Initiation au C -Configuration

Cet article ne va beaucoup parler du langage C, mais plutôt de l'environnement défini par ORCA/C. D'ailleurs, tout ce qui est dit dans cet article s'applique aussi pratiquement tel quel à ORCA/M et à ORCA/Pascal.

En fait, on peut distinguer 2 environnements : celui des disquettes telles qu'elles sont fournies par ByteWorks, et celui mis en place lors de l'installation du compilateur sur votre disque dur. Vous avez maintenant tous un disque dur, non ?

Si tel n'était pas le cas, l'environnement cible doit être à priori le même que celui fourni initialement, mais franchement, je n'ai pas le courage de le vérifier. Comme je vous l'ai dit dès le premier article, il me parait indispensable de posséder un disque dur pour travailler sérieusement dans un environnement de programmation (il en est d'ailleurs de même avec le Pascal ou l'assembleur, à partir du moment où vous souhaitez écrire de véritables programmes). La suite de cet article ne fera essentiellement référence qu'à l'utilisation de l'environnement depuis un disque dur.

La version étudiée est la v1.2 du compilateur. Cette version corrige la plupart des bugs de la version 1.1, notamment ceux de l'optimiseur. Elle n'en est malheureusement pas totalement exempte elle même, comme nous aurons l'occasion de le voir par la suite.

Si vous utilisez encore une version antérieure à celle-ci, je vous recommande vivement de faire la mise à jour, d'autant qu'elle ne coûte que $10 si vous faites l'impasse sur la documentation (sa mise à jour vous coûtera $20 de plus; je pense cependant que si vos moyens vous le permettent, il est souhaitable de l'avoir aussi à jour). Mais peut-être est-il déjà trop tard pour bénéficier de ces tarifs ? J'espère en tout cas que vous êtes un possesseur légal du compilateur : je tiens en effet à ce que Mike Westerfield puisse continuer à faire évoluer ses produits (et à bénéficier des mises à jour pour si peu cher), et ne soit pas condamné à cesser son activité à cause des pirates.

Le contenu des disquettes

La boîte de ORCA/C 1.2 comprend 4 disquettes dont voici une description succinte, ainsi que du contenu de chacun de leurs répertoires :

• La disquette "Apple IIGS System Disk" contient, comme son nom l'indique, le système du GS dans sa version 5.0.4. Il n'y a rien de particulier à dire sur cette disquette, sinon qu'il s'agit d'une version partielle, et que vous préférerez donc installer la version complète que vous aurez obtenu par ailleurs.

• La disquette "ORCA/C Program Disk" contient l'essentiel de l'environnement nécessaire à l'utilisation de ORCA/C en mode bureau, à l'exception de quelques fichiers employés dans certains cas très particuliers. Si vous ne possédez pas de disque dur, c'est en fait la disquette avec laquelle il vous faudra travailler, du moins avec une copie que, j'en suis sûr, vous n'aurez pas oublié de réaliser.

Les fichiers présents sur cette disquette se répartissent de la façon suivante :

:ORCA.C

ORCA.SYS16	Il s'agit du shell mode texte v1.2.1
SYSTEM	Catalogue contenant les fichiers système d'ORCA
LOGIN	Instructions à exécuter lors du démarrage du shell
SYSCMND	Table des commandes reconnues par le shell, et liste des langages installés et des utilitaires disponibles
SYSTABS	Tabulations des éditeurs pour les différents langages
LANGAGES	Catalogue contenant les compilateurs et le linker
CC	Le compilateur ORCA/C
LINKER	Le linker d'ORCA
LIBRARIES	Catalogue contenant les librairies de ORCA/C utilisées lors de la phase de link
ORCALIB	Librairie spécifique d'ORCA/C
SYSLIB	Librairie des fonctions générales d'ORCA que l'on retrouve avec l'assembleur ou le Pascal
ORCACDEFS	Catalogue contenant la plupart des fichiers "header" à inclure au début de vos programmes C (le contenu de ce catalogue n'est pas listé car trop long).
UTILITIES	Catalogue des utilitaires ou commandes externes
HELP	Catalogue des fichiers d'aide des commandes du shell (ce catalogue est vide sur la disquette, faute de place).
PRIZM	Environnement en mode bureau lancé par le LOGIN
MAKELIB	Gestion de librairies utilisables par le linker
ICONS	ORCA.ICONS Les icônes d'ORCA pour le Finder™
SAMPLES	Catalogue de quelques exemples de programmes C (vous pouvez le supprimer si vous travaillez sur disquettes pour gagner un peu de place).

• La disquette "ORCA/C Extras" contient le reste de l'environnement tel qu'il sera installé sur votre disque dur. Il contient tous les fichiers qui ne sont pas indispensables à la bonne marche du compilateur, mais qui peuvent être nécessaires dans certains programmes. L'environnement en mode texte, notamment l'éditeur, se trouve aussi sur cette disquette, ainsi que les outils d'installation standard de GS/OS.

Elle contient essentiellement les mêmes catalogues que la disquette programme, chacun d'entre eux possédant les fichiers suivants :

:CC.EXTRAS

INSTALLER	Programme d'installation de GS/OS
SCRIPTS	Catalogue des scripts d'installation de ORCA (nous allons les voir en détail plus loin)
SYSTEM	Compléments au catalogue système pour le mode texte
EDITOR	Editeur en mode texte
SYSHELP	Fichier d'aide de l'éditeur
SYSEMAC	Macros définissables dans l'éditeur
TEXT.LOGIN	Fichier de login remplaçant le précédent lorsque l'on choisit le mode texte
SYSCMND.PAS.ASM	Table des commandes du shell lorsque l'on possède aussi le Pascal et/ou l'assembleur.
LIBRARIES	Catalogue de librairies complémentaires
PASLIB	Librairie du langage Pascal, nécessaire si on doit faire cohabiter les 2 environnements.

ORCAGLIB	Librairie C lorsque l'on utilise le modèle de mémoire dit large, à installer et à utiliser à la place de la précédente (ORCALIB) uniquement dans ce cas. Dans la pratique, vous ne devriez pas en avoir besoin, car contrairement à ce que dit la documentation, on peut continuer à utiliser le petit modèle même si l'on alloue plus de 64K de mémoire d'un seul tenant, pourvu que l'on y accède par des pointeurs.
ORCACDEFS	Catalogue de fichiers "header" jugés moins indispensables. On y trouve néanmoins GSOS.H qui est nécessaire si vous appelez directement les primitives du système. Les problèmes commenceront à se poser aux non possesseurs de disque dur qui voudront utiliser les outils correspondant à ces fichiers, par exemple les outils musicaux, faute de place sur leur disquette, sauf en utilisant la technique décrite plus loin.
AINCLUDE:M16.CC	Macros à employer avec ORCA/M lorsque vous désirez mixer du C et de l'assembleur. Son utilisation est décrite dans la documentation de ORCA/C.
UTILITIES	Catalogue de commandes supplémentaires
HELP	Catalogue de tous les fichiers d'aide (celui-ci est plein !)
CRUNCH	Utilitaire de compactage des fichiers objets après des compilations partielles
INIT	Commande de formatage d'une disquette
RELEASE.NOTES	Notes de mise à jour de la version 1.2
TECH.SUPPORT	Qui appeler en cas de problème.

• La disquette "Samples" contient un ensemble d'exemples démontrant les différentes facettes du langage. Il n'y a pas grand chose de plus à en dire.

L'installation sur le disque dur

Cette installation consiste essentiellement à créer l'arborescence des catalogues listés précédemment et d'y copier la plupart des fichiers des disquettes "Program" et "Extras". Cependant, cette arborescence peut démarrer d'un catalogue quelconque de votre disque, et pas obligatoirement du principal.

Par exemple, chez moi, les catalogues du premier niveau ci-dessus sont localisés dans le sous-sous-catalogue *:DEV:ORCA. Notez que dans ce dernier cas, le catalogue SYSTEM de ORCA/C n'aura rien à voir avec celui défini par GS/OS, puisqu'il doit se trouver dans le même catalogue que le shell (ORCA.SYS16).

Différentes options d'installation vous sont offertes. Ces options sont accessibles à travers plusieurs scripts de l'"Installer". Ces scripts, au nombre de 7, répondent à la plupart des cas possibles; il est donc préférable d'installer ou de mettre à jour votre environnement ORCA avec l'Installer, quite à l'adapter à vos besoins après.

En voici la liste commentée et classée par fonction :

• Si vous installer ORCA/C la première fois, il vous faudra choisir entre :

"New System" :

Ce script installe la totalité de l'environnement, aussi bien le mode bureau de PRIZM que le mode texteclassique. Cette installation ne peut se faire que sur un disque dur, puisque le contenu des disquettes :ORCA.C et :CC.EXTRAS est recopié.

Cependant, le login est configuré pour lancer PRIZM automatiquement; si vous préférez travailler en mode texte, il vous faudra soit l'éditer, soit le remplacer par le fichier TEXT.LOGIN décrit plus haut.

Dans la pratique, c'est le script que vous serez sans-doute amenés à employer.

Si l'utilisation principale de votre GS est la programmation, et que vous souhaitez démarrer automatiquement dans ORCA, il vous faudra choisir le catalogue de votre volume de boot comme cible de l'installation d'une part, et vous assurer que le fichier ORCA.SYS16 est le premier fichier S16 suffixé .S16 ou SYS suffixé .SYSTEM de ce catalogue et qu'il n'y a pas de fichier START dans votre catalogue :SYSTEM, d'autre part.

Ce script correspond au fichier :SCRIPTS:ORCA.HD.

"New Text System" :

Ce script n'installe que l'environnement texte, à l'exclusion de tout ce qui concerne le bureau, notamment les fichiers header d'interface avec la toolbox.

A moins que vous n'envisagiez de programmer qu'en C standard pur et dur et portable, ce script ne sert à rien.

A la limite, vous pouvez l'utilisez puis copier manuellement tous les fichiers header des disquettes :ORCA.C et :CC.EXTRAS, mais c'est loin d'être le plus pratique.

Ce script correspond au fichier :SCRIPTS:ORCA.TEXT.

• Si vous avez déjà installé une version précédente de ORCA/C ou que vous disposez d'un autre des compilateurs de ByteWorks, il vous faudra choisir parmi les scripts de mise à jour, à savoir :

"Update System" :

Ce script est le pendant de "New System". Il installe la même chose, à l'exception des fichiers de données que l'on trouve dans le sous-catalogue :SYSTEM (LOGIN, SYS?) car ceux-ci ont probablement été adaptés par vos soins.

Si vous utilisez ce script alors que vous n'avez pas déjà installé le C, il vous faudra modifier la table de commande (SYSCMND) manuellement en ajoutant la ligne :

```
CC              *L              8
```

Si vous installez le C alors que vous avez déjà installé le Pascal, il vous faudra procéder à une deuxième étape, décrite un peu plus loin.

Ce script correspond au fichier :SCRIPTS:ORCA.UPDATE.

"Update Text System" :

Ce script correspond à la mise à jour de "New Text System" et suivant les mêmes principes que le précédent.

Ce script correspond au fichier :SCRIPTS:ORCA.TUPDATE.

"Update Text System, No Editor" :

Ce script est le même que le précédent, sauf qu'il ne met pas à jour l'éditeur. La raison est qu'il existe maintenant plusieurs éditeurs de substitution à l'éditeur standard et sensiblement plus sophistiqués (EdIt-16, MaxEdit, ...).

Malheureusement, cette possibilité n'est offerte que pour la mise à jour la moins intéressante des 2 précédentes; si vous disposez de l'un de ces éditeurs et que vous voulez mettre à jour l'ensemble des fichiers, il vous faudra le sauvegarder avant l'installation, puis le restaurer après.

Ce script correspond au fichier :SCRIPTS:ORCA.TNE.

• Deux scripts complètent cet ensemble :

"ORCA Icons" :

Ce script vous permet d'installer les icônes du shell et des différents types de fichiers gérés (sources, objets, exécutables ...) pour le Finder™. Si vous n'utilisez pas ce dernier (par exemple, si vous bootez directement dans ORCA), vous pouvez ne pas les installer.

Ce script correspond au fichier :SCRIPTS:ORCA.ICONS.

"ORCA Pascal, C, Asm Libraries" :

Ce script installe la librairie Pascal, tout en mettant l'ensemble des librairies dans le bonordre (qui doit être ORCALIB — ou ORCAGLIB —, PASLIB et enfin SYSLIB), sinon des erreurs peuvent apparaître pendant le link (même si vous n'utilisez qu'un seul des 2 langages).

Si vous n'avez pas ORCA/Pascal, il n'est pas nécessaire d'utilisez ce script, car la présence de la librairie ralentira légérèment la phase de link et consommera un tout petit peu plus de mémoire.

Si vous avez ORCA/M et ORCA/C, vous n'avez pas besoin de l'utiliser non plus, car la librairie de l'assembleur (SYSLIB) est installée d'office.

Ce script met aussi à jour la table de commandes (SYSCMND) avec la définition des 3 compilateurs (il faut bien sûr les posséder pour pouvoir les utiliser). Si vous aviez modifié ce fichier, par exemple pour ajouter vos propres commandes externes, n'oubliez pas de le sauver avant !

Ce script est à utiliser en général après l'installation ou la mise à jour principale.

Il correspond au fichier :SCRIPTS:ORCA.LIBRARIES.

Fonctionnement général de l'environnement

Que vous ayez choisi de travailler en mode bureau avec PRIZM ou en mode texte, le shell est toujours exécuté en premier. Dès qu'il prend le contrôle du GS, il charge sa table de commandes située dans le fichier :SYSTEM:SYSCMND, et qui doit être un fichier texte (ou SRC). Si il ne peut la trouver ou la lire, ou qu'elle est incorrecte, le shell s'interrompra avec une erreur fatale, et il vous faudra redémarrer (attention si vous lancez le shell au moment du boot ! J'ai peur que dans ce cas, il vous faille booter à partir d'un autre disque système).

Le shell définit ensuite la plupart des préfixes originaux de ProDOS-16 (ceux de numéros inférieurs à 8) de façon à ce qu'ils correspondent aux différents sous-catalogues créés lors de l'installation. En voici la liste (le préfixe numéro 1 est positionné automatiquement par GS/OS pour pointer sur le catalogue du programme, ici le shell ORCA.SYS16) :

- 2 1/Libraries
- 3 1
- 4 1/System
- 5 1/Languages
- 6 1/Utilities

Personnellement, j'ai ajouté dans mon fichier LOGIN les définitions suivantes :

- 3 1/Temp
- 7 2/OrcaCDefs

Le préfixe 3 pointe sur le catalogue dans lequel seront stockés d'une part le résultat des commandes copier/couper de l'éditeur (ce fichier s'appelle SYSTEMP), ainsi que les fichiers temporaires créés par l'utilisation des "pipes" (le symbole | qui permet de joindre plusieurs commandes, la sortie de la première devenant l'entrée de la seconde), ces fichiers sont nommés SYSPIPEn, n étant le numéro d'ordre de la commande dans la série (on peut en effet en enchaîner plusieurs de cette façon).

Le préfixe 7 est inutilisé normalement par ORCA; la définition ci-dessus me permet d'accéder directement aux fichiers de description de l'interface avec la ToolBox.

Le shell exécute enfin le fichier LOGIN (si il existe; dans le cas contraire, le shell affiche directement son prompt "#") qui doit se trouver dans le catalogue pointé par le préfixe 4. Ce fichier permet de personnaliser son environnement, par exemple en définissant des alias pour certaines commandes, en positionnant certaines des variables standard; par exemple, vous pouvez y placer les instructions "set KeepName $" et "export KeepName" de façon à ne pas avoir à utiliser l'option "keep=" lors de l'utilisation des commandes "cmpl" ou "link" si vous utilisez le mode texte. Vous pouvez aussi définir les préfixes que vous souhaitez (rappelez-vous qu'il y en a 32 disponibles) comme indiqué précédemment.

Si vous préférez l'environnement de bureau, votre fichier LOGIN devra exécuter à sa fin la commande PRIZM (ce qui est le cas si vous avez effectué une installation complète), qui n'est en fait qu'une commande comme les autres, si bien que vous pouvez le lancer et le quitter à tout moment.

Le catalogue SYSTEM identifié par le préfixe 4 est utilisé essentiellement par l'éditeur, une fois que le shell a terminé son initialisation. Lorsqu'en effet, vous tapez une commande EDIT en mode texte, le shell cherche un programme EDITOR de type EXE dans ce catalogue. Si vous utilisez un éditeur de remplacement, il vous suffira donc de remplacer le programme EDITOR par cet éditeur et de lui donner le type EXE (au cas où ce serait S16); c'est tout ! L'éditeur utilise le fichier SYSTABS décrivant les tabulations souhaitées en fonction du numéro du langage; ce fichier est aussi utilisé de la même façon par PRIZM).

