Historique du langage C

par Marc Mongenet (Site personnel)

Date de publication :

Dernière mise à jour :

Cet article est un bref historique du langage C ; de ses origines (CPL, BCPL, B) à ses évolutions (K&R, ANSI, C++, ...)

Votre avis et vos suggestions nous intéressent !
De même que des informations complémentaires (citations, URL, images, codes, ...) ou des corrections.
Alors après votre lecture, n'hésitez pas :

I - Origines

Le langage C possède trois ancêtres : le CPL, le BCPL et le B.

I-A - Le CPL

Le langage CPL (pour Combined Programming Language) a été conçu au début des années 1960 par les universités **de Cambridge** et de Londres. C'était un grand projet académique consistant à créer un langage englobant beaucoup de concepts. Le CPL devait notamment être fortement typé, avec de nombreux types comme des nombres entiers, réels, booléens, caractères, tableaux, listes...

Le CPL était trop complexe pour l'époque et il semble que personne n'ait réussi à terminer l'écriture d'un compilateur. Ce langage a disparu sans laisser de trace quelque part dans les années 1970.

Réf. : **LinuxGuruz**, **Wikipédia**

I-B - Le BCPL

Le langage BCPL (pour Basic CPL) a été conçu à Cambridge en 1966 par **Martin Richards**. L'année suivante, il alla au **MIT** et écrivit un premier compilateur. Le BCPL est une version fortement simplifiée du langage CPL avec notamment un seul type de donnée, le mot machine, c'est-à-dire le nombre typique de bits que le processeur d'un ordinateur traite en une instruction-machine (addition, soustraction, multiplication, copie...). La notion est devenue un peu floue avec les processeurs actuels qui peuvent traiter des données de toutes sortes de tailles. Cependant, on peut raisonnablement classer les Pentium et PowerPC parmi les processeurs 32 bits, contre 64 bits pour les Alpha, Itanium ou Opteron. Du temps du BCPL, on trouvait des architectures à mots de 40 bits, 36 bits, 18 bits, 16 bits...

Le BCPL opère sur les mots de la machine. Il est donc à la fois portable et proche du matériel, donc efficace pour la programmation système. Le langage BCPL a servi à écrire divers systèmes d'exploitation de l'époque, dont un (TripOS) qui s'est retrouvé partiellement porté sur l'Amiga pour devenir la partie AmigaDOS du système.

Mais aujourd'hui, le BCPL ne semble plus être utilisé que par son inventeur. C'est sans doute dû au fait que le langage C a repris et étendu la plupart des qualités de BCPL.

Référence : **manuel BCPL**

Exemple de code BCPL (tiré de **Clive Feather on CPL and BCPL**) :

```
MANIFEST ${ TOPFACT = 10 $} // Equivalent of "const int TOPFACT = 10"
LET infact (n) = VALOF
$(
  LET f, j = 1., 0.

  FOR i = 0 TO n              // Declares i for the next block only
  $(
    f #*:= j;                 // := is assign, = is compare
    fact!i := f;              // assignment doesn't return a value
    j #+:= 1.
  $)
  RESULTIS f
$)
AND fact = VEC TOPFACT;       // As in B, allocates 0 to TOPFACT
```

I-C - Le B

Le langage B a été créé par **Ken Thompson** vers 1970 dans les **laboratoires Bell** d'**AT&T**. L'année précédente, il avait écrit en assembleur la première version d'**UNIX** sur un **PDP-7** contenant 8 kilo mots de 18 bits. Lorsqu'il voulut

proposer un langage sur ce nouveau système d'exploitation, il semble qu'il ait d'abord pensé à porter **Fortran**, mais que très vite (en une semaine) il conçut son propre langage : B.

Le B est une simplification de BCPL, un peu forcée par les limitations du **PDP-7**. Mais la syntaxe très sobre du B (et des commandes **UNIX**) toute en lettres minuscules correspond surtout aux goûts de **Thompson**. Le langage C a repris la syntaxe du B avec un minimum de changements.

