Introduction à la programmation en C

Séance 3 : conversions de type, flot de contrôle

Luca SAIU

http://ageinghacker.net

IUT de Villetaneuse, Université Paris 13

Septembre 2018

Sommaire

Rappel

- Mon site web : `http://ageinghacker.net`
- vous trouvez la page web officielle du cours en suivant le lien "*Teaching*" ;
- je suis facile à contacter (*n'utilisez pas l'ENT*) ;
- si vous voulez le livre sur C ("*GNU C Intro and Reference*", Stallman and Rothwell, en anglais) demandez-moi.

Compilation : warnings et erreurs

- En cas de *warnings*, la compilation n'échoue pas, et GCC produit un fichier exécutable—même si *suspect*.
- En cas d'**erreurs** GCC ne va **pas générer un fichier exécutable** : si vous avez déjà un fichier compilé ayant le même nom, l'ancien fichier reste là.
  - Donc **tester le fichier exécutable est inutile** après une erreur de compilation : c'est une ancienne version !

Compilation : warnings et erreurs

- En cas de *warnings*, la compilation n'échoue pas, et GCC produit un fichier exécutable—même si *suspect*.
- En cas d'**erreurs** GCC ne va **pas générer un fichier exécutable** : si vous avez déjà un fichier compilé ayant le même nom, l'ancien fichier reste là.
  - Donc **tester le fichier exécutable est inutile** après une erreur de compilation : c'est une ancienne version !

Compilation : warnings et erreurs

- En cas de *warnings*, la compilation n'échoue pas, et GCC produit un fichier exécutable—même si *suspect*.
- En cas d'**erreurs** GCC ne va **pas générer un fichier exécutable** : si vous avez déjà un fichier compilé ayant le même nom, l'ancien fichier reste là.
  - Donc **tester le fichier exécutable est inutile** après une erreur de compilation : c'est une ancienne version !

Erreurs fréquentes : indentation

- Les sous-commandes d'une commande sont écrits, dans le bon code, décalées à doite :

```
if (x < 0)
  printf ("x est négative\n");
```

- Ce décalage (en anglais *"indentation"*) n'est pas important pour le compilateur C (différemment de Python).
  - Pour les humains, c'est **très** important.
    - Quand vous écrivez du code en raisonnant, car vous n'êtes pas encore convaincus de sa correction, *vous* devez savoir qu'est-ce vous êtes en train d'essayer.

      Dans le doute, groupez avec des {...}.

Erreurs fréquentes : indentation

- Les sous-commandes d'une commande sont écrits, dans le bon code, décalées à doite :

```
if (x < 0)
  printf ("x est négative\n");
```

- Ce décalage (en anglais *"indentation"*) n'est pas important pour le compilateur C (différemment de Python).
  - Pour les humains, c'est **très** important.
    - Quand vous écrivez du code en raisonnant, car vous n'êtes pas encore convaincus de sa correction, *vous* devez savoir qu'est-ce vous êtes en train d'essayer.

      Dans le doute, groupez avec des {...}.

Erreurs fréquentes : indentation

- Les sous-commandes d'une commande sont écrits, dans le bon code, décalées à doite :

```
if (x < 0)
  printf ("x est négative\n");
```

- Ce décalage (en anglais *"indentation"*) n'est pas important pour le compilateur C (différemment de Python).
  - Pour les humains, c'est **très** important.
    - Quand vous écrivez du code en raisonnant, car vous n'êtes pas encore convaincus de sa correction, *vous* devez savoir qu'est-ce vous êtes en train d'essayer.

      Dans le doute, groupez avec des {...}.

Erreurs fréquentes : indentation

- Les sous-commandes d'une commande sont écrits, dans le bon code, décalées à doite :

```
if (x < 0)
  printf ("x est négative\n");
```

- Ce décalage (en anglais *"indentation"*) n'est pas important pour le compilateur C (différemment de Python).
  - Pour les humains, c'est **très** important.
    - Quand vous écrivez du code en raisonnant, car vous n'êtes pas encore convaincus de sa correction, *vous* devez savoir qu'est-ce vous êtes en train d'essayer.

      Dans le doute, groupez avec des {...}.

Erreurs fréquentes : indentation : exemple 1

Du code mal écrit (version 1) :

```
int a = -10;
if (a > 0)
printf ("foo");
printf ("bar");
printf ("quux");
```

Qu'est ce q'il affiche ?

