École de technologie supérieure

THÈME 6
Nos énoncés sont clairs

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

1. Du tunnel vers la voie de service

COMPÉTENCE VISÉE:
Comprendre le sens technique de la notion d'énoncé et celle d'expression. Savoir programmer correctement des tests à branches multiples avec des "if" ou un "switch". Connaître les particularités du "switch" et savoir expliquer la présence de l'énoncé "break" à l'intérieur de celui-ci.

A. Retour technique sur la notion d'énoncé

Le C est un ogre vorace d'expressions à évaluer. L'énoncé le plus simple en C est toujours constitué d'une expression à évaluer:

```
expression;
```

Les énoncés d'assignation et les appels de fonctions sont également des expressions:

```
nombre = 0;
```

et

```
printf("allo");
```

Lorsqu'une expression est évaluée, le résultat est perdu! La seule façon de récupérer le résultat est de faire en sorte que l'expression produise un effet de bord. Ainsi, l'expression suivante est tout à fait valide mais elle est inutile:

```
n + 1;
```

de même que:

```
estPremier(n);
```

Parfois ces expressions inutiles peuvent être difficiles à détecter, comme dans l'exemple suivant::

```
for (i=0; i < 100; i+1)
  for (j=0; j < 100; j++)
     printf("%i", i * j);
```

Avez-vous détecté le problème?

L'opérateur virgule que nous avons vu dans le dernière thème permet d'écrire une longue expression formée elle-même d'expression à évaluer:

```
c++, a = s * 10, n--;
```

Cette expression retourne évidemment aussi un résultat (le résultat de la dernière expression...)

Nous savons que cette longue expression peut être réécrite en utilisant la notion de bloc. Le bloc contiendra alors 3 expressions distinctes:

```
{ c++;
  a = s * 10;
  n--; }
```

Ces trois expressions distinctes peuvent évidemment être écrites sur une seule ligne. Sur une ligne ou plusieurs, un bloc est un bloc:

```
{ c++; a = s * 10; n--; }
```

Notre petit programme MOTS.C pourrait d'ailleurs être réécrit en utilisant intempestivement la virgule:

SANS VIRGULE	AVEC VIRGULE

`int main(void){`
`   int carac, dansMot;`
`   dansMot = 0;`
	`int main(void){`
	`   int carac, dansMot;`
	`   dansMot = 0;`

`   while ((carac = getchar()) != EOF){`	`   while (carac = getchar(), carac != EOF){`

`     if (isalpha(carac)){`
`       putchar(carac);`
`       dansMot = 1;`
`     }`
`     else if (dansMot){`
`       putchar(' ');`
`       dansMot = 0;`
`     }`
	`     if (isalpha(carac))`
	`       putchar(carac), dansMot = 1;`
	`     else if (dansMot){`
	`       putchar(' '), dansMot = 0;`

`   }`
`   return EXIT_SUCCESS;`
`}`
	`   }`
	`   return EXIT_SUCCESS;`
	`}`

B. Sortir des chemins battus

LE IF

Intuitivement, nous connaissons maintenant la syntaxe générale du IF et nous savons que l'énoncé IF peut en inclure un autre. Voici, en guise d'illustration, un petit programme qui fait la lecture d'une note pour ensuite afficher la catégorie correspondante (A si la note est 100, C si la note est inférieure à 50 et B dans tous les autres cas). Il existe plusieurs façons de coder une solution. Examinons la solution suivante:

```
int main(void){
   int note, categorie;
   scanf("%i", &note);
   categorie = 'B';
   if (note > 50)
      if (note == 100)
         categorie = 'A';
   else
      categorie = 'C';
   printf("Votre catégorie = %c\n", categorie);
   return EXIT_SUCCESS;
}
```

Ce programme est défectueux. Effectuons quelques tests. Si l'on fourni 100 à ce programme, il affiche A. Si l'on fourni 80, il affiche C (oups!) et si l'on fourni 40, il affiche B (oups! encore une fois). Le compilateur ne sait pas ici que le "else" doit être attaché au "if" le plus externe. Par défaut, le compilateur rattache toujours le "else" au "if" le plus près à moins que le bloc soit explicitement mentionné grâce aux accolades, comme dans la version correcte suivante:

```
int main(void){
   int note, categorie;
   scanf("%i", &note);
   categorie = 'B';
   if (note > 50){
      if (note == 100)
         categorie = 'A';
   }
   else
      categorie = 'C';
   printf("Votre catégorie = %c\n", categorie);
   return EXIT_SUCCESS;
}
```

Il faut donc se rappeler que la syntaxe du C est libre de sorte que l'alignement artificielle que vous créez à l'intérieur de vos IF pour vos lecteurs, n'est pas prise en compte par le compilateur.