Le langage B a été porté et utilisé sur quelques autres systèmes. Mais cette même année 1970, le succès du projet **UNIX** justifia l'achat d'un **PDP-11**. Celui-ci avait des mots machine de 16 bits, mais il était aussi capable de traiter des octets (24 ko de mémoire vive en tout) dans chacun desquels pouvait être stocké un caractère. Le B ne traitait que des mots machine, donc le passage de 18 à 16 bits n'était pas problématique, mais il était impossible de traiter efficacement les caractères de 8 bits. Pour bien exploiter les capacités de la machine, le B a donc commencé à être étendu en ajoutant un type pour les caractères...

Référence : **Thompson's BManual**

Exemple de code B identique au code BCPL ci-dessus (tiré de **Clive Feather on CPL and BCPL**) :

```
infact (n)
{
  auto f, i, j;    /* no initialization for auto variables */
  extrn fact;      /* "What would I do differently if designing
                    *  UNIX today?  I'd spell creat() with an e."
                    *  -- Ken Thompson, approx. wording */

  f = 1.;          /* floating point constant */
  j = 0.;
  for (i = 0; i <= n; ++i) {
    fact[i] = f =#* j;       /* note spelling =#* not #*= */
    j =#+ 1.;                /* #+ for floating add */
  }

  return (f);      /* at least, I think the () were required */
}

TOPFACT = 10;   /* equivalent of #define, allows numeric values only */
fact[TOPFACT];
```

I-D - Le C

Le langage C a été développé par un collègue de **Ken Thompson**, **Dennis Ritchie** qui pensait d'abord faire uniquement un New B ou NB. Mais en plus des caractères, Ritchie ajouta les tableaux, les pointeurs, les nombres à virgule flottante, les structures... 1972 fut l'année de développement la plus productive et sans doute l'année de baptême de C. En 1973, le langage C fut suffisamment au point pour que 90 % d'**UNIX** puisse être récrit avec. En 1974, les **laboratoires Bell** ont accordé des licences **UNIX** aux universités et c'est ainsi que le langage C a commencé à être distribué.

Références : **The Development of the C Language**, **Very early C compilers and language**

Exemple de code C identique aux codes BCPL et B ci-dessus (tiré de **Clive Feather on CPL and BCPL**) :

```
float infact (n) int n;
{
  float f = 1;
  int i;
  extern float fact[];

  for (i = 0; i <= n; ++i)
    fact[i] = f *= i;

    return d;
```

```
  }

#define TOPFACT 10
float fact[TOPFACT+1];
```

Voir Ken Thompson (assis) et Dennis Ritchie devant un PDP-11 fonctionnant avec **UNIX** vers 1972.

II - Évolutions

Une fois rendu public, le langage C a peu changé. Pratiquement toutes les extensions se sont faites dans **C++**, qui est une gigantesque extension du C. Une autre évolution du C est **Objective-C**, qui se concentre sur l'orientation objet. De nombreux autres langages comme **Java**, **JavaScript** ou **C#** ont largement repris la syntaxe du C, mais sans être compatibles.

II-A - 1978 - K&R C

```
#include <stdio.h>
main(argc,
     argv)
int argc;
char ** argv;
{
 printf("hello, world\n");
}
```

La plus ancienne version du langage C encore en usage a été formalisée en 1978 lorsque **Brian Kernighan** et **Dennis Ritchie** ont publié la première édition du livre **Le langage C**. Ce livre décrit ce qu'on appelle actuellement le K&R C, le C traditionnel, voire le vieux C. Peu après sa publication, de très nombreux compilateurs C ont commencé à apparaître.

Les programmes portables écrits dans les années 1980 sont donc en K&R C. De fait, quelques programmes très portables sont encore en K&R C (par exemple **GNU Make**). En effet, de nombreux systèmes commerciaux ne proposent qu'un compilateur K&R C en standard, le compilateur ANSI C devant être acheté séparément. Heureusement, la disponibilité presque universelle de **GCC** résout pratiquement ce problème.