Le catalogue LANGUAGES identifié par le préfixe 5 contient les compilateurs des langages installés ainsi que le linker. Lorsque vous lancez la compilation d'un programme C, le shell identifie le langage à l'aide du champ "auxtype", qui dans ce cas vaut 8, puis à l'aide de sa table de commande associe ce numéro au langage C et donc au compilateur CC. Notez que le nom du langage défini dans la table SYSCMND doit être le même que le nom du programme compilateur; c'est aussi ce nom qui est affiché par la commande CATALOG, et enfin le nom de la commande que vous devrez taper avant d'éditer un nouveau fichier, de façon à l'identifier correctement; dans le cas contraire, il vous faudra le modifier après coup avec la commande CHANGE. Si vous travaillez essentiellement avec un langage, par exemple le C, il peut être judicieux de placer la commande CC dans votre fichier LOGIN; ainsi tous les nouveaux programmes seront identifiés correctement.

Le catalogue LIBRARIES identifié par le préfixe 2, ainsi que ses sous-catalogues (il y en a un pour chacun des langages installés, leurs contenus correspondant aux interfaces de ces langages avec entre autres la ToolBox), n'est pas utilisé par le shell, mais plutôt par les compilateurs et le linker.

Par exemple, lorsqu'en C vous employez l'instruction "#include <xxx.h>", vous indiquez au compilateur qu'il doit aller chercher ce fichier dans le catalogue "2/OrcaCDefs".

ORCA/Pascal va lui chercher les Units référencées dans l'instruction USES dans le catalogue "2/OrcaPascalDefs". En revanche, lorsqu'en C vous utilisez la syntaxe "#include "xxx.h"", le compilateur regardera dans votre catalogue courant (identifié par le préfixe 0) si le nom est partiel et à l'endroit indiqué si vous avez spécifié un chemin complet.

Le linker lui n'utilise que les fichiers de type LIB de ce catalogue et ne va jamais regarder dans les sous-catalogues. Il les consulte dans l'ordre de leur apparition dans le catalogue, et une fois qu'il a fini d'en traiter une, il passe à la suivante, sans jamais revenir aux précédentes, même si des références externes n'ont pas pu être résolues; c'est pourquoi l'ordre est très important. Si vous êtes amenés à écrire des librairies et que vous souhaitez que le linker les regarde automatiquement, il vous faudra les placer avant les librairies standard — et donc utiliser un programme de tri de catalogue, tel que Prosel-16 — notamment si elles sont écrites en C ou en Pascal, car vous ferez alors inévitablement référence aux librairies système.

Le catalogue UTILITIES identifié par le préfixe 6 contient les commandes externes du shell, ainsi que le sous-catalogue HELP des fichiers d'aide.

Si vous développez un programme fonctionnant dans le mode texte du shell et que vous souhaitez

pouvoir l'utiliser quelque soit le catalogue dans lequel vous vous situez, le plus simple est d'ajouter une ligne dans la table de commandes SYSCMND prenant la forme "mon_utilitaire *U", le caractère "*" étant utilisé si vous voulez pouvoir redémarrer ce programme directement en mémoire sans le recharger (pourvu que la place soit suffisante pour le conserver), car vous l'utilisez souvent. Vous devrez aussi bien entendu recopier ce programme dans le catalogue 6. Une autre possibilité est de définir un alias vers ce programme en spécifiant le chemin complet, si vous préférez ne pas modifier ce catalogue ou la table de commande; en revanche, vous ne pourrez pas le faire redémarrer en mémoire. Dans les 2 cas, il vous suffira alors de taper son nom, éventuellement suivi de ses paramètres pour l'exécuter automatiquement.

Ce programme peut aussi bien être un fichier de type EXE que S16 ou SYS ou encore EXEC (dans les cas S16 et SYS, le shell les lancera par un QUIT GS/OS et vous ne pourrez alors pas leur passer de paramètres — le shell n'utilise pas (encore) Message Center), mais à mon avis il est préférable que ce soit un fichier de type EXE, sauf dans le cas où il utilise des ressources, et qu'il n'ouvre pas lui même son resource fork (StartUpTools ouvrira celui du shell, qui de toutes façons n'existe pas). La table de commandes peut être rechargée par le shell sans le quitter, en utilisant la commande COMMANDS 4/SYSCMND; celle-ci a cependant aussi pour effet de purger la mémoire de tous les programmes résidents, suite à l'utilisation de "*U" et "*L".

Pratiquement tous les programmes C (et Pascal) sont redémarrables en mémoire :
il suffit d'initialiser les variables globales dans le programme principal ou dans un sous-programme quelconque, et non pas en utilisant la syntaxe d'initialisation automatique des variables présentée la dernière fois (qui elle n'aura lieu que lors du chargement depuis le disque).

Lorsque vous employez la commande HELP, si vous n'avez pas spécifié la commande pour laquelle vous souhaitiez avoir de l'aide, elle vous affiche le contenu de la table de commandes du shell, sinon elle effectue un simple affichage du fichier ayant le même nom que le paramètre et étant situé dans le catalogue 6/HELP.

Conclusion

Voilà, je pense avoir fait le tour de l'environnement d'ORCA, qui de part son utilisation des préfixes est assez souple.

Si par exemple, vous n'avez pas de disque dur, mais 2 lecteurs de disquettes, vous pouvez envisager de répartir l'environnement sur 2 disquettes, et modifier les préfixes dans le fichier LOGIN pour pointer sur les bons catalogues; attention cependant au fait que GS/OS aime bien avoir le disque de boot toujours présent, notamment si vous utilisez des programmes avec des ressources; on doit cependant arriver à s'en sortir en installant le shell à la place du Finder™, ainsi que le catalogue SYSTEM de ORCA dans le catalogue SYSTEM d'origine, éventuellement PRIZM (et le reste de UTILITIES à l'exclusion des fichiers d'aide) si il reste suffisamment de place et que vous souhaitez utiliser l'environnement de bureau, le reste étant installé sur la deuxième disquette.

Vous pouvez aussi envisager d'avoir 2 arborescences de librairies, une pour le petit modèle et une autre pour le grand, et faire pointer le préfixe 2 sur l'une ou sur l'autre selon les besoins (les autres fichiers, notamment le sous-catalogue ORCACDEFS, devront être en double), encore que je n'ai pas trouvé de cas où il faille utiliser le grand modèle.

Je n'ai pas parlé du processus d'édition/compilation/link, car il varie selon que vous utilisez le shell en mode texte ou l'environnement du bureau de PRIZM. Il est cependant assez simple dans les 2 cas pour que vous puissiez vous débrouiller tout seuls.

Ah oui ! J'ai failli oublier de parler des bugs. En fait, je n'en ai vraiment trouvé qu'un jusqu'à présent, mais qui m'a fait passer beaucoup de temps, étant donné que cela marchait avec la v1.1.

Les fichiers d'interface de la ToolBox sont fournis par Apple et livrés tels quels par ByteWorks. Dès que vous voulez en utiliser un, il vous faut tout d'abord inclure le fichier "Types.h" (c'est de toute façon

fait automatiquement par l'ensemble des fichiers header de la ToolBox) qui définit les constantes et les types de base, notamment la constante NULL de la façon la plus standard :

```
#define NULL 0x0L
```

Cependant ORCA/C génère du code faux (qui sème la panique dans la pile) lorsque l'on affecte cette constante à un pointeur, ce qui est quand même le BA BA de l'utilisation des pointeurs ! Heureusement la correction est simple : il vous suffit de modifier Types.h de sorte que la définition soit :

```
#define NULL (void *) 0x0L
```

C'est à dire que l'on transforme explicitement la constante en pointeur, ce que devrait faire implicitement le compilateur, comme c'était le cas en v1.1.

Cette définition est faite ainsi dans le fichier stdio.h que l'on inclut à la place de Types.h lorsque l'on n'utilise pas la ToolBox.

Si vous avez besoin d'autres précisions sur l'environnement de ORCA, n'hésitez pas à m'en faire part. J'essaierai d'y répondre, dans la mesure de mes moyens, soit par un article, soit par un petit mot dans la rubrique Forum C.

Vous trouverez la suite de l'initiation au langage C dans le prochain numéro de GS Infos, dans lequel nous finirons enfin par parler des opérateurs et des expressions ...

Initiation au langage C

Introduction

Cet chapitre est le premier d'une nouvelle (et je l'espère longue) série d'initiation au langage C.

Cette série suppose que vous possédez déjà des rudiments de programmation, par exemple en Basic ou en Pascal, que vous pouvez avoir acquis après avoir étudié les articles sur le Pascal ou l'algorithmique parus dans les précédents numéros de GS Infos.

Il sera fait de temps à autre référence au Pascal, qui est le langage le plus connu dans la communauté des utilisateurs programmeurs de l'Apple IIGS, à titre de comparaison et pour faciliter les explications.

Cependant, si vous n'avez aucune connaissance de programmation, restez quand même avec nous, j'essaierai de vous faire découvrir ce langage et peut être vous allez figurer à votre tour dans la longue liste des programmeurs sur Apple IIGS.

Ces articles doivent servir de forum sur le langage C, et j'ai par conséquent besoin de savoir si le contenu et/ou le niveau vous satisfont (ou si ce dernier est trop élevé ou trop simpliste), si vous voulez voir traiter certains sujets particuliers, etc ...

Pour ce faire, vous pouvez me contacter à l'adresse suivante :

Philippe Manet
40 rue Victor Hugo
94700 Maisons Alfort

Après les présentations d'usage, entrons maintenant dans le vif du sujet, en rappelant d'abord l'historique du langage, puis en présentant les compilateurs disponibles sur l'Apple IIGS. Une bibliographie et nos premiers programmes C complètent ce premier article.

Un peu d'histoire

Le langage C est né au début des années 70 (à peu près en même temps que Pascal) de l'imagination fertile de Dennis Ritchie, chercheur aux laboratoires BELL de AT&T (les téléphones américains). L'objet de l'inventeur était de définir un langage d'assez haut niveau fournissant des structures de contrôle complètes permettant la programmation dite structurée, mise en avant par Algol puis Pascal. Il avait cependant aussi besoin d'un langage ayant une grande puissance dans le traitement des expressions de calcul et ayant des types de données de base proches de ceux disponibles dans la machine, puisqu'il voulait se servir de ce langage pour développer un système d'exploitation, ce qu'il fit avec Unix.

C est dérivé d'un langage plus simple (il n'avait par exemple qu'un type entier) appelé, je vous le donne dans le mille ... B.

Je vous rassure tout de suite, C est complètement indépendant d'Unix, comme nous le verrons dans les prochains articles. On peut donc apprendre et programmer en C, sans jamais penser à Unix ou à aucun autre système d'exploitation particulier.

Bien que déjà ancien, le langage n'a connu une véritable popularité que dans les années 80, avec l'avènement de la micro-informatique, et plus particulièrement du monde MS-DOS; la plupart des applications de cet environnement sont d'ailleurs écrites en C. Comme c'est le cas avec Unix, le langage C n'a (heureusement !) rien à voir avec la galère MS-DOS (à titre d'anecdote, les américains disent en général MeSs (ou MeSsy)-DOS, ce qui veut dire à peu près Système d'Exploitation M.....).

Malgré son ancienneté, le langage n'a subi que très peu d'évolutions depuis ses origines, preuve de son efficacité. Les plus importantes d'entre elles ont été apportées par la normalisation du langage par l'ANSI (l'équivalent américain de notre AFNOR) qui n'est toujours pas validée à ma connaissance. Ces nouvelles fonctionnalités concernent plus particulièrement la qualité des programmes sources grâce à un mécanisme appelé prototypage (nous verrons plus loin que ce terme savant recouvre un concept de base de Pascal). Précédemment, il était possible de faire à peu près n'importe quoi dans les passages de paramètres à des sous-routines (c'est d'ailleurs toujours possible, le prototypage étant optionnel), ce qui contribue à la puissance, mais aussi à l'opacité (pour ne pas parler des difficultés de mise au point) qui ont fait la réputation de C. La norme ajoute aussi la notion de type énuméré (qui existe depuis toujours en Pascal) ainsi que d'autres petites choses qu'il n'est pas utile de lister ici.

Une autre évolution du langage C a débuté au début des années 80 par un autre chercheur des laboratoires Bell (Bjarne Stroustrup), visant à intégrer au langage des nouveaux concepts dits "orientés objets" mis en avant par SmallTalk (pour ceux qui en ont entendu parler). Ces recherches ont conduit à un nouveau langage, non pas appelé D, mais C++. En dehors de ces nouveaux concepts, C++ apporte des nouvelles fonctions au C, permettant d'améliorer sensiblement la qualité du code source; une partie de ces améliorations a d'ailleurs servi de base aux travaux de normalisation du langage C. C++ se présente le plus souvent sous forme d'un pré-prossesseur traduisant le programme en C et qui est compilé ensuite par ce dernier; il commence aussi à exister des compilateurs natifs C++. Le démarrage de C++ est fulgurant, surtout dans le monde des clones PC et plus modérément sur le MacIntosh. A ce jour, il n'y a pas de compilateur, ni de pré-processeur C++ pour Apple IIGS.

Le C et l'Apple IIGS

Il existe aujourd'hui 2 implémentations du langage C sur Apple IIGS (il existe aussi des implémentations en ProDOS 8, dont la plus connue est AZTEC C, mais nous n'en parlerons pas ici) :

- APW C est la version proposée par Apple et la première disponible. Elle est aujourd'hui un peu dépassée, et elle n'est pas vraiment supportée. De plus, elle ne respecte pas la norme ANSI. Cette version requiert l'environnement APW/ORCA et ne peut s'utiliser qu'en mode texte. La librairie run-time (les fonctions standards fournies par le langage) est très mauvaise, si bien que le moindre programme (y compris un programme vide ne faisant rien !) génère un exécutable de taille impressionnante (plusieurs dizaines de KO). Ceci contraint à disposer d'un minimum de mémoire suffisant, 2 MO semblant raisonnable.

- ORCA/C est la version développée et vendue par Byte Works. Il en est aujourd'hui à la version 1.2 (qui vient juste de sortir). Cette version est maintenant assez stabilisée et de bonne qualité. ORCA/C est à la norme ANSI, ce qui facilite la mise au point des programmes, si on en utilise les possibilités, et évite ainsi des maux de tête causés autrement par les nombreux crashes. ORCA/C peut être utilisé dans l'environnement texte APW/ORCA ou dans l'environnement de bureau PRIZM fournis en standard avec le compilateur. La librairie run-time est de bonne qualité, évitant de générer des programmes de grosse taille.

Bien que fonctionnant dans la configuration de base de 1,2 MO, il est recommandé d'augmenter sa mémoire à 1,5, voire 2 MO pour des programmes de bonne taille, sinon les accès disques sont nombreux pendant les compilations (surtout si vous utilisez la toolbox, car il y a alors beaucoup de fichiers à inclure). Le compilateur dispose aussi d'une fonction d'optimisation du code objet, diminuant la taille du

programme final et améliorant les temps d'exécution. Enfin, et ce n'est pas le moindre de ses avantages, ORCA/C possède une fonction de mise au point (debugger en anglais) du code source, c'est à dire que vous pouvez placer directement des points d'arrêt et examiner les variables dans le programme C original et non pas dans le programme objet résultant de la compilation.

Pour toutes les raisons évoquées ci-dessus, et pour vous initier à une "bonne" écriture en C, les articles suivants seront basés sur ORCA/C.

Vous pourrez cependant utiliser APW C au prix de légères adaptations des programmes présentés. Le présent article vous montrera le type de modifications que vous serez amenés à effectuer.

Quelque soit votre choix, vous devez envisager l'achat d'un disque dur. Les 2 compilateurs fonctionnent correctement avec 2 lecteurs de disquette, voire même 1 pour ORCA/C, mais si vous vous lancez dans le développement d'un programme de bonne taille, vous vous apercevrez rapidement qu'un disque dur devient rapidement indispensable.

Bibliographie

Il existe une foule de livres d'initiation au langage C en français, la plupart étant basés sur Turbo-C ou MicroSoft-C pour les PC. Je ne saurai vous en recommander un en particulier, ne les connaissant pas. Attention,cependant, la plupart des livres informatiques, surtout ceux d'initiation, sont mauvais, la plupart des auteurs se contentant de traduire le manuel d'utilisation original. A vous de faire le bon choix.

Je peux malgré tout vous recommander la "bible", à savoir le livre "Le Langage de Programmation C" par Dennis Ritchie et Brian Kernighan paru chez Masson (veillez à ce que ce soit la 2ème édition prenant en compte les nouveautés de la norme ANSI). Il s'agit de l'ouvrage de référence, écrit par les inventeurs du langage. Sans être forcément simple, il est assez didactique et peut servir de manuel d'initiation au C. Un autre ouvrage complémentaire peut être utile (je ne me souviens pas du titre, mais il est aussi publié chez Masson et fait référence au précédent); il contient la solution des différents exercices proposés dans la bible, et peut ainsi être utilisé comme guide d'exemples pratiques du langage.

Bien qu'il ne s'agisse pas d'un livre d'initiation au C, et qu'il commence à dater (il correspond à la version 3.1 du disque système), vous pouvez envisager l'achat du livre "Boîte à outils de l'Apple IIGS" de J.P. Curcio paru aux éditions du P.S.I. si il est encore disponible. Comme son nom l'indique, il décrit l'utilisation de la boîte à outils du GS, et tous les exemples sont en C.

