

Plus de joie avec G77

Thierry Boudet, aka "Tonton Th"

4 juillet 2007

Chapitre 1

Introduction

Coder en FORTRAN de la variété 77, c'est vraiment une nouvelle expérience pour vous autres, les petits jeunes élevés au jus de web sémantique 2.0 et aux langages à designs patterns. Heureusement que framework, ça en fait au moins un qui bosse. On trouve en même temps dans ce langage, d'intenses facilités pour la spleytisation du code, opposées à d'immenses (en théorie) facilités de calcul.

Je vais essayer, à travers ce document, de vous faire découvrir les moyens de vous amuser avec ce langage de dinosaure, et le compilateur G77 qui va avec. Ma démarche n'est pas vraiment rigoureuse et fera probablement hurler les puristes de `comp.lang.fortran`, que je remercie au passage pour la pertinence de leurs trolls parfois diablement techniques.

```
program hello
write(6, 100)
100 format('Hello, world')
end
```

Certains chapitres font référence à des bibliothèques de fonctions extérieures, dont certaines ont été écrites¹ par l'auteur de ce document. Il est clair que dans certains cas, d'autres solutions de meilleure qualité existent, et seront peut-être abordées un autre jour. Regardez en page 50 le futur de ce document.

1.1 Conventions typographiques.

Dans les exemples de code, les mots-clefs du FORTRAN seront en MAJUSCULES, et les nom de variables en minuscules. En fait, après une relecture sur une version imprimée, je me suis rendu compte que je ne fait pas toujours ça. Mais comme il faut (conventuellement) établir ce genre de chose, je le fait.

¹bricolées serait un terme plus exact.

1.2 Avertissement.

Ce document ne doit **en aucun cas** être pris au pied de la lettre. Il est avec certitude rempli d'erreurs et de mauvais conseils. Certaines des choses que j'expliquent proviennent de déductions hatives. Les exemples de code n'ont pas été soumis à une procédure de test extensif. Le compilateur, G77, utilisé pour mettre au point et tester les divers exemples et logiciels décrits dans ce document est, semble-t-il, un produit en fin de vie. Il sera à terme remplacé par le GFORTTRAN (voir p. 50). D'ailleurs, ça nous promet de franches rigolades en compagnie de notre copain le debugger.

1.3 L'auteur.

Thierry Boudet, membre fondateur et président honoraire du Club des Vieux Cons, utilise des ordinateurs depuis l'époque des cartes perforées. Son tempérament intégriste lui interdit de toucher à tout ce qui sort de Redmont. Il habite à Toulouse et cherche actuellement un emploi : <http://tboudet.free.fr/cv.html>.

Chapitre 2

Notions de bases

2.1 Structure générale

Traditionnellement, le code-source d'un programme FORTRAN s'écrit sur des cartes perforées à 80 colonnes, de préférence avec une perforatrice 029. Les cinq premières de ces colonnes sont réservées aux labels numériques, la sixième au caractère de continuation et vous pouvez aller de la colonne 7 jusqu'à la 72 pour coder vos futures divagations.

Une instruction longue peut être écrite avec un nombre arbitraire de lignes de continuation : celles-ci doivent contenir uniquement des espaces dans les colonnes 1 à 5, et un caractère non-blanc en colonne 6. Généralement, on utilise un '+'.
C +

Les lignes de commentaires sont signalées par un 'C' ou une '**' en première colonne. On peut aussi introduire un commentaire par un '!', mais je ne m'étendrais pas sur cette chose.

Un programme FORTRAN au delà du deuxlignes est divisé en différentes unités, dites souvent "unités de compilation" : le programme principal, et éventuellement un ou plusieurs sous programmes. Chaque unité doit être terminée par un END. Le programme principal est obligatoire et doit être unique. Il existe deux types de sous programmes : SUBROUTINE et FUNCTION, que nous verrons un peu plus loin.

```
C      Debut de l'execution ici.  
      PROGRAM exemple  
      CALL plop()  
      END  
  
C      Sous programme qui ne fait rien.  
      SUBROUTINE plop  
      END
```

2.2 Types de données

Le FORTRAN 77 supporte six types de données, G77 en rajoutant un septième, dont voici la liste :

- Entiers (type générique INTEGER)
- Réels (type générique REAL)
- Réels en double précision
- Nombres complexes (type générique COMPLEX)
- Booléens (type générique LOGICAL)
- Caractères (type générique CHARACTER)
- Doubles Complexes (une Gnuserie) ¹

Théoriquement, un nom de variable ne peut avoir plus de 6 caractères, pris dans l'ensemble des lettres et des chiffres. mais cette limitation n'a plus lieu d'être quand on utilise G77, qui permet aussi d'utiliser le `_` (underscore). Il doit obligatoirement commencer par une lettre. Par défaut, pour le typage des variables numériques qui ne sont pas déclarées, la règle est simple, mais fourbe : les variables dont le nom commence par une des lettres `I JKLMN` seront de type entier, et toutes les autres seront de type réel simple précision. En général, il est quand même plus prudent de déclarer ses variables, et de se prémunir contre un éventuel oubli : voir en page 12 pour savoir comment faire.

En ce qui concerne les nombres flottants, vous trouverez plus de détails au chapitre des mathématiques (page 18), en particulier sur le *cast*.

2.3 Chaines et tableaux

En FORTRAN, par défaut, les indices de tableaux commencent à 1. On peut bien entendu changer ça à la déclaration du tableau, ce qui engendre une foultitude de *of by one errors* rendant l'exécution du programme quasiment stochastique.

Si mes souvenirs sont exacts, on peut utiliser des tableaux ayant jusqu'à 7 dimensions, ce qui est largement suffisant pour aller bomber dans l'hyper-espace.

2.4 Structures de contrôle

Pour s'assurer que toutes ces petites instructions se déroulent dans le bon ordre, il faut le demander gentiment. Nous allons voir les comparaisons, les tests, les sauts et divers genres de boucles.

2.4.1 Les comparaisons numériques.

Il existe 6 opérateurs de comparaison pour les valeurs numériques. Ils sont de la forme `.LL.` où LL désigne la comparaison effectuée. Ils retournent une valeur de type logique.

<code>.eq.</code>	<i>Equal</i>	<code>==</code>
<code>.ne.</code>	<i>Non Equal</i>	<code>!=</code>
<code>.lt.</code>	<i>Letter</i>	<code><</code>
<code>.gt.</code>	<i>Greater</i>	
<code>.le.</code>	<i>Letter or Equal</i>	
<code>.ge.</code>	<i>Greater or Equal</i>	

¹Le terme « gnuserie » n'est pas péjoratif. Il désigne juste des choses que vous risquez de ne pas retrouver facilement si vous utilisez une plate-forme propriétaire, genre Origin.

Il est ensuite possible de combiner le résultat de plusieurs comparaisons avec des opérateurs logiques tels que `.and.`, `.not.` ou `.or.`.

2.4.2 Les comparaisons textuelles.

Les opérateur vus un peu plus haut sont aussi utilisables pour comparer des chaînes de caractères. La comparaison s'effectue de la gauche vers la droite, caractère par caractère. Dès qu'ils sont différents, celui dont le code EBCDIC/ASCII est le plus grand est supérieur.

2.5 IF

Maintenant que nous avons une valeur qui dit 'oui/non'², nous pouvons prendre une décision. On trouve deux variétés de IF en FORTRAN : le simple, et le multiligne.

En mode simple : `IF (condition) INSTRUCTION`. L'instruction pouvant être un `GOTO 25673` pour rester dans la cuisine italienne.

Le test multiligne est plus intéressant, puisqu'il permet d'une part d'exécuter conditionnellement plusieurs instructions, et d'autre part de coder les deux alternatives en utilisant l'instruction `ELSE`. Le tout étant terminé par un `END IF` que vous pouvez aussi écrire `ENDIF`.

```
IF (foo .EQ. 42) THEN
    points = points + 1
ELSE
    points = points - 1
ENDIF
```

Remarquez bien la nuance : c'est le mot-clef `ELSE` qui fait toute la différence. Notez bien aussi que ce n'est pas tout à fait pareil qu'en GW-BASIC.

2.6 Les boucles

Entre deux générations du langage FORTRAN (66 et 77), il y a eu une révolution terrible au sujet des boucles `DO`. C'est pour ça que c'est délicat d'en parler, parce qu'on risque de choquer un ultradino et de blesser certaines susceptibilités.

The meaning of a DO loop in Fortran is precisely specified in the Fortran standard... and is quite different from what many programmers might expect. *lol*.

²comme une poupée ?

2.6.1 La boucle DO

Ce type de boucle nous permet de répéter un certain nombre de fois une suite d'opérations. Une variable entière va servir de compteur. Il y a une valeur de début, une valeur de fin et un pas d'incrément facultatif.

```

DO 5 i=1, 10
    foo = foo + i
5 CONTINUE

```

Quand on est à l'intérieur de la boucle, il est interdit de tenter de changer la valeur de la variable qui contrôle la boucle.

2.6.2 La boucle WHILE

C'est pas un peu moderne, ce truc, mais ça ne va pas nous empêcher d'y porter un regard objectif, quoique légèrement narquois³..

2.7 Sous programmes

Ils sont de deux sortes : les fonctions, qui retournent une valeur avec un type, et les sous-routines qui ne retournent rien, sauf si elles le veulent bien.

Un sous-programme peut prendre un nombre quelconque d'arguments. Il faut bien entendu faire très attention à la concordance des types entre l'appel et la déclaration. FTNCHEK (décrit en page 14) sera d'un grand secours pour éviter les petites inconveniences génératrices d'angoisse.

2.7.1 Fonction

Une fonction va retourner une valeur au (sous-)programme appelant. Il faut donc faire attention à la déclarer afin que le type (entier, réel...) soit connu de l'appelant et de la fonction. A défaut, c'est la règle du IJKLMN qui sera appliquée, avec mauvaises surprises à la clef. Exemple :

```

INTEGER FUNCTION reponse (param)
INTEGER param
reponse = 42
END

```

Vous remarquerez que la valeur que l'on veut renvoyer est affectée à une pseudo-variable portant le même nom que la fonction appelée.

