Style de programmation

Vladimir Makarenkov
Professeur au département d'informatique de l'Université du Québec à Montréal.
Adapté par Emmanuel Chieze

Introduction

- N'avez vous jamais:
  - dépensé inutilement du temps à coder un mauvais algorithme ?
  - utilisé une structure de données trop complexe ?
  - testé un programme, mais laissé passer un problème évident ?
  - passé beaucoup de temps à essayer de corriger un bogue que vous auriez dû trouver en quelques minutes ?

Introduction

- N'avez vous jamais:
  - dû vous battre pour déplacer un programme d'une station de travail (Unix) vers un PC (Windows) ou vice-versa ?
  - essayé de réaliser une modification mineure dans le programme d'un autre ?
  - réécrit un programme parce que vous ne compreniez pas son fonctionnement ?
  - eu besoin que votre programme s'exécute trois fois plus vite et utilise moins de mémoire ?

Introduction

- Régler ces problèmes s'avère difficile.
- La pratique de la programmation n'est généralement pas au centre des enseignements d'informatique.
- Certains programmeurs acquièrent cette connaissance au fur et à mesure de leur expérience grandissante.
- La simplicité, la précision et la standardisation sont souvent perdues de vue.

Introduction

- La **simplicité** permet de conserver les programmes courts et maniables.
- La **clarté** qui garantit la facilité de compréhension des programmes tant pour les personnes que pour les machines.
- La **standardisation** qui garantit le bon fonctionnement des programmes dans une diversité de situations et s'adapte parfaitement aux nouvelles situations.
- L'**automatisation** qui laisse la machine faire le travail à notre place.

Introduction

- Un bon style de programmation concourt à une bonne programmation.
- Ils possèdent moins d'erreurs et sont donc plus faciles à déboguer et à modifier.
- Les principes de tout style de programmation:
  - sont fondés sur le sens commun;
  - guidés par l'expérience;
  - non guidés par des règles arbitraires ni des prescriptions.

Introduction

- Le code doit:
  - être simple et consistant;
  - avoir une logique claire et évidente;
  - utiliser des expressions naturelles;
  - utiliser un langage conventionnel;
  - utiliser des noms compréhensibles et significatifs;
  - éviter les spécificités astucieuses ainsi que les constructions inhabituelles;
  - avoir un aspect cohérent.

Les noms

- Un nom de variable ou de fonction:
  - catalogue un objet;
  - transmet l'information le concernant.
- Un nom doit être:
  - informatif;
  - concis;
  - facile à mémoriser;
  - si possible prononçable.
- Plus la portée d'une variable est importante, plus son nom devra être évocateur de ce qu'elle représente.

Les noms

```
#define UN 1
#define DIX 10
#define VINGT 20
```

- Pourquoi ces noms sont discutables ?
- Imaginer qu'on ait un tableau de vingt éléments qui doit être agrandi.
- `#define TAILLE_TABLEAU 20`

Les noms: variables globales et locales

- Les variables globales:
  - peuvent être employées n'importe où dans un programme;
  - leurs noms doivent être suffisamment longs et descriptifs pour suggérer au programmeur leur signification.

```
int nbreElemCourant = 0; // le nombre d'éléments
                         // courant dans la file d'attente.
```

- Les fonctions, les classes et structures de données globales doivent être également désignées par des noms descriptifs.

Les noms: variables globales et locales

- Les variables locales peuvent être définies par des noms plus courts.
- Pour décrire une variable “nombre de points” dans une fonction:
  - nbre est suffisant;
  - nbrePoints est excellent;
  - nombreDePoints péchant par excès.