LE IF À BRANCHES MULTIPLES

Lorsque nous effectuons plusieurs tests en série, on utilise plusieurs IF imbriqués les uns dans les autres, comme dans:

```
   if (note > 90)
     categorie = 'A';
```

```
else if (note > 80)
  categorie = 'B';
else if (note > 70);
  categorie = 'C';
else if (note > 60);
  categorie = 'D';
else
  categorie = 'E';
```

Il faut évidemment comprendre que cette façon d'écrire, qui est la plus utilisée, revient à écrire:

```
if (note > 90)
  categorie = 'A';
else
  if (note > 80)
    categorie = 'B';
  else
    if (note > 70);
      categorie = 'C';
    else
      if (note > 60);
        categorie = 'D';
      else
        categorie = 'E';
```

Somme toute, la première est plus lisible. Qu'en pensez-vous? Rappelez-vous. La syntaxe est libre.

On pourrait également imbriquer le tout comme suit:

```
if (note <= 90)
  if (note <= 80)
    if (note <= 70);
      if (note <= 60);
        categorie = 'E';
      else
        categorie = 'D';
    else
      categorie = 'C';
  else
    categorie = 'B';
else
  categorie = 'A';
```

Pour le compilateur, il n'y a pas d'ambiguïté possible. Chaque "else" est rattaché au dernier "if" ouvert. Ici tout devient une question de lisibilité. Cette structure est plutôt lourde, convenons-en. Le tout consiste à débuter avec le bon test comme dans:

```
if (note <= 60)
  categorie = 'E';
else if (note <= 70)
  categorie = 'D';
else if (note <= 80);
  categorie = 'C';
else if (note <= 90);
  categorie = 'B';
else
  categorie = 'A';
```

Examinons maintenant la solution suivante:

```
if (note <= 60)
  categorie = 'E';
if (note <= 70)
  categorie = 'D';
if (note <= 80);
  categorie = 'C';
if (note <= 90);
  categorie = 'B';
if (note > 90);
  categorie = 'A';
```

Vous remarquerez qu'il n'y a aucun "else". On a donc affaire à une série de petits tests en série. Est-ce que ce petit bout de code fonctionne bien? Pour le savoir, testons toutes les possibilités.

valeur de la variable "note"	valeur de "categorie" après les tests
95	A (tout va bien)
85	B (tout va bien)
75	B OUPS!
65	B OUPS!
55	B OUPS!

Pour coder une solution correcte avec une série de "if", il faudrait utiliser des conditions plus complexes:

```
if (note <= 60)
  categorie = 'E';
if (note <= 70 && note > 60)
  categorie = 'D';
if (note <= 80 && note > 70);
  categorie = 'C';
if (note <= 90 && note > 80);
  categorie = 'B';
if (note > 90);
  categorie = 'A';
```

Cela est inutile et coûteux dans la mesure où le "else" existe!

Examinons maintenant le programme COUTEM.C, il effectue le décompte du nombre d'espaces, de caractères alphabétiques, numériques et de ponctuations dans une source texte (interactif ou fichier).

Code	Explication

`#include <stdio.h>`
`#include <stdlib.h>`
`#include <ctype.h>`
	La librarie "ctype" a été incluse afin d'avoir accès aux fonctions "isalpha", "isspace", "isdigit" et "ispunct".

`#define  CNT_WIDTH  11`	Nous reviendrons plus loin sur l'utilisation de cette constante...
`double pct(unsigned long count,` `           unsigned long total);`	Puisque nous vous suggérons de prendre l'habitude de placer vos prototypes à l'extérieur du "main", voici celui de la fonction "pct". Cette fonction calcule le pourcentage correspondant à la quantité "count" relativement au total.