Vous pourrez trouver tous ces livres soit à la FNAC, soit chez Infothèque, 32 rue de Moscou, 75008 Paris (c'est près de la Gare St Lazare). Soit dit en passant, Infothèque vend l'ensemble de la documentation technique Apple IIGS, qui se revélera indispensable (3 volumes des Toolbox Reference Manual et GS/OS Reference Manual entre autres) dès que vous voudrez réaliser des programmes utilisant les spécificités de votre IIGS préféré.

Notre premier programme C ! Enfin, me direz vous :-)

Je ne vais pas entrer dans les détails d'utilisation des différents environnements de programmation disponibles. Pour cela, vous devrez vous référer à la documentation fournie avec le compilateur choisi. Un atout de plus pour ORCA/C est qu'il dispose d'un environnement en mode "bureau" beaucoup plus simple à apréhender et très bien expliqué dans le manuel.

En tout état de cause, je suppose que vous savez éditer un programme et le compiler.

Tout livre d'initiation au C propose comme premier programme l'affichage du message "hello, world". Alors allons y :

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

Si vous compilez puis exécutez ce programme que vous aurez appelé par exemple "hello.cc" (par convention, les programmes C ont le suffixe ".cc" indiquant leur type), vous devriez voir apparaître le fameux message "hello, world" sur votre écran. Si vous avez utilisé l'environnement PRIZM, celui-ci apparaîtra dans une fenêtre appelée "shell"; cette fenêtre permet d'exécuter les commandes du shell ORCA de la même manière qu'en mode texte.

Tout programme C est composé d'un ou plusieurs fichiers sources (C supporte naturellement le concept de "compilation séparée"; nous reviendrons sur ce sujet dans un prochain article), eux mêmes composés d'une ou plusieurs "fonctions". Une fonction C est équivalente à une procédure (ou une fonction) Pascal, et de manière plus éloignée, à une sous-routine Basic.

La fonction principale (celle par laquelle débute automatiquement l'exécution du programme) doit s'appeler obligatoirement "main"; elle correspond au PROGRAM de Pascal.

Vous avez sans doute remarqué que le programme précédent est en minuscules :
ceci est extrêmement important, car le langage C, contrairement aux autres, fait la différence entre minuscules et majuscules. Ainsi, les mots clefs du langage doivent être impérativement en minuscules, sinon ils ne seront pas reconnus par le compilateur. En revanche, vous pouvez utiliser aussi bien des minuscules que des majuscules dans les noms des variables ou des fonctions, sauf pour la fonction principale qui doit s'appeler "main" en minuscules. Attention cependant au fait que vous devrez toujours écrire un symbole de la même manière : par exemple, "toto" est différent de "TOTO" et de "Toto".

Une fonction C se présente de la manière suivante :

```
type-du-résultat  nom-de-la-fonction  ( paramètres-de-la-fonction )
{
      code-de-la-fonction;
}
```

Contrairement à Pascal qui dispose de procédures effectuant des opérations mais ne retournant rien et des fonctions retournant une valeur, C ne dispose que de fonctions retournant en principe toujours quelque chose; c'est pourquoi, il n'y a pas de mot clé indiquant que l'on définit une fonction.

Originellement, le champ "type-du-résultat" pouvait être vide, ce qui ne voulait pas dire que l'on ne retournait rien (bien que ce fût généralement le cas), mais que l'on retournait un entier. A des fins de compatibilité, c'est toujours le cas. Cependant, la norme a ajouté un nouveau type de données réservé à la déclaration d'une fonction et appelé "void" (comme dans notre exemple précédent), ce qui veut dire vide. Une fonction retournant void (c'est à dire rien) est donc strictement équivalente à une procédure Pascal. Dans le cas d'ORCA/C, il est utile de déclarer les procédures comme des fonctions retournant void, car le compilateur utilise cette information pour optimiser le code généré. APW C ne dispose pas de ce type de données; on se contente alors de ne rien mettre du tout.

Lorsque void est utilisé dans le champ "paramètres-de-la-fonction", cela signifie que celle-ci n'accepte aucun paramètre, et demande au compilateur de vérifier qu'effectivement, aucun paramètre n'est passé à cette fonction.

C'est ce mécanisme, réalisé d'office en Pascal, que j'ai appelé plus haut "prototypage" : il s'agit simplement de dire au compilateur de vérifier la cohérence des paramètres d'une fonction entre sa définition et ses appels.

Vous l'avez deviné, ce contrôle n'était pas effectué dans les versions de C antérieures à la norme; pour des raisons de compatibilité avec des programmes anciens, le mécanisme de prototypage est optionel, et si les types des paramètres ne sont pas spécifiés dans la déclaration d'une fonction, le compilateur n'effectuera aucun contrôle, d'où une mise au point nettement plus délicate, des erreurs évidentes n'étant pas détectées à la compilation.

Ceci permet en revanche de faire des choses très puissantes, comme par exemple de déclarer une fonction avec 2 paramètres entiers et de lui passer un entier long.

Le programme précédent n'utilisant pas le prototypage (et donc compilable avec APW C) s'écrit donc :

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

Dans cet exemple, le compilateur C ne vérifiera pas qu'aucun paramètre n'est effectivement passé à cette fonction, alors qu'il l'aurait fait avec la version précédente.

Il est à noter que l'essentiel des modifications que vous aurez à apporter aux exemples donnés consistera en la suppression des prototypes, des types void (que vous pourrez remplacer par le type entier, si besoin est) et autres particularités de la norme que j'indiquerai par la suite.

Vous avez déjà deviné que les "{" et "}" servaient à délimiter le corps d'une fonction; en fait, ils sont strictement équivalents aux "BEGIN" et "END" de Pascal.

Le ";" est utilisé identiquement à Pascal pour séparer les instructions. Il n'y a pas, en revanche, de "." final, la fonction principale "main" pouvant en fait se trouver n'importe où dans le fichier source.

Comme Pascal, C ne possède pas d'instructions d'entrée/sortie. A la place, il offre un ensemble de fonctions standards présentes dans la librairie "run-time". "printf" fait partie de ce jeu de fonctions; il signifie affichage formaté, et comme nous le verrons par le suite, il est beaucoup plus sophistiqué que ses équivalents WRITE/WRITELN du Pascal.

Une chaine de caractères C est délimitée par le caractère ". Un certain nombre de séquences prédéfinies correspondent aux caractères de contrôle les plus utilisés. Ainsi la séquence "\n" correspond à "newline", et permet de passer à la ligne suivante; elle permet de réaliserl'équivalent du WRITELN, excepté qu'elle peut se situer n'importe où dans la chaine de caractères.

Si, par exemple notre programme précédent avait été :

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

l'affichage aurait été constitué des 2 lignes :

```
hello
world
```

Les espaces de part et d'autre des () ne sont là que pour améliorer la lisibilité du programme, et sont donc tout à fait optionnels. Notre programme pourrait donc s'écrire (à l'extrême, j'en conviens) :

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

ce qui est nettement moins lisible !

Allons un peu plus loin

Nous avons vu dans l'exemple précédent que la fonction "main" était une fonction ordinaire, sauf qu'elle s'appelait "main" et que c'était la première exécutée. Comme toute fonction, elle accepte des paramètres, qui sont cependant prédéfinis, du fait qu'elle est appelée par une routine standard de la librairie run-time exécutée avant elle.

Ces paramètres correspondent à la ligne de commande qu'un utilisateur saisit dans un environnement de type "shell", tel celui de APW/ORCA. En revanche, si le programme est exécuté depuis le finder, par exemple, la ligne de commande ne correspond à rien, et aucun paramètre n'est donc passé à la fonction "main".

Dans le shell APW/ORCA, il existe une commande standard "echo" qui affiche simplement la liste des arguments qui lui sont fournis. Si par exemple, je tape la commande :

```
echo Ceci est un message
```

Je verrai apparaître sur mon écran sur la ligne suivante :

```
Ceci est un message.
```

Dans l'exemple qui suit, nous allons voir comment réaliser en C un programme qui réalise exactement la même fonction que la commande "echo".

Un dessin valant mieux qu'un grand discours, voici ce que cela donne :

```
void main ( int argc, char argv[][] )

{

   int i;

   for ( i = 1; i < argc; i++ ) {

        printf ( argv[i] );
      if ( i < argc - 1 )
          printf ( " " );
    else
          printf ( "\n" );

  }

}
```

Appelez le echoc.cc puis compilez le et faites l'édition de liens. Si maintenant vous tapez la commande (si vous êtes dans PRIZM, utilisez la fenêtre "shell") :

echoc ca marche.

Vous devriez alors voir le message :

ca marche.

juste en dessous.

Si vous avez déjà pratiqué le Pascal, vous devez reconnaître certains éléments du langage. Cependant, et afin de conserver un peu de suspense, les explications détaillées ne vous seront données que dans le prochain article.

La fonction "main" accepte 2 paramètres qui sont le nombre d'arguments de la ligne de commande et un tableau de chaines de caractères (en C, une chaine de caractères est un tableau de caractères, d'où la description du paramètre "argv" en tant que tableau à 2 dimensions). Un argument d'une commande est simplement une suite de caractères quelconques exceptés l'espace et la fin de la ligne qui délimitent chacun de ces arguments; le découpage est fait automatiquement avant l'exécution de la fonction "main". Si l'on veut regrouper plusieurs mots séparés par des espaces au sein d'un même argument, il suffit de délimiter ces mots par des "; dans le cas du programme "echoc", le résultat sera strictement identique.

Les noms de variable "argc" et "argv" sont ceux couramment utilisés pour matérialiser les arguments d'une commande; le "c" veut dire "count" (le nombre) et le "v" correspond à "vecteur", c'est à dire la liste des arguments proprement dits.

Et après ?

Dans le prochain article, nous commencerons l'étude de la syntaxe du langage C, et plus particulièrement des types de données scalaires et des opérateurs disponibles dans les expressions arithmétiques et logiques.

Nous reprendrons le programme "echoc" précédent, en expliquant chacune de ses instructions, et en montrant comment il peut être écrit de manière beaucoup plus synthétique; c'est ce qui fait la puissance du langage et si trop poussé à l'extrême sa complexité.

Initiation au Langage C (deuxième partie)

Avant d'entrer dans le vif du sujet, je voudrais faire un petit retour en arrière sur la bibliographie donnée dans le premier article.

Je me suis aperçu que j'avais donné un titre inexact pour la bible, que l'on appelle d'ailleurs le K&R (du nom de ses auteurs, Kernighan et Ritchie); son nom exact est le "Langage C" 2ème édition et il est effectivement paru chez Masson.

Depuis la dernière fois, je me suis renseigné sur les meilleurs livres disponibles, et on m'a recommandé ces 3 titres (mais je n'ai pas vérifié par moi-même), qui sont des traductions de livres américains publiées par Masson :

• "Langage C : manuel de référence" de Samuel Harbison et Guy Steele, 2ème édition. Il s'agit d'un manuel de référence très complet et non pas d'un livre d'initiation, mais une fois que vous posséderez les rudiments du langage, vous pourrez l'utiliser.

• "Langage C : problèmes et exercices" de Alan Feuer. C'est un livre d'exercices couvrant pratiquement tous les aspects du langage. Son intérêt est qu'il présente les erreurs les plus courantes, que vous serez sans aucun doute amenés à faire dans vos propres programmes, et donne bien entendu une correction à ces erreurs.

• "Langage C : solutions" de Clovis Tondo et Scott Gimpel, 2ème édition. Ce livre donne une solution à l'ensemble des exercices proposés dans le K&R.
J'en avais vaguement parlé dans le précédent article, j'ai juste retrouvé son titre et ses auteurs.

Pour terminer, je vous donne l'adresse d'une autre librairie informatique sur Paris :
Le Monde en Tique, 18 rue Maître Albert, Paris 5ème.

Contrairement à ce que j'avais annoncé dans le précédent article, nous n'allons étudier que les constantes et les types de base dans cette partie, tandis que nous traiterons des opérateurs que l'on peut utiliser pour construire des expressions à partir de ces types dans 2 mois. La raison en est que je pense qu'il y a déjà pas mal d'informations à ingurgiter dans cet article, et qu'il n'est donc pas souhaitable de vous encombrer l'esprit dès le début. Par conséquent, nous ne reprendrons le programme echoc.cc que dans le prochain article.

Le langage C

Nous pouvons maintenant entrer de plain pied dans la syntaxe du langage C.

Les données et leur types

Un programme, par essence, manipule des données. Dans les langages modernes, ces données sont typées. De plus, elles représentent soit des constantes (la donnée a une valeur fixe prédéterminée qui ne change jamais pendant le déroulement du programme), soit des variables (la valeur de la donnée varie au cours de l'exécution du programme, par exemple à la suite d'un calcul).
Le type d'une donnée est utilisé par le compilateur pour déterminer les opérations possibles sur cette donnée, ainsi que l'espace mémoire qu'elle occupe.

Une donnée, lorsqu'elle est matérialisée par une variable, est représentée par 2 entités : pour le programmeur, il s'agit de son nom qui l'aide à l'identifier dans le corps de son programme, tandis que pour l'ordinateur, il s'agit de son adresse, qui lui sert à la localiser dans sa mémoire. Il faut bien comprendre que ces 2 dénominations ne recouvrent qu'une seule et même chose; c'est simplement le point de vue qui change.

Format général d'un programme C

Une instruction peut utiliser une ou plusieurs lignes, et une ligne peut comporter plusieurs instructions (bien qu'en général et pour des raisons de clarté on n'écrit qu'une seule instruction par ligne); chaque instruction est terminée par le caractère ";". Les éléments syntaxiques formant des mots doivent être séparés de leurs voisins par des espaces, des tabulations, des lignes blanches ou des commentaires; lorsqu'il s'agit d'un symbole définissant la ponctuation du langage, celui-ci peut être accolé à un mot (par exemple x=3), cependant pour améliorer la lisibilité du programme, je vous recommande d'utiliser un espace de part et d'autre du symbole (comme dans x = 3).

Un commentaire est délimité par les séquences "/*" et "*/" qui ne peuvent être imbriquées (il n'est pas possible d'avoir la séquence /* ... /* ... */ ... */).
En revanche, on peut intercaler un début de commentaire au milieu d'un commentaire, par exemple /* ... /* ... */.

Il est aussi possible de couper un élément en 2 (un mot réservé, une chaîne de caractères ...) en utilisant le caractère de continuation "\" immédiatement suivi d'un retour chariot. Attention, si au moins un espace vient à se glisser entre les 2, le compilateur n'identifiera plus cette séquence de continuation, et vous donnera une erreur; par conséquent, je ne vous conseille pas d'utiliser cette possibilité (d'ailleurs elle est rarement nécessaire).

Identificateurs

Le langage définit la syntaxe des identificateurs utilisés pour symboliser les constantes et les variables, mais aussi pour les fonctions. En C, un identificateur est constitué d'une lettre ou du caractère souligné "_", éventuellement suivi d'un ou plusieurs lettres, chiffres ou soulignés (il s'agit bien entendu des lettres de A à Z et de a à z sans aucun accent); ORCA/C limite la longueur d'un tel identificateur à 255 caractères, ce qui laisse tout de même pas mal de liberté pour définir des noms significatifs.
Je vous rappelle qu'en C, les minuscules et les majuscules sont traitées différemment dans un identificateur ("toto" est différent de "TOTO" et de "Toto").

Les mots réservés du langage (ceux définissant sa sémantique, c'est à dire les instructions de boucle, de test, de définition de types ...) ne peuvent être utilisés comme identificateurs. La liste des mots réservés est donnée dans le manuel de référence du compilateur; elle varie légèrement entre ORCA/C et APW C. Les symboles spéciaux (par exemple = * { }) sont aussi réservés et ne peuvent donc être utilisés que dans le contexte de la fonction qui leur a été assignée, sauf bien sûr lorsqu'ils sont utilisés dans une chaîne de caractères ou un commentaire. Une liste de ces symboles est aussi donnée dans le manuel de référence du compilateur.

Les noms suivants sont des identificateurs valides en C :

main, array (non ce n'est pas un mot clef), laFenetre, _special,
un_tres_tres_tres_long_identicateur, z12

Les noms suivants ne sont pas des identificateurs C :

45t, laFenêtre, i%

Constantes

Les constantes sont utilisées pour représenter des nombres entiers ou réels et des caractères, soit unitairement, soit sous forme de chaînes. Chacune de ces entités a une valeur préderminée au moment de l'écriture du programme, et cette valeur ne peut pas être modifiée au cours de l'exécution de ce programme.

Constantes entières

Les constantes entières peuvent être représentées sur un mot (qui occupe 2 octets ou 16 bits) ou sur un mot long (4 octets, c'est à dire 32 bits). Elles peuvent être aussi signées ou pas. Une constante entière peut enfin être définie en décimal, en octal ou en hexadécimal.