II-B - 1983 - C++

```
#include <iostream.h>
int main(int argc,
         char * argv[])
{
 cout << "hello, world"
      << endl;
 return 0;
}
```

À partir de 1980, **Bjarne Stroustrup** a étendu le langage C avec le concept de classe. Ce langage étendu a d'abord été appelé C with Classes, puis le C++ en 1983. Le langage C++ a énormément évolué (surcharge d'opérateurs, héritage, références, types génériques, exceptions...), mais en restant le plus compatible possible avec C. Il est souvent possible de compiler un programme en C avec un compilateur C++.

II-C - 1983 - Objective-C

```
void main()
{
 printf("hello, world\n");
}
```

Le langage Objective-C a été créé par Brad Cox. C'est un strict sur ensemble de C. Il lui apporte un support de la programmation orientée objet inspiré de **Smalltalk**. Ce langage est à la base de **NeXTSTEP**. Avec le rachat de **NeXT** par **Apple**, l'Objective-C s'est retrouvé à la base de l'interface graphique **Cocoa** de **Mac OS X**.

II-D - 1989 - ANSI C

```
#include <stdio.h>
main(int argc,
     char * argv[])
{
 printf("hello, world\n");
 return 0;
}
```

Un comité de standardisation a été créé en 1983 pour éviter que les quelques ambiguïtés et insuffisances du K&R C ne conduisent à des divergences importantes entre les compilateurs. Il a publié en 1989 le standard appelé ANSI C. Il a repris quelques bonnes idées du C++ comme les prototypes de fonction, tout en restant très compatible avec K&R C.

Le degré de compatibilité atteint est suffisant pour que **Kernighan** et **Ritchie** n'aient eu qu'à adapter légèrement la seconde édition du livre **Le langage C** pour qu'il décrive ANSI C. En plus, selon **Stroustrup**, tous les exemples de cette seconde édition sont aussi des programmes C++.

Le standard ANSI C est devenu l'évolution la plus utilisée du C.

II-E - 1998 - Standard C++

```
#include <iostream>
int main(int argc,
         char * argv[])
{
 std::cout << "hello, world"
           << std::endl;
}
```

Le C++ a évolué très longtemps. Ce n'est qu'en 1998, 8 ans après la création d'un comité, que le standard ISO C++ a été officiellement publié. Ce standard est tellement complexe (et légèrement incohérent), qu'en 2003, **le compilateur GCC** ne le met pas complètement en œuvre, et ce n'est pas le seul. Les syntaxes obsolètes et problématiques de K&R C ont été abandonnées ; pour le reste, la compatibilité avec le C reste excellente.

II-F - 1999 - C99

```
#include <stdio.h>
int main(int argc,
         char * argv[])
{
 printf("hello, world\n");
}
```

Le dernier-né de l'histoire est le C99 (standard ISO de 1999) qui est une petite évolution de l'ANSI C de 1989. Les évolutions ne sont pas compatibles avec le C++ et n'ont pas attiré beaucoup d'intérêt.

GCC supporte le C99 et le **noyau Linux** en tire profit. Côté compatibilité, le support des syntaxes obsolètes de K&R C a été abandonné.

III - Héritage historique

De nombreuses propriétés étranges du langage C s'expliquent par l'évolution historique du langage.