Les noms: variables globales et locales

- Les variables locales employées de façon conventionnelle peuvent posséder des noms très courts:
  - i et j pour des indices de tableaux;
  - p et q pour des pointeurs;
  - s et t pour des chaînes de caractères.
- Comparez:

```
for (theElementIndex = 0; the ElementIndex <
      numberOfElement;  theElementIndex++)
      elementArray[theElementIndex] = theElementIndex;
```

- Et

```
for (i = 0; i < nbreElems; i++)
      elem[i] = i;
```

Les noms: convention

- En général:
  - Les noms commençant et se terminant par "_" sont réservés.
  - Évitez les noms qui diffèrent seulement par la casse (tete et Tete) ou le souligné (nbre_Elem et nbreElem).
  - Évitez les noms qui se ressemblent (i et l).
  - Les noms globaux doivent avoir un préfixe identifiant leur module.
  - Utilisez p pour désigner un pointeur (pNoeud ou noeudP).
  - Mettez les constantes en majuscules (MAX).
  - Des règles plus fines existent (pStrNom, strToNom, strFromNom).

Les noms: utilisez des termes actifs pour les fonctions

- Les noms doivent être basés sur des verbes actifs:

```
now = date.getTime();
putchar('\n');
```

- Les fonctions retournant une valeur booléenne doivent être nommées de façon à ce que la valeur retournée ne soit pas équivoque:

```
if (checkOctal(c))        /* doit être remplacé par */
if (isOctal(c))
```

Les noms: soyez minutieux

- Un nom transmet de l'information
- Un nom ambigu peut entraîner des bogues difficiles à détecter.
- Que fait la fonction suivante ?

```
bool inChaine(char * chaine, int taille, char c) {
  int i = 0;
  while ( (i < taille) && (chaine[i] != c)) i++;
  return (i == taille);
}
```

La fonction retourne faux si c est dans la chaîne.

Les expressions et instructions

- Écrivez les expressions et instructions de façon compréhensible à tout le monde.
- Formatez avec soin (utilisez des espaces autour des opérateurs).
- Analogie: si vous gardez votre bureau rangé, alors vous pouvez retrouver vos affaires facilement.
- Contrairement à votre bureau, vos programmes seront sûrement examinés par d'autres personnes.

Instructions: indentez votre code

- L'indentation permet de montrer la structure du code.
- Comparez les trois codes suivants:

```
for (n++; n<100; table[n++]='\0');
 *i = '\0'; return ('\n');
```

```
for (n++; n < 100; table[n++] = '\0')
    ;
*i = '\0';
return ('\n');
```

```
for (n++; n < 100; n++)
    table[n] = '\0';
*i = '\0';
return ('\n');
```


Expressions

- Employez la forme de lecture naturelle.
- Remplacez:
  - if (!(age < ageMinDor) || !(age >= ageMaxEtudiant)) par
  - if ((age >= ageMinDor) || (age < ageMaxEtudiant))
- Utilisez les parenthèses pour dissiper les ambiguités:
  - anneeBissex = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
  - anneeBissex = ((y%4== 0) && (y % 100 !=0)) || (y % 400 == 0);
- Que signifie l'expression suivante ?

```
if (x & MASK == BITS)  :
```

  - if ((x & MASK) == BITS) ou
  - if (x & (MASK == BITS))

Expressions

- Séparez les expressions complexes.
- Remplacez l'expression:

```
*x += (*xp = (2 * k < (n-m)? c[k+1] : d[k--]));
```

par

```
if (2*k < n-m)
  *xp = c[k+1];
else
   *xp = d[k--];
*x += *xp;
```

- Soyez clair, le but est d'écrire un code explicite et non un code ingénieux.