Nombres décimaux

Un entier sur 16 bits permet de représenter un nombre décimal compris entre -32768 et +32767 si il est signé, ou de 0U à 65535U si il ne l'est pas; remarquez que pour signaler au compilateur qu'il s'agit d'un nombre non signé, on accole un "U" (cela signifie "unsigned" en anglais; on peut aussi utiliser le "u") juste après la valeur du nombre (sans aucun espace entre les 2). Si le nombre est en dehors des limites précédentes ou que l'on lui accole la lettre "L" ou "l" (pour "long"), il occupera 32 bits, ce qui permet de représenter des nombres entre -2147483648L et +2147483647L ou 0UL à 4294967295UL. Dans le cas d'un entier long non signé, les symboles "L" et "U" peuvent être spécifiés dans un ordre quelconque (c'est à dire soit "LU" ou "UL"). L'utilisation de nombres non signés a des implications sur les expressions dans lesquels ils apparaissent; ceci sera traité dans le prochain article.

Nombres octaux

Un nombre octal utilise les chiffres de 0 à 7 au lieu des chiffres de 0 à 9 du système décimal. C distingue ces nombres en les préfixant par le chiffre 0.

Dès qu'un nombre commence par 0 (et que ce n'est pas le nombre 0 bien sûr !), il s'agit d'un nombre octal qui ne peut donc utiliser les chiffres 8 et 9.

Attention, car celà peut parfois créer des confusions et donc donner des résultats pour le moins inattendus ! Comme les nombres décimaux, les nombres octaux peuvent être soit courts ou longs et signés ou non signés.

Les limites sont alors de 0100000 à 077777 pour un entier court signé, de 0 à 0177777U pour un entier court non signé, de 020000000000L à 017777777777L pour un entier long signé et de 0 à 037777777777UL pour un entier long non signé. Notez que le nombre 0 a la même représentation dans toutes les bases.

Nombres hexadécimaux

Un nombre hexadécimal utilise les chiffres de 0 à 9 et les lettres de A à F (ou de a à f) pour représenter les 16 unités de sa base. Les nombres hexadécimaux sont distingués des nombres décimaux et

octaux par le préfixe 0x (ou 0X). Bien entendu, les nombres hexadécimaux peuvent aussi être courts ou longs, signés ou non signés. Les limites sont alors de 0x8000 à 0x7FFF pour les entiers courts signés, de 0 à 0xFFFFu pour les entiers courts non signés, de 0x80000000L à 0x7FFFFFFF pour les entiers longs signés de de 0 à 0xFFFFFFFFuL pour les entiers longs non signés.

Constantes caractères

Les constantes caractères sont formées en entourant le caractère en question entre des apostrophes (comme dans 'a'). Une constante caractère est traitée de la même manière qu'une constante entière dont la valeur est le code ASCII du caractère. De fait, ces 2 types sont interchangeables, ce qui, dans la pratique, permet de définir des variables entières sur un octet (8 bits) et donc d'économiser de la mémoire. D'ailleurs, en C, le type caractère peut être signé, comme nous le verrons plus loin.

Séquences d'échappement

Tous les caractères du code ASCII ne sont pas directement accessibles au clavier; C permet de les représenter sous la forme de séquences dites d'"escape" : une telle séquence commence par le caractère "\" et est suivie soit d'un autre caractère indiquant la fonction de la séquence ou d'un nombre octal ou hexadécimal donnant le code ASCII du caractère (notez que dans ce cas, un nombre hexadécimal commence simplement par "x" et non pas par "0x" tandis qu'un nombre octal n'est pas obligatoirement précédé par un "0"). Si le caractère suivant le "\" ne fait pas partie d'une séquence prédéfinie, le compilateur considère ce caractère comme si il n'avait pas été précédé par le "\" (ainsi par exemple '\z' et 'z' sont strictement identiques).

Les séquences prédéfinies peuvent être catégorisées de la manière suivante :

• Représentation des caractères déjà utilisés dans la définition d'une constante. Il s'agit de "\", de "'" et de """, le guillemet étant utilisé pour délimiter les chaînes de caractères, comme nous allons le voir juste après.

• Représentation de caractères de contrôle : les plus utilisés sont :

"\n" (code ASCII 10,soit Line-Feed) pour newline, c'est à dire le passage au début de la ligne suivante.

"\r" (code ASCII 13, soit Carriage-Return) ; il permet de revenir au début de la ligne courante.

"\t" (code ASCII 9, soit Horizontal-Tab); c'est la tabulation classique.

"\f" (code ASCII 12, soit Form-Feed); c'est le saut de page.

"\b" (code ASCII 8, soit Backspace); permet de revenir un caractère en arrière.

"\0" (code ASCII 0, soit Null); utilisé pour matérialiser la fin d'une chaîne de caractères. Il s'agit d'un cas particulier du "\" suivi d'un nombre octal, mais qui est probablement le plus utilisé.

Il en existe quelques autres que vous trouverez dans la documentation du compilateur.

• La séquence "\p" est spécifique au GS et ne peut de plus s'utiliser qu'en première position d'une chaîne de caractères. Elle demande au compilateur de remplacer l'octet qu'elle occupe par la longueur de la chaîne qui suit, afin de former une p-string utilisée par la ToolBox; cette p-string est limitée à 255 caractères. Dans les autres cas (emploi de cette séquence comme une constante d'un caractère ou ailleurs qu'en première position), la séquence est traitée comme un simple "p".

Chaînes de caractères

Une constante chaîne de caractères est constituée d'une suite quelconque de caractères entourés par des guillements "" excepté newline et le guillemet; ceux-ci étant représentés par des séquences d'échappement telles qu'elles ont été décrites ci-dessus. Une chaîne de caractères occupe une suite continue d'octets en mémoire, à raison d'un octet par caractère, et elle est terminée par un octet à 0; sa longueur peut donc être quelconque sans la limite à 255 caractères comme une p-string.

Attention à ne pas confondre une constante caractère (par exemple 'x') et une constante chaîne de caractères (par exemple "x"). Dans le deuxième cas, la constante occupe 2 octets (un octet pour le x et un octet pour le 0 de terminaison. De plus l'accès à la chaîne de caractères se fait au travers d'un pointeur, comme nous aurons l'occasion de le voir dans un prochain numéro, alors que la constante caractère est directement représentée. Une chaîne vide se réprésente par """", mais elle occupe cependant 1 octet, celui du 0 final; la séquence "" n'est pas valide en C.

Comme la longueur d'une chaîne de caractères peut être quelconque, cette dernière peut utiliser plusieurs lignes dans le programme source. Avant la norme ANSI, il fallait utiliser le caractère "\" suivi immédiatement d'un retour chariot (attention, il ne s'agit pas de la séquence "\n") puis continuer la suite de la chaîne au début de la ligne suivante; cette méthode présente l'inconvénient de casser l'indentation du programme, car si l'on souhaite positionner la suite du texte au même niveau d'indentation que la ligne précédente, les blancs insérés en début de ligne sont pris en compte dans la chaîne de caractères, ce qui n'est certainement pas désiré !

De plus, si un espace a été ajouté juste après le "\", le retour chariot suivi n'est plus traité comme continuation de la chaîne, et la ligne suivante est prise comme une nouvelle instruction. Un exemple valant mieux qu'un long discours, voici comment on utilise ce mécanisme :

```
printf ( "Hello, \
world.\n" );
```

Vous noterez que cela ne fait pas très joli. Heureusement, la norme ANSI définit une nouvelle syntaxe beaucoup plus pratique : il suffit simplement d'accoler 2 constantes chaînes de caractères, chacune entourée de ses "", sans autre chose entre les 2 que des espaces, tabulations et retours chariot, comme :

```
printf ( "Hello, "
         "world.\n" );
```

Ce qui est tout de même nettement mieux ! Cependant, si vous utilisez APW C, il faudra vous contenter de la première méthode. Encore un atout pour ORCA/C ;-)

Nombres réels

La définition d'une constante représentant un nombre réel est similaire aux autres langages, notamment Pascal. Le format général est une séquence de chiffres, éventuellement suivi d'un point décimal puis d'une autre séquence de chiffres, puis d'un exposant matérialisé par le caractère "e" ou "E".

Par exemple : 314.15926e-2. Une constante réelle est stockée sous le format étendu de SANE. C autorise l'utilisation des lettres "f" (ou "F") et "l" (ou "L") accolées après le nombre pour indiquer respectivement que la constante représente un nombre flottant ou double; ces spécifications sont cependant ignorées par ORCA/C qui représente toujours les constantes réelles dans le format étendu de SANE.

Symbolisation des constantes

ORCA/C permet de symboliser (c'est à dire de donner un nom) à une constante et ce de 2 manières différentes : soit en utilisant l'instruction "#define", soit en utilisant "const". Le mot clef "const" fait partie des apports de la norme ANSI, et n'est donc pas disponible avec APW C; cependant, les constantes restent encore généralement définies avec "#define". Nous verrons comment est utilisé le mot clef "const" lorsque nous aborderons les définitions de types et de variables.

Il est fortement recommandé de nommer ses constantes, car cela rend le programme nettement plus compréhensible, y compris à son auteur lorsqu'il doit le reprendre.

Les évolutions sont aussi beaucoup plus simples, puisqu'il suffit de changer la définition d'une constante au lieu de chercher son utilisation partout dans le programme.

Pré-processeur

La commande "#define" n'est pas une instruction du langage C. En fait, elle fait partie d'un ensemble de commandes d'un composant appelé "pré-processeur".

Au début du C, il s'agissait d'un programme séparé qui s'exécutait avant le compilateur; aujourd'hui, il est intégré dans le compilateur, mais son action s'effectue avant le travail de compilation. Dans le cas d'ORCA/C, il permet d'effectuer 4 types d'opérations :

• La définition de constantes et de macros (pour ces dernières, nous verrons ultérieurement de quoi il s'agit) par la commande "#define". Cette commande permet d'associer un nom à une valeur (ou une expression) numérique ou chaîne de caractères. La définition peut elle-même faire référence à d'autres symboles définis de cette manière. Lors de l'exécution du pré-processeur, celui-ci va remplacer toutes les occurences du symbole par la valeur associée. Le compilateur n'a donc jamais connaissance de ces symboles.

De ce fait, s'agissant de substitution textuelle, ces symboles n'ont pas d'existence dans le programme compilé et n'occupent donc pas de mémoire, tout en permettant une utilisation symbolique des valeurs constantes.

En revanche, elles ne sont pas examinables avec un debugger, et ne permettent pas au compilateur de faire des contrôles de type. La syntaxe prend la forme suivante (les crochets représentent des espaces et des tabulations optionnels) :

```
[]#[]define symbole expression
```

Avant la norme, le symbole "#" devait être en colonne 1, et le mot clef définissant l'action du pré-processeur devait lui être accolé. Ce n'est aujourd'hui plus obligatoire, mais cela permet de garder la compatibilité avec des compilateurs plus anciens. Le symbole est un identificateur C défini par les règles précédentes; les conventions veulent qu'il soit en majuscules pour le différencier d'un identificateur normal reconnu par le compilateur, mais ce n'est nullement obligatoire, et c'est une convention rarement respectée sur le GS. L'expression doit suivre les règles de construction d'une expression C, que nous verrons dans le prochain article.

Comme #define permet de faire de la substitution de texte, rien n'empêche de s'en servir pour redéfinir par exemple les mots clefs et les symboles du langage, bien que je vous le déconseille fortement, afin de ne pas créer chacun son propre dialecte C. Ainsi, vous pourriez "pascaliser" le C en redéfinissant certains symboles :

```
#define begin {
#define end }
```

etc ... Mais ce n'est plus du C ! et je ne pense pas que cela améliore vraiment la lisibilité du programme.

Nous verrons dans un prochain numéro une autre forme de la commande #define permettant de définir des macros.

• L'inclusion de fichiers par la commande "#include". Elle permet d'importer les définitions d'un autre module dans un contexte de compilation séparée; par exemple, la toolbox utilise ce mécanisme pour définir les constantes et les fonctions de chacun des toolsets. En théorie, et pour être complet, le programme "hello.cc" du précédent article aurait dû s'écrire :

```
#include <stdio.h>

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

Le fichier stdio.h contient toutes les définitions de la librairie standard d'entrée/sortie du C, notamment la définition du prototype de "printf".

Dans cet exemple, son inclusion n'est pas absolument indispensable, mais elle est souhaitable pour la clarté du programme.

• La compilation conditionnelle. On entend par ce terme la possibilité de compiler une partie du programme en fonction de la valeur booléenne d'une expression ou de la définition préalable (par #define) d'un symbole. Nous traiterons dans un prochain article de ce sujet.

• Les directives de compilation par la commande "#pragma" complétée par un nom de directive et éventuellement d'attributs. L'interprétation de cette commande varie avec chaque compilateur; par exemple ORCA/C utilise (entre autres) cette commande pour permettre au programmeur de lui indiquer le type du programme (normal, NDA, CDA). Pour plus d'informations, je vous renvoie à la documentation d'utilisation du compilateur, s'agissant d'un domaine trop spécifique pour être traité dans ces articles.

Types et variables scalaires

A la lecture de la syntaxe des constantes, vous devez déjà entrevoir les types disponibles. Par type scalaire, on entend tout type de donnée élémentaire (ou ordinal), par opposition aux types composites, comme par exemple les tableaux.

C possède relativement peu de types scalaires, mais pour la plupart d'entre eux, il permet de leur associer un ou plusieurs attributs modifiant soit l'espace que les données de ce type occupent en mémoire, soit le type d'espace qu'elles vont occuper (c'est ce que l'on appelle la "classe de stockage"), ou encore la manière dont la variable est modifiable, ou enfin si elle est signée ou pas.

Comme dans tout langage, C différencie les variables globales (qui sont connues dans l'ensemble du programme) des variables locales (qui ne sont connues que dans la fonction qui les définit). Les variables globales sont donc définies à l'extérieur de toute fonction (y compris la fonction "main" présentée dans le précédent article). En C, les variables globales ne sont pas nécessairement définies avant la première fonction (bien que ce soit généralement le cas pour des raisons de clarté); dans tous les cas, elles ne deviennent connues (et donc référençables) qu'à partir de l'endroit où elles ont été définies (et sont donc inconnues dans les fonctions précédant cette définition). Il a été dit plus haut qu'un programme C pouvait être réparti sur plusieurs fichiers sources. Les variables globales peuvent être globales soit à l'ensemble du programme, soit être restreintes aux fonctions du fichier source dans lequel elles ont été définies; ceci permet d'implémenter une véritable notion de module.

Essentiellement la déclaration d'une variable scalaire se présente de la manière suivante (les zones entre crochets sont optionnelles, sachant que l'une de celles précédent le nom de la variable doit être présente) :