Erreurs fréquentes : indentation : exemple 2

Version 2 :

```
int a = -10;
if (a > 0)
  printf ("foo");
  printf ("bar");
printf ("quux");
```

Qu'est ce q'il affiche ?

C'est écrit de façon **encore pire** par rapport à la version 1 !

Erreurs fréquentes : indentation : exemple 2

Version 2 :

```
int a = -10;
if (a > 0)
  printf ("foo");
  printf ("bar");
printf ("quux");
```

Qu'est ce q'il affiche ?

C'est écrit de façon **encore pire** par rapport à la version 1 !

Erreurs fréquentes : indentation : exemple 2 bien écrit

Version 2, bien écrite :

```
int a = -10;
if (a > 0)
  printf ("foo");
printf ("bar");
printf ("quux");
```

Qu'est ce q'il affiche ?

Erreurs fréquentes : indentation : exemple 3

Une version encore modifiée :

```
int a = -10;
if (a > 0)
  {
    printf ("foo");
    printf ("bar");
  }
printf ("quux");
```

Qu'est ce q'il affiche ?

Erreurs fréquentes : branches et groupage

Erreurs typiques :

- Mettre un `;` après la parenthèse fermée à droite de la condition dans un `if` est presque toujours une erreur ;
- si votre code n'est pas bien décalé (*"indenté"*) il devient beaucoup plus difficile à comprendre—non juste par moi, mais surtout par vous !

Suggestion :

- dans le doute, désambiguïsez en utilisant des accolades. (Ce n'est pas une erreur, ni « moche », utiliser deux accolades de plus.)

  Rendre votre code plus lisible par vous-même est une raison excellente pour le faire, indépendemment de moi.

Erreurs fréquentes : branches et groupage

Erreurs typiques :

- Mettre un `;` après la parenthèse fermée à droite de la condition dans un `if` est presque toujours une erreur ;
- si votre code n'est pas bien décalé (“*indenté*”) il devient beaucoup plus difficile à comprendre—non juste par moi, mais surtout par vous !

Suggestion :

- dans le doute, désambiguïsez en utilisant des accolades. (Ce n'est pas une erreur, ni « moche », utiliser deux accolades de plus.)

  Rendre votre code plus lisible par vous-même est une raison excellente pour le faire, indépendemment de moi.

Erreurs fréquentes : branches et groupage

Erreurs typiques :

- Mettre un `;` après la parenthèse fermée à droite de la condition dans un `if` est presque toujours une erreur ;
- si votre code n'est pas bien décalé (*"indenté"*) il devient beaucoup plus difficile à comprendre—non juste par moi, mais surtout par vous !

Suggestion :

- dans le doute, désambiguïsez en utilisant des accolades. (Ce n'est pas une erreur, ni « moche », utiliser deux accolades de plus.)

  Rendre votre code plus lisible par vous-même est une raison excellente pour le faire, indépendemment de moi.

Erreurs fréquentes : branches et groupage

Erreurs typiques :

- Mettre un `;` après la parenthèse fermée à droite de la condition dans un `if` est presque toujours une erreur ;
- si votre code n'est pas bien décalé (*"indenté"*) il devient beaucoup plus difficile à comprendre—non juste par moi, mais surtout par vous !

Suggestion :

- dans le doute, désambiguïsez en utilisant des accolades. (Ce n'est pas une erreur, ni « moche », utiliser deux accolades de plus.)

  Rendre votre code plus lisible par vous-même est une raison excellente pour le faire, indépendemment de moi.

Conseil : nommez les valeurs intermédiaires

Exemple :

```
int bit_de_poids_faible = n & 1;
```

Donnez des noms raisonnables à vos variables.

Si vous ne savez pas comment les nommer, vous n'avez pas compris ce que vous voulez faire.

Conseil : nommez les valeurs intermédiaires

Exemple :

```
int bit_de_poids_faible = n & 1;
```

Donnez des noms raisonnables à vos variables.

Si vous ne savez pas comment les nommer, vous n'avez pas compris ce que vous voulez faire.

Erreurs fréquentes : ne pas respecter l'énoncé

Ne pas respecter l'énoncé est une bonne façon de **perdre des points** au contrôle final.

Conditionnelles `if` imbriquées

Syntaxe de la commande conditionnelle (rappel, avec une correction : plus de `;` ici [Le fichier `slides-2.pdf` sur la page web du cours a été corrigé presque immédiatement, mais la version que j'avais montré en classe avait les `;` redondants]) :

commande ::= `if (` expression `)` commande
| `if (` expression `)` commande `else` commande

Cette syntaxe permet d'imbriquer les conditionnelles. Ce n'est pas une nouvelle partie du langage, juste une façon commune de l'utiliser.