Que fait l'expression suivante ?

```
sousCle = sousCle >> (bitOff - ((bitOff >> 3) << 3));
sousCle >>= bitOff & 0x7;
```


Expressions

- Faites attention aux effets de bords. Remplacez :

```
str[i++] = str[i++] = ' ';
```

par

```
str[i++] = ' ';
str[i++] = ' ';
```

- Faites attention à l'ordre d'évaluation des paramètres. Remplacez :

```
scanf("%d %d", &yr, &profit[yr]);
```

par

```
scanf("%d", &yr);
scanf("%d", &profit[yr]);
```

- Comment améliorer cette expression ?

```
if (!(c == 'y' || c == 'Y'))
  return;
```

Cohérence

- Effectuez une opération de la même manière, à chaque utilisation:
  - il ne faudrait pas copier une chaîne de caractères par ***strcpy*** quelques fois et caractère par caractères d'autres fois.
- Employez une indentation cohérente et un style d'accolade:
  - adopter l'une des conventions et restez fidèle.
- Lorsque vous travaillez sur un programme que vous n'avez pas écrit, conservez le style dans lequel il a été créé.

Expressions idiomatiques

- Employez les expressions idiomatiques dans un but de cohérence.
- Initialisation d'un tableau:

```
i = 0;
while ( i <= n-1)
    table[i++] = 1.0;
```

ou

```
for (i = 0; i < n;)
    table[i++] = 1.0;
```

ou

```
for (i = 0; i < n; i++)
    table[i] = 1.0;
```

- Employez la boucle standard pour dérouler une liste:

```
for (p = list; p != NULL; p = p->next)
```

- Pour une boucle infinie: `for (;;)` ou `while (1)`.

Expressions idiomatiques (2)

- Utilisez l'instruction idiomatique de lecture conditionnelle:

```
while ((c = getchar()) != EOF)
    putchar (c);
```

- Utilisez do-while lorsqu'il y a au moins une itération dans une boucle.

Expressions idiomatiques: if et else if

- Employez des else-if pour des branchements conditionnels à choix multiples :

```
if (condition1)
   instruction1
else if (condition2)
   instruction2
 ...
else if (conditionn)
   instructionn
else
   instruction par défaut
```

Expressions : if et else if

Remplacez le code ci-dessous:

```
if (argc == 3)
  if ((fin = fopen (argv[1], "r")) != NULL)
    if ((fout = fopen (agv[2], "w")) != NULL) {
       while ((c = getc (fin)) != EOF)
           putc (c,fout);
    }
    else
       printf("Impossible d'ouvrir la sortie %s\n", argv[2]);
  else
     printf("Impossible d'ouvrir l'entrée %s\n", argv[1]);
else
  printf("Usage: cp fichier_entree fichier_sortie\n");
```


Expressions : if et else if (2)

Par:

```
if (argc != 3)
  printf ("Usage: cp fichier_entree fichier_sortie\n");
else if ((fin = fopen (argv[1], "r")) == NULL)
  printf ("Impossible d'ouvrir l'entrée %s\n", argv[1]);
else if ((fout = fopen (agv[2],"w")) != NULL) {
    printf ("Impossible d'ouvrir la sortie %s\n", argv[2]);
    fclose (fin);
}
else {
  while ((c = getc (fin)) != EOF)
      putc (c, fout);
  fclose (fin);
  fclose (fout);
}
```

Expressions : switch vs if-else

- Les tentatives de réutilisation de portions de code conduisent souvent à des programmes condensés.

  Remplacez le code suivant:

```
switch (c) {
case '-': sign = -1;
case '+': c = getchar ();
case '.': break;
default: if (!isdigit (c))
             return 0;
}
```


Expressions : switch vs if-else (2)

Par une structure else-if plus adéquate :

```
if (c == '-') {
   sign = -1;
   c = getchar ();
}
else if (c == '+') {
   c = getchar ();
}
else if ((c != '.') && (!isdigit (c))) {
   return 0;
}
```

Les nombres magiques

- Les nombres magiques représentent:
  - les constantes, les tailles de tableau,
  - les positions de caractère, les facteurs de conversion,
  - les valeurs littérales numériques.
- Donnez des noms aux nombres magiques (tout nombre autre que 0 et 1).
- Définissez les nombres sous forme de constantes, non de macros.
  - enum { MAX_ELEM = 30, …}; // en C
  - const int MAX_ELEM = 30; // en C++
- Utilisez des constantes caractères, non des entiers:
  - if ((c >= 'A') && (c <= 'Z')) au lieu de if ((c >= 65) && (c <= 90))
  - mais au lieu des 2 instructions, utilisez isUpper()

Les nombres magiques

- Utilisez les valeurs appropriées lors des affectations.