³Après tout, avec le IF et le GOTO, on peut refaire toutes les boucles du monde

2.7.2 Subroutine

On les appellent avec le mot-clef `CALL`. On ne peut donc pas affecter leur hypothétique valeur de retour à une *lvalue*. Par contre, les arguments étant (en général) passés par référence, il est donc probablement possible de les modifier.

```
SUBROUTINE plop
...
END
```

Il est possible de quitter, à tout moment, une procédure (fonction ou subroutine) par le mot-clef `RETURN`.

2.8 Compiler votre oeuvre

Voyons la création d'un exécutable à partir de deux fichiers sources, l'un contenant (par exemple, hein...) le *main*, et l'autre quelques sous-programmes. Il faut tout d'abord traduire les sources vers un format intermédiaire, appelé objet, puis lier ces objets en un fichier directement exécutable.

```
tth@pouffre $ g77 -c main.f
tth@pouffre $ g77 -c ssprg.f
tth@pouffre $ g77 main.o ssprg.o -o machin
```

Que l'on pourra bien entendu résumer en :

```
tth@pouffre $ g77 main.f ssprg.f -o machin
```

Ce qui, en fait, est simple, mais pas très malin non plus. Le `FORTRAN` est un langage délicieux, car les bugs sont quasiment implicites. Vous trouverez plus de détails à la page 12 avec quelques explications sur les diverses options disponibles permettant de compiler de manière plus saine.

Et enfin, pour contempler l'absurdité de votre création :

```
tth@pouffre $ ./machin
```

Chapitre 3

Perversions de bases

3.1 Les GOTO

Le truc vraiment délassant, avec le FORTRAN, c'est que l'on peut sauter, tel la grenouille sur le plat de nouilles entremêlées, d'un endroit à un autre, butinant ainsi les instructions à exécuter. Certains prétendent que le GOTO est le mal absolu, mais c'est juste parce que ils sont jaloux. En plus, cerise sur le gâteau, il existe plusieurs formes de GOTO.

Un GOTO doit être « local » à une unité de programme. C'est-à-dire qu'il ne peut pas sortir d'une unité pour rentrer subrepticement dans une autre unité. Les *power-users* pourront en partie contourner cette limitation absurde en utilisant les entrées.

3.1.1 Le GOTO simple

Chaque ligne d'une unité de programmation peut être numérotée par une valeur entre 1 et 99999 (cinq colonnes sur la carte, en fait). Ce numéro de ligne sera ensuite utilisé comme cible lors d'un saut.

```
[bla-bla-bla]
27005 CONTINUE
[...500 lignes de code...]
GOTO 27005
```

C'est la première forme, relativement simple et facilement utilisable. Après tout, c'est comme ça que ça marchait en GW-BASIC, et même plutôt bien, donc nous y sommes donc habitués depuis 25 ans...

3.1.2 Le GOTO ASSIGNED

Nous allons maintenant passer par une variable intermédiaire, ce qui nous permettra de modifier dynamiquement la destination du saut :)

```
PROGRAM toto
INTEGER cible
ASSIGN 100 TO cible
GOTO cible
STOP 'FINI'
100 STOP 'LIGNE CENT'
END
```

Vous devez saisir, après quelques minutes de réflexion, toutes les possibilités d'obfuscation que procure le goto assigné. Un vrai tir sur cibles mouvantes.

Mais attendez, il y a mieux...

3.1.3 Le GOTO calculé

Ou comment reproduire, à relativement peu de frais, le switch/case du C, avec, bien entendu, quelques limitations prises de tête. Le principe est simple, la mise en oeuvre bogogène, c'est l'esprit *morefun*.

Voici un exemple, que nous allons détailler un jour ou l'autre.

```
PROGRAM cgoto
INTEGER foo
foo = 3
GOTO (100, 200, 300) foo
PRINT *, 'huhu ?'
100 PRINT *, 'un'
200 PRINT *, 'deux'
300 PRINT *, 'troix'
END
```

3.2 Lire et écrire des fichiers

Plus sérieusement, vous trouverez en page 22 des explications plus détaillées sur la gestion des fichiers en G77.

3.2.1 Les FORMAT

Découvrons ensemble la subtilité des champs HOLLERITH. Nan, j'rigole... Cette agréable perversion est obsolète, et elle l'a bien mérité. Après tout, l'époque des cartes perforées est révolue.

3.2.2 Les modes d'ouverture

C'est le paramètre `STATUS=` de la fonction `OPEN`. Il détermine le comportement au moment de l'ouverture du fichier, selon que celui-ci existe déjà ou non. Il est facultatif.

SCRATCH Utilisé pour les fichiers temporaires. Ils seront détruits dès que l'on n'en aura plus besoin.

NEW Génère une erreur si le fichier existe déjà.

OLD Génère une erreur si le fichier n'existe pas.

REPLACE Tente, avec succès, d'écraser vos anciennes données.

Suite en page 22. . .

3.3 Les COMEFROM

Nan, là, je plaisante. Enfin, à moitié. Il paraît que certains l'ont fait. Ceux qui veulent en savoir plus peuvent toujours aller fouiller les bas-fonds du réseau¹. Ils trouveront des horreurs réjouissantes.

Le principe est simple dans l'idée, mais assez délicat à conceptualiser. En gros c'est le symétrique du `GOTO`. Voilà. Vous en savez plus que ce que vous ne l'auriez voulu.

3.4 Et hop, le COMMON

Un système bien pratique qui permet de gérer finement les variables globales² et le partage de données entre modules. En prime, la subtilité des bugs semble infinie, puisqu'il y a de potentiels problèmes d'alignement des données en mémoire.

3.4.1 Common anonyme

Il n'a pas de nom, et à ce titre, il est anonyme. En plus on risque de le perdre... Donc on ne va pas s'étendre dessus, puisque c'est quasiment inutilisable dès que le deuxlignes a un peu grandi.

3.4.2 Common nommé

On lui donnera donc un nom, ce qui évite de se mélanger les pinceaux. Ensuite, une liste des variables à partager, lesquelles ayant été préalablement déclarées.

```
COMMON /nom/ var1, var2, var3
```

Cette ligne doit être placée dans chacun des blocs (`main`, `fonc` ou `sub`) qui doivent se partager les variables en question. Nous découvrons donc ici un premier usage pour la directive `INCLUDE`.

Si vous voulez en savoir plus, je vous suggère de faire un détour par le chapitre sur les frameworks (page 40), et de revenir ici après une pause Guinness.

¹mot-clef : `datamation`.

²Ouééé, `_o/` `super`, des globales.

3.5 EQUIVALENCE

all your coredumps are belong to us.

L'équivalence permet d'utiliser la même zone mémoire sous deux noms différents. C'est une sorte d'*aliasing*.

3.6 SAVE

Une variable qui est déclarée dans une fonction ou une subroutine n'a pas vraiment de mémoire : elle ne se souvient plus de son contenu d'un appel à l'autre, et n'a même pas de valeur définie quand elle arrive dans le champ de visibilité. C'est parfois gênant. Voyons ça sur un exemple :

```
PROGRAM demosave
CALL plop
CALL plop
END
```

Ce programme principal va appeler deux fois d'affilée une procédure, chargé d'une modeste addition et d'un non moins modeste affichage de nombre entier, en quelque sorte un compteur du nombre de fois où cette procédure a été appelée.

```
SUBROUTINE plop
INTEGER compteur
compteur = compteur + 1
WRITE(6, 1) compteur
1  FORMAT(i5)
END
```

Hélas, le résultat est décevant. *Plop* ne fait rien qu'à radoter. Il nous dit que `compteur`, même incrémenté, ne fait rien qu'à garder la même valeur, d'ailleurs incohérente, puisque la variable n'a jamais été initialisée. Nous allons donc dire au compilateur que notre compteur doit garder sa valeur entre deux appels à la subroutine. Il suffit de rajouter cette ligne juste après la déclaration de la variable :

```
SAVE compteur
```

Et ça marche...

Chapitre 4

Compilation

Les compilateurs de la lignée gcc, et G77 en particulier, comprennent un gazillion d'options de ligne de commande. Nous allons voir ici celles qui sont indispensables pour une première approche, en permettant de détecter à l'avance les problèmes potentiels par l'émission de warnings parfois pertinents.

4.1 L'indispensable

- Wall** Garde-fou de base. Signale les variables qui sont inutilisées ou qui sont utilisées avant d'avoir été initialisées. Ne pas utiliser cette option est réservé aux *37it35*.
- Wimplicit** Protection contre toutes les erreurs de typage qui surviennent quand on n'a pas au moins quarante-deux ans d'expérience dans le typage pervers du FORTRAN.
- pedantic** Cette option est absolument indispensable aux puristes de la programmation *alternative* qui tiennent quand même à garder leur tête sur leurs épaules en restant dans la ligne dur de la norme ANSI

4.2 Les *includes*.

On en parle dans le chapitre à propos des frameworks. Il faudrait détailler les chemins de recherche implicites et les précautions à prendre.

4.3 Compilation séparée.

Il est évident qu'un code source de 42000 lignes est quasiment ingérable. On peut donc séparer les diverses unités de compilation en le mettant dans plusieurs fichiers.

C'est là qu'on découvre finalement l'utilité de `ftnchek` qui permet de faire certains contrôles de cohérence entre ces divers modules, par exemple sur la cohérence des paramètres lors des appels de sous-programmes.

4.4 Le superflu

Superflu ? Futile ? Léger ? Donc indispensable. Certaines options, d'après ce que j'ai pu lire dans la documentation¹ des compilateurs de la Gnu Collection, permettent d'introduire une légère dose d'indécision dans la chaîne de décision, ou au contraire de se rassurer sur la pertinence de ses choix.

4.4.1 Le profilage.

Pour savoir où ce foutu machin que l'on vient de goretcoder passe le plus de temps à glander, il faut faire appel au profiler. On utilisera pour ça l'option **-pg** à la compilation et à l'édition de lien. Ensuite, on lance le programme et on attend avec impatience qu'il se termine.

On peut alors examiner le résultat avec la commande `gprof -b nom_du_binaire` qui donnera plein d'informations sur le temps d'exécution des diverses fonctions et procédures du logiciel. A vous d'en tirer les conclusions, en n'oubliant pas qu'un programme qui va très vite pour fournir un résultat faux n'est pas très utile².

4.4.2 Le débogage

Il faut utiliser l'option **-g** à la compilation et à l'édition de liens afin que les informations nécessaires soient enregistrées dans le binaire final. Ensuite, la page man de `gdb` est votre amie. Mais il serait bon que je prépare un exemple, et même que je le colle aux environs de la page 16 pour que tout le monde le retrouve.

