```

```
// inverser (toggle) un ou plusieurs bits
unsigned valeur     = 125u;                                  // 0111 1101
unsigned mask       =  15u;                                  // 0000 1111
unsigned toggleBits = valeur ^ mask;                         // 0111 0010
```

5. Les entrées / sorties avec stdio.h

Instructions basiques d'affichage

La librairie `stdio.h` fournit une collection de constantes, de macros et de fonctions pour les opérations d'entrées / sorties

`EOF` qui indique une fin de fichier, vaut souvent -1

`int putchar(int c)`

- écrit la valeur de c, convertie en `unsigned char` sur la sortie standard
- retourne le caractère écrit (promu `en int`), ou `EOF` si erreur

`int getchar(void)`

```
#define CLEAR_BUFFER while (getchar() != '\n')
```

- lit le prochain caractère du flux d'entrée standard
- retourne le caractère (promu `en int`) ou `EOF` en cas d'erreur

Instruction printf

La fonction `printf` permet des affichages formatés sur `stdout`

```
int printf(const char* format, ...);
```

- `format` représente la chaîne de caractères à afficher
  … qui peut contenir des descripteurs préfixés par %
- si `format` contient un ou plusieurs descripteurs
  la ou les valeurs de remplacement suivront en séquence
- `printf` retourne le nombre de caractères écrits,
  … ou une valeur négative si erreur

```
printf("bonjour\n");            // bonjour         // retourne 8 (7+\n)
printf("il est %d:%d", 12, 34); // il est 12:34    // retourne 12
```

Instruction `printf` - descripteurs

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: `%e` or `%f`	392.65
G	Use the shortest representation: `%E` or `%F`	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
%	A `%` followed by another `%` character will write a single `%` to the stream.	%

Sources : http://www.cplusplus.com/reference/cstdio/printf

Instruction `printf` - descripteurs

Il est important de noter que le descripteur définit le type et l'interprétation de la valeur correspondante mais ne fait pas de conversion de type

```
// interprétation
printf("|%d|", 'A');        // |65|
printf("|%c|",  65);        // |A|
printf("|%d|", -30);        // |-30|
printf("|%d|",  30);        // |30|
printf("|%o|",  30);        // |36|
printf("|%x|",  30);        // |1e|
printf("|%X|",  30);        // |1E|

printf("|%u|",  12u);       // |12|
printf("|%u|", -1);         // |4294967295|

printf("|%f|",  3.14);      // |3.140000|
printf("|%e|",  3.14);      // |3.140000e+00|

// pas de conversion (warning à la compilation)
printf("|%f|",  12);        // indéterminé
printf("|%d|",  3.14);      // indéterminé
```

Instruction `printf` - descripteurs

- Optionnellement, les descripteurs peuvent être complétés par

```
%[flags][width][.precision][length]descripteur
```

- `flags` `- + (space) # 0`
- `width` `valeur` ou `*`
- `precision` `valeur` ou `*`
- `length` `hh h l ll j z t L`

Détails sur http://www.cplusplus.com/reference/cstdio/printf

Instruction printf - length

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

```
printf("%hhd", ..); // signed   char
printf("%hhu", ..); // unsigned char
printf("%hd",  ..); // signed   short
printf("%hu",  ..); // unsigned short
printf("%ld",  ..); // signed   long
printf("%lu",  ..); // unsigned long
```

```
printf("%lld", ..); // signed   long long
printf("%llu", ..); // unsigned long long
printf("%f",   ..); // float
printf("%f",   ..); // double
printf("%Lf",  ..); // long double
printf("%zu",  ..); // size_t
```

NB : Pas tous supportés, mieux d'utiliser <inttypes.h>

Sources : http://www.cplusplus.com/reference/cstdio/printf

Instruction `printf` - flags

flags	**description**
–	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or –) even for positive numbers. By default, only negative numbers are preceded with a – sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with `o`, `x` or `X` specifiers the value is preceeded with `0`, `0x` or `0X` respectively for values different than zero. Used with `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see *width* sub-specifier).

```
printf("|%+d|",    1234);   // |+1234|
printf("|%+d|",   -1234);   // |-1234|
printf("|% d|",    1234);   // | 1234|
printf("|% d|",   -1234);   // |-1234|
printf("|% f|",   12.34);   // | 12.340000|
printf("|%+f|",   12.34);   // |+12.340000|
```

```
printf("|%x|",    1234);   // |4d2|
printf("|%X|",    1234);   // |4D2|
printf("|%o|",    1234);   // |2322|
printf("|%#x|",   1234);   // |0x4d2|
printf("|%#X|",   1234);   // |0X4D2|
printf("|%#o|",   1234);   // |02322|
```

Sources : http://www.cplusplus.com/reference/cstdio/printf

Instruction printf - width

width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

```
                                    // |1234567890|
printf("|%10s|", "HEIG-VD");        // |   HEIG-VD|
printf("|%10d|",   1234);           // |      1234|
printf("|%3d|",    1234);           // |1234|
printf("|%+10d|",  1234);           // |     +1234|
printf("|%-+10d|", 1234);           // |+1234     |
printf("|%010d|",  1234);           // |0000001234|
```

Sources : http://www.cplusplus.com/reference/cstdio/printf

Instruction printf - precision

.precision	**description**
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, `0` is assumed.
`.*`	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