III-A - Pointeurs et tableaux

III-A-1 - Déclaration avec []

Les toutes premières versions du langage C ne permettaient pas d'utiliser * pour déclarer un pointeur, il fallait utiliser []. Les **sources du premier compilateur C** montrent cependant qu'une variable déclarée avec [] était un pointeur, déréférençable avec l'opérateur * et incrémentable avec l'opérateur ++ :

```
/* Exemple des premières versions de C, désormais illégal ! */
int np[];   /* Ceci déclarait en fait un pointeur sur un entier  */
/*...*/
*np++ = 1;  /* qui était utilisable comme les pointeurs actuels. */
```

Les premières versions du C n'avaient donc en fait que des pointeurs. La trace la plus visible de cet héritage se retrouve dans la déclaration classique de la fonction main :

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

Encore aujourd'hui, l'opérateur [] dans un paramètre formel de fonction déclare un pointeur. Cet usage trompeur est inusité, sauf pour main. La déclaration précédente est donc tout à fait équivalente à :

```
int main(int argc, char** argv);
```

Il est impossible de déclarer un paramètre formel de type tableau. En revanche, il est possible de déclarer un pointeur sur un tableau. Les déclarations suivantes sont équivalentes, car f reçoit en fait un pointeur de tableau de 13 int :

```
void f(int tab[2][13]);  /* Le 2 est ignoré ! */
void f(int tab[][13]);
void f(int (*tab)[13]);  /* Usage le plus clair */
```

III-A-2 - Pointeur et premier élément de tableau

Le fait que int t[]; déclarait un pointeur explique les liens très étroits entre pointeurs et tableaux. En effet, si t pointait sur un int et si ce dernier était suivi en mémoire d'autres int, alors t pointait sur le premier élément d'un tableau d'int.

Aujourd'hui int t[]; déclare un tableau de nom t. Mais un nom de tableau est automatiquement utilisé comme un pointeur sur son premier élément dans un contexte ou un pointeur est attendu, c'est-à-dire pour initialiser un pointeur ou comme opérande de +, -, *, [], ==, !=, <, <=, >, >=, !... En revanche, un nom de tableau ne se comporte pas comme un pointeur avec les opérateurs unaires &, ++, --, sizeof ou à gauche de =.

```
void f(int* q)  /* la fonction f prend un pointeur d'entier */
{
  int *p, t[4]; /* p est un pointeur d'entier, t un tableau de 4 entiers */
  p = t;        /* assigne &t[0] (adresse du premier élément de t) à p */
  *t = 6;       /* assigne 6 au premier élément de t */
  p[1] = 6;     /* assigne 6 au second élément de t */
  *(t+2) = 6;   /* assigne 6 au troisième élément de t */
  t[3] = 6;     /* assigne 6 au quatrième élément de t */
  f(t);         /* initialise q avec &t[0] (adresse du premier élément de t) */
}
```

III-A-3 - Opérateur []

Alors que les tableaux n'existaient pas encore, l'opérateur d'indexation [] existait déjà. Ses opérandes sont donc un pointeur et un entier : l'expression E1[E2] est équivalente à *((E1)+(E2)) et une des deux expressions doit être de type pointeur et l'autre de type entier. Si une des deux expressions est un nom de tableau, elle sera alors automatiquement convertie en un pointeur sur le premier élément. Le résultat de l'addition est un pointeur sur l'élément voulu, déréférencé par *.

À noter que E1[E2] est équivalent à *((E1)+(E2)) qui est équivalent à *((E2)+(E1)) qui est équivalent à E2[E1]. Les deux expressions suivantes sont donc équivalentes :

```
t[3] = 6;  /* *(t+3) = 6; */
3[t] = 6;  /* *(3+t) = 6; */
```

III-B - Conversions de type

Le C a tendance à convertir automatiquement les valeurs entre des types qui ne partagent aucune sémantique :

```
char c = 133333.14;  /* conversion double -> char */
float x = 'a';       /* conversion int -> float (char -> float en C++) */
char * p = 123;      /* conversion int -> char* possible en vieux C */
```

Cela vient du fait que les types n'ont pas été ajoutés au langage C pour permettre au compilateur de faire des vérifications sémantiques. À l'origine, les types ont été ajoutés simplement pour traiter des variables de différentes tailles, notamment des caractères de 8 bits lorsque les mots machine mesurent 16 bits. Les conversions automatiques de type étaient alors une puissante fonctionnalité.