4.4.3 Les listings

Le dino aime le papier zoné, et les bandes Carrols³. Pour masquer l'affreuse lumière jaune qui vient du plafond bleu de la grande salle de l'autre côté de la porte, il aime imprimer le listing de ses plus belles routines, et coller le papier ainsi obtenu sur les fenêtres de son bureau, à côté d'une Britney en `ascii-art`.

¹Excellente, il faut presque le dire.

²Ceci dit, il va très très vite.

³Pas sûr de l'orthographe, là.

Chapitre 5

ftnchek

Les vrais habitués du C connaissent bien lint, et souhaitent retrouver un outil du même genre pour les accompagner dans leur plongée vers les abysses du FORTRAN 77.

5.1 Démonstration

```
tth@flo:/d2b/tth/MoreFunWithG77$ cat toto.f
PROGRAM toto
INTEGER foo, bar
DO 10 bar=1,foo
    WRITE(6, 2) foo, bar
10 CONTINUE
2  FORMAT(i5, ' : ', i5)
END
```

Ce code contient une erreur vraiment grossière, essayons de la retrouver. (l'option **-nonovice** demande un affichage des messages de diagnostic en version courte)

```
tth@flo:/d2b/tth/MoreFunWithG77$ ftnchek -nonovice toto

FTNCHEK Version 3.3 November 2004

File toto.f:

"toto.f", line 3: Warning in module TOTO: Variables used before set
"toto.f", line 3:      FOO used; never set

0 syntax errors detected in file toto.f
1 warning issued in file toto.f
```

5.2 Explications

Ftnchek est un gros morceau. Première étape : `$ ftnchek -help`, pour avoir une idée de la foultitude des options disponibles, qui sont quand même, pour la plupart, bien expliquées dans la man-page.

Deuxième étape (elle est pour moi, celle-ci) : Au fur et à mesure¹ que je fabrique les exemples pour ce HOWTO, je vais étudier ce que peut faire Ftnchek pour améliorer la précision de la craditude du code.

¹Bien belle chanson, il faut le dire.

Chapitre 6

Pif, paf, gdb.

Quand rien ne marche, quand tout va mal, quand vous soupçonnez qu'il se passe des choses cha-fouines dans votre dos, que le Général Failure discute avec monsieur Crash CoreDump pour savoir qui va segfault le premier, vos nerfs commencent à craquer, votre lucidité ne vous parait pas si digne de confiance que ça, le moment est venu : il faut faire appel à **gdb**, l'ami des petits et des gros bugs.

6.1 Démonstration rapide

Nous allons étudier un cas très classique : oublier la déclaration de type d'une fonction, et découvrir un résultat déraisonnable. Ne rigolez pas, ça vous arrivera plus qu'à votre tour.

```
program bug
integer  foo
foo = barfunc(42)
write (6, 100) foo
100  format (i8)
end

integer function barfunc(arg)
integer  arg, temp
temp = arg * 2
barfunc = temp
end
```

Ce programme va afficher sans complexe un résultat digne d'une bonne analyse. Puisque je suis un vieux dino, je connais le problème : dans la fonction, nous préparons une valeur de retour stockée dans un integer sur 32 bits.

```
morefun77 $ gdb bug
GNU gdb 6.4-debian
[blabla...]
(gdb) break barfunc_
```

```

Breakpoint 1 at 0x80486ea: file bug.f, line 12.
(gdb) run
Starting program: /d2b/tth/Documents/MoreFunWithG77/bug
Breakpoint 1, barfunc_ (arg=0x80488b8) at bug.f:12
12      temp = arg * 2
(gdb) print arg
$1 = (PTR TO -> ( integer )) 0x80488b8
(gdb) print *arg
$2 = 42
(gdb)

```

Nous constatons que le paramètre passé lors de l'appel est correct. Avançons maintenant d'une ligne dans notre code source, et vérifions la qualité de notre calcul :

```

(gdb) step
13      barfunc = temp
(gdb) print temp
$4 = 84
(gdb) x &temp
0xbffff9d0:    0x00000054
(gdb) step
14      end
(gdb) x &barfunc
0xbffff9d4:    0x00000054

```

Jusque là, tout va bien¹. Le calcul a donné un résultat globalement inoffensif, et la préparation du retour est assez encourageante, puisque nous retrouvons bien la double réponse universelle.

A suivre...

6.2 Un truc un peu plus compliqué.

Par exemple, la recherche d'un problème de passage de paramètres entre le G77 et le C, à propos d'une chaîne de caractère.

6.3 Quelques astuces.

A ce jour², la consultation de la manpage de gdb n'est pas encourageante, il y est dit ceci : Fortran support will be added when a GNU Fortran compiler is ready., alors que G77 me semble quand même assez utilisable.

¹Nous venons de passer le quinzième étage.

²Aout 2006

Chapitre 7

Les mathématiques

Après tout, FORTRAN étant l'abréviation de *FORMULA TRANSLATOR*, il est normal d'aborder le calcul mathématique à un moment ou à un autre. Et il aurait beaucoup à raconter sur ce sujet.

7.1 Simple et double précision.

Une variable flottante se déclare avec le mot-clef `REAL`. Pour avoir plus de digits, il faut utiliser la `DOUBLE PRECISION`.

Attention, il y a un piège. Si vous voulez une constante en double précision, il faut l'écrire sous la forme `3.14d0` et non pas `3.14`, qui sera interprétée comme une valeur en simple précision. Il faut oublier les règles implicites de promotion du C.

7.2 Conversions de types.

Les habitués du C connaissent bien le *cast*, ou transtypage, ou magouille, qu'il soit explicite ou implicite. En FORTRAN, c'est assez différent. Il est souvent préférable de bien expliciter les choses, en particulier dans les passages de paramètres. N'oubliez pas `ftnchec`, décrit en page 14, qui évite de perdre des heures sur des idioties finalement contre-gruikives.

7.3 Les nombres complexes.

Là, désolé, je ne suis pas trop expert, mais je me soigne. Ceci dit, ce truc semble très pratique, pour les fractales, ou la synthèse sonore...

Le FORTRAN à la sauce GNU propose deux extensions pour récupérer les parties réelles et imaginaires d'un nombre complexe : `REALPART (EXPR)` et `IMAGPART (EXPR)`.

7.4 Quelques *intrinsic*.

sin, cos, abs...

Chapitre 8

Quelques astuces

8.1 Les arguments de la ligne de commande.

Pour savoir combien de paramètres ont été donnés sur la ligne de commande, il faut utiliser la fonction `iargc()`. Ensuite, on peut récupérer ces arguments un par un avec `call getarg (n, cv)`, où `cv` est une variable chaîne de caractères, qu'il conviendra de dimensionner correctement.

Pas besoin d'en savoir beaucoup plus, voyons un exemple :

```
PROGRAM demoarg
  INTEGER foo, bar
  CHARACTER*42 chaine
  foo = IARGC()
  WRITE(6,1) foo
  DO 10 bar=1,foo
    CALL GETARG(bar, chaine)
    WRITE(6, 2) bar, chaine
10  CONTINUE
1   FORMAT('argc = ', i2)
2   FORMAT(i2, ' -> ', a20)
END
```

...qui nous donnera à l'exécution :

```
tth@flo:~/Documents/MoreFunWithG77$ ./demoarg toto 42 "coin pan" Pl0p
argc = 3
1 -> toto
2 -> 42
3 -> coin pan
4 -> Pl0p
tth@flo:/d2b/tth/Documents/MoreFunWithG77$
```

Maintenant, c'est à vous de faire une émulation bien générique de `getopt`, qui me rendra probablement bien service...

Attention, le nom du programme en cours d'exécution n'est pas disponible dans l'argument d'indice 0. Il est peut-être possible de le retrouver par une Gnuserie, à vous de chercher...

8.2 Les variables d'environnement

Là, je suis vraiment désolé, mais je n'ai pas la moindre idée. D'un autre côté, je pense qu'il doit exister une gnuserie adéquate.

8.3 L'équivalent du `printf`.

Il faut utiliser le `WRITE`, et un `FORMAT`, en remplaçant le numéro du fichier par une variable texte. C'est vraiment très facile. Il suffit d'essayer.

Attention quand même, si la taille de la chaîne de destination est supérieure à la taille du format, il y aura des espaces à la fin.

```
CHARACTER*15  chaine
WRITE(chaine, 100) 42
100  FORMAT(' reponse:', I2)
```

Si vous voulez imprimer un entier avec tous ses zéros de tête (par exemple pour générer des noms de fichiers séquentiels), vous utiliserez `I5.5` dans la chaîne de format, ce qui vous génèrera `00042`.

Je ne sais pas si on peut cadrer à gauche un champ d'édition.

8.4 Ecrire sur la sortie d'erreur.

Je ne sais pas comment faire. grml...

Il y a peut-être une magouille possible en bidouillant dans le fs virtuel `/proc`, mais ce ne sera vraiment pas portable.

8.5 Chaînes en paramètres

Comment passer une chaîne de caractères en paramètre à une fonction ou à une procédure ?

Chapitre 9

Les fichiers

Le sujet de la gestion des fichiers a déjà été abordé à divers endroits de ce document, et il est temps de rentrer dans les détails. Une grande partie du code montré dans ce chapitre sera testé sur un Cyrix 166 sous OpenBSD 3.8, avec un *gcc version 3.3.5 (propolice)*. Nous verrons probablement en fin de chapitre les éventuelles adaptations pour une Daubian avec un gcc 3.4.6 pre-release.

9.1 Paramètres des fonctions d'IO

Nous avons déjà vu précédemment quelques exemples sommaires. Il est temps de rentrer un peu plus dans les détails. Chacune des instructions ayant trait aux fichiers accepte un grand nombre de paramètres, appelés *specifiers*. Certains d'entre eux sont communs à plusieurs instructions, et d'autres sont spécifiques.

UNIT=*n* *n* est le numéro de fichier, qui peut être compris entre X et Y. Les numéros 5 et 6 sont en général affectés à l'entrée et à la sortie standard. Quand à la sortie d'erreur, rendez vous page 21, où vous verrez que ce n'est pas si simple...

IOSTAT=*ios* *ios* est une variable entière qui contiendra un code d'erreur. Si ce code est différent de zéro, c'est qu'il y a eu une erreur, et la signification de cette valeur est dépendante du système¹.