`int main()`
`{`
`  int           c;`
`  unsigned long spaces;`
`  unsigned long letters;`
`  unsigned long digits;`
`  unsigned long puncts;`
`  unsigned long others;`
`  unsigned long t;`
	Le début du programme. "c" sera utilisé pour stocker le dernier caractère lu. "spaces", "letters", "digits", "puncts" et "others" seront utilisées pour comptabiliser le nombre d'espaces, de caractères alphabétiques, numériques, de ponctuation et les autres respectivement. Finalement, la variable "t" sera utilisée pour comptabiliser le nombre total de caractères.

`spaces = letters = digits = puncts = others = 0;`	On démarre avec tous les compteurs à zéro.

`  while ((c = getchar()) != EOF)`
`    if (isspace(c))`
`      spaces++;`
`    else if (isalpha(c))`
`      letters++;`
`    else if (isdigit(c))`
`      digits++;`
`    else if (ispunct(c))`
`      puncts++;`
`    else`
`      others++;`
	La boucle de lecture. Examinez la structure du "if". Notez comment sont utilisées les fonctions. Inutile d'écrire (isspace(c) != 0) n'est-ce pas?
	
	Notez qu'il n'y a pas d'accolades pour cerner le bloc du "while". Cela est inutile puisque cette boucle ne contient qu'un seul (GROS) énoncé. Vous pouvez mettre les accolades cependant. Cela est tout à fait correct.

`t = spaces + letters + digits + puncts + others;`	On pouvait se demander pourquoi la variable "t" n'était pas initialisée à zéro. En voici la raison: "t" est calculée grâce à l'addition de toutes les autres compteurs, tout simplement.
`printf("Total   %*lu\n\n", CNT_WIDTH, t);`	Impression du total... mais quel est donc l'usage de cette astérisque dans %*lu ??? Et pourquoi donc semble-t-on afficher la valeur de la constante CNT_WIDTH ??? Faisons une pause avant de regarder le reste...

L'astérisque permet de fournir le nombre de positions utilisées pour afficher l'entier en utilisant une variable, une constante ou toute expression retournant un entier au lieu de fournir ce chiffre directement. C'est très pratique lorsqu'on désire ajuster la "beauté" de nos affichages. Ainsi, puisque la constante CNT_WIDTH vaut 11, le printf est l'équivalent de:

```
printf("Total   %11lu\n\n", t);
```

La valeur de CNT_WIDTH vient donc remplacer l'astérique et n'est donc pas affichée comme telle. Si la constante vaut 5, alors le printf est l'équivalent de:

```
printf("Total   %5lu\n\n", t);
```

Nous pouvons maintenant poursuivre la lecture du programme:

```
  if (t != 0)
  {
    printf("spaces  %*lu %5.1f%%\n", CNT_WIDTH, spaces, pct(spaces, t));
    printf("letters %*lu %5.1f%%\n", CNT_WIDTH, letters, pct(letters, t));
    printf("digits  %*lu %5.1f%%\n", CNT_WIDTH, digits, pct(digits, t));
    printf("puncts  %*lu %5.1f%%\n", CNT_WIDTH, puncts, pct(puncts, t));
    printf("others  %*lu %5.1f%%\n", CNT_WIDTH, others, pct(others, t));
  }
```

Donc, si le total permet de calculer correctement les pourcentages, on effectue l'affichage de chaque compteur et du pourcentage correspondant tel que calculé par la fonction "pct".

```
  return EXIT_SUCCESS;
}
```

Le programme est terminé.