```
                                  // |1234567890|
printf("|%10.6d|", 1234);         // |    001234|
printf("|%.6d|",   1234);         // |001234|
printf("|%10.4f|", 123.456789);   // |  123.4568|
printf("|%.4f|",   123.456789);   // |123.4568|
printf("|%*.*d|",  10, 6, 1234);  // |    001234|
```

Sources : http://www.cplusplus.com/reference/cstdio/printf

Affichage de types entiers spécifiques

- Pour afficher précisément une variable dont le type est défini dans `stdint.h`, il faudra généralement avoir recours à la librairie `inttypes.h`, qui met à disposition, sous forme de macros, les codes de format nécessaires

int64_t	"%" PRId64
uint64_t	"%" PRIu64
intptr_t	"%" PRIdPTR
uintptr_t	"%" PRIuPTR
intmax_t	"%" PRIdMAX
uintmax_t	"%" PRIuMAX

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

int main(void) {
   printf("%s", PRId64);     // Windows => I64d
                             // MAC     => lld
   int64_t n = 100;
   printf("%" PRId64, n);    // 100
}
```

Caractères spéciaux dans printf

Escape sequence	Description	Representation
Simple escape sequences		
`\'`	single quote	byte 0x27 in ASCII encoding
`\"`	double quote	byte 0x22 in ASCII encoding
`\?`	question mark	byte 0x3f in ASCII encoding
`\\`	backslash	byte 0x5c in ASCII encoding
`\a`	audible bell	byte 0x07 in ASCII encoding
`\b`	backspace	byte 0x08 in ASCII encoding
`\f`	form feed - new page	byte 0x0c in ASCII encoding
`\n`	line feed - new line	byte 0x0a in ASCII encoding
`\r`	carriage return	byte 0x0d in ASCII encoding
`\t`	horizontal tab	byte 0x09 in ASCII encoding
`\v`	vertical tab	byte 0x0b in ASCII encoding
Numeric escape sequences		
`\`*nnn*	arbitrary octal value	byte *nnn*
`\x`*nn*	arbitrary hexadecimal value	byte *nn*
Conditional escape sequences[1]		
`\`*c*	Implementation-defined	Implementation-defined
Universal character names		
`\u`*nnnn*	arbitrary Unicode value; may result in several code units	code point U+*nnnn*
`\U`*nnnnnnnn*	arbitrary Unicode value; may result in several code units	code point U+*nnnnnnnn*

```
printf("name\t: %s\n", "Abraham Lincoln");
printf("date\t: %4d\\%4d\n", 1809, 1865);

// name  : Abraham Lincoln
// date  : 1809\1865

printf("bient\u00f4t l'\u00e9t\u00e9 \u263A\n");
printf("\u2660 \u2663 \u2665 \u2666");

// bientôt l'été ☺
// ♠ ♣ ♥ ♦
```

Lien utile : https://www.fileformat.info/info/unicode/char

Sources : https://en.cppreference.com/w/cpp/language/escape

Variantes de printf

A noter encore les variantes de `printf`

Defined in header <cstdio>

```
int printf( const char* format, ... );
int fprintf( std::FILE* stream, const char* format, ... );
int sprintf( char* buffer, const char* format, ... );
int snprintf( char* buffer, std::size_t buf_size, const char* format, ... );
```

```
char chaine[80];          // réservation nécessaire !

sprintf(chaine, "date : %02d-%02d-%4d", 14, 2, 2012);
printf("%s", chaine);     // date : 14-02-2012

snprintf(chaine, 10, "date : %02d:%02d:%4d", 14, 2, 2012);
printf("%s\n", chaine);   // date : 14
```

NB : `fprintf` sera vu avec les fichiers

Sources : https://en.cppreference.com/w/cpp/io/c/fprintf

Instruction scanf

La fonction `scanf` permet la lecture et la conversion de caractères depuis `stdin` dans des variables

```
int scanf(const char* format, ...);
```

- `format` est une chaîne indiquant le(s) format(s) de conversion
- les arguments suivants doivent être des adresses (lvalue) où les valeurs lues seront stockées
- retourne le nombre de données saisies avec succès

```
int i, j, n;
char s[80];
scanf("votre saisie : %d", i);      ❌

printf("votre saisie : ");          // msg de saisie
n = scanf("%d %d", &i, &j);         // 3 4↵          => n=2 i=3 j=4
n = scanf("%s", s);                 // Jean Paul↵    => n=1 s=Jean
```

Fonctionnement de scanf

- `scanf` essaie de matcher les suites de caractères saisis au clavier avec ceux de la chaîne `format` comprenant des caractères et des code de conversion en ignorant généralement les espaces
- pour chaque code de conversion
  - `scanf` lit alors l'item en s'arrêtant au premier caractère qui ne correspond pas au type
  - scanf convertit et stocke l'item à l'adresse donnée en paramètre

```
int    i, j, n;
double x;

n = scanf("%d",     &i);        // 123abc↵   => n=1 i=123
n = scanf("%d",     &i);        // ab123cd↵  => n=0 i=?
n = scanf("%d %d", &i, &j);     // 1 2↵      => n=2 i=1 j=2
n = scanf("%d %d", &i, &j);     // 1a2↵      => n=1 i=1 j=?
n = scanf("%lf",    &x);        // 12.345a6↵ => n=1 x=12.345
```