FILE=*'foobar.data'* C'est le nom du fichier concerné par l'opération. Les blancs en fin de chaîne sont éliminés.

9.1.1 OPEN

STATUS=*'foo'* Si *'foo'* est *'REPLACE'*, il y a écrasement d'un éventuel fichier existant.

ACCESS=*'foo'*

¹faire une liste des codes d'erreur pour g77 dans Linux/OpenBSD...

9.1.2 READ

9.1.3 WRITE

9.1.4 CLOSE

9.1.5 INQUIRE

Grâce à cette instructions, nous pouvons obtenir plein d'informations diverses sur un fichier. Il serait bien que j'examine ça de plus près.

9.1.6 REWIND

Chacun sait qu'au bon temps des vrais ordinateurs, les données importantes étaient stockées sur des bandes magnétiques. Si on voulait relire à partir du début ; il fallait rembobiner la bande. C'est clair, non ?

9.2 Fichiers 'texte'

Ce premier exemple, tout simple, permet d'écrire un fichier contenant une réponse universel.

```
PROGRAM essai
INTEGER*4 foo
foo = 42
OPEN(10, NAME='toto.dat')
WRITE(10, 1) foo
CLOSE(10)
1  FORMAT(i5)
END
```

9.3 Fichiers binaires

Pour continuer dans l'esprit du premier exemple de la section précédente, nous allons tenter d'écrire une réponse en binaire dans un fichier. Mais ça n'a rien d'évident, *Crash Coredump* me fait des siennes. C'est probablement que je n'ai pas vraiment lu la documentation, un peu incomplète sur ce domaine, il fallait le préciser.

Nous allons donc proceder² de façon empirique... Voici quelques extraits de code (pour ce qui manque, voir l'exemple complet un peu plus haut).

```
OPEN(10, NAME='toto.dat', ACCESS='DIRECT')
WRITE(10) foo
```

²Si la loi DADVSI m'attrape, vous me porterez des Guinness en tôle, hein ?

Nous utilisons un `ACCESS='DIRECT'` et remarquez bien l'absence de label de `FORMAT` dans le `WRITE`. Un dump du fichier obtenu nous donne ceci, que nous allons essayer d'interpréter : `04 00 00 00 2a 00 00 00 04 00 00 00`. A priori, nous sommes en présence de trois longs, dont le second contient le nombre sacré³.

Quand au premier et au troisième, nous pouvons supposer qu'ils contiennent la taille de l'enregistrement qui vient juste après, ce qui est vérifié par l'expérience.

Si nous essayons de remplacer `'DIRECT'` par `'KEYED'` ou `'SEQUENTIAL'`, paf, ça coredumpe avec le message « *sue : unformatted io not allowed, Abort trap (core dumped)* », pas franchement encourageant...

9.4 "Images Cartes"

Retour vers le passé, non ? Pour moi, une « image carte » est une suite d'enregistrements contenant chacun 80 caractères.

9.5 Séquentiel Indexé

Voilà enfin un peu de prise de tête... Pour cette partie du document, je vais me baser sur de vieux livres du siècle dernier, consacrés à la « gestion de fichier en basic ». Umpff. Ces deux ouvrages ont été écrits par Jacques Boisgontier en 1984, et publiés par les éditions du P.S.I.

³Ah, c'est une sacrée réponse, ça.

Chapitre 10

Advanced perversity

10.1 ENTRY

Mot-clef fabuleux ouvrant la porte aux GOTOS non-locaux. En gros, on peut entrer dans un sous-programme à n'importe quel endroit. Je n'ai pas le souvenir d'avoir utilisé ça, mais je pense que ça doit être délicieux. Nous allons donc essayer de vous montrer un petit exemple...

```
PROGRAM demoentry
CALL bonne
CALL mauvaise
END
c -----
SUBROUTINE reponse
INTEGER valeur
valeur = 0
GOTO 900
ENTRY bonne
valeur = 42
GOTO 900
ENTRY mauvaise
valeur = 27
900 WRITE(6,378) valeur
378 FORMAT('la reponse est ', i5)
END
```

Maintenant que vous savez comment ça marche, et que vous avez constaté que l'utilité est toute relative, lâchez la bride de votre imagination...

10.2 STOP

Faites vos jeux, plus rien ne va, vous êtes le maillon faible. Ce mot-clef permet d'arrêter un programme, avec affichage d'un petit texte sur la console, mais ne permet pas de retourner un code d'erreur

au shell. Dommage. Il faut utiliser `CALL exit(42)` à la place.

10.3 CYCLE

Alors, là, celui-ci je ne le connaissais pas, et j'ai découvert son existence en lisant la documentation de `Ftnchek`. Il faut absolument que je découvre à quoi il sert, bien que je pense que ce soit en rapport avec un contrôle avancé des diverses formes de boucle disponibles. Peut-être même ça permet de débarquer d'une façon impromptue sur un `CONTINUE`.

10.4 ASM

Hélas, il semble impossible d'inclure du code assembleur dans le `G77`. Avec ça, il aurait probablement été possible de bien naxoriser le bouzin.

10.5 COMEFROM

Finalement, à la réflexion, ce n'est pas si mal, ce truc. Dommage que ça n'existe pas vraiment. Ou alors, je n'ai pas réussi à le retrouver dans le grand Bitnet.

10.6 Et maintenant ?

Bah oui, maintenant, on fait quoi ? Comme il fait très chaud¹, nous n'allons pas faire grand chose. . .

¹23 Juillet 2006, à Toulouse, 33°

Chapitre 11

Interface avec le C

11.1 Avertissement préliminaire.

Attention, tout ce que vous allez voir dans ce chapitre n'a été testé que sur les machines x86, sous Linux et avec le compilateur **gcc/g77**. C'est des choses que j'ai découvertes de façon plus ou moins empirique, donc YMMV.

D'autre part, il est important de faire l'édition de lien avec la commande **G77** qui sait comment faire la cuisine interne pour que tout se passe bien.

11.2 La décoration des symboles.

Le FORTRAN est un langage *case insensitive*, contrairement à d'autres, comme le C. Les trolls sur cette différence sont morts depuis longtemps. Par défaut, les symboles du G77 seront convertis en minuscules, puis le compilateur leur rajoutera `_` (un¹ caractère "souligné") à la fin du nom. Ainsi, une fonction `TOTO` sera traduite en `toto_`.

11.3 Tout est référence.

Dans tous les cas, les paramètres visibles d'un appel vers une routine externe sont passées par référence, c'est à dire que le sous-programme recevra l'adresse de la variable en mémoire, et sera donc en mesure de modifier le contenu d'une variable² dans l'appelant. Ceci est considéré par certains comme une bonne chose. On peut donc, par exemple, tenter d'avoir une valeur de Π légèrement supérieur à 3 d'une façon variable en fonction du sens du vent.

Voyons un exemple. Le code FORTRAN appelant est :

¹En fait, parfois il y en a deux. J'ai pas encore tout compris.

²Et dans certains cas, une constante. More fun again.

```

...
integer    foo, bar
foo = 42
bar = VERIFICATION(foo)
...

```

Et le sous-programme en C sera donc :

```

int verification_(int *pfoo)
{
  if (*pfoo == 42)    return 1;
  return 0;
}

```

11.4 Les chaînes de caractères.

Les chaînes de caractères sont traitées en FORTRAN d'une façon assez différente de celle du C. Elle ne sont pas terminées par un caractère spécial, du genre 0x00. En contrepartie, les fonctions C appelées reçoivent un (ou plusieurs) paramètres *cachés* contenant la taille de la ou les chaînes concernées.

Bien sur (ou hélas, c'est selon...) on ne peut, depuis le sous-programme C, modifier cette longueur. Ce serait trop simple, et générateur de trolls.

11.5 Les tableaux.

En dehors des tableaux à une dimension, c'est *aspirine* assurée, FORTRAN et C ne rangeant pas les données de la même façon et ne commencent pas, en général, au même indice : 0 pour le C et 1 pour le FORTRAN.

Après cet avertissement, voyons rapidement un exemple :

```

INTEGER tableau(100)
tableau(10) = 42
CALL C_FUNC(tableau)

```

Ce fragment de code déclare un tableau d'entiers (qui seront des nombres signés sur 32 bits) et appelle une fonction C, que voici :

```

void c_func_(int *ptable)
{
  printf("element 10 = %d\n", ptable[9]);
}

```

... et qui affichera joyeusement une réponse universelle.

11.6 Valeurs de retour.

On retrouve un mécanisme analogue pour retourner certains types de données depuis une FONCTION.

Gni ???

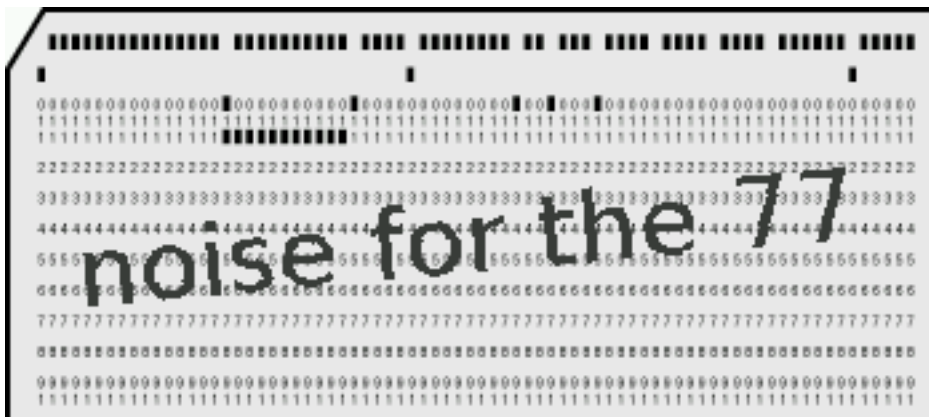
11.7 Mélanger les I/O.

C'est assez mal vu, probablement. Pensez toujours à flusher votre sortie standard quand vous avez fini de vous en servir.

Chapitre 12

Les fichiers de sons.

Quand on a goûté à la grande puissance du FORTRAN 77, on a parfois envie d'enregistrer ses petits murmures dans un fichier. Et ensuite, on veut relire ces octets si précieux... Vu l'immense diversité des formats de fichier sonore, nous n'allons pas nous prendre la tête, nous allons utiliser la `libsndfile` qui fait ça très bien à notre place.