La définition de la fonction "pct" est toute simple:

```
double pct(unsigned long count, unsigned long total)
  { return count * 100.0 / total; }
```

LE SWITCH

Lorsque nous testons ainsi la même expression de manière successive, il est courant d'utiliser, à la place du IF à branches multiples, l'énoncé SWITCH. La syntaxe générale du SWITCH est la suivante:

```
switch (expression)
{
   case étiquette : liste d'énoncés
   case étiquette : liste d'énoncés
   ...
   case étiquette : liste d'énoncés
   default : liste d'énoncés
}
```

Lors de l'évaluation du "switch" la valeur de <expression> est comparée à celle des "étiquettes". Si les deux valeurs correspondent, la liste d'énoncés suivant le point est exécutée. Si aucun cas ne correspond, c'est la liste d'énoncés sous "default" qui sera exécutée. Mais cette simplicité cache en fait une relative complexité.

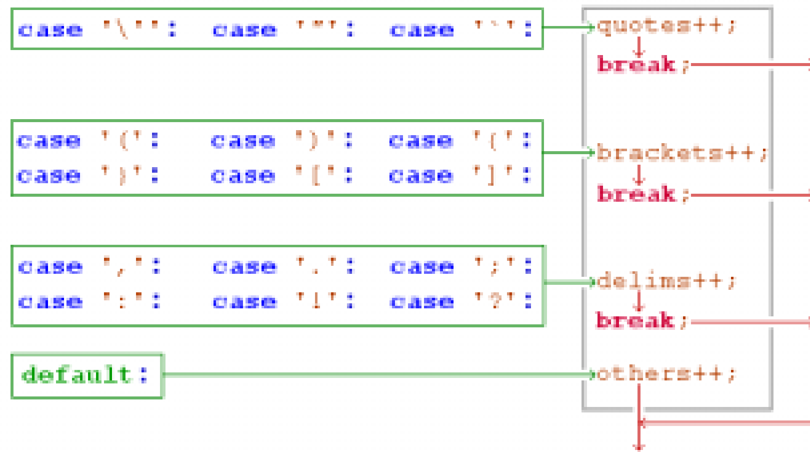
Il serait pénible de commenter en long et en large la syntaxe du "switch". La meilleure façon de le comprendre est de regarder un exemple. Examinez et faites exécuter le programme COUTPCT.C qui effectue le décompte du nombre de tous les types de «parenthèses» possibles (incluant '(', ')', '[', ']', '{', '}'), de tous les types «d'apostrophes» (incluant, ', ", et `) et de tous les types de délimiteurs possibles (espace, '.', ',', '!', '?', ':', et ';'). Voici l'énoncé "switch" utilisé par ce programme:

```
switch(c)
{
  case '\'':
  case '"':
  case '`':
    quotes++;
    break;
  case '(':
  case ')':
  case '{':
  case '}':
  case '[':
  case ']':
    brackets++;
    break;
  case ',':
  case '.':
  case ';':
  case ':':
  case '!':
  case '?':
    delims++;
    break;
  default:
    others++;
}
```

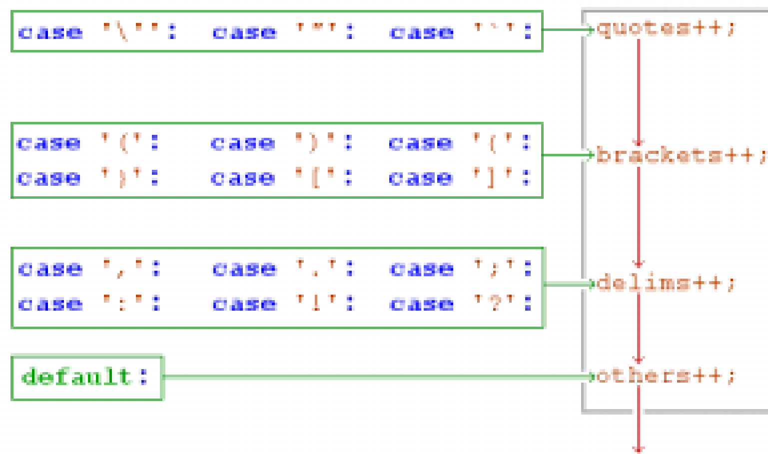
Notez bien la syntaxe de chaque "case":

```
case <valeur> :
```

C'est uniquement l'énoncé "break" qui permet de sortir du "switch". En fait, il faut imaginer la structure du "switch" comme suit:



Dès qu'un des cas mentionnés est respecté, on pénètre alors dans l'antre du "switch". Seul le "break" peut nous en faire sortir! Si le "break" est absent, l'exécution se poursuit dans les autres groupes du "switch" même si la valeur testée ne correspond à aucun des cas qui y sont identifiés!