Chaîne de format : caractères ordinaires

- Lorsqu'il rencontre un caractère qui n'est pas un espace dans une chaîne de format, `scanf` le compare avec le caractère d'entrée suivant
  - s'ils sont égaux, `scanf` prélève le caractère d'entrée et poursuit le traitement de la chaîne de format
  - s'ils ne le sont pas, `scanf` remet le caractère dans le tampon d'entrée, puis échoue

```
int i, j;

scanf("%d/%d",   &i, &j);   // échoue avec  " 5 / 96↵"
scanf("%d/%d",   &i, &j);   // réussit avec " 5/ 96↵"
scanf("%d / %d", &i, &j);   // réussit avec " 5 / 96↵"
```

Cas spéciaux

A noter encore les variantes de `scanf`

Defined in header <cstdio>

```
int scanf( const char* format, ... );
```

```
int fscanf( std::FILE* stream, const char* format, ... );
```

```
int sscanf( const char* buffer, const char* format, ... );
```

NB : `fscanf` sera vu avec les fichiers

Cas spéciaux

- `%*` la valeur est ignorée
- `%[listOfChars]` lit seulement des caractères qui sont dans `listOfChars`
- `%[^listOfChars]` lit seulement des caractères qui ne sont pas dans `listOfChars`
- `%Nc` lit N caractères
- `%n` `scanf` retourne le nombre de caractères lus

Qu'affiche ce code ?

```c
int   i, j;
float x, y;
char  str1[10], str2[4];

char input[] = "25 54.32E-1 Thompson 56789 0123 56";
// parse as follows:
// %d: an integer
// %f: a floating-point value
// %9s: a string of at most 9 non-whitespace characters
// %2d: two-digit integer (digits 5 and 6)
// %f: a floating-point value (digits 7, 8, 9)
// %*d an integer which isn't stored anywhere
// ' ': all consecutive whitespace
// %3[0-9]: a string of at most 3 digits (digits 5 and 6)

int n = sscanf(input, "%d%f%9s%2d%f%*d %3[0-9]", &i, &x, str1, &j, &y, str2);
printf(" n = %d \n i = %d \n x = %f \n str1 = %s \n j = %d \n y = %f \n str2 = %s",
         n, i, x, str1, j, y, str2);
```

```c
// output
 n = 6
 i = 25
 x = 5.432000
 str1 = Thompson
 j = 56
 y = 789.000000
 str2 = 56
```

Sources : https://en.cppreference.com/w/cpp/io/c/scanf

6. Résumé

Résumé

- Le langage C est aujourd'hui encore un des langages les plus utilisés.
- Des compilateurs C existent pour pratiquement tous les OS.
- Le langage C est (presque) un sous-ensemble du langage C++.
- **C est un langage procédural**, C++ est un langage procédural et OO.
- Nombre de concepts C++ n'existent pas en C : surcharge des fonctions, exceptions, généricité, classes et objets, etc.
- **C est un langage de haut niveau**, concis et donc relativement facile à apprendre, qui permet d'écrire du code de bas niveau (mais pas seulement), très rapide et compact.

Résumé

- La rapidité du code C est essentiellement due au fait que le compilateur présuppose que le programmeur sait ce qu'il fait.
  Corollaire : le code généré réalise très peu de vérifications (!).
- **Les domaines d'application du C sont nombreux** : programmation système, systèmes embarqués, réalisation de compilateurs, drivers, …
- Le **préprocessing**, la **compilation** et l'**édition de liens** constituent les trois étapes nécessaires pour transformer un code source C en code exécutable.
- **Le préprocesseur** a pour mission de "préparer le terrain" pour la compilation en traduisant, notamment, les directives (#include, #define, de compilation conditionnelle) qui lui sont adressés dans le code source.

Résumé

- **La librairie <stdint.h>** fournit un ensemble de types entiers particuliers (p. ex. codés sur un nombre de bits fixé, comme `int64_t`) dans le but de garantir une meilleure portabilité des applications.
- **La librairie <inttypes.h>** propose, elle, des macros (p. ex. PRId64) permettant de faire des E/S (`printf/scanf`) sur les types proposés par <stdint.h>
- A l'instar de C++, C propose six opérateurs permettant d'opérer au niveau des bits : **les opérateurs de manipulation de bits**.
  - 4 **opérateurs logiques bit à bit** : &, ^, | et ~
  - 2 **opérateurs de décalage** : << (à gauche) et >> (à droite)

Résumé

- Les E/S en C se réalisent principalement au travers des fonctions **`printf`** et **`scanf`**.
- Ces deux fonctions font appel au concept de **descripteur de format** pour préciser la nature et la forme de ce qui est censé être affiché à l'écran ou lu à partir du clavier.