12.1 `libsound77`.

J'ai écrit, en C, une interface pour cette bibliothèque. Vous la trouverez dans <http://tontonh.free.fr/libsound77.html>. Vous pourrez lire et écrire des fichiers sons au format `.WAV`, en 16 bits, à diverses fréquences d'échantillonnage et sur un ou deux canaux.

L'installation est très simple : décompression du tarball, `make`, su `-c 'make install'`. Vous trouverez deux ou trois exemples dans le répertoire `LibSound77/`, dont un petit *player* de `.WAV` qui reprend certaines des notions que nous avons vues dans les chapitres précédents.

12.2 Ouvrir ou créer un fichier.

Avant de pouvoir lire ou écrire dans un fichier son, il faut l'ouvrir avec `sndfopen` qui prend quatre paramètres : le nom du fichier, le mode (lecture ou écriture), la fréquence d'échantillonnage en Hertz et le nombre de canaux (en général, 1 ou 2).

```
INTEGER fichier
fichier = sndfopen('sinus.wav', 'O', 24000, 1)
IF (fichier .LT. 0) THEN
    STOP 'erreur fichier'
END IF
```

- Le mode est **I** pour la lecture et **O** pour l'écriture. Il est actuellement impossible d'accéder à un fichier en mode "mise-à-jour".
- Le *samplerate* est en nombre d'échantillon par seconde. Les valeurs que l'on peut trouver sont, par exemple, 8000, 44100 ou 96000.
- Le nombre de canaux doit être un (mono) ou deux (stéréo).

La valeur retournée par cette fonction est un entier. Si il est positif ou nul, il permettra par la suite de faire référence à ce fichier. Si il est négatif, c'est qu'il y a eu une erreur. *Insérer ici une table avec les divers codes d'erreur.*

Bien entendu, dans le cas d'une lecture, les deux derniers arguments de la fonction doivent être des variables entières dans lesquelles on récupérera les valeurs correspondant au contenu du fichier.

Attention, il est possible que le prototype de cette procédure change dans un futur proche pour pouvoir sélectionner le type de fichier que l'on veut obtenir en sortie.

12.3 Lire et écrire.

Les valeurs sont traitées sur 16 bits en signé, même si le contenu du fichier est différent. Cette limitation sera peut-être levée dans une hypothétique future version de `libsound77` et n'est **pas** une limitation de la `libsndfile` sous-jacente. Cela implique aussi que vos buffers de travail doivent être déclaré en `INTEGER*2`.

L'écriture se fait avec `CALL sndfput(fn, sample)` ou `call sndfput2(fn, left, right)`. Il n'y a pas vraiment de contrôle d'erreur, il faut donc faire attention à ne pas remplir tout le disque.

La lecture se fait avec `err = sndfget(fn, sample)` ou `err = sndfget2(fn, left, right)`. En retour, si tout s'est bien passé, on obtient le nombre d'échantillons lus : 1 pour la mono et 2 pour la stéréo. Si la valeur retournée est différente, c'est probablement qu'on est arrivé à la fin de la chanson, ou qu'une erreur inconnue est arrivée.

Vous avez remarqué avec déplaisir qu'on ne peut lire qu'un échantillon à la fois, et vous regrettez de ne pas pouvoir travailler avec des zones tampons afin d'optimiser les performances¹. Soyez rassurés, c'est prévu et ça sortira quand ce sera prêt. Après usage, il faut fermer le fichier afin de libérer les ressources utilisées. C'est la procédure `sndfclose(fn)` qui s'en charge. Rappelez vous : vous ne pouvez pas avoir plus de 42 fichiers ouverts simultanément. Cette limitation doit pouvoir être modifiée en recompilant la `libsound77`.

¹... qui sont, il faut bien l'avouer, déplorables.

Chapitre 13

Attaquer les haut-parleurs

Quand on a goûté à la grande puissance du FORTRAN 77, on a parfois envie de crier, très fort, en direction de l'oreille du monde. Ici aussi, nous allons utiliser une bibliothèque extérieure de haut niveau et une paire de *Fender* de 100W.

13.1 libsound77

Nous retrouvons ici quelque chose de connu (voir page 30) qui recouvre la `libao`, une bibliothèque générale de sortie sonore, développée par la fondation Xiph.org et portée sur de multiples plateformes.

13.2 beep77

Voici un extrait de code d'un exemple que vous trouverez dans le *tarball* de la `libsound77`. Il manque les déclarations des variables. C'est pô grave, je vais bientôt changer l'exemple.

Sachez quand même que les échantillons sonores doivent être impérativement stockés dans des `integer*2`, puisque ce sont des entiers signés sur 16 bits.

```
CALL ao77ini
CALL ao77set(rate, 1)
nbre = rate * 2
sweep = 3.55555
DO 100, foo=0,nbre
    di = DBLE(foo) / DBLE(nbre)
    phi = di * 3.141592654 * 2
    ve = (1.0 - COS(phi)) / 2.0
    dfoo = SIN(DBLE(foo)/sweep)
    sample = INT(dfoo*32000.0*ve)
    CALL ao77out(sample)
    sweep = sweep - 0.00001
100 CONTINUE
CALL ao77end
```

Voilà, c'est (presque) tout simple. Pour une sortie en stéréo, il faut appeler `ao77set` avec 2 en second paramètre, et utiliser `ao77out2(left,right)`. Ne pas appeler `ao77end` avant la fin du programme est l'équivalent logiciel du croisement des effluves. Vous n'avez pas vraiment besoin d'explications complémentaires, je suppose...

Chapitre 14

Traiter les images

Euh, là, j'ai un peu honte, là, parce que le code que je vais vous suggérer d'utiliser est vraiment crade. Et la documentation (<http://tboudet.free.fr/libimage/>) est lamentable¹. Ceci dit, c'est assez utilisable. La preuve, j'arrive parfois à m'en servir.

Cette libimage a quand même quelques défauts, dont le premier est le nombre très limité de formats de fichier connus : essentiellement le format TARGA en RGB et RGBA non compressé. Il y a aussi l'écriture au format Portable Net Map, en diverses variations.

14.1 Une première image.

Nous allons créer une petite image rgb de dimensions normalisées, avec un fond uni et un petit point noir au milieu. Voici le code :

```
PROGRAM point
INTEGER image
INTEGER img_create
image = img_create(42, 42, 3)
CALL img_clear(image, 10, 100, 255)
CALL img_plot(image, 21, 21, 0, 0, 0)
CALL img_tga_sa(image, 'dot.tga', 0)
CALL img_free(image)
END
```

Le troisième paramètre de la fonction `img_create` doit obligatoirement être 3. Les trois paramètres de `img_clear` sont respectivement les valeurs des composantes rouge, verte et bleue de la couleur choisie, avec des valeurs comprises entre 0 et 255. Les coordonnées dans l'image vont de 0 à (largeur/hauteur) - 1. Le troisième paramètre de la procédure `img_tga_sa` doit toujours être 0.

¹Bonne nouvelle, le développement a repris en Mars 2007, et la doc est en cours de ré-écriture.

14.2 Utiliser les fichiers images.

Nous avons vu dans l'exemple minimum comment enregistrer une image au format TARGA non compressé. Maintenant, nous désirons charger un fichier contenant une image².

Actuellement, il n'est pas possible de charger un fichier dans une image déjà allouée. La fonction `tga_aload(nom, w, h, foo)` réalise les deux opérations, et retourne, si tout s'est bien passé, le numéro de l'image. Une valeur négative indique à coup sûr une erreur. Les variables `w` et `h` sont renseignées avec les dimensions en nombre de pixel de l'image. Quand à 'foo', je le dis sans arrière-pensée politique, mais il ne sert pas à grand-chose, sauf à `coredump` si ce n'est pas une variable de type integer.

Quand on a fini de triturer cette image, il nous arrive parfois d'avoir une envie soudaine de l'enregistrer pour la postérité. Il existe actuellement trois fonctions pour ça.

Voyons la première : `foo = img_tga_sa(img, fname, 0)` enregistre votre image au format TGA non³ compressé. On retrouve la même chose pour les Portable Pix Map : `img_ppm_sa`. Quand à la troisième possibilité, je vous laisse le soin de lire les sources.

14.3 Tripoter les pixels.

14.3.1 Lecture.

On peut lire chacune des composantes d'un pixel par les fonctions `img_Xpix` avec `X` étant `r`, `g`, `b` ou `a` pour le rouge, le vert, le bleu ou le canal alpha. Si tout se passe bien, la valeur retournée sera comprise entre 0 et 255. Dans le cas contraire, paf, *coredump*.

On peut également lire les trois canaux `rgb` d'un coup en appelant la subroutine `img_rgb` (`img, x, y, r, g, b`). Actuellement, il n'est pas possible de lire en une fois le `rgba`.

Si on essaye de lire en dehors de l'image, le programme va se terminer avec un message précisant les coordonnées `x` et `y` fautives. Rappelons que ces coordonnées commencent à zero⁴ et que le dernier pixel est en (`hauteur-1`, `largeur-1`).

14.3.2 Ecriture.

De la même façon qu'en lecture, on peut écrire sur chacune des composantes, avec les subroutines de la forme `img_plt_X`.

On peut aussi écrire tous les canaux *couleur* de l'image d'un coup, avec `img_plot` (`img, x, y, r, g, b`). Pour gérer la transparence, on utilisera `img_plota` qui rajoutera en dernier paramètre la valeur du canal alpha, lui aussi compris entre 0 et 255.

Contrairement à la lecture, cette fonction est protégée contre les coordonnées en dehors de l'image.

²Qui devra être au format targa non compressé, bien entendu.

³En fait, un de ces jours, je vais virer ce 'non'

⁴Contrairement aux tableaux natifs du Fortran.

14.3.3 Opérations.

Rien à dire pour le moment. Dommage...

14.4 Fonctions utilitaires.

`i2 = img_clone(i1)` alloue une image `i2` ayant les mêmes dimensions que `i1`. Les valeurs des pixels sont également recopiés.

`call img_clear(im, r, g, b)` efface le contenu d'une image. On ne peut pas directement effacer le canal alpha.

`call img_dims(im, w, h)` permet de récupérer la largeur et la hauteur d'une image qui est en mémoire.