```
[classe-de-stockage] [caractéristiques ...] [taille] [type] nom-variable
[= valeur-initiale] [, variable [= valeur-initiale] ...]
```

J'espère que vous n'êtes pas noyés ! Chacun des éléments précédents a la signification suivante (l'ordre ci-dessous n'est pas celui dans lequel ces éléments doivent apparaître; il a été choisi pour la clarté des explications) :

• Un nom de variable reprend la syntaxe d'un identificateur donnée plus haut; je ne reviendrais donc pas dessus.

• Les types de base sont très peu nombreux puisqu'il n'y a que les entiers, les réels, les pointeurs (que nous traiterons dans un prochain article), et pour les compilateurs à la norme, les énumérations et un type spécial.

Le type entier est défini par le mot clef "int", c'est le type par défaut d'une variable lorsqu'aucun type n'est spécifié (ce qui n'est possible que si un des autres éléments de la déclaration d'une variable a été utilisé, notamment la zone taille). La taille d'une variable de type int dépend de la machine sous-jacente; dans le cas du GS qui est une machine16 bits, un int occupe donc 2 octets. Le champ taille permet de modifier la représentation de la variable : il peut prendre l'une des 2 valeurs "short" ou "long" indiquant respectivement un entier de 2 (16 bits) ou de 4 (32 bits) octets. Vous noterez que dans le cas du GS, le type "int" est identique au type "short int" ou plus simplement "short". Comme le type int peut varier d'une machine à l'autre, et pour éviter de se poser des questions, je vous conseille fortement de toujours déclarer les variables entières en utilisant les types "short" et "long". Par défaut le type int est signé, cependant vous pouvez explicitement déclarer une variable entière signée ou non signée en préfixant son type par l'un des 2 mots clefs "signed" et "unsigned". Sachez qu'il est souhaitable de déclarer les variables non signées à chaque fois que c'est possible, car le code généré par ORCA/C est

plus efficace (du moins c'est ce que dit sa documentation).

Vous avez sans doute remarqué que je n'ai pas cité le type caractère comme type de base; en fait, il ne s'agit que d'une variante du type int, mais qui se déclare cependant avec le mot clef "char" qui peut malgré tout stocker n'importe quelle valeur entière sur 1 octet (soit de -128 à +127 si la variable est signée et de 0 à 255 si elle ne l'est pas).

Contrairement aux autres types entiers, les variables déclarées de type char sont non signées par défaut depuis ORCA/C 1.2 (elles étaient signées dans les versions antérieures), mais cela peut être spécifié explicitement avec les mots clefs "signed" et "unsigned" (notez que la plupart des autres compilateurs ont un type char signé par défaut; il peut donc être souhaitable de toujours indiquer le signe lorsqu'on utilise une variable de type char pour y stocker des entiers).

Vous avez aussi sans aucun doute noté que je n'ai pas parlé de type booléen, ni décrit de constante TRUE et FALSE. La raison en est que ce type n'existe pas en C. Au contraire, toute variable entière peut être utilisée comme booléen, puisque C définit 0 comme FALSE, et toute autre valeur comme TRUE. C'est pourquoi il est contraire à l'esprit de C de comparer explicitement un résultat booléen à une éventuelle constante TRUE définie à 1, car rien ne garantit qu'une fonction va retourner une telle valeur.

En revanche, l'évaluation d'une expression booléenne donne toujours un résultat de 1 si elle est vraie et de 0 dans le cas contraire, ce qui permet de l'utiliser dans une expression arithmétique (comme au bon vieux temps de l'AppleSoft ;-).

Il existe enfin un type entier un peu spécial, appelé "comp" et qui occupe 8 octets. Ce type est défini par SANE et est donc spécifique au GS (et accessoirement au Mac). Le compilateur ORCA/C les traite d'ailleurs dans la même catégorie que les nombres réels, puisqu'il doit passer par SANE pour les manipuler.

Les variables réelles sont déclarées grâce à 3 mots clefs représentant chacun une des tailles et précisions possibles; il s'agit de "float" occupant 4 octets, de "double" nécessitant 8 octets (qui peut aussi être précédé par "long", sans que cela ne change quoi que soit) et de "extended" qui requiert 10 octets. Je vous renvoie à la doc du compilateur pour plus d'informations sur les variables réelles.

ORCA/C (et les autres compilateurs à la norme ANSI) définissent un type spécial (que nous avons déjà abordé dans le premier article) appelé "void" (vide en français). Ce type ne s'applique qu'à la déclaration des fonctions, ou pour définir un pointeur générique; nous en parlerons donc plus en détail lorsque nous aborderons ces sujets.

Le type énuméré est presque équivalent à son homologue Pascalien, avec cependant beaucoup plus de libertés, puisqu'il n'y a aucun contrôle.

Une variable de ce type est déclarée avec le mot clef "enum". Ce type permet de définir une liste de constantes entières et dont la variable de ce type peut prendre l'une d'entre elles à un instant donné. Ce type est identique à la définition de constantes par #define, la première constante de la liste prenant la valeur 0, et les suivantes les nombres successifs; il est aussi possible d'affecter une valeur donnée à un des symboles, ceci réinitialisant le compteur à cette valeur pour les symboles suivants. Il est possible d'utiliser des valeurs négatives ou plusieurs fois la même valeur pour des symboles différents. A l'exécution, les variables énumérées peuvent être librement mélangées avec d'autres variables entières, et on peut même leur affecter des valeurs en dehors de celles définies dans la déclaration.

- Il est possible d'associer une valeur initiale à une variable lors de sa déclaration.

Cette valeur pourra être modifiée lors de l'exécution de la fonction, sauf dans le cas où il s'agit d'une constante, auquel cas la valeur initiale devra être impérativement spécifiée.

- La classe de stockage (si elle est présente) peut être l'un des mots clefs suivants :

"auto" (pour automatique); cette classe s'applique uniquement aux variables locales

d'une fonction, dont c'est d'ailleurs la classe par défaut, si aucun mot clef n'est spécifié. Cette classe signifie simplement que la mémoire requise pour le stockage de la variable doit être allouée au début de la fonction (ORCA/C réserve de la place sur la pile pour ces variables) puis libérée lorsque cette dernière se termine; par conséquent, la valeur de ces variables est perdue d'un appel à l'autre de la fonction.

"register"; cette classe s'applique aussi uniquement aux variables locales. Son but est d'indiquer au compilateur d'utiliser de préférence un registre du microprocesseur pour stocker la variable associée, en général parce qu'elle va être utilisée très fréquemment. Comme le 65816 ne dispose que de très peu de registres, cette classe est ignorée et traitée de la même manière que "auto". Dans un contexte plus général, les compilateurs C actuels sont optimisants, c'est à dire qu'ils sont capables de déterminer automatiquement quelles sont les variables à stocker dans les registres disponibles, sans qu'il soit nécessaire de leur indiquer; cette classe reste donc à titre de compatibilité historique.

"extern"; cette classe signifie simplement que la variable ne doit pas être définie ici, mais qu'elle l'est quelque part ailleurs dans le programme, et que l'on souhaite y faire référence. Cette classe de stockage ne peut bien entendu s'appliquer qu'aux variables globales, puisqu'elle permet de référencer des variables qui survivent à l'exécution d'une fonction. C'est d'ailleurs la classe de stockage par défaut de ces variables.

Cependant, la présence ou l'absence du mot clef a une signification différente : si il est présent, on référence une variable globale, tandis que si il est absent, on définit cette variable, ce qui n'est possible qu'une seule fois pour une variable donnée pour un même programme. Si une variable est déclarée "extern" dans une fonction, cette variable globale ne peut être référencée que dans cette fonction, tandis que si elle est déclarée à l'extérieur de toute fonction, elle est accessible pour toutes les fonctions suivant sa déclaration. Cette classe peut aussi s'appliquer à la déclaration d'une fonction, avec les mêmes règles que pour les variables globales.

"static"; cette classe a une signification différente, selon qu'elle s'applique à une variable locale ou globale. Dans le cas d'une variable locale, la classe static indique au compilateur que la valeur de la variable doit être préservée entre les appels à la fonction dans laquelle elle a été déclarée. Dans le cas d'une variable globale ou d'une fonction, la classe static restreint la globalité au fichier source dans lequel cette variable ou cette fonction a été définie, c'est à dire qu'une variable ou une fonction définie ainsi reste accessible globalement à l'intérieur du même module (le même fichier source) mais est invisible au linker, et par conséquent aux autres modules (fichiers sources) du programme. Cela permet de réaliser des librairies en garantissant que les variables globales internes de la librarie ne seront pas partagées avec les programmes utilisant cette librairie.

"typedef"; bien que catégorisée comme telle, typedef n'est pas à proprement parler une classe de stockage. En effet, il sert à définir de nouveaux types. Cependant, la syntaxe est identique à celle décrite précédemment, sauf que la liste des variables devient une liste de nouveaux noms de types, et que bien entendu, la zone "valeur initiale" n'a pas de sens. Typedef peut être considéré comme l'équivalent C de la commande #define du préprocesseur définie plus haut, en ce sens qu'elle permet de définir un synonyme à un type scalaire ou composite, éventuellement modifié par certains des attributs disponibles, et non pas comme la déclaration de type du Pascal.

• ORCA/C permet d'associer de façon optionnelle l'une des 2 caractéristiques suivantes à la déclaration d'une variable (celles-ci sont de par leur fonction mutuellement exclusives, bien que l'on puisse trouver les 2 dans la même déclaration, mais elles s'appliquent à des éléments différents — voir l'exemple ci-après) :

Le mot "const" permet d'indiquer au compilateur que la variable est en fait une constante, et qu'elle ne sera donc pas modifiée par l'exécution de la fonction (ce que le compilateur garantit en générant une erreur si ce n'était pas le cas). Une valeur initiale doit être impérativement donnée lors de la déclaration d'une constante.

Le mot "volatile" indique au compilateur que la variable ainsi désignée peut être modifiée en dehors du code dans lequel elle est définie, et donc de façon non identifiable par le compilateur. Ceci est utile si la variable est modifiée par exemple par une routine d'interruption ou s'il s'agit d'un pointeur sur l'un des softswitches comme le clavier. Le compilateur utilise cette information pour ne pas faire d'optimisations qui seraient contraires à l'effet désiré.

Les compilateurs C considèrent ces attributs comme des spécifications de type, si bien que l'ordre dans lequel ils apparaissent par rapport au type lui-même n'a pas d'importance (sauf dans le cas où ils sont appliqués à un pointeur sur une variable et non pas à la variable pointée), et qu'il est même possible de ne spécifier aucun type, auquel cas le type int est pris par défaut.

Cependant, je vous recommande d'indiquer ces attributs avant le type de la variable et de toujours spécifier ce dernier.

Maintenant que nous avons fait le tour de la déclaration d'une variable, voyons avant de terminer cet article quelques exemples concrets :

```
int  i;
/* i est une variable entière sur 16 bits */
short int i;                        /* la même chose, mais il est préférable de le préciser */
short  i;
/* toujours pareil, mais c'est la forme que je préfère */
unsigned i;                         /* entier court non signé */
unsigned short i;
/* je trouve que cela est plus cohérent */
long int j;                         /* j est une variable entière sur 32 bits */
long  j;
/* c'est la même chose, mais je trouve plus lisible */
unsigned long k = 10;               /* entier long non signé et initialisé à la valeur 10 */
char c;
/* un caractère */
signed char l = -3;                 /* un entier négatif stocké sur un octet valant -3 */
extern short z;
/* z est défini ailleurs, mais on en a besoin */
static long t;
/* si on est dans une fonction, la valeur de t doit être */
/* préservée jusqu'au prochain appel */
/* si t est une variable globale, ne pas l'exporter */
float f;
/* pourquoi pas un nombre réel ? */
double d,e;                         /* et 2 doubles */
extended ex;                        /* et un étendu */
const extended pi = 3.141592653;    /* pi est déclaré comme constante */
typedef int entier;                 /* définition du type entier */
entier var;                         /* et déclaration d'une variable de ce type */
enum { rouge, vert, bleu } couleur; /* définition d'une variable énumérée */
enum couleur { rouge, vert, bleu }; /* ici il s'agit d'une déclaration de type */
enum couleur col;
          /* et d'une variable de ce type */
enum couleur couleur;
/* on peut utiliser le même nom */
typedef enum { false, true } boolean; /* définition d'un type booléen équivalent à celui de Pascal */
typedef enum { false = 0, true = 1 } boolean; /* la même chose en forçant les valeurs de façon
                                    compatible avec le C (ce sont aussi les valeurs normales)*/
volatile char * const kb = 0x00C000; /* déclaration d'un pointeur constant sur une zone de 1
```

caractère qui est volatile (vous avez reconnu l'adresse du clavier) */

Et voilà, vous savez quasiment tout sur la déclaration des variables scalaires en C. Comme le montre le dernier exemple, cela peut parfois être complexe (nous aurons l'occasion de voir des déclarations pour lesquelles il faut son brevet lorsque nous traiterons des pointeurs; en fait je plaisante, car il existe des règles d'écriture et donc de lecture assez simples, et l'on peut encore améliorer la lisibilité en utilisant des typedef sur les types intermédiaires; nous verrons tous çà dans le chapitre traitant des pointeurs).

Dans le prochain article, nous aborderons la syntaxe des expressions permettant d'exploiter ces variables, que nous illustrerons par un petit programme.

Forum C

Dans cette section, nous traiterons des points particuliers que vous aurez soulevés dans vos courriers.

Les premières lettres que j'ai reçues me demandent de décrire l'environnement APW/ORCA et comment compiler et linker un programme. Je n'ai pas eu le temps d'aborder ce sujet pour cet article, mais dans le prochain numéro de GSInfos (celui d'après les vacances), vous trouverez un article répondant à vos demandes, en complément de celui sur les expressions. Continuez donc à m'écrire !

Chapitre 3

==========

Résumé des épisodes précédents

==========

Dans les chapitres précédents, nous avons vu les types de base disponibles dans le langage C, ainsi que la manière de déclarer des variables en leur donnant un de ces types, et enfin la syntaxe des différentes constantes et le moyen de leur donner un nom symbolique à l'aide du pré-processeur.

Dans cet article, nous allons traiter des opérateurs disponibles en C. Ces opérateurs sont utilisés dans la construction des expressions. Ils sont principalement arithmétiques (ils s'appliquent donc à des données numériques, entières ou réelles) et relationnels (ils servent à effectuer des comparaisons entre des types scalaires).

C est caractérisé par la grande richesse de ses opérateurs. Ceux-ci couvrent en effet toute l'étendue des instructions de calcul offertes par la plupart des microprocesseurs, voire même plus. La difficulté qu'éprouve le débutant vient souvent du fait que chacun est représenté par un symbole spécial (ou une séquence de symboles), et non pas par un mot clef. De plus, la précédence (priorité) affectée à ces opérateurs impose souvent l'utilisation des parenthèses pour obtenir le résultat désiré, ce qui ajoute à la complexité que l'on peut ressentir.

Les détracteurs de C (les pascaliens en général) disent que le C est illisible, entre autre à cause des expressions utilisant ces opérateurs. Je pense qu'il s'agit plutôt de jalousie face à des possibilités qu'ils n'ont fait que rêver. Par exemple, tous les opérateurs de manipulation de bits n'existent pas en Pascal standard. Les 2 Pascals disponibles sur GS les ont implémentés en tant qu'extension, mais chacun de façon différente : ORCA/Pascal utilise les mêmes opérateurs que C, tandis que TML Pascal emploie des fonctions. Je ne parle même pas des opérateurs d'incrémentation/décrémentation ou d'affectation composée qui n'existent pas tout simplement.

Opérateurs arithmétiques

==========

Il s'agit bien entendu des opérateurs permettant d'effectuer les opérations classiques que l'on retrouve dans tous les langages, à savoir l'addition, la soustraction, la multiplication et la division avec respectivement les symboles "+", "-", "*" et "/". Contrairement au Pascal, C ne dispose que d'un seul opérateur de division qui donne soit un résultat réel, soit un résultat entier en fonction des opérandes (il vous faudra faire une conversion de type explicite pour faire une division flottante avec des nombres entiers). C possède aussi l'opérateur modulo permettant d'obtenir le reste d'une division entière, à travers le symbole "%" (cet opérateur ne peut donc s'appliquer qu'à des opérandes de type entier).

Les opérateurs "+" et "-" sont soit binaires (c'est à dire entre 2 opérandes, afin d'effectuer une addition et une soustraction), soit unaires (c'est à dire immédiatement devant une constante, de façon à déterminer son signe).

La notion de nombre négatif n'existe pas pour un compilateur (que ce soit C ou un autre langage). Par conséquent la présence du signe "-" et d'un nombre correspond à 2 entités : cela revient en fait à soustraire le nombre de 0.

Il en est de même pour l'opérateur unaire "+", excepté que cela ne génère pas de code (c'est un NOP); on a donc parfaitement le droit d'écrire l'expression : a + +2, le premier "+" est l'opérateur binaire, tandis que le second est unaire. Notez que le + unaire n'existe pas dans tous les langages; en l'occurence, il n'existait pas dans la définition originale de C.

Les règles de précédence habituelles sont respectées, à savoir que "+" et "-" ont la même priorité, mais qu'elle est inférieure à celle de "*", "/" et "%" (qui est identique pour ces 3 opérateurs). Ainsi l'expression 3+4*5 vaudra 23 et non pas 35. Comme dans les autres langages, C permet de regrouper des opérations entre des parenthèses ("(" ")") afin de forcer l'ordre d'évaluation; dans l'exemple précédent, il aurait fallu écrire (3+4)*5 pour obtenir 35.

Les opérandes sont automatiquement convertis de façon à avoir le même type, avant que l'opération soit effectuée. Par exemple, si un opérande est entier et l'autre flottant, le nombre entier sera converti en flottant avant l'opération et éventuellement reconverti dans le type de la variable dans laquelle sera stocké le résultat.

Si une soustraction est effectuée avec des nombres non signés, et que le premier est inférieur au second, le résultat sera la valeur non signée correspondant au nombre signé généré par le 65816, c'est à dire qu'on lui aura virtuellement ajouté 2 puissance la taille utilisée (8, 16 ou 32).
Par exemple, la soustraction non signée sur 16 bits 1 - 2 aura pour résultat -1 + 65536, soit 65535.

Les débordements de capacité d'un calcul entier (soit signé, soit non signé) donneront lieu à un résultat qui sera modulo du nombre maximal représentable dans la taille utilisée; en d'autres termes, les bits les plus significatifs seront perdus. Pour les nombres flottants, cela dependra du paramétrage de SANE qu'il faudra faire par des appels directs.

Opérateurs d'affectation

En C, l'opérateur d'affectation (permettant de stocker le résultat d'un calcul dans une variable) est le symbole "=", contrairement à Pascal où il sert à tester l'égalité entre 2 variables ou expressions. La raison en est que dans un programme normal, on effectue bien plus souvent une affectation à une variable qu'une comparaison; ceci permet donc d'économiser de la saisie lors de l'édition du programme.

Cet opérateur peut être utilisé avec tous les opérateurs arithmétiques précédents. Le résultat du calcul est éventuellement converti dans le type de la "valeur-g" de l'affectation.

Notion de "valeur-g" et de "valeur-d"

La notion de "valeur-g" (qui signifie "valeur-gauche"; notez que vous lirez plus souvent le nom anglais de "l-value" pour "left-value") et de "valeur-d" (vous aurez deviné qu'il s'agit de "valeur-droite" ou de "r-value") existe dans tous les langages. Elle prend cependant une importance toute particulière en C, du fait de ses opérateurs.
Ces 2 valeurs correspondent simplement à leurs positions respectives par rapport au signe "=" de l'affectation. La notion de "valeur-g" est la plus importante des 2, car elle conditionne l'utilisation d'une expression en tant que membre gauche d'une affectation. Toute expression légale en C est une "valeur-d".
Une "valeur-g" est en fait une expression qui réfère à l'adresse d'un objet.
Par essence, toute variable simple est une "valeur-g", alors qu'une constante ou une expression arithmétique ne l'est pas : on ne peut pas écrire a+b = ...
(tous les opérateurs décrits précédemment ne peuvent jamais donner une "valeur-g").

Un autre exemple est celui du tableau : un élément d'un tableau est une "valeur-g" alors que le tableau pris comme un tout ne l'est pas (on ne peut rien affecter à un tableau), il peut cependant être une "valeur-r" dans certains cas.

C possède des opérateurs qui permettent d'obtenir une "valeur-g" comme résultat de l'évaluation d'une expression. Ces expressions correspondent en général à la manipulation des pointeurs, et seront donc traitées en détail dans un prochain numéro.

Affectations multiples

En C, le résultat d'une affectation peut lui-même est affecté à une autre "valeur-g"; il peut donc être aussi considéré comme une "valeur-r". Supposez que vous voulez initialiser les 3 variables a,b et c à 0. En Pascal, vous devriez écrire :