À mesure que le C a été utilisé pour de grands projets, le besoin de vérifications de type s'est plus fait sentir. Les compilateurs ont commencé à signaler les conversions entre pointeur et entier ou entre pointeurs incompatibles. Les conversions automatiques impliquant des pointeurs ont finalement été retirées de C++. En effet, la vérification statique des types est un des points forts de C++.

III-C - int implicite

Le type char a été créé pour représenter les caractères, les autres variables tenant dans un mot machine. Le type int a été donné au mot machine, mais le nom du type est resté optionnel pour déclarer une variable int. Ainsi, le fait d'ajouter les types n'a pas rendu les sources existantes incompatibles.

La règle du int implicite est même restée couramment utilisée jusqu'au K&R C. Elle est encore admise dans ANSI C et C++, mais a disparu du C++ standard et de C99. On peut noter que la déclaration d'une variable automatique locale sans donner son type requiert l'usage du mot clé auto, devenu totalement obsolète en C moderne :

```
/* Exemple de int implicite en K&R C */
/*int*/ *g;
/*int*/ main()
{
  auto /*int*/ i;
  g = &i;
  *g = 0;
  return i;
}
```

III-D - Conversions arithmétiques

Lorsqu'une opération arithmétique implique des valeurs de différents types, les opérandes sont d'abord automatiquement convertis vers un type commun. Ces conversions sont naturelles et donnent des résultats intuitifs (sauf lors d'un mélange entre types signé et non signé).

```
int func() {
  short s = 2;
  long l = 1;
  /* Valeur de s convertie en long, puis addition en long, puis résultat long
     converti en float pour être assigné à f. */
  float f = l + s;
}
```

Une particularité est qu'aucune opération ne s'effectue avec une précision inférieure à int. Autrement dit, si l'on additionne deux short, ils seront chacun convertis en int avant que l'opération ait lieu. C'est encore un héritage de l'importance du mot machine :

```
short f(short a, short b) {
  /* Valeurs de a et b converties en int avant l'addition.
     Le résultat int de l'addition est converti en short pour être
     retourné, d'où possibilité d'avertissement du compilateur ! */
  return a + b;
}
```

III-E - Paramètres de fonction

Les vérifications de type des paramètres de fonction ont été ajoutées progressivement au langage. Mais seuls le C ++ et le C99 rendent les vérifications obligatoires.

III-E-1 - Déclaration implicite de fonction

En C, il est possible d'appeler une fonction déclarée nulle part. Dans ce cas, le compilateur se crée une déclaration implicite de la fonction d'après l'appel. Bien sûr, un exécutable peut être produit uniquement si l'éditeur de liens trouve la fonction appelée dans un fichier objet. Cependant, si la déclaration implicite du compilateur n'est pas compatible avec la fonction trouvée, alors l'éditeur de lien ne le verra pas, et l'exécution du programme sera erronée.

```
int main(void)
{
  /* La fonction printf n'est déclarée nulle part, mais présente dans la libc,
     donc un exécutable peut être produit par un compilateur C. */
  printf("hello\n");  /* Appel compatible avec printf, affichera "hello". */
  printf(123);   /* Appel non compatible, causera une erreur d'exécution. */
  return 0;
}
```

III-E-2 - Déclaration de fonction

En C, une déclaration de fonction ne donne pas son prototype, c'est-à-dire qu'une déclaration de fonction ne donne ni le nombre ni le type des paramètres et ne permet donc pas de vérifications :

```
int main(void)
{
  int printf();  /* Déclaration de la fonction printf, ne précise pas
                    les paramètres de printf. */
  printf(123);   /* Appel non compatible, causera une erreur, mais
                    seulement à l'exécution ! */
  return 0;
}
```