Exemple :

```
if (a < 0)
  printf ("a est négative\n"));
else if (a == 0)
  printf ("a est zéro\n"));
else
  printf ("a est positive\n"));
```

if imbriqués : exemple expliqué

Ce code :

```
if (a < 0)
  printf ("a est négative\n"));
else if (a == 0)
  printf ("a est zéro\n"));
else
  printf ("a est positive\n"));
```

marche exactement comme ce code :

```
if (a < 0)
  printf ("a est négative\n"));
else
  {
    if (a == 0)
      printf ("a est zéro\n"));
    else
      printf ("a est positive\n"));
  }
```

Évaluation *short-circuit*

Les expressions contenant `&&` et `||` sont évaluées d'une façon *short-circuit*.

Ce n'est pas le cas pour les opérateurs bit à bit `&` et `|`.

Évaluation *short-circuit*

Les expressions contenant `&&` et `||` sont évaluées d'une façon *short-circuit*.

Ce n'est pas le cas pour les opérateurs bit à bit `&` et `|`.

Entrée (simplifiée)

Pour le moment, acceptez cette syntaxe d'une commande comme « magique » :

- Lire un entier (du terminal) et stocker la valeur écrite par l'utilisateur dans une variable de type `int` :

```
scanf ("%i", & variable);
```

Je vais expliquer le vrai signifié de `&` à la fin, s'il y aura du temps. Ce n'est pas un *« et » bit à bit* : en fait, ce `&` a un seul opérande.

Pour jouer avec le langage, cette forme d'entrée suffit. Bien sûr ce n'est pas la seule.

Entrée (simplifiée)

Pour le moment, acceptez cette syntaxe d'une commande comme « magique » :

- Lire un entier (du terminal) et stocker la valeur écrite par l'utilisateur dans une variable de type `int` :

```
scanf ("%i", & variable);
```

Je vais expliquer le vrai signifié de `&` à la fin, s'il y aura du temps. Ce n'est pas un *« et » bit à bit* : en fait, ce `&` a un seul opérande.

Pour jouer avec le langage, cette forme d'entrée suffit. Bien sûr ce n'est pas la seule.

Entrée (simplifiée)

Pour le moment, acceptez cette syntaxe d'une commande comme « magique » :

- Lire un entier (du terminal) et stocker la valeur écrite par l'utilisateur dans une variable de type `int` :

```
scanf ("%i", & variable);
```

Je vais expliquer le vrai signifié de `&` à la fin, s'il y aura du temps. Ce n'est pas un *« et » bit à bit* : en fait, ce `&` a un seul opérande.

Pour jouer avec le langage, cette forme d'entrée suffit. Bien sûr ce n'est pas la seule.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type

Les littéraux et les variables ont un type. *Les expressions ont un type aussi*.

Disons que `i` et `j` soient deux variables de type `int` et que `f` soit une variable de type `float`.

- Quel est le type de `i + j` ?
  - Bien sûr `int`.
- Quel est le type de `i / j` ?
  - Encore `int` ! (Division entière, car les deux opérandes sont entiers.)
- Quel est le type de `i + f` ?
  - `float` ! (Le type d'une expression « se propage » aux sur-expressions où elle est contenue.)

Dans le dernier exemple on a une conversions de type **implicite**.

Conversions de type explicites : *cast*

Parfois vous souhaitez convertir explicitement le résultat d'une expression, ayant un type, dans une valeur d'une autre type.

C a une syntaxe spéciale pour cette opération appelée en anglais *"cast"* (au sense de *modeler* de l'argile, pas de lancer un projectile).

Syntaxe : expression ::= (type) expression

Le type donnée est le *type destination*.

Sémantique : On évalue l'expression, puis on fait une conversion du résultat vers le type donné. Ce résultat converti est le résultat de l'expression cast.

Conversions de type explicites : *cast*

Parfois vous souhaitez convertir explicitement le résultat d'une expression, ayant un type, dans une valeur d'une autre type.

C a une syntaxe spéciale pour cette opération appelée en anglais *"cast"* (au sense de *modeler* de l'argile, pas de lancer un projectile).

Syntaxe : expression ::= (type) expression
Le type donnée est le *type destination*.

Sémantique : On évalue l'expression, puis on fait une conversion du résultat vers le type donné. Ce résultat converti est le résultat de l'expression cast.