```
a := 0;
b := 0;
c := 0;
```

En C, vous écrirez plus simplement : a = b = c = 0;

L'affectation a une associativité droite-gauche (au lieu de gauche-droite pour les opérateurs arithmétiques précédents); ceci signifie que l'opération la plus à droite est évaluée en premier (ce qui paraît assez logique).

Une affectation intermédiaire peut aussi faire intervenir un calcul : par exemple, a = (b = c + d) * e; Dans cette expression, l'addition de c et d est stockée dans b, le tout est multiplié par e et stocké dans a. Il est clair que ce genre de possibilité peut conduire à l'illisibilité du programme quand elle est utilisée systématiquement et n'importe comment. Elle a néanmoins son utilité, notamment dans les macros.

Affectations composées

Vous avez peut être remarqué que j'avais écris comme titre de cette section 'opérateurs d'affectation' au pluriel. En effet, en plus de l'opérateur d'affectation simple que l'on vient de voir, C permet de composer une affectation et une opération arithmétique dans un même opérateur (note : cette composition s'applique aussi aux opérateurs de manipulation de bits que nous verrons plus loin). L'effet est d'appliquer l'opération spécifiée entre la "valeur-g" et la "valeur-r", le résultat étant aussi une "valeur-r". Ces opérateurs sont "+=", "-=", "*=", "/=" et "%=". Vous aurez par exemple i += 5; ou i -= j * 7 / k;

Ce qui signifie respectivement ajouter 5 à i et soustraire j*7/k à i.

A priori, vous devez penser que cela ne sert pas à grand chose et qu'on aurait pu tout aussi bien écrire i = i + 5; et i = i - j * 7 / k;. Ce n'est pas exact, car dans la première forme l'adresse de i n'est évaluée qu'une seule fois, alors que dans la deuxième, elle l'est 2 fois (certains compilateurs très sophistiqués peuvent optimiser suffisamment pour supprimer la 2ème évaluation, mais ce n'est certainement pas le cas sur le GS). Dans ce cas précis, l'adresse de i peut être obtenue rapidement, mais supposez que vous vouliez accéder à un élément d'un tableau et que vous ayiez une fonction permettant de calculer l'indice voulu; il est clair que c'est nettement plus performant de ne le faire qu'une seule fois.

Concernant la lisibilité, je trouve qu'on voit ainsi clairement ce que l'on fait avec le membre gauche, ce qui n'est forcément le cas lorsqu'il est perdu au milieu d'une expression.

Incrémentation et décrémentation

C possède 2 opérateurs permettant d'incrémenter de 1 ("++") ou de décrémenter (de 1 aussi) ("--") le contenu d'une variable, soit avant (pré-incrémentation/décrémentation), soit après (post-incrémentation/décrémentation) l'évaluation de cette variable; c'est à dire que ces opérateurs peuvent être placés avant ou après la variable (ils ne peuvent donc s'appliquer qu'à des valeurs-g), par exemple ++i; ou j--;. Dans le premier cas, i est incrémenté et c'est cette valeur qui sera rendue, tandis

que dans le deuxième cas, j donnera sa valeur, puis sera décrémenté. Cela ne prend son sens que si i et j sont utilisés dans une expression plus complexe ou que leur valeur est affectée à une autre variable (sinon les 2 opérations sont identiques). Par exemple avec k = i++; k prendra la valeur de i qui sera ensuite incrémenté. Avec l = --j; j sera d'abord décrémenté et le résultat sera affecté à l.

Les pascaliens m'objecteront que l'on aurait pu écrire tout aussi bien :

```
k = i;
i = i + 1;
j = j - 1;
l = j;
```

pour arriver au même résultat. En dehors du fait que c'est nettement plus fastidieux à écrire (pour ne pas dire lourd), le code généré n'est pas exactement le même, du fait que la plupart des microprocesseurs ont des instructions d'incrémentation et de décrémentation, et qu'il faut déjà un bon optimiseur pour reconnaître cette opération et utiliser ces instructions. Ainsi, on aurait dans cet exemple :

Dans le premier cas :

```
lda i
inc i
sta k
dec j
lda j
sta l
```

Dans le second cas :

```
lda i
sta k
lda i
clc
adc #1
sta i
lda j
sec
sbc #1
sta j
lda j
sta l
```

Dans un cas aussi simple, l'optimiseur de Byte Works est capable de générer le code du premier cas, en utilisant la séquence du deuxième. Rien ne garantit cependant que cela sera le cas dans des situations beaucoup plus complexes.

Enfin, je trouve qu'une fois qu'on en a pris l'habitude, ces opérateurs n'entravent en rien la lisibilité du programme, bien au contraire.

Opérateurs relationnels

Les opérateurs relationnels se classent dans 2 catégories : les opérateurs permettant d'effectuer des comparaisons et les opérateurs logiques.

Opérations de comparaison

C possède les 6 opérations de comparaison classique, à savoir :

==	égal
!=	différent
<=	inférieur ou égal
<	inférieur
>=	supérieur ou égal
>	supérieur

C ne possédant pas de type booléen, une opération de comparaison donne un résultat numérique qui est 1 lorsque la comparaison est positive et 0 lorsqu'elle est négative. Cette valeur (qui n'est pas bien entendu une valeur-g) peut être utilisée dans une expression arithmétique, comme au bon vieux temps de l'AppleSoft. Par exemple, i = j + (k < l);

Il faut faire attention entre l'opérateur d'affectation "=" et l'opérateur de comparaison "==", car les 2 ont une valeur et peuvent donc être utilisés dans les mêmes conditions. Par exemple, avec l'instruction de test "if" (que nous verrons en détail dans le prochain numéro),
les 2 syntaxes "if (a = b)" et "if (a == b)" sont légales bien que n'ayant pas du tout le même effet; dans la première b est affecté à a et la condition est vraie si b est non nul, tandis que dans le deuxième, la condition n'est vraie que si a et b sont égaux. Cette ambiguïté est à l'origine de beaucoup de bugs, même chez les pros, car on n'est pas à l'abri d'une faute de frappe.

Opérateurs logiques

C possède les 3 opérateurs logiques classiques and "&&", or "||" et not "!".

Le point le plus important les concernant est que && et || sont toujours évalués de gauche à droite et que l'évaluation s'arrête dès que la vérité de l'expression est déterminée. Par exemple, si on a "a < b && c > d", la deuxième expression ne sera pas évaluée si la première est fausse; si l'opérateur || avait été utilisé, la deuxième expression n'aurait pas été évaluée si la première avait été vraie.
Notez que cela a l'air normal, mais ce n'est pas nécessairement vrai dans tous les langages. Par exemple, la norme Pascal indique que toutes les expressions doivent être évaluées lorsqu'on n'utilise pas l'option d'optimisation afin d'exercer les différents cas de figure. Dans le cas d'ORCA/Pascal, l'évaluation de toutes les expressions est effectuée systématiquement. Dans l'exemple précédent, cela n'a pas d'importance, mais dès que vous faites intervenir des pointeurs, vous devez complexifier votre programme inutilement, et on se demande alors lequel est le moins lisible ;-)
Par exemple en C, vous pouvez écrire en toute confiance "if (ptr != NULL && *ptr ...)"
En Pascal, il vous faudra écrire :
IF ptr <> NIL THEN
IF ptr^ ... THEN
si vous ne voulez pas vous exposer à un plantage dans le cas ou le pointeur est vraiment NIL. La future norme "Pascal étendu" en cours d'établissement prévoie les 2 opérateurs AND_THEN et OR_ELSE pour forcer l'arrêt de l'évaluation; on pourra alors écrire : IF PTR <> NIL AND_THEN ptr^ ... THEN.

L'opérateur "!" est la négation. Il convertit donc son opérande soit de 0 en non-zéro, soit le contraire. On l'utilise en général de la manière suivante :
if (!variable)
à la place de
if (variable == 0)

ce qui est pratique et clair lorsque la variable représente un booléen. Dans d'autres cas, il peut être souhaitable de faire une comparaison explicite pour améliorer la lisibilité du programme.

Expression conditionnelle

C possède un opérateur très particulier; cet opérateur est ternaire, c'est à dire qu'il requiert 3 opérandes, alors que les autres opérateurs sont soit binaires (ils travaillent sur 2 opérandes) soit unaires (ils ne travaillent qu'avec un seul opérande). Il est utilisé pour exécuter une instruction de façon conditionnelle.

Prenons par exemple le cas du calcul du maximum de 2 nombres. Classiquement, on écrirait :

```
if ( a > b )
                                        c = a;
else
                                        c = b;
```

En C, on peut aussi utiliser l'opérateur "?:" pour parvenir au même résultat.

On écrirait alors : c = (a > b) ? a : b; (les parenthèses ne sont là que pour améliorer la lisibilité et ne sont pas nécessaires, car la précédence de cet opérateur est très faible).

En généralisant, on obtient "e1 ? e2 : e3"; l'expression e1 est d'abord évaluée, si elle vraie (non nulle), e2 est évaluée et donne le résultat de l'opération, sinon c'est e3 qui est utilisée. Il faut bien voir qu'une seule des 2 expressions e2 et e3 est évaluée. e1 est une expression quelconque, donc pas nécessairement une comparaison. e2 et e3 doivent en général être de types compatibles (une conversion de type pouvant être effectuée si nécessaire), puisque leur résultat sera utilisé identiquement.

Manipulations de bits

C possède un jeu très complet et très puissant d'opérateurs de manipulation de bits; ces opérateurs peuvent être utilisés avec tous les types entiers. Ce sont :

&	et
|	ou inclusif
^	ou exclusif
~	complément à 1
<<	décalage à gauche
>>	décalage à droite

Les opérateurs &, | et ^ effectuent l'opération correspondante bit à bit entre leurs 2 opérandes (éventuellement après conversion). L'opérateur ~ inverse tous les bits de son opérande. Les opérateurs de manipulation de bits &, | et ~ ne doivent pas être confondus avec les opérateurs logiques &&, || et ! décrits précédemment.

Par exemple 1 & 2 vaut 0, tandis que 1 && 2 vaut 1 (je vous laisse chercher pourquoi).

L'opérande de gauche des opérateurs << et >> est décalé respectivement à gauche et à droite du nombre de bits indiqué par l'opérande de droite. Par exemple x << 2 décale x à gauche de 2 bits et les remplace par des 0 (c'est exactement la même chose que si j'avais écrit x * 4, excepté que cela ira nettement plus vite, étant donné que le 65816 a un opérateur de décalage et pas de multiplication).

Pour le décalage à droite, les bits décalés sont soit remplacés par des 0 si le nombre est non signé, soit par la valeur du bit de signe pour un nombre signé.

Un décalage de 0 bit est un NOP, tandis qu'un décalage d'un nombre négatif de bits est à prohiber, car le résultat est imprévisible.

Un exemple de la puissance de C (pris dans K&R) : supposez que vous vouliez extraire n bits du nombre x (déclaré en unsigned int) à partir de p, sans connaître à l'avance la taille de int (qui peut être 16 ou 32 bits selon la machine), et le tout ramené à droite. Vous pouvez écrire : (x >> (p + 1 - n)) & ~(~0 << n)
La partie "x >> (p + 1 - n)" déplace le champ désiré à droite de l'entier (on est sur que les bits décalés seront remplacés par des 0 en ayant déclaré x en unsigned).
~0 génère un nombre ne contenant que des 1 (soit sur 16 soit sur 32 bits);
~0 << n crée un masque contenant des 0 sur les n dernières positions et des 1 sur les autres; le complément à 1 inverse tout, soit un masque avec des 1 sur les n dernières positions et des 0 sur le reste; le & extrait enfin ces bits du nombre décalé précédemment.

Tous les opérateurs binaires peuvent aussi être utilisés dans une affectation composée, sous la forme : &=, |=, ^=, <<= et >>=. Par exemple c &= 0x7F permet de supprimer le bit de poids fort du caractère c.

Conversions de types

Lors de l'évaluation d'une expression, le type de chacun des opérandes peut être converti implicitement par le compilateur, soit pour les rendre compatibles entre eux, soit pour les rendres compatibles avec le type de la valeur-g qui recevra le résultat de l'affectation. Ces conversions sont décrites en détail dans le manuel du compilateur, et n'ont que peu d'intérêt pour une initiation.

Vous pouvez aussi vouloir forcer la conversion de type de façon explicite; ceci se fait en indiquant le nom du type entre parenthèses "(type)" juste devant l'expression à convertir. Cet opérateur s'appelle un "cast". Par exemple (long) (i + 2) donnera un résultat sur un long plutôt qu'un short si tel était le type de i. Bien entendu, le tout sera reconverti en short, si le résultat est stocké dans une variable de ce type. Ce type de conversion explicite peut être utile lorsqu'un résultat intermédiaire peut dépasser la capacité d'un short, et que le résultat final tient dans un short, de façon à éviter une troncation du résultat intermédiaire.

Evaluation séquentielle

C dispose d'un opérateur (la ",") permettant de forcer l'évaluation séquentielle de plusieurs expressions. Le résultat de cet opérateur est celui de la dernière expression évaluée. Il permet d'écrire par exemple :
a = (b = c + d, e = b + f) + g; auquel cas a vaudra e + g.

En fait, cette utilisation reste marginale, et est même à décourager, car elle n'apporte pas grand chose, si ce n'est du pain aux détracteurs de C (il faut avouer que l'expression précédente n'est pas très lisible). En revanche, cet opérateur est très utile (et très utilisé) avec l'instruction for, comme nous le verrons dans 2 mois.

Il ne faut pas confondre l'opérateur "," avec le séparateur "," tel qu'il est utilisé par exemple dans la déclaration de plusieurs variables de même type.

Précédence des opérateurs

Chacun des opérateurs que nous venons de voir est affecté d'une précédence permettant de déterminer l'ordre dans lequel une expression les contenant sera évaluée.

Le tableau ci-dessous résume cet ordre :

Opérateur	Associativité()	
! ~ ++ -- -	unaire +unaire (type)	droite-gauche
* / %		gauche-droite
+ -		gauche-droite
<< >>		gauche-droite
< <= > >=		gauche-droite
== !=		gauche-droite
&		gauche-droite
^		gauche-droite
|		gauche-droite
&&		gauche-droite
||		gauche-droite
?:		droite-gauche
= += -= *= /= %= <<= >>= &= ^= |=		droite-gauche
,		gauche-droite

Note : cette table est incomplète, car nous n'avons pas encore vu les opérateurs de manipulation de pointeurs.

Les opérateurs d'une même ligne ont une précédence identique et sont évalués dans l'ordre indiqué dans la colonne associativité. Les opérateurs d'une ligne ont une précédence supérieure à ceux de la ligne inférieure. Les parenthèses (qui ont la priorité la plus élevée) permettent de changer l'ordre d'évaluation.

Conclusion

Nous avons maintenant étudié les bases du langage C, et nous pouvons désormais mettre en œuvre nos connaissances dans un petit programme. C'est ce que je vous propose dans le programme "nombres" que vous trouverez sur la disquette GS Infos.

Ce programme a pour but de convertir un nombre dans sa forme textuelle (par exemple si vous saisissez 123, le programme vous répondra cent vingt-trois).

Même pour un programme aussi simple, il a été nécessaire d'utiliser des éléments du langage que nous n'avons pas encore vu. C'est pourquoi, je vous ai aussi mis le même programme écrit avec ORCA/Pascal (je vous laisse faire les adaptations éventuelles pour TML Pascal), afin que vous puissiez vous y retrouver (et comparer par la même occasion entre les 2 langages !).

La prochaine fois, nous continuerons notre croisière à bord de C, en abordant les structures de contrôle du langage.

chapitre 4

==========

Les instructions de contrôle

Dans cet article, nous allons aborder les éléments du langage qui permettent de commencer à construire des programmes. Vous en avez déjà découvert quelques uns dans le programme "Nombres" que je vous ai proposé la dernière fois. Aujourd'hui, nous allons voir ce que j'ai appelé plus haut les "instructions de contrôle". Avec ces éléments, vous devriez pouvoir écrire vos premiers programmes C !

Le langage C, qui a été défini dans les années 70, a, comme Pascal, bénéficié des principes de la programmation structurée, et offre donc toute la panoplie classique des instructions permettant de réaliser :

- la séquence
- l'alternative
- l'itération

Notions d'expression et d'instruction

==========

On dit que C est un langage d'"expressions" (par opposition à Pascal qui est un langage d'"instructions"). Je vous rappelle qu'une expression en C est construite à partir des opérateurs qui ont été décrits dans le précédent GS Infos (ainsi que de quelques autres que nous traiterons ultérieurement).

Cette terminologie se réfère à l'élément de base du langage qui est l'expression pour le C, et l'instruction pour le Pascal (notez que Pascal est juste un exemple; pratiquement tous les langages de haut niveau sont des langages d'instructions).

Une expression devient une instruction lorsqu'on la termine par le caractère ";".
Si cette expression ne correspond pas à une affectation, le résultat de son évaluation est perdu. Par exemple l'expression "i++" a 2 effets : le premier est d'obtenir la valeur actuelle de 'i', et le second d'incrémenter 'i'.
En transformant cette expression en l'instruction "i++;", la valeur de 'i' est aussi obtenue, mais comme elle n'est pas utilisée ensuite, elle est donc écartée.

C'est là qu'est la différence fondamentale entre un langage d'expressions et un langage d'instructions (donc entre C et Pascal) : dans le premier, le ";" termine l'expression et en fait une instruction, tandis que dans le deuxième le ";" sépare 2 instructions (c'est par exemple pour cela qu'il ne faut pas en mettre devant le ELSE, puisqu'il fait partie de la même instruction que le IF correspondant). Notez bien la différence entre les 2 verbes "terminer" et "séparer".

Ainsi, une instruction C peut comporter plusieurs expressions (par exemple grâce à l'opérateur d'évaluation séquentielle "," que nous avons vu il y a 2 mois). Inversement, une expression n'est pas obligatoirement une instruction (par exemple, une affectation est une expression, sans être nécessairement une instruction), comme nous le verrons dans la suite de cet article.

Séquence

Une séquence d'instructions est donc ce qu'elle dit, c'est à dire simplement une suite d'instructions (sans préciser séparées par ";", car celui-ci est partie intégrante de l'instruction).

Blocs

Un "bloc" (ou "instruction composée") est une séquence d'instructions encadrée par les caractères accolades ouvrante "{" et fermante "}".

Le ";" faisant partie intégrante des instructions, la dernière instruction d'un bloc devra être aussi terminée normalement. En revanche, le bloc n'étant pas une instruction, il ne doit pas y avoir de ";" après la fermeture du bloc par "}".

Un bloc est considéré par le compilateur comme une instruction simple, et est donc interchangeable avec elle. Il est donc possible de démarrer un nouveau bloc n'importe quand, dès lors qu'une instruction est autorisée par la syntaxe du langage, et pas uniquement à la suite d'une instruction de contrôle ou de la déclaration d'une fonction. On peut ainsi avoir la séquence d'instructions suivante dans un programme :