Le break est très important. Considérez le "switch" suivant:

```
switch (categorie) {
   case 'A' : printf("Excellent\n");
   case 'B' : printf("Très bon\n");
   case 'C' : printf("Bon\n");
   case 'D' : printf("Passable\n");
   case 'E' : printf("Échec\n");
}
```

Supposons que la variable "categorie" contienne 'A', ce bout de code affichera:

```
Excellent
Très bon
Bon
Passable
Échec
```

Si la variable "categorie" contient 'C', ce bout de code produira:

```
Bon
Passable
Échec
```

Le "break" sert donc à sortir du "switch"... au bon moment.

EXERCICE (DIFFICILE): Si vous exécutez le programme IDENTIF.C (du thème 5) sur lui-même, vous allez récupérer la liste de mots suivante:

```
include stdio h include stdlib h include ctype h int main
void int x23 Uniquement pour tester un identificateur avec
des chiffres int carac dansMot dansMot while carac getchar
EOF if isalpha carac dansMot isdigit carac putchar carac
dansMot else if dansMot putchar dansMot return EXIT SUCCESS
```

Vous remarquerez que les mots présents en commentaires sont également récupérés. La question est la suivante: modifiez le programme afin que les mots situés entre /* et */ ne soient pas récupérés. Voici une solution envisageable: IDENTIF2.C.

2. De la ronde au saut à ski

COMPÉTENCE VISÉE:
Être capable d'interpréter des boucles "do...while" et de les traduire en "while" et vice-versa. Être capable d'interpréter des boucles "for" dans lesquelles une des sections est absente. Savoir utiliser correctement l'énoncé "break". Savoir interpréter du code dans lequel l'énoncé "continue" est utilisé. Être capable d'interpréter du code dans lequel apparaît l'énoncé nul.

A. Tourner jusqu'à en avoir des hauts le coeur

WHILE

La syntaxe du WHILE devrait désormais être claire. Rappelons son fonctionnement brièvement:

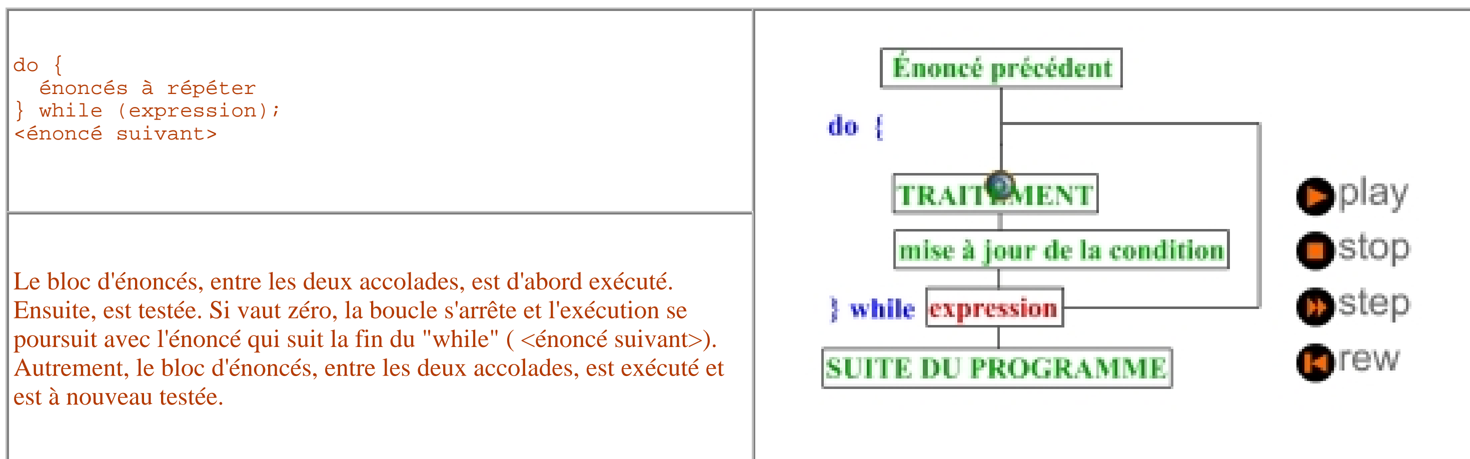
```
while (expression){
  énoncés à répéter
}
<énoncé suivant>
```