Conversions de type explicites : *cast*

Parfois vous souhaitez convertir explicitement le résultat d'une expression, ayant un type, dans une valeur d'une autre type.

C a une syntaxe spéciale pour cette opération appelée en anglais *"cast"* (au sense de *modeler* de l'argile, pas de lancer un projectile).

Syntaxe : expression ::= (type) expression
Le type donnée est le *type destination*.

Sémantique : On évalue l'expression, puis on fait une conversion du résultat vers le type donné. Ce résultat converti est le résultat de l'expression cast.

Conversions de type : démo

[Démo : conversions de type en C, implicites et explicites]

Flot de contrôle *non structuré* : saut `goto`

Syntaxe :

commande	::=	étiquette `:` commande
	|	`goto` étiquette `;`
étiquette	::=	identifiant

Sémantique : La sémantique d'une commande précédée par une *étiquette* est la même de la commande.
Quand on exécute « `goto` étiquette `;` » l'exécution continue à l'étiqutte donnée au lieu de passer à la prochaine commande après le « `;` ». Une *étiquette* indique un *point dans le programme*.

`goto` est souvent considéré comme difficile à utiliser dans des gros programmes sans les rendre incompréhensibles.

Flot de contrôle *non structuré* : saut goto

Syntaxe :

commande	::=	étiquette : commande
	|	goto étiquette ;
étiquette	::=	identifiant

Sémantique : La sémantique d'une commande précédée par une *étiquette* est la même de la commande.
Quand on exécute « goto étiquette ; » l'exécution continue à l'étiqutte donnée au lieu de passer à la prochaine commande après le « ; ». Une *étiquette* indique un *point dans le programme*.

goto est souvent considéré comme difficile à utiliser dans des gros programmes sans les rendre incompréhensibles.

Flot de contrôle *non structuré* : saut goto

Syntaxe :

commande	::=	étiquette : commande
	|	goto étiquette ;
étiquette	::=	identifiant

Sémantique : La sémantique d'une commande précédée par une *étiquette* est la même de la commande.
Quand on exécute « goto étiquette ; » l'exécution continue à l'étiqutte donnée au lieu de passer à la prochaine commande après le « ; ». Une *étiquette* indique un *point dans le programme*.

goto est souvent considéré comme difficile à utiliser dans des gros programmes sans les rendre incompréhensibles.

Flot de contrôle : boucle `while`

Syntaxe :

commande ::= `while (` expression `)` commande

L'expression s'appelle garde et la commande s'appelle corps.

Sémantique : On évalue la garde, booléenne : si elle est vraie on exécute le corps et puis on recommence, en réévaluant la garde et, si elle est encore vraie, en reexécutant le corps... On s'arrête quand la guarde est fausse.

Si la garde est fausse au début, *le corps n'est jamais exécuté*.

Flot de contrôle : boucle `while`

Syntaxe :

commande ::= `while (` expression `)` commande

L'expression s'appelle garde et la commande s'appelle corps.

Sémantique : On évalue la garde, booléenne : si elle est vraie on exécute le corps et puis on recommence, en réévaluant la garde et, si elle est encore vraie, en reexécutant le corps... On s'arrête quand la guarde est fausse.

Si la garde est fausse au début, *le corps n'est jamais exécuté.*

Flot de contrôle : boucle `while`

Syntaxe :

commande ::= `while (` expression `)` commande

L'expression s'appelle garde et la commande s'appelle corps.

Sémantique : On évalue la garde, booléenne : si elle est vraie on exécute le corps et puis on recommence, en réévaluant la garde et, si elle est encore vraie, en reexécutant le corps... On s'arrête quand la guarde est fausse.

Si la garde est fausse au début, *le corps n'est jamais exécuté*.

Flot de contrôle : boucle do...while

Syntaxe :

commande ::= `do` commande `while ( ` expression ` ) ;`

Encore, l'expression s'appelle garde et la commande s'appelle corps.

Sémantique : Comme la boucle `while`, mais la condition est évaluée *après* le corps à chaque itération. Encore, on s'arrête quand la guarde est fausse.

Le corps d'une boucle `do...while` est exécuté *au moins une fois*.

Flot de contrôle : boucle do...while

Syntaxe :

commande ::= `do` commande `while ( ` expression ` ) ;`

Encore, l'expression s'appelle garde et la commande s'appelle corps.

Sémantique : Comme la boucle `while`, mais la condition est évaluée *après* le corps à chaque itération. Encore, on s'arrête quand la guarde est fausse.

Le corps d'une boucle `do...while` est exécuté *au moins une fois*.