  Remplacez les expressions suivantes:

```
ptr = 0;
nom[n-1] = 0;
double x = 0;
```

  Par

```
ptr = NULL;
nom[n-1] = '\0';
double x = 0.0;
```

- Utilisez sizeof pour calculer la taille d'un objet:
  - n'utilisez pas une taille explicite
  - employez sizeof (int) au lieu de 2 ou 4
  - employez sizeof (array[0]) au lieu de sizeof (int)

Les commentaires

- Ils aideront la personne qui lira votre code.
- Ils ne doivent pas indiquer ce que le code dit de façon manifeste, ni le contredire.
- Ils doivent clarifier les choses subtiles.
- Commentez toujours les fonctions et les données globales

```
struct State {              /*préfixe + liste de suffixes */
   char *pref[NPREF];    /*mots préfixes */
   Suffix *suf;             /*liste de suffixes */
   State *next;             /*le suivant dans la table de hachage*/
 };
```


Les commentaires inutiles

Ne soulignez pas ce qui est évident

```
while (((c = getchar ()) != EOF) && isspace (c));   /* saute les espaces */

if (c = EOF)                                        /* fin de fichier atteinte */
   type = endoffile;
else if (c == '(')                                  /* parenthèse à gauche */
   type = leftparen;
else if (c ==')')                                   /* parenthèse à droite*/
   type = rightparen;
else if (c == ';')                                  /* point-virgule*/
   type = semicolon;
else if (is_op (c))                                 /* opérateur */
   type = Operator;
else if (is_digit (c))                              /* nombre */
   type = digit;
```

Les commentaires

- Ne commentez pas du mauvais code, réécrivez le.

```
/* Si "result" est à 0 une correspondance a été trouvée
   et true (non-zéro) est donc retourné. Dans les
   autres cas,"result" n'est pas égal à zéro et
   false (zéro) est retourné. */

#ifdef DEBUG
   printf ("*** isword renvoie !result = %d\n",!result);
   fflush (stdout);
#endif
return (!result);
```


Les commentaires

- Ne soyez pas en contradiction avec le code. Après la correction de bogues, revoyez vos commentaires.
- Clarifiez les choses, ne les rendez pas confuses.

```
int strcmp (char *s1, char *s2)  {
    /*la routine de comparaison des chaînes de caractères*/
    /*retourne  -1 si s1 est au-dessus de s2 dans une liste */
    /*à ordre croissant, 0 en cas d'égalité et 1 si s1 est en */
    /*dessous de s2 */
```

- Un bon code nécessite moins de commentaires.

Règles d'écriture d'un programme en C

Afin d'écrire des programmes C lisibles, il est important de respecter un certain nombre de règles de présentation

- ne jamais placer plusieurs instructions sur une même ligne;
- utiliser des identificateurs significatifs;
- grâce à l'indentation des lignes, on fera ressortir la structure;
- syntaxique du programme. Les valeurs de décalage les plus utilisées;
- sont de 2, 4 ou 8 espaces;

Règles d'écriture d'un programme en C (suite)

- on laissera une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions;
- une accolade fermante est seule sur une ligne (à l'exception de l'accolade fermante du bloc de la structure **do** ... **while**) et fait référence, par sa position horizontale, au début du bloc qu'elle ferme;
- aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces;
- il est nécessaire de commenter les listings; évitez les commentaires triviaux.

Avantages d’un bon style

- Un code bien écrit est :
  - plus facile à lire et à comprendre;
  - comporte assurément moins d’erreurs;
  - probablement moins volumineux;
  - assurément plus portable et plus facile à déboguer.