INITIALISATION
while expression
{
TRAITEMENT
mise à jour de la condition
}
SUITE DU PROGRAMME
play
stop
step
rew

Si vaut zéro, la boucle s'arrête et l'exécution se poursuit avec l'énoncé qui suit l'accolade fermante <énoncé suivant>. Autrement, le bloc d'énoncés, entre les deux accolades, est exécuté et est à nouveau testée.

DO...WHILE

Il existe une autre boucle disponible, il s'agit du do...while.



Très utile pour sauter des espaces ou dans toute situation où l'on est certain que le passage une fois dans la boucle est toujours nécessaire. Ainsi, les deux boucles suivantes sont équivalentes et servent toutes deux à "sauter" les caractères qui ne sont pas alphabétiques.

```
c = getchar();
while (!isalpha(c)){
  c = getchar();
}
```

```
do {
  c = getchar();
} while (!isalpha(c));
```

FOR

Voici encore une boucle que vous connaissez relativement bien, le "FOR". Rappelons sa syntaxe:

```
for (init; test; action){
  énoncés à répéter
}
```

Habituellement, init et action sont constituées d'assignation ou d'appel de fonctions et test est constitué d'une expression relationnelle. En fait, n'importe quelle expression peut être utilisée et de plus, les trois sections sont optionnelles. Ainsi,

```
for (;;)
```

est une boucle infinie. Les point-virgule doivent toujours être présents cependant.

Voici un exemple avec la section laissée vide. Il s'agit d'une fonction permettant de calculer X^n si "n" est un entier positif. Notez que la section laissée vide.

```
double power(double x, int n)
{
  double p;

  p = 1.0;
  for ( ; n > 0; n--)
      p *= x;
  return p;
}
```

Il convient de noter que , et peuvent être composés d'une seule expression. On ne peut donc pas faire plusieurs initialisations du type:

```
for (x = 0; i = 0; i < 15; i++)
```

C'est souvent là que l'opérateur virgule est utilisé. Il est fréquent que des programmeurs se laissent séduire par l'écriture de code très concis en utilisant des raccourcis partout où ils le peuvent. de mon point de vue, ces programmeurs sont en fait de grands associaux qui ont besoin de ce petit pouvoir que leur confèrent les possibilités syntaxiques du C! Voici deux versions du même programme. Ils effectuent la numérotation des lignes dans une source texte. Le premier est effectué avec un "while" et comporte quelques procédés raccourcis que je qualifierais de "digestibles". Le second utilise un "for" assez alambiqué! Notez l'utilisation en suffixe de l'opérateur ++. Il nous assure que la valeur de la variable "lineno" sera affichée APRÈS avoir été ncrémentée.

```
int main()
{
  int           c;
  int           lastch;
  unsigned long lineno;

  lineno = 0;  lastch = '\n';
  while ((c = getchar()) != EOF)
  {
    if (lastch == '\n')
      printf("%8lu ", ++lineno);
    putchar(lastch = c);
  }
  return EXIT_SUCCESS;
}
```

```
int main()
{
  int           c;
  int           lastch;
  unsigned long lineno;

  for (lineno = 0, lastch = '\n';
       c = getchar(), c != EOF;
       lastch = c, putchar(c))
    if (lastch == '\n')
      printf("%8lu ", ++lineno);

  return EXIT_SUCCESS;
}
```

L'UTILISATION DE CES RACCOURCIS (VIRGULE ETC.) NE REND PAS DU TOUT VOTRE PROGRAMME RAPIDE! Si vous respectez vos lecteurs, SOYEZ CLAIRS!