`call img_dumpslots(flag)` liste les images en mémoire. Si le flag est différent de zéro, vous avez plein d'infos supplémentaires.

`call img_version(flag)` affiche la version de libimage, et certaines options utilisées à la compilation.

14.5 Effets spéciaux.

En fait, il serait préférable de dire « effets traditionnels », parce qu'ils n'ont rien de révolutionnaires. Mais bon, ce n'est pas non plus le genre de choses que l'on fait tout les jours avec un langage de vieux dino. Même si ces « effets spéciaux » ne sont pas vraiment écrits en FORTRAN.

Chapitre 15

Les height-fields.

La libimage propose également quelques fonctions pour travailler sur les fichiers d'altitude permettant de modéliser des collines et des montagnes avec Persistence Of Vision, le célèbre *ray-tracer* presque libre.

Un height-field est en fait un tableau d'entiers compris entre 0 et 32767, stocké d'une manière particulière dans un fichier au format TARGA. La bibliothèque libimage permet de gérer facilement cette perversion d'un fichier rgb.

15.1 Le début.

On utilisera une structure d'image tout-à-fait classique, créée par la fonction `img_create`, mais avec des accès aux pseudos-pixels par des fonctions spécialisés. La sauvegarde du fichier se fera par une des deux fonctions d'enregistrement de fichier .TGA (`img_tga_sa` et `CHÉPLUKOA`).

Pour écrire une donnée d'altitude dans le height-field, on appellera la subroutine `img_hf_plot` (`img`, `x`, `y`, `hauteur`) en faisant attention de ne pas dépasser les bornes en `x` et en `y`.

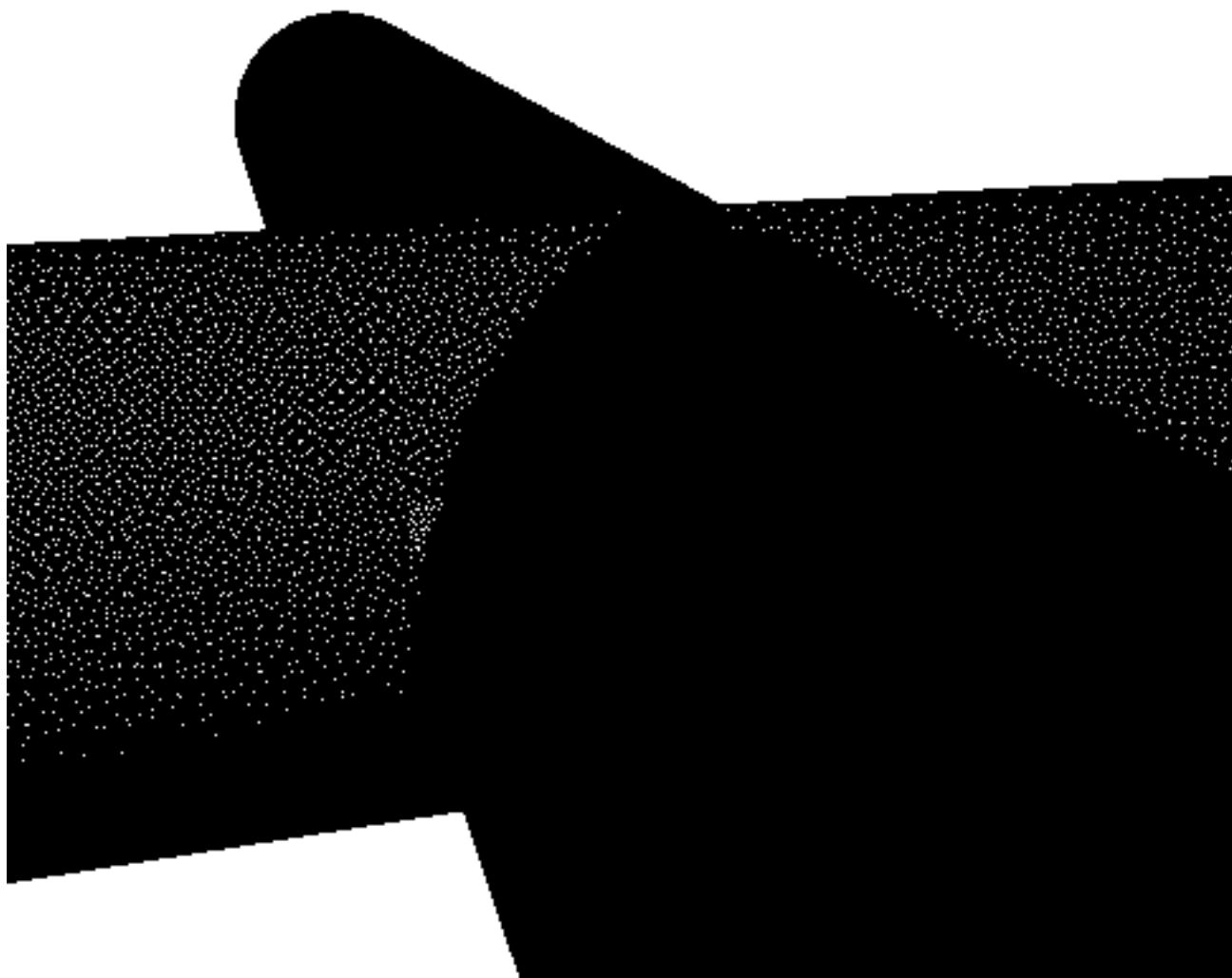
Pour récupérer la hauteur d'un point, c'est la fonction integer `img_hf_height` (`img`, `x`, `y`) qui fera le travail pour nous.

On pourra également utiliser la subroutine `img_hf_add` (`img`, `x`, `y`, `delta`) pour ajouter une valeur (positive ou négative) à un point de notre champ de hauteurs, en faisant attention à ne pas sortir des limites : entre 0 et 32767.

Avec tout ça, vous en savez assez pour commencer quelques expériences. Je vous renvoie vers la documentation de Povray pour trouver comment utiliser ces fichiers d'altitude.

15.2 Une démonstration

Oula, mais il faut que j'en fasse une...



15.3 Traitements.

Il existe quelques traitements de base sur ces champs d'altitudes. Il semble néanmoins que beaucoup d'entre eux n'aient pas encore d'interface *G77*. Il faudra probablement attendre la Saint Sarge¹.

Pour lisser un champ d'altitude, nous avons `CALL img_hf_smooth(hfsrc, hfdst, coef)`. Bien entendu, l'image source doit être différente de l'image destination, sinon, les pixels se mélangent un peu trop. Quand au coef, je ne sais plus trop à quoi il correspond, mais une petite valeur, entre 2 et 6, doit faire l'affaire, à vous d'expérimenter.

Pour normaliser un champ, vous avez `CALL img_hf_norm((hfsrc, hfdst, hmin, hmax)`.

¹laquelle ne saurait tarder, puisque la patate est cuite.

15.4 Faire autrement.

Bien entendu, vous n'êtes pas obligé d'utiliser tout cet enrobage autour des primitives de gestion des HF de la libimage, mais de ne faire la conversion de votre tableau FORTRAN que seulement au moment d'écrire le fichier dans la mémoire de masse. Seul inconvénient : l'occupation mémoire est plus que doublée, puisque pour chaque pixel du height-field, il y a deux octets dans votre tableau, plus trois octets pour la structure dans la libimage.

15.4.1 Un exemple.

Voyons, par exemple, l'écriture d'une matrice carrée dans un fichier d'altitude.

```
INTEGER*2    tableau(100, 100)
CALL remplir_tableau(tableau, 100)
```

15.4.2 Mais encore ?

On peut remarquer aussi que certaines fonctions de PGPLOT (voir page ??) sont tout à fait utilisables sur le genre de données que l'on peut mettre dans un height-field.

Chapitre 16

Créer un framework.

Maintenant que vous possédez les notions indispensables du FORTRAN, nous allons utiliser ces connaissances en créant une ossature de base pour une application complexe, mettant en oeuvre plusieurs modules, plusieurs exécutable, et des fichiers de configuration et de commandes.

Le but de cette application est la construction d'un petit court-métrage en images de synthèse, avec une bande-son également calculée. Bien entendu, tout ne pourra pas être fait en FORTRAN, l'encodage vidéo, par exemple, sera fait avec un programme externe : *mencoder*. Chaque séquence du film est générée par un programme séparé qui fabriquera une succession d'images numérotées séquentiellement.

Certaines séquences utilisant des images pré-calculées, il faudra tenir compte des dépendances entre séquences. Ce qui sera la tâche du Makefile.

16.1 Paramètres globaux.

Nous allons commencer par définir les paramètres globaux, les deux plus évidents étant la largeur et la hauteur en pixels de nos images. Il semble logique de les déclarer dans un fichier séparé qui sera inclus dans chaque unité concernée. Et c'est l'utilisation d'un COMMON qui s'impose naturellement.

Voici l'ébauche du fichier `commons.inc` :

```
C
C Shared multi-module global variables.
C
      INTEGER          width, height, frame, nbimages
      INTEGER          tdebut
      COMMON /globales/ width, height, frame, nbimages
+                tdebut
```

La variable `nbimages` contiendra le nombre total d'images, et `frame` contiendra le numéro de l'image en cours de calcul. Ces deux repères temporels devront être largement disponible, certaines procédures en ayant besoin pour se situer dans le temps relatif de la séquence.

On y trouve aussi quelques valeurs statistiques. La variable `tdebut` permettra de calculer le temps-mur¹ et donc une estimation de l'heure d'arrivée.

Il semble également nécessaire que chacun des paramètres faisant partie de la globalité du monde de l'entier soit accessible par un *accesseur-filtrant* afin de veiller à la cohérence de l'univers.

16.2 Paramètres de séquence.

Chacune des séquences durera un temps prédéfini, qui sera en fait un nombre de *frames*. Cette durée sera également utilisée pour générer le fichier audio qui accompagnera la séquence.

Chacune des séquences se verra aussi attribuer un identifiant unique (probablement une séquence de deux lettres) afin de générer la racine des noms des fichiers images.

Nous allons donc stocker ces données dans un fichier centralisé, et créer quelques fonctions d'interrogation². Ces fonctions seront dans un fichier source séparé, ce qui sera l'occasion de revoir l'édition de lien et la fabrication de `.a`.

16.3 Synchro son/image.

La vidéo sera calculée en 25 images par secondes, et le son sera probablement en 44100 Hz, ce qui nous donne 1764 échantillons sonores par image. Un chiffre bien abstrait, en vérité...