III-E-3 - Style K&R C de définition de fonction

Les définitions de fonction de style K&R C ne sont pas utilisées pour vérifier le type, ni même le nombre des paramètres :

```
/* Style de programmation K&R C */
void repeter(c, s, n)  /* 3 paramètres */
char c, *s;            /* n implicitement int */
{
  while (n--) s[n] = c;
}

main()
{
   char t[10+1];
   t[10] = 0;
   repeter('a', t, 10);  /* écrit aaaaaaaaaa dans t */
   repeter(123);         /* erreur, mais seulement à l'exécution ! */
   printf("%s\n", t);
}
```

III-E-4 - Promotion des paramètres

Avec les fonctions de style K&R C, les compilateurs ne connaissent le type des paramètres des fonctions appelées. Ils promeuvent donc les entiers (short, char... en int ou unsigned) et les float en double, avant de les passer à la fonction. La promotion en int laisse des traces dans plusieurs prototypes de fonction de la bibliothèque standard qui traitent des caractères :

```
int putchar(int c);
int isalnum(int c);
int tolower(int c);
int toupper(int c);
void * memset(void * s, int c, size_t n);
```

Malgré l'introduction des prototypes, on trouve encore une trace de cela avec les fonctions à nombre variable de paramètres, comme int printf(const char*,...);. Pour afficher un short, on utilise le formatage %d comme pour un int. En effet, le short est promu en int avant d'être passé à printf :

```
#include <stdio.h>
int main()
{
  int i = 1; short s = 2;
  printf("%d %d\n", i, s);  /* affiche 1 2 */
  return 0;
}
```

III-E-5 - Prototypes, void, procédure

Les prototypes ont été introduits en C sous l'impulsion de C++. En C++, le prototype de toute fonction appelée doit être connu. En outre, en C++, int f(); est un prototype qui indique que la fonction f n'a pas de paramètres. En C, int f(); est une déclaration et ne précise rien sur les paramètres. En C, le prototype d'une fonction f sans paramètres est int f(void);.

```
              /* En C         */    /* En C++      */
int f();      /* déclaration */    /* prototype   */
int f(int);   /* prototype   */    /* prototype   */
int f(int i)  /* définition  */    /* définition  */
{ return i; }
int f() {}    /* erreurs     */    /* définition */
```

La dernière ligne est une erreur en C à deux titres. D'abord, la redéfinition de fonction est interdite (et la surcharge n'existe pas). Ensuite, la définition n'est pas compatible avec le prototype. En revanche, on remarque que le prototype est compatible avec la déclaration.

Pour C++, il y a simplement deux prototypes et deux définitions, pour deux fonctions surchargées.

L'introduction du type void rend également possible la création de procédures en C, c'est-à-dire des fonctions ne retournant rien.

III-F - Chaînes littérales de caractères

À l'origine de C, les chaînes littérales de caractères étaient des tableaux statiques anonymes de caractères, initialisés avec les caractères. Lors de la standardisation de C, le mot clé const a été introduit et les chaînes littérales ont été rendues constantes pour pouvoir être partagées ou mises en mémoire morte. Il aurait donc été logique de donner aux chaînes littérales le type « tableau de caractères constants.» Cependant, l'usage suivant était extrêmement répandu :

```
char t[] = "hello";
char * p = "hello";
```

Or, cet usage aurait causé un avertissement du compilateur à cause de la conversion de const char[] en char[]. Dans le standard ANSI C, les chaînes littérales gardent donc le type char[], bien qu'elles soient considérées comme constantes.

Référence : **explication de Dennis Ritchie**.

Lors de la standardisation de C++, le type des chaînes littérales a tout de même été changé en « tableau de caractères constants ». Cela est nécessaire pour choisir la bonne fonction parmi deux fonctions surchargées dont seule la constance du pointeur change. Toutefois la construction char*p="texte"; reste permise, mais dépréciée.

IV - Remerciements

L'équipe C de Developpez.com tient à remercier **Wachter** pour la relecture orthographique de cet article