```
{                       /* début d'un bloc */
  instructions;
.
.
.
{                       /* début d'un bloc imbriqué */
  instructions du bloc imbriqué;
.
.
.
 }                      /* fin du bloc imbriqué */
  suite des instructions du premier bloc;
.
.
.
 }                      /* fin du premier bloc */
```

A quoi cela sert-il ? Pour que vous le compreniez, je dois vous dévoiler la forme exacte d'un bloc, qui est la suivante :

```
{       /* début bloc */
déclarations;
instructions;
}       /* fin du bloc */
```

En effet, en C, on peut déclarer des variables au début de chaque bloc (elles sont donc locales à ce bloc), et non pas seulement au début d'une fonction.

Bien évidemment, c'est une possibilité qu'il faut employer avec parcimonie, et à bon escient, sous peine de rendre complètement incompréhensible le programme.

Mais en faisant bien attention, ceci permet dans un certain nombre de cas d'améliorer la lisibilité du programme, et d'éviter de déclarer une variable en permanence, alors qu'elle n'est utilisée que dans un cas bien précis.

Supposez, par exemple, que vous souhaitez permuter les nombres entiers 'a' et 'b' lorsque 'a' est plus grand que 'b'; vous pouvez écrire le code suivant :

```
if ( a > b ) {
    int temp = a;
    a = b;
    b = temp;
}
```

Note : J'ai ici combiné la déclaration d'une variable dans un bloc avec la possibilité de l'initialiser lors de sa déclaration.

Dans ce cas précis, il me semble plus clair de déclarer la variable temporaire juste pour la permutation et non pas en permanence dans la fonction; ainsi, on n'a pas besoin de chercher où elle est référencée. Remarquez que, dans cet exemple, le bloc est conditionné par l'instruction if, mais qu'il aurait très bien pu être imbriqué simplement dans un autre bloc (sans instruction de contrôle préalable); je pense cependant que cette pratique est à décourager, et qu'il vaut mieux alors déclarer les variables en tête de la fonction.
Elle est néanmoins intéressante dans la définition d'une macro; voir pour cela le programme d'accompagnement de cet article.

Lorsqu'une variable est redéclarée dans un bloc imbriqué (bien que ce soit possible, je ne saurais vous recommander cette pratique), elle masque la variable déclarée dans le bloc externe, pendant la durée de validité du bloc dans lequel elle est redéclarée, c'est à dire que lorsque l'on ressort de ce bloc pour exécuter l'instruction suivant l'"}", la variable masquée redevient active.
Par exemple :

```
{
short  i;
...
      {              /* bloc imbriqué */
       char i;      /* redéclaration de i pour ce bloc avec un autre type */
     ...
      }             /* fin du bloc imbriqué */
...     /* le premier i redevient actif */
}
```

Toutes les classes de stockage sont disponibles. Par exemple, on peut déclarer une variable statique dans un bloc imbriqué; cette variable perdurera donc après la terminaison de ce bloc. Ceci revient exactement au même qu'une variable statique locale déclarée en début de fonction, sauf que la variable locale au bloc ne pourra être utilisée que dans ce bloc (on dit qu'elle n'est visible que dans ce bloc). C'est bien entendu une hérésie, et il ne faut pas employer de telles méthodes.

Instruction vide

Un ";" tout seul constitue une instruction vide, qui peut donc être utilisée partout où une instruction est autorisée. Ainsi, en C, la séquence "...;;" est légale, bien que d'un intérêt limité (le premier ";" correspond à la fin d'une instruction, tandis que le second est l'instruction vide). De même, si un ";" suit une "}", il s'agit en fait de 2 instructions différentes (la première étant un bloc et la seconde, une instruction vide), et non pas d'un ";" terminant l'instruction composée "{ ... }" (d'ailleurs certains compilateurs peuvent rejeter ces constructions; ORCA/C, lui, les accepte).

A quoi celà peut bien donc servir, vous demandez-vous perplexes devant l'écran de votre GS préféré. Je vous rassure tout de suite, c'est très utile, notamment avec les instructions que nous allons étudier juste après dans cet article.

Dans la suite de cet article, toute référence au terme instruction correspondra indifférement à une instruction simple, composée (un bloc) ou vide.

Alternative

Comme vous l'avez sans doute déjà constaté (par la lecture des articles précédents ou du programme "Nombres"), l'instruction de test est le "if" (comme dans pratiquement tous les langages). En C, cette instruction a la syntaxe suivante (dans sa forme la plus simple) :

```
if (expression) instruction
```

Notez que les parenthèses sont obligatoires, et qu'elles délimitent l'expression. D'autre part, il n'y a pas de mot clef "then"; les concepteurs du C ont en effet jugé qu'il était redondant, et qu'il ne faisait qu'alourdir les programmes.
Vous remarquerez enfin que je n'ai pas mis de ";" après l'instruction, puisque celui-ci en fait partie.

L'expression est évaluée : si elle vraie (toute expression dont la valeur est différente de 0 est considérée comme vraie), l'instruction (qui peut être composée ou vide, mais dans ce dernier cas, ça n'a vraiment aucun intérêt) est exécutée. Dans le cas contraire (l'expression vaut 0), l'exécution passe à l'instruction suivante.

Remarquez que je n'ai nulle part indiqué "expression conditionnelle" : l'expression n'a en effet pas besoin d'utiliser les opérateurs de comparaison décrits dans le précédent GS Infos (pour mémoire, il s'agit de "==", de "!=", de "<", "<=", ">" et ">=").

Nous pouvons donc écrire : if (variable) ... Si la variable est différente de 0, la condition est vérifiée, et l'instruction exécutée; ainsi toute variable scalaire peut être utilisée comme booléen. C'est d'ailleurs une bonne pratique de ne pas faire de test explicite (== 0 ou != 0) lorsque la variable représente un booléen. En revanche, lorsqu'elle représente un nombre, le programme sera plus clair avec un test explicite.

Un autre exemple est celui d'une chaîne de caractères; je vous rappelle qu'une chaîne C est une suite de caractères terminée par un le caractère de code ascii 0 (écrit en général '\0') : on peut donc écrire "if (caractère) ..." pour déterminer si l'on est en fin de chaîne ou pas; personnellement, je préfère utiliser la forme "if (caractère != '\0') ..." qui me paraît indiquer beaucoup plus clairement ce que l'on fait.

ATTENTION : comme l'expression peut être quelconque, elle cache un piège dans lequel on peut très facilement tomber (même quand on est un pro), et occasionner des heures de recherche du bug (car bien évidemment, lorsqu'on connaît son programme, on ne voit jamais le bug qui est pourtant en plein en face des yeux).

L'expression peut en effet être une affectation; j'ai donc tout à fait le droit d'écrire "if (a = b) ...", ce qui affecte 'b' à 'a' et si 'b' était non nul, provoque l'exécution de l'instruction associée au if.
Malheureusement, très souvent, je voulais écrire "if (a == b) ..." pour vérifier l'égalité entre les 2 variables, et une petite faute de frappe de rien du tout peut faire perdre un temps fou (ne rigolez pas, ça m'est bien sûr déjà arrivé plusieurs fois).
Cela fait partie à la fois de la puissance, mais aussi de la complexité de C, dont ses détracteurs raffolent.

Bien entendu, cette possibilité est intéressante dans un certain nombre de cas; par exemple si 'b' est en fait un appel d'une fonction ou un élément de tableau, et que le test porte sur plusieurs valeurs de 'b'. Pour éviter de recalculer cette valeur (ce qui peut coûter cher), on peut écrire :

```
if ( ( a = fct(paramètres) ) == valeur1 || a == valeur2) ...
```

Comme l'ordre d'évaluation est garanti par le compilateur, cela évite de rappeler la fonction. Bien sûr, j'aurais pu tout aussi bien écrire à la Pascal ;-) :

```
a = fct(paramètres);
if ( a == valeur1 || a == valeur2 ) ...
```

Il est vrai que lorsqu'on débute, il est sans doute souhaitable d'employer cette deuxième approche. D'un autre côté, je n'aime pas trop la programmation en C à la manière de Pascal (je dis ça dans un cas général, et pas spécialement pour l'instruction précédente, les 2 formes sont également acceptables, et j'utilise indifféremment l'une ou l'autre); je trouve que cela alourdit inutilement les programmes. Comme toutes les bonnes choses, il faut en user, et pas en abuser.

Lorsque vous voulez réellement faire une affectation dans un test (ou en fait dans n'importe laquelle des instructions de contrôle, puisque ceci est possible avec quasiment toutes), précisez le dans un commentaire ou écrivez
"if ((a = b) != 0) ...".

Forme générale

Dans sa forme générale, l'instruction "if" se présente ainsi :

```
if (expression) instruction1
else instruction2
```

L'instruction2 est exécutée lorsque l'expression est évaluée à faux (c'est à dire à 0).

Notez que, puisque le ";" fait partie intégrante d'une intruction, il en faut 1 après instruction1 (donc avant le else) lorsque celle-ci est une instruction simple. Attention, néanmoins, à ne pas en mettre après une "}", car vous auriez alors 2 instructions (le bloc et l'instruction vide), et par conséquent, le "else" ne se rapportera plus au "if" (une erreur sera signalée par le compilateur, sauf si ce "if" est imbriqué et que le "if" extérieur n'a pas de "else", mais votre programme serait tout de même faux).

Tests multiples

L'instruction de tests multiples prend la forme :

```
if (expression) instruction
else if (expression) instruction
else if (expression) instruction
else instruction
```

Le dernier "else" correspond à la situation "aucun des précédents", c'est à dire l'instruction à exécuter lorsqu'aucune des conditions n'est satisfaite. Si il n'y a pas besoin d'une telle instruction, alors il n'y a pas de "else" final (dans ce cas, on utilise souvent ce dernier "else" pour détecter une erreur et afficher un message approprié).

Imbrication des if

Plusieurs instructions "if" peuvent être imbriquées; il faut seulement faire attention que le "else" se rapporte toujours au dernier "if". Si l'on souhaite qu'il s'applique à un "if" de niveau supérieur, on emploie les délimiteurs de bloc "{}".
Par exemple, si vous avez :

```
if (...)
    if (...) ...
else ...
```

le "else" se rapporte au deuxième "if" (le compilateur ne tient pas compte de votre indentation !).
Pour qu'il se rapporte au premier, il faut écrire :

```
if (...) {
    if (...) ...
} else ...
```

Sélections

A la place d'une batterie de "if ... else if ... else", vous pouvez utiliser l'instruction de sélection "switch", qui a la syntaxe :

```
switch (expression-scalaire) {
    case constante : instruction
    .
    .
    .
    default : instruction
}
```

Un certain nombre de commentaires sont nécessaires sur cette instruction :

• L'expression associée à l'instruction "switch" doit donner un résulat entier (cela peut néanmoins être aussi un caractère ou un énuméré qui sont considérés comme des entiers).

• En théorie, l'instruction switch est de la forme "switch (expression) instruction", mais dans la pratique, il s'agit toujours d'une instruction composée (un bloc), d'où la présence quasi-obligatoire des "{}". Par conséquent, on peut trouver des déclarations de variables juste après l'"{", mais elles ne pourront être initialisées, sauf si elles sont statiques (l'initialisation étant faite au moment de la compilation/link). Bien évidemment, je ne vous recommande pas d'utiliser cette possibilité.

• N'importe qu'elle instruction du bloc peut être introduite par le préfixe "case" suivi d'une constante (ou d'une expression donnant une constante).
Plusieurs "case" peuvent correspondre à la même instruction. En revanche, chaque constante ne doit être listée qu'une seule fois. Le type des constantes doit être le même que celui de l'expression associée à l'instruction "switch", aux conversions de types automatiques près.

• L'étiquette "default" permet d'introduire une série d'instructions qui seront exécutées lorsque l'expression ne correspond à aucune des constantes listées par les différents "case". Il ne doit aussi y avoir qu'un seul "default", mais il peut être placé n'importe où (donc pas nécessairement à la fin).

Cela a l'air bien compliqué, mais c'est en fait assez simple. Ce qu'il faut bien voir, c'est que les mots clefs "case" et "default" ne sont que des étiquettes, et non pas des instructions. En conséquence, la série d'instructions du bloc introduit par switch doit être considéré comme un tout, certaines de ces instructions sont simplement préfixées par une ou plusieurs étiquettes.

Ainsi, si plusieurs instructions sont associées à la même étiquette, cela ne constitue pas une instruction composée pour le "case" (puisque ce n'est pas une instruction), et il n'est donc pas nécessaire de les encadrer par des "{}" (encore que si vous le faites, cela n'aura aucune incidence, de par les vertus des blocs).

Le fonctionnement est alors le suivant :

• L'expression associée au "switch" est évaluée.

• On recherche dans l'instruction composée suivant le "switch" si il n'y a pas une étiquette "case" avec une constante correspondant à la valeur trouvée.

• Si oui, on débute l'exécution du bloc à l'instruction associée à cette étiquette.

• Dans le cas contraire, l'exécution du bloc démarre à l'étiquette "default" si elle existe. Sinon, aucune instruction du bloc ne sera exécutée, et le déroulement du programme se poursuit avec l'instruction suivant ce bloc.

• L'exécution se poursuit jusqu'à la fin du bloc, sans plus tenir compte des étiquettes "case" et "default" qui pourraient suivre. En Pascal, la rencontre d'un nouveau cas provoque la sortie de l'instruction "case". Ce n'est pas le cas en C.

• Pour sortir du "switch", sans exécuter les instructions correspondant aux autres étiquettes "case" (ou "default"), on utilise l'instruction "break", qui effectue un branchement sur l'instruction suivant le bloc associé au "switch" (on peut aussi utiliser les autres instructions de branchement de C, "goto" et "return", que nous verrons plus loin pour "goto" et dans un autre chapitre pour "return"). On obtient ainsi un comportement identique à celui de Pascal.

Au niveau du GS (et en simplifiant), le compilateur construit une jump-table avec une entrée pointant sur chacun des cas, puis il effectue un saut en indexant cette table par la valeur donnée par l'expression. Seule l'instruction "break" (ou la fin du bloc, bien sûr) génère à son tour un branchement vers l'instruction suivant la jump-table. Du moins, ceci est le cas général, car si les différentes valeurs possibles de l'expression (donc des "case") sont disparates (par exemple 1 et 10000), ORCA/C générera des comparaisons plutôt qu'une jump-table qui aurait contenu 9998 entrées se branchant vers l'instruction suivant le switch.

Voyons tout cela sur un exemple :