Il faut quand même songer à une chose : les données sonores peuvent parfaitement servir à piloter un générateur d'images. Inversement on peut créer des sons à partir d'une image, par exemple en convertissant des lignes de pixels en formes d'onde.

16.4 Fonctions de support.

Quelques fonctions seront écrites en C, pour des raisons de performances, ou d'interface avec le système d'exploitation. Il sera peut-être possible pour ces fonctions d'avoir accès aux variables globales décrites plus haut.

Il est important pour l'équilibre de l'univers de limiter autant que possible le recours au langage C. Ce qui ne peut être fait en FORTRAN ne mérite pas d'être fait. Telle est la dure loi du framework de dino.

¹C'est à dire, celui que l'on peut lire sur la *wall-clock*

²Peut-être, un de ces jours, un *binding* sur la `libpq`.

Chapitre 17

ncplop77 :)

Ceci est une toute nouvelle idée, et tout ce que vous pouvez lire dans ce chapitre n'est que le fruit d'une imagination torturée par l'ennui et l'abus de bière australienne¹. En gros, il s'agit de donner à notre fidèle compagnon *G77* une interface yuser-friendly, en s'appuyant sur la technologie éprouvée issue des *vt100* et *adm3a*².

17.1 Initialisation.

Avant d'utiliser quoi que ce soit, il faut mettre en place un certain nombre de choses. Il y a deux fonctions pour cela, et elles doivent être appelées dans l'ordre.

```
INTEGER foo, bar
foo = 0
bar = ncp_init0(foo)
foo = 0
bar = ncp_init1(foo)
```

Pourquoi deux fonctions? En fait, je ne sais pas vraiment. Disons que la première construit le contexte de base, et que la seconde permet de remettre les choses en l'ordre quand ça va vraiment mal.

Après cette initialisation, le contrôle TOTAL de l'écran texte est confié à *ncplop77*. Il est donc malvenu d'utiliser les fonctions classiques du FORTRAN pour afficher quelque chose. Si cela arrivait quand même, pour tenter de nettoyer le garnage sur l'écran, il faut faire un `CALL ncplop_refresh(2)` en espérant que cela fonctionne.

¹Foster roxor.

²A ce propos, si vous avez une *adm3a* qui traîne dans votre grenier, je suis preneur.

17.2 Terminaison.

MERCI DE LAISSER CET ENDROIT AUSSI PROPRE EN SORTANT QUE CE QUE VOUS AURIEZ AIMÉ LE TROUVER EN ARRIVANT.

En fait, il y a un `atexit` qui se charge à votre place de faire les nettoyages indispensables.

17.3 Fonctions de base.

Vu que la bibliothèque d'interface n'est pas encore écrite, il est difficile de donner des explications sur les fonctions de base. C'est d'ailleurs pour cette raison que le tarball n'est pas encore disponible dans le grand Ternet mondial.

17.4 Fonctions auxiliaires.

Ces fonctions ne sont pas directement liées à la bibliothèque, mais sont parfois utiles. Elles permettent d'apprendre des choses sur le contexte d'exécution, par exemple.

17.5 Exemples.

Vu que la bibliothèque d'interface n'est pas encore écrite, il est difficile de fournir des exemples, mais j'ai déjà quelques idées...

17.6 Astuces.

Vu que la bibliothèque d'interface n'est pas encore écrite, il est difficile de fournir des astuces pertinentes.

17.7 Ncidgets

Un *ncidget* est à ncurses ce qu'un widget est à Tkinter. Sauf que ça n'est pas tout à fait pareil. C'est même radicalement différent, vu que ça n'existe pas encore vraiment.

Chapitre 18

Le réseau :)

Ah..., se connecter au grand Ternet mondial avec du bon vieux code de dino serait du meilleur mauvais gout. Après tout, il y a bien eu un protocole réseau qui transférait des images-cartes dans les fils. Si un dino a plus d'informations, je suis preneur¹.

Et si un djeunz pense être mentalement préparé à l'écriture d'un RFC, il peut me contacter sur le 3615 DINO. On trouvera bien un serveur subversion prêt à nous établir un court-circuit pour ça.

18.1 Comment faire ?

Il est évident que *G77* ne propose absolument rien pour utiliser les chaussettes à la mode Berkeley. Nous allons donc passer par un pot de colle bien crade écrit en C, une fois de plus...

Hélas, intégrer *nicely* la gestion du réseau dans le *G77* me semble proche de l'impossible. Peut-être est-ce mon manque de compétence qui est en cause, mais je pense plutôt qu'il faudrait rentrer trop profondément dans la bibliothèque runtime de la chose pour que cela soit possible à mon modeste niveau.

18.2 Formaliser une adresse IPv4.

Une adresse IP classique est un nombre entier long, codé sur 32 bits. Pour une utilisation pratique par les humains, on la traite en général comme quatre octets, de la forme 192.168.232.2 qui est l'adresse de mon portable². Deux façons équivalentes de contempler la même donnée, ce qui nous un peu trop vite à ceci :

INTEGER*1	addrh(4)
INTEGER*4	addrl
EQUIVALENCE	(addrl, addrh)

¹Un truc dont le nom était BITNET ?

²Pas la peine de pinguer, il est éteint.

Oups, c'est une vilaine approximation. En effet, FORTRAN ne connaît pas les entiers non signés, et cette façon de faire a de fortes chances de ne pas fonctionner. Il va falloir trouver autre chose.

18.2.1 Ce p*t**n de resolver.

Ah, oui, il va falloir s'y frotter un jour ou l'autre. Bon, on se lance. Puisque nous avons vu un peu plus haut (*de façon approximative* comment traiter une adresse IP numérique, nous pouvons tenter de convertir une adresse symbolique, du genre `tth.vaboofer.com`, en sa valeur numérique.

18.3 Avec UDP.

UDP est l'acronyme de "user datagram protocol".

18.3.1 Quelle heure est-il ?

Il y a deux façons de connaître l'heure d'une machine distante. On peut demander l'information sous une forme humainement lisible : `Fri Aug 11 14 :51 :39 2006`³ ou sous la forme d'un entier sur 32 bits représentant le nombre de secondes écoulées depuis la Sainte Epoch⁴.

18.4 Avec TCP.

Pour le moment, je n'ai pas trouvé d'exemple bien percutant à traiter.

18.5 Et le Web 2.0 dans tout ça ?

Cet gadget de djeunz est abordé dans le chapitre ??, et ne nécessite pas, à priori, un gros traitement de bas-niveau.

³Essayez : "telnet localhost daytime"

⁴A ne pas confondre avec la Saint Sarge.

Chapitre 19

Images fractales

En fait, je viens de retrouver un *très* vieux projet de générateur d'images fractales que j'avais commencé à écrire en 1998, et que j'avais mis de côté, probablement parce que le g77 que j'avais à l'époque m'avait posé quelques soucis, en particulier sur les blocs COMMON.

Il est donc temps, en ce beau début d'automne 2006, de le faire revenir sur la scène. Et comme c'est écrit quand même un peu trop crade, nous allons reprendre tout ça dans un esprit plus framework.

D'autre part, vu comment a été construit la chose, ce sera un bon prétexte pour écrire un getopt décent, utiliser fnchek, goûter aux nombres complexes, et d'autres choses assez réjouissantes.

19.1 Généralités

19.2 Procédures communes

Chapitre 20

PostgreSQL.

Nous avons vu, lors de l'étude du framework (page 40) toute l'importance des variables globales. Nous avons également vu les immenses possibilités du détergent 2.0 (page ??). Il est peut-être temps de faire cohabiter toutes ces choses-là.

Nous avons fondamentalement deux choix : des fichiers placés dans un répertoire commun, ou l'utilisation d'une bédédé multi-valuée. Nan, j'déconne, pas de multivalué parmi nous.

20.1 Pourquoi PostgreSQL ?

Essentiellement parce que ça roxe et que c'est libre. Mais surtout parce qu'un de ses langages internes, Pg/plsql est un moteur de guikerries extraordinaires. Nous demandons une minute de silence pour toutes ces pouliches mortes au champ d'honneur des *delete on cascade*.

20.2 La base.

Nous allons commencer par quelque chose de très simple. Notre table, que nous baptiserons "essai" contiendra deux champs : un numéro et un texte. Nous devons écrire deux procédures FORTRAN permettant de lire et d'écrire une ligne dans cette table.

20.3 Raffinons le truc.

Maintenant, passons à quelque chose de plus complexe.

Chapitre 21

Autres fortraneries

21.1 Après la norme 77

Les gentils membres de la secte GNU sont en train de nous concocter un compilateur conforme à la norme 95 qu'il va bien falloir que j'essaye un de ces jours. D'autant plus qu'avec l'arrivée des branches 4.1.x de la *Gnu Compiler Collection*¹, les évolutions et même le support du G77 classique seront arrêtés. On peut quand même espérer que nous aurons une option de ligne de commande pour garder toutes les subtilités qui font la joie du Fortran 77.

Une rumeur prétend même que les travaux sont en cours pour normaliser le Fortran 2008. Je me demande bien quel bloatware ils vont réussir à nous pondre.

21.2 Exemples d'applications.

Il va falloir chercher dans les applications scientifiques, vu que le code de **MUSIC V** semble, à ce jour, introuvable. Et c'est pas faute d'avoir cherché...

Vous pouvez aussi consulter l'excellent wiki musical² de monsieur Alex³ pour quelques idées tournant autour du traitement du son et de la pseudo-musique.

Sinon, pour les courageux, il est possible d'envisager l'écriture d'un CoinCoin en g77, moyennant les supports du réseau, du XML, du HTML et du désormais indispensable bigornophone.

21.3 Et les jeux ?

Dans le passé, de nombreux jeux ont été écrits en FORTRAN. On en retrouve quelques traces (fouillez par exemple <http://catb.org/~esr/>) mais la plupart ne sont plus activement maintenus, et certains sont même difficiles à compiler avec un g77 moderne.

¹Bah oui, ça fait longtemps que GCC ne veut plus dire Gnu C Vompiler...

²<http://wiki.alpage.org/Fortran>

³<http://al.smeuh.org/>

Chapitre 22

Bibliothèque de fonctions.