B. Je me repose

Plusieurs d'entre-vous ont été confrontés à l'énoncé nul en écrivant des IF du style (notez le point-virgule après la condition):

```
if ();
```

ou encore des boucles comme:

```
for(x = a; x <= b; x += 0.1);
{
  énoncés traîtreusement indentés!
  ce bloc ne sera pas répété!
}

while (x <= b);
{
  énoncés traîtreusement indentés!
  ce bloc ne sera pas répété!
}
```

L'utilisation de cet énoncé nul doit être prudente, est-il nécessaire de le mentionner? On peut ainsi construire des boucles très concises. Voici la réécriture comparée du "while" et du "do... while" que nous avons vus plus haut. Nous avons utilisé ici l'énoncé nul dans le cas du "while" et des accolades vides, dans le cas du "do...while". On notera que ces accolades sont obligatoires pour le "do...while".

```
while (!isalpha(c = getchar()))
  ;
```

```
do {
} while (!isalpha(c = getchar()));
```

En général, pour s'assurer que vous êtes bien compris (à moins de participer au concours international du programme le plus obscur), placer donc votre énoncé nul sur une ligne distincte. vous vous remercierez vous-même lorsque vous lirez votre code 1 mois plus tard!

C. Sortons d'ici! Je n'en peux plus! Ne t'en fais pas. Je connais un racourci pour continuer.

BREAK

Souvent appelé, le GOTO en cravate, le break sert à sortir immédiatement d'une BOUCLE (il est également utilisé dans le SWITCH... parce que les concepteurs du C ont eu la paresse de ne pas utilisé un autre mot réservé!).

Vous vous rappelez la petite fonction "lireEntre" permettant de faire des lectures répétitives jusqu'à ce que l'utilisateur saisissse un nombre entre deux bornes? Comparez les deux versions. La première version utilise une variable entière pour sortir de la boucle, la seconde utilise un énoncé "break" pour effectuer la même tâche.

```
int lireEntierEntre(int borneInf, int borneSup){
  int valeur;
  int fini;
  fini = 0;
  while (!fini) {
    printf("\nDonnez-moi un entier entre %i et %i : ", borneInf, borneSup);
    if (scanf("%i", &valeur) == 1) {
      if ((valeur >= borneInf) && (valeur <= borneSup))
        fini = 1;
    }
    fflush(stdin);
  }
  return valeur;
}
```

```
int lireEntierEntre(int borneInf, int borneSup){
  int valeur;
  while (1) {
    printf("\nDonnez-moi un entier entre %i et %i : ", borneInf, borneSup);
    if (scanf("%i", &valeur) == 1) {
      if ((valeur >= borneInf) && (valeur <= borneSup))
        fflush(stdin), break;
    }
    fflush(stdin);
  }
  return valeur;
}
```

Pouvez-vous expliquer pourquoi un "fflush" a été ajouté dans cette version?

Notez également la virgule qui n'était pas du tout nécessaire. Sans elle le "fflush" et le "break" doivent cependant être enveloppés d'accolades.

CONTINUE

Voici un autre GOTO en cravate: le CONTINUE. Il permet de sauter le reste des énoncés d'une boucle et provoque une évaluation immédiate du test de cette boucle (s'il s'agit d'un FOR, l'incrémentation est effectuée d'abord et le test ensuite). Voici un programme permettant de convertir les heures exprimées en format 24h (ex. 21:45, 23:17) en format 12h (9:45, 11:17). Notez 'utilisation du "continue" et lisez bien les commentaires...

```
int main()
{
  unsigned int mhour;
  unsigned int min;
  unsigned int stdhour;
```

"mhour" contient les heures lues en format 24h (ex. pour 21:15, "mhour" contiendra 21).

"min" contient les minutes soit en format 24h ou en format 12h.

"stdhour" contient les heures en format 12h.

```
  while (printf("Enter military time (as xx:xx): "),
         scanf("%u:%u", &mhour, &min) == 2)
  {
```

Les auteurs appelle le format 24h, l'heure militaire...

Lecture des heures et des minutes en format 24h dans la condition du "while". Notez bien l'utilisation de la VIRGULE.