```
switch (i) {
   case 1 : ...
            break;

   case 2 : ...
            break;

   default : ...
             break;
}
```

Une bonne habitude à prendre est de mettre un "break", bien entendu après chaque cas, mais aussi après le dernier. Ainsi, si vous avez besoin de rajouter un nouveau cas, vous n'aurez pas à penser à rajouter le break au cas qui devient l'avant dernier, et donc d'éviter des surprises qui peuvent être désagréables.

D'un autre côté, comme chaque constante ne peut être présente qu'une seule fois, il est parfois nécessaire de ne pas mettre de "break" entre 2 cas. Par exemple, si un cas doit effectuer un traitement supplémentaire à d'autres cas, on le met avant, on effectue ce traitement, puis on met les autres cas en dessous, sans les séparer par break, le premier cas continue alors son exécution avec les instructions communes. Il est souhaitable de préciser par un commentaire que l'absence de "break" est volontaire.

Voici un exemple typique de l'utilisation de ces différentes possibilités (vous noterez que l'on peut sortir du "switch" par l'instruction "break" de façon conditionnelle — à l'intérieur d'un "if") :

```
switch (c) {
    case '\' : /* traiter les cas \n, \t ... */
                if (caractère suivant == n || t ...) {
                ...
                 break;
               }
                /* caractère ordinaire précédé par escape */
               c = caractère suivant;
               /* on tombe dans le cas normal, d'où pas de break */

    default : /* traitement des caractères ordinaires */

              ...
               break;
}
```

Branchement inconditionnel

C dispose de l'instruction "goto" (comme pratiquement tous les autres langages). Elle est cependant encore moins nécessaire qu'en Pascal, car C possède naturellement des instructions de sortie et de poursuite de boucle sans utiliser aucun artifice, comme nous allons le voir par la suite (j'ai déjà écrit plusieurs dizaines de milliers de lignes de C, sans jamais avoir eu besoin de faire de goto).

Le "goto" se fait à une étiquette qui a la syntaxe d'un identificateur et est suivie du caractère ":". Cette étiquette se rapporte à l'instruction qui la suit, et doit être unique dans la fonction dans laquelle elle est utilisée. Le goto ne peut être fait que sur une étiquette localisée dans la même fonction (on ne peut pas faire de branchement par goto d'une fonction à une autre). On peut en revanche effectuer un saut au milieu d'un bloc (le comble de l'horreur !), et ignorer notamment l'initialisation des variables locales à ce bloc si il y en a, avec tous les effets de bord que cela peut impliquer. Une étiquette peut enfin être déclarée, sans pour autant qu'aucun goto n'y fasse référence.

Je ne peux que vous recommander de bannir cette instruction inutile dès maintenant.

Ne croyez pas que je sois un fanatique de la programmation structurée (d'ailleurs, si je l'avais été, j'aurais continué avec le carcan qu'impose Pascal plutôt que de passer à la liberté du C), mais je pense néanmoins qu'elle a quelques mérites.

L'utilisation du "goto" dans les langages modernes montre en général un problème de conception du programme (les puristes appellent cela de la programmation spaghetti).

Lorsque le problème est clairement énoncé, la solution s'écrit toujours simplement, et il vaut mieux écrire un ensemble de fonctions traitant chacune un de ses aspects, au lieu d'une méga-routine avec des tonnes de tests et de boucles imbriquées (3 ou 4 me paraît constituer un maximum maintenable; de ce point de vue, le programme d'accompagnement de cet article est limite, mais personne n'est parfait :-).

C'est dans ce dernier cas que l'évolution d'un programme se réduit souvent à faire des branchements dans tous les sens pour intégrer les modifications. Au contraire, il est beaucoup plus facile de modifier une petite fonction pour traiter un nouvel aspect du problème. Il faut cependant faire attention à ce qu'une évolution ou une correction de bug ne vienne pas compliquer la fonction modifiée; bien souvent, lorsque cela se produit, il est souhaitable de repenser cette fonction et de la redécouper à son tour en sous-problèmes plus petits (sur le GS, l'appel à une fonction ne coûte pas très cher : quelques PUSH/PULL, un JSL, un RTL, et un réajustement de la pile et de la direct page).

Je vous donne cependant un exemple, pour que vous voyez comment cela fonctionne :

```
fonction ( arguments )
{
    ...
      goto plus_loin;
    ...
      plus_loin : ... /* c'est ici que le goto se branchera */
    ...
}
```

Branchements longs

Le "goto" du C ne peut être utilisé que dans le contexte d'une seule fonction.

Bien qu'il ne s'agisse pas d'une instruction, mais d'un couple de 2 fonctions de la librairie standard du langage (que l'on doit donc trouver dans toutes les bonnes implémentations dont ORCA/C), C permet d'effectuer un branchement de presque n'importe quel point du programme à presque n'importe quel autre.

Ces fonctions sont "setjmp" et "longjmp" : la première permet de définir un repère, vers lequel la seconde permettra de se brancher directement. Pour le programme, le saut sera matérialisé par un retour de la fonction "setjmp"; la valeur retournée étant 0 lorsqu'il s'agit de l'appel normal initialisant le mécanisme, et une valeur passée en paramètre à "longjmp" autrement (bien entendu, si le programme utilise cette valeur, l'appel à "longjmp" ne devra pas utiliser la valeur 0).

Plus haut, j'ai indiqué 'presque' car le "longjmp" ne pourra être effectué que depuis une fonction qui est dans la séquence d'appel de celle ayant initialement appelé "setjmp", sans quoi vous pouvez vous attendre à avoir un crash.

L'utilisation typique de ces fonctions est dans un système de menus : la fonction proposant le menu principal appelle "setjmp" avant d'afficher le menu, et les fonctions traitant chacune des options (et qui peuvent avoir été appelées par un certain nombre de fonctions intermédiaires), peuvent ainsi revenir au menu principal directement par "longjmp", par exemple en cas d'erreur ou sur commande spéciale (un 'escape'). Néanmoins, je pense qu'il existe des manières "moins sauvages" de parvenir au même résultat sans utiliser ces fonctions; de plus, avec la programmation évènementielle du GS, c'est d'un intérêt assez limité.

D'un point de vue pratique, et en schématisant, la fonction "setjmp" va mémoriser dans la zone mémoire qu'on lui passe en paramètre (on peut ainsi avoir plusieurs "setjmp" en attente) le pointeur de programme et celui de la pile de la fonction appelante. Le "longjmp" n'aura plus qu'à les restaurer, un RTL faisant le reste.
C'est pour cela qu'il faut que la fonction faisant le "longjmp" doit être dans l'arbre d'appel de celle ayant fait le "setjmp", de façon à ce que son contexte soit toujours sur la pile.

Les boucles (ou les itérations)

C (comme Pascal) dispose de 3 instructions pour réaliser des boucles; 2 d'entres elles sont d'ailleurs pratiquement équivalentes dans les 2 langages, tandis que la troisième, bien qu'ayant le même nom, est beaucoup plus puissante que son homologue pascalienne.

Tant que

Commençons par la plus simple des 3 : le "while". Sa syntaxe a la forme suivante :

```
while (expression) instruction;
```

L'expression est évaluée, et tant qu'elle est non nulle (donc considérée comme vraie), l'instruction est exécutée. Cette instruction peut être simple ou un bloc ou une instruction vide. Cette dernière possibilité est intéressante dans un certain nombres de cas, notamment lorsque la totalité de l'action est réalisée par l'expression conditionnant la boucle. Par exemple, supposez que vous souhaitez savoir si une touche du clavier a été enfoncée, vous pouvez utiliser l'instruction :

```
while ( kbd < 127 );
```

L'exemple n'est pas tout à fait exact, mais il requiert des notions que vous ne connaissez pas encore. Disons simplement que "kbd" correspond à 0xE0C000.

Pour que l'instruction soit exécutée une première fois, il faut que la condition soit initialement vraie (donc que la première évaluation de l'expression soit différente de 0). Vous aurez constaté que cette instruction fonctionne de façon strictement identique à son homologue Pascal.

Contrôle des boucles

Deux instructions annexes sont très utilisées dans le contexte d'une boucle, et servent à contrôler leur exécution :

• L'instruction "break", que nous avons déjà rencontrée, permet de sortir à tout moment de la boucle dans laquelle elle est employée; dans le cas de boucles imbriquées, elle ne permet de sortir que de la boucle interne, mais par un système de drapeau (flag), on peut cascader les "break" pour sortir de la boucle la plus externe.

Le programme d'accompagnement de cet article en constitue un bon exemple. Si vous avez un "switch" imbriqué dans une boucle, le "break" utilisé pour terminer un "case" ne s'appliquera, bien entendu, qu'au "switch" et pas à la boucle englobante.

On peut aussi bien sûr utiliser l'instruction goto ... hum!

• L'instruction "continue" permet, elle, de passer directement à la fin du bloc (et donc de réévaluer l'expression pour le tour de boucle suivant), sans exécuter toutes les instructions intermédiaires. En gros, cela revient à faire un goto à une étiquette précédent immédiatemment la fin du bloc.

Pour synthétiser, voyons sur un exemple, à quoi correspondent ces 2 instructions :

```
while (expression) {
  ...;
   etiquette_continue:
}
etiquette_break:
```

L'instruction "continue" correspond donc à un branchement à "etiquette_continue", tandis qu'un "break" correspond à un branchement à "etiquette_break".

Enfin, l'instruction "return" que nous traiterons dans le prochain numéro, avec les fonctions, permet bien entendu aussi d'interrompre une boucle.

Faire ... tant-que

Dans certains cas, on veut pouvoir exécuter au moins une fois l'instruction à répéter. C dispose à cet effet de l'instruction "do ... while", dont la syntaxe est :

```
do instruction while (expression);
```

L'instruction est donc exéutée au moins une fois, puis l'expression est évaluée : si elle vraie (non nulle), la boucle reprend; dans le cas contraire, l'exécution se poursuit avec l'instruction suivant le "while".

Vous remarquerez que l'expression est évaluée dans les mêmes conditions que l'instruction "while" décrite précédemment, la seule différence étant que l'instruction est exécutée au moins une fois.

Cette instruction n'est donc pas strictement équivalente au "repeat ... until" de Pascal, puisque cette dernière continue la boucle tant que la condition est fausse.

Cette boucle peut être aussi interrompue prématurément par les instructions "break" et "continue" (ainsi que par "goto" et "return").

Pour

L'instruction la plus puissante de C est, à mon avis, l'instruction "for".

Dans la plupart des langages (Pascal en représentant l'implémentation la plus limitée), la boucle "for" permet de répéter un certain nombre de fois, connu à l'avance, la ou les instructions associées, éventuellement avec un pas d'incrémention ou de décrémentation différent de 1 (la limitation particulière de Pascal vient du fait que le pas est obligatoirement de 1).

Bien entendu, C permet de réaliser cette opération. Cependant, la syntaxe du "for" C est telle que celui-ci peut être utilisé dans bien d'autres occasions.
Cette syntaxe est donc :

for (expression1; expression2; expression3) instruction

Les expressions peuvent être quelconques, et elles sont même optionnelles. En revanche, les ";" séparant ces expressions sont obligatoires.

Avant de détailler le fonctionnement du "for" (et de vous le rendre plus compréhensible), sachez qu'il est strictement équivalent à la séquence d'instructions suivante :

```
expression1;
while (expression2) {
      instruction
      expression3;
}
```

Le principe général est alors :

• Expression1 (si elle existe) est évaluée. En général, elle va consister à initialiser la ou les variables correspondant au compteur de la boucle (en se référant au "for" des autres langages).

• Expression2 est évaluée. Si elle est vraie (non nulle), l'instruction associée au "for" est exécutée. Cette instruction peut être un bloc ou vide, si le "for" se suffit en lui-même; cela arrive très fréquemment, voyez, par exemple, le programme d'accompagnement de cet article. Si l'expression2 est fausse, l'exécution continue à la suite du "for", sans que expression3 ne soit évaluée, ni l'instruction exécutée. Si l'expression2 est absente, elle est alors considérée systématiquement comme vraie, et la boucle devra être interrompue, par exemple par "break".

• Expression3 est évaluée après l'instruction, et est utilisée généralement pour faire avancer le compteur de boucle. Le processus se répète avec la réévaluation de l'expression2, puis l'exécution éventuelle de l'instruction, et une nouvelle évaluation de expression3, jusqu'à ce que la boucle se termine.

L'exemple le plus classique de cette instruction est de la forme :

```
for ( i = 0; i < nombre_de_repetitions; i++ ) ...
```

ou (ce qui revient au même) :

```
for ( i = 1; i <= nombre_de_repetitions; ++i ) ...
```

Notez que l'on peut utiliser les 2 formes d'incrémentation dans ce cas, sans aucune incidence sur le résultat. C'est juste une affaire de goût.

L'exemple typique d'une boucle "for" se suffisant à elle-même, c'est à dire avec une instruction vide, est celui de la recherche d'un élément dans une table, ce qui donne :

```
for ( i = 0; i < nombre_elements && element_i != element_recherche; i++ );
```

Notez bien le ";" juste après le "for", signifiant qu'il n'y a aucune instruction à répéter.

A la fin de la boucle, "i" sera soit l'indice de l'élément si il a été trouvé, soit le nombre d'éléments en cas d'échec. En effet, contrairement à d'autres langages (tel Pascal), il n'y a pas à proprement parler de variable de contrôle de boucle en C, c'est à dire que, dans notre exemple, "i" a une valeur significative et utilisable à la fin de la boucle, et aussi qu'il peut être modifié dans le corps de la boucle (voir le programme d'accompagnement pour un exemple).

En Pascal, la variable "i" aurait été 'réservée' pendant la durée de la boucle, et n'aurait donc pu être modifiée, ni utilisée après la fin de la boucle.

La valeur de "i" pourra aussi être utilisée en cas d'interruption prématurée de la boucle (ce qui peut être effectué, comme pour les autres types de boucle, par "break", mais aussi par "goto" ou "return"); ainsi l'exemple précédent aurait pu être écrit de la manière suivante :

```
for ( i = 0; i < nombre_elements; i++)
      if ( element_i == element_recherche )
        break;

if ( i != nombre_elements )        /* element trouve */
   ...
else                               /* element non trouve */
   ...
```

Les 2 types de construction sont également employés. En général, on peut considérer que lorsque les conditions d'arrêt de la boucle sont multiples ou que des traitements annexes à la simple recherche sont nécessaires, il est préférable d'utiliser la deuxième forme, tandis que pour une recherche simple, la première méthode a l'avantage de la concision et de la clarté.

Ce processus est donc tel qu'il a été représenté en utilisant l'instruction "while". Il y a cependant une différence énorme, qui justifie à elle seule toutes les possibilités de cette instruction : si on utilise l'instruction "continue", on reprend avec la réévaluation de expression3, tandis qu'avec un while, on ne peut continuer qu'avec expression2, à moins de coder une copie de expression3 avant le "continue", ce qui alourdit le programme, et rend sa maintenance plus difficile (si cette expression doit être modifiée, il faudra changer toutes ses occurences).

Si les 3 expressions sont vides, on a alors la forme "for (;;)" qui signifie 'boucler indéfinimment'; il faudra alors employer une des instructions d'interruption précédentes.

Quand utiliser "for" plutôt que "while" et vice-versa : disons que lorsque la boucle ne nécessite pas d'initialisation ou de mise à jour d'une ou plusieurs variables, il n'y a aucun besoin d'employer le "for" et que le "while" convient parfaitement à cette situation.

En revanche, dès que l'on rentre dans une boucle de balayage d'une liste d'éléments, dans laquelle on commence à initialiser un index sur le premier élément, et que cet index doit ensuite correspondre à chacun des éléments, le "for" C montre ici toute sa supériorité (notamment lorsque le pas d'incrémentation est différent de 1) :

• Il permet de localiser en tête de la boucle toutes les informations de contrôle, et donc d'éviter à les rechercher dans le corps de la boucle, comme cela serait le cas avec "while". Ceci est particulièrement important dans le cas de boucles imbriquées, où il devient difficile de déterminer ce qui se passe, lorsque le corps de la boucle est mélangé avec son contrôle.

• Comme les expressions sont quelconques, un indice de boucle n'a pas à être nécessairement un entier, ni à subir une progression arithmétique. Nous verrons par exemple, lorsque nous traiterons les pointeurs, que l'instruction "for" est très puissante pour balayer des structures chaînées.

Bien entendu, on peut tout faire avec l'instruction "for". Encore une fois, la sagesse dicte de l'utiliser à bon escient, c'est à dire dans un contexte répétitif, et non pas pour faire n'importe quoi.

Une dernière remarque pour en terminer avec cette instruction; je vous avais dit la dernière fois que l'opérateur "," était surtout utilisé avec l'instruction "for". Voici donc comment : je vous rappelle que cet opérateur permet de grouper deux expressions qui sont évaluées l'une à la suite de l'autre; par conséquent, les différentes expressions constituant l'instruction "for" peuvent être plusieurs expressions séparées par ce fameux opérateur ",". Cela sert surtout