Au cours de l'écriture de cet ouvrage, et des exemples de programme, je me suis trouvé plusieurs fois face aux mêmes petits soucis. A force de réinventer des roues carrées, j'ai bien fini par regrouper ces bidules dans un `.so` ré-utilisable à volonté.

22.1 Les fonctions en Fortran.

22.2 Les fonctions en C.

22.3 Les outils annexes.

Chapitre 23

Le futur de ce document

Je n'en ai aucune idée. J'ai commencé à l'écrire un soir de blues, sur un petit coup de tête, et il a fait ce qu'il n'aurait jamais du faire : il a grandi au delà de toute expectation. Maintenant, il me faut assumer, et supporter tout les caprices d'un jeune document rebelle et plein de fougue.

Je vais probablement (*dès que j'aurais un Nerim¹ à la nouvelle maison à moi que j'ai*) chercher quelques bonnes bibliothèques de fonctions (genre PGPLOT, p. ??). Je pense en particulier à quelque chose pour faire de la 3d fil-de-fer, outil graphique qui manque à mon arsenal.

Un autre projet qui me tente est une bibliothèque, peut-être basée sur ncurses, pour faire des interfaces plus *conviviales*, avec fenêtres, menus², formulaires de saisie, tout ce genre de truc. Même que j'appellerais ce truc ncplop77, d'abord. Même qu'on en cause à la page 42, ensuite.

Par la suite, pourquoi ne pas songer à un wrapper autour de la Xlib et des widgets Athena afin de faire claquer le WIMP sur nos écrans graphiques à haute résolution ? Et même, tant qu'à faire dans le crade, une interface vers **Qt** et ses abstractions objet ne serait-elle pas du meilleur mauvais goût ³ ?

Le Fortran 95, et son incarnation GNU : le GFORTRAN , peut être aussi un bon support pour l'écriture de programmes futiles et non-conventionnels. D'autant plus qu'il va bientôt prendre la place du vénérable G77, lequel va partir vers une retraite bien méritée.

23.0.1 Janvier 2007

Pourquoi ne pas profiter de ce changement d'année pour attaquer des versions plus récentes de ce vénérable langage ? Mais que choisir comme domaine d'exploration ? Un bon gros truc bien gourmand, bien numérique et bien crade à implémenter ? Un peu de **tomographie**, ça vous tente ? Moi aussi, mais je n'avance pas aussi vite que je le voudrais. *modern life is a mess...*

23.0.2 Juin 2007

Ces derniers mois, je n'ai pas trop touché au dino-jouet, d'autres trucs à faire. Je suis en train de m'y remettre petit-à-petit.

¹Nom de domaine à trouver.

²Non, je n'ai pas dit "un installateur Debian en Fortran"

³Alex, si il y a un dépôt Subversion, tu veux bien jouer avec moi ?

Chapitre 24

Conclusion.

◇
En
conclu-
sion, si vous
être en train de lire
ces lignes, il vous suffit
de dire que « jusque-là, tout
va bien ». Je vous remercie de
votre attention. Plop les moules. **Vim**
est votre ami, **Gdb** aussi. Consultez «
Viande Fraiche » régulièrement. Transcript
written on morefun.log. EHLO irc.reynerie.org.
On ne devrait jamais quitter Montauban. TRI-X roxore.
L'année prochaine, nous verrons le COBOL. Florence, je
t'aime vraiment. Tu es le soleil de ma vie. Respectez
la nature, n'utilisez que des électrons recyclés. Ne
trollez jamais sur les licences des logiciels
libres. Document réalisé avec L^AT_EX et
obstination. N'abusez pas de l'Ani-
sette, gardez ça pour les bons
soirs. Pensez à donner à
manger aux poissons
rouges. Slack-
ware, c'est
bien :-
)
◇

Index

édition de lien, 27, 41

029, 24
42, 31
72, 3
80, 3
95, 48

adresse IP, 44
alpha, 35
argc & argv, 20
array, 28
ASM, 26

bédédé, 47
bug, 6, 7

C, 41
CALL, 7
cast, 4, 18
CHARACTER, 4
CLOSE, 23
Cobol, 51
CoinCoin, 48
COMEFROM, 26
commentaire, 3
COMMON, 10, 40
compilation, 7, 12
COMPLEX, 4, 18
CONTINUE, 26
coredump, 11, 35
CYCLE, 26

décoration, 27
djeunz, 44
DO, 5
DOUBLE PRECISION, 18

ELSE, 5
END, 3
END IF, 5
ENTRY, 8, 25

EQUIVALENCE, 11

FORMAT, 9, 21
framework, 1, 40, 46
ftnchek, 12, 14, 18, 46
FUNCTION, 6
futur, 50

g95, 48, 50
games, 48
gcc, 12
gdb, 13, 16, 51
getopt, 21, 46
gFortran, 2, 50
Gnuserie, 4, 48
GOTO, 8
Guinness, 10
gw-basic, 5, 8

height-field, 37

IF, 5
ijklmn, 4, 6
INCLUDE, 10
INQUIRE, 23
INTEGER, 4

libao, 32
libimage, 34, 37
libpq, 41
libsndfile, 31
libsound77, 30
lint, 14
Linux, 27
LOGICAL, 4
lvalue, 7

Makefile, 40
mencoder, 40

ncurses, 50

OPEN, 10, 22

pgplot, 39
pixel, 35, 40
Please, Do..., 4
pnm, 34
pom, 27
POV, 37
profiler, 13
punched cards, 3, 9

référence, 7, 27
READ, 23
REAL, 4, 18
resolver, 45
retro, 23
RETURN, 7
REWIND, 23
rgb, 34, 37
rgba, 34

SAVE, 11
Spleyt, 1, 47
sprintf, 21
status, 10
stderr, 21
stdout, 29
STOP, 25
SUBROUTINE, 7

targa, 34, 37
TCP, 45
Ternet, 44
tkinter, 43
troll, 1

UDP, 45

warning, 12
wav, 30
Web 2.0, 1
WHILE, 6
widget, 43
WRITE, 21, 23

X11, 50
XXX, 29, 31, 32

yymm, 27

Table des matières

1	Introduction	1
1.1	Conventions typographiques.	1
1.2	Avertissement.	2
1.3	L'auteur.	2
2	Notions de bases	3
2.1	Structure générale	3
2.2	Types de données	3
2.3	Chaines et tableaux	4
2.4	Structures de contrôle	4
2.5	IF	5
2.6	Les boucles	5
2.7	Sous programmes	6
2.8	Compiler votre oeuvre	7
3	Perversions de bases	8
3.1	Les GOTO	8
3.2	Lire et écrire des fichiers	9
3.3	Les COMEFROM	10
3.4	Et hop, le COMMON	10
3.5	EQUIVALENCE	11
3.6	SAVE	11
4	Compilation	12
4.1	L'indispensable	12
4.2	Les <i>includes</i>	12
4.3	Compilation séparée.	12
4.4	Le superflu	13
5	ftnchek	14
5.1	Démonstration	14
5.2	Explications	15
6	Pif, paf, gdb.	16
6.1	Démonstration rapide	16
6.2	Un truc un peu plus compliqué.	17
6.3	Quelques astuces.	17
7	Les mathématiques	18

TABLE DES MATIÈRES	55
7.1 Simple et double précision.	18
7.2 Conversions de types.	18
7.3 Les nombres complexes.	18
7.4 Quelques <i>intrinsics</i> .	19
8 Quelques astuces	20
8.1 Les arguments de la ligne de commande.	20
8.2 Les variables d'environnement	21
8.3 L'équivalent du <code>sprintf</code> .	21
8.4 Ecrire sur la sortie d'erreur.	21
8.5 Chaines en paramètres	21
9 Les fichiers	22
9.1 Paramètres des fonctions d'IO	22
9.2 Fichiers 'texte'	23
9.3 Fichiers binaires	23
9.4 "Images Cartes"	24
9.5 Séquentiel Indexé	24
10 Advanced perversity	25
10.1 ENTRY	25
10.2 STOP	25
10.3 CYCLE	26
10.4 ASM	26
10.5 COMEFROM	26
10.6 Et maintenant ?	26
11 Interface avec le C	27
11.1 Avertissement préliminaire.	27
11.2 La décoration des symboles.	27
11.3 Tout est référence.	27
11.4 Les chaines de caractères.	28
11.5 Les tableaux.	28
11.6 Valeurs de retour.	29
11.7 Mélanger les I/O.	29
12 Les fichiers de sons.	30
12.1 <code>libsound77</code> .	30
12.2 Ouvrir ou créer un fichier.	31
12.3 Lire et écrire.	31
13 Attaquer les haut-parleurs	32
13.1 <code>libsound77</code> .	32
13.2 <code>beep77</code> .	32
14 Traiter les images	34
14.1 Une première image.	34
14.2 Utiliser les fichiers images.	35
14.3 Tripoter les pixels.	35
14.4 Fonctions utilitaires.	36
14.5 Effets spéciaux.	36

TABLE DES MATIÈRES	56
15 Les height-fields.	37
15.1 Le début.	37
15.2 Une démonstration	37
15.3 Traitements.	38
15.4 Faire autrement.	39
16 Créer un framework.	40
16.1 Paramètres globaux.	40
16.2 Paramètres de séquence.	41
16.3 Synchro son/image.	41
16.4 Fonctions de support.	41
17 ncplop77 :)	42
17.1 Initialisation.	42
17.2 Terminaison.	43
17.3 Fonctions de base.	43
17.4 Fonctions auxiliaires.	43
17.5 Exemples.	43
17.6 Astuces.	43
17.7 Ncidgets	43
18 Le réseau :)	44
18.1 Comment faire ?	44
18.2 Formaliser une adresse IPv4.	44
18.3 Avec UDP.	45
18.4 Avec TCP.	45
18.5 Et le Web 2.0 dans tout ça ?	45
19 Images fractales	46
19.1 Généralités	46
19.2 Procédures communes	46
20 PostgreSQL.	47
20.1 Pourquoi PostgreSQL ?	47
20.2 La base.	47
20.3 Raffinons le truc.	47
21 Autres fortraneries	48
21.1 Après la norme 77	48
21.2 Exemples d'applications.	48
21.3 Et les jeux ?	48
22 Bibliothèque de fonctions.	49
22.1 Les fonctions en Fortran.	49
22.2 Les fonctions en C.	49
22.3 Les outils annexes.	49
23 Le futur de ce document	50
24 Conclusion.	51