Notez la chaîne de caractères utilisée dans le "scanf". L'utilisateur DEVRA, pour que le "scanf" fonctionne, saisir les deux-points mentionnés dans la chaîne de formattage.

```
    if (mhour >= 24 || min >= 60)
    {
      printf("Error: military time is from 0:00 to 23:59\n");
      continue;
    }
```

Voilà le fameux "continue". Si l'utilisateur a saisi des données inacceptable, un petit message est affiché et le "continue" nous ramène sur l'évaluation du "while". Le reste du code dans la boucle ne sera pas effectué.

```
    stdhour = mhour;              /* assume AM */
    if (mhour <= 12)
    {
      if (mhour == 0)
        stdhour = 12;             /* midnight */
    }
    else
      stdhour = mhour - 12;       /* PM */
    printf("Standard time: %u:%02u\n", stdhour, min);
```

Traitement normal s'il n'y a pas d'erreur.

```
  }
  return EXIT_SUCCESS;
}
```

Fin de la boucle et fin du programme

L'intérieur de la boucle pourrait facilement être réécrit comme suit:

NOTEZ l'utilisation du "else" pour "imiter" l'effet du "continue".

```
    if (mhour >= 24 || min >= 60)
      printf("Error: military time is from 0:00 to 23:59\n");
    else {
      stdhour = mhour;                   /* assume AM */
      if (mhour <= 12)
      {
        if (mhour == 0)
          stdhour = 12;                  /* midnight */
      }
      else
        stdhour = mhour - 12;            /* PM */
    }
    printf("Standard time: %u:%02u\n", stdhour, min);
```

CONCLUSION: Utilisez cet énoncé avec parcimoni et sachez que le CONTINUE peut toujours être remplacé par un if..else...

GOTO

Le tour des énoncés du C ne serait pas complet si on ne mentionnait pas l'infâme GOTO.

Sa syntaxe comprend une étiquette (un identificateur valide), qui sera placé à l'endroit où l'on désire se rendre, et l'énoncé "goto étiquette" placé judicieusement(!) dans le code.

"Where time or intelligence are lacking, a goto may do the job."

-- M.E. Hopkins, "A Case for the GOTO," 1972.

Voici un GOTO qui imite l'énoncé "break" dans la fonction "lireEntierEntre":

```
int lireEntierEntre(int borneInf, int borneSup){
  int valeur;
  while (1) {
    printf("\nDonnez-moi un entier entre %i et %i : ", borneInf, borneSup);
    if (scanf("%i", &valeur) == 1) {
      if ((valeur >= borneInf) && (valeur <= borneSup))
      {  fflush(stdin); goto idiot; }
    }
    fflush(stdin);
  }
  idiot:
  return valeur;
}
```

Je n'élaborerai pas sur le sujet car il existe très très très peu de cas où le GOTO soit nécessaire pour rendre le code clair. On peut trouver certains exemples dans Robert Sedgewick (1992) Algorithms in C : Fundamentals, Data Structures, Sorting, Searching, Addison-Wesley (niveau intermédiaire ou avancé)

Références

Les débats sur l'utilisation des GOTO sont nombreux et ressurgissent de temps à autre dans la littérature informatique. L'un des premiers à avoir soulevé le problème est Edsger W. Dijkstra dans Go To Statement Considered Harmful (article paru originalement dans "Communications of the ACM", Vol. 11, No. 3, March 1968, pp. 147-148)

Andrew Koenig (chercheur chez AT&T Bell Laboratories) a écrit un texte intitulé "C Traps and Pitfalls" qui illustre quelques embuches de la syntaxe du C. Certaines sont assez surprenantes. Vous l'apprécierez encore plus à la fin du cours. Cet article ne constitue qu'un fragment de ce qu'on peut maintenant trouver dans son livre portant le même titre, publié chez Addison-Wesley en 1989.

Les citations de Edsger W. Dijkstra sont toujours savoureuses. En voici quelques unes en vrac:

- "If debugging is the process of removing bugs, then programming must be the process of putting them in."
- "Computer science is no more about computers than astronomy is about telescopes."
- "It is potentially impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration. "
- "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence. "
- "Program testing can be used to show the presence of bugs, but never to show their absence!." in Notes on Structured Programming

(1972)