Flot de contrôle : boucle do...while

Syntaxe :

commande ::= `do` commande `while ( ` expression ` ) ;`

Encore, l'expression s'appelle garde et la commande s'appelle corps.

Sémantique : Comme la boucle `while`, mais la condition est évaluée *après* le corps à chaque itération. Encore, on s'arrête quand la guarde est fausse.

Le corps d'une boucle `do...while` est exécuté *au moins une fois*.

Flot de contrôle : boucle `for`

Syntaxe :

commande ::= `for (` expression `;` expression `;` expression `)` commande

Les trois expressions sont appelées, en ordre, *initialisation*, *garde* et *incrément*. La commande est le *corps* de la boucle.

Sémantique : Complètement équivalent à
`{` initialisation `;` `while (` garde `) {` corps `;` incrément `; } }`.

À utiliser quand vous connaissez le nombre d'itérations.

Flot de contrôle : boucle `for`

Syntaxe :

commande ::= `for (` expression `;` expression `;` expression `)` commande

Les trois expressions sont appelées, en ordre, *initialisation*, *garde* et *incrément*. La commande est le *corps* de la boucle.

Sémantique : Complètement équivalent à
`{` initialisation `;` `while (` garde `) {` corps `;` incrément `; } }`.

À utiliser quand vous connaissez le nombre d'itérations.

Flot de contrôle : boucle `for`

Syntaxe :

commande ::= `for (` expression `;` expression `;` expression `)` commande

Les trois expressions sont appelées, en ordre, *initialisation*, *garde* et *incrément*. La commande est le *corps* de la boucle.

Sémantique : Complètement équivalent à
`{` initialisation `;` `while (` garde `) {` corps `;` incrément `; } }`.

À utiliser quand vous connaissez le nombre d'itérations.

Flot de contrôle : boucle `for` : exemple

Exemple d'utilisation typique d'une boucle `for` :

```
int i;
for (i = 0; i < 10; i = i + 1)
  printf ("i est %i\n", i);
```

C'est typique de compter avec une variable *à partir de zéro*, en la comparant dans la garde avec un *mineur strict*.

Dans l'exemple on exécute le corps *10 fois*, pour toute `i` *entre 0 et 9* (compris) : on sort, sans exécuter le corps une autre fois, quand `i` devient égale à 10.

Flot de contrôle : boucle `for` : exemple

Exemple d'utilisation typique d'une boucle `for` :

```
int i;
for (i = 0; i < 10; i = i + 1)
  printf ("i est %i\n", i);
```

C'est typique de compter avec une variable *à partir de zéro*, en la comparant dans la garde avec un *mineur strict*.

Dans l'exemple on exécute le corps *10 fois*, pour toute `i` *entre 0 et 9* (compris) : on sort, sans exécuter le corps une autre fois, quand `i` devient égale à 10.

Don't panic ! (again)

Vos programmes ne marcheront pas à la première tentative.

C'est normal, même pour les experts. Ne vous inquiétez pas !

Welcome to the world of *debugging*.

Bibliography I

Saiu, L. (2018). La page web de mes cours.
`http://ageinghacker.net/teaching`
La page web officielle du cours contient des pointeurs à des ressources web, et une copie des mes transparents.

Stallman, R. and Rothwell, T. (2019). *GNU C Intro and Reference*. Free Software Foundation.
(Manuscrit non encore officiellement publié. Demandez-moi un exemplaire si vous êtes intéressés.).

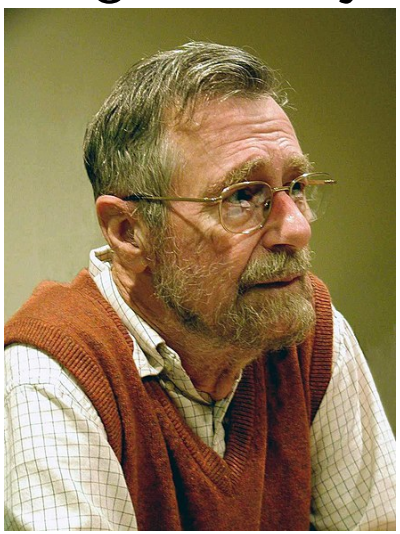
[To do : Backus]

John W. Backus

ALGOL 58, ALGOL 60.

[To do : Dijkstra]

Edsger W. Dijkstra



"*Go To Statement Considered Harmful*" (titre par Niklaus Wirth)

[To do : Knuth]

Donald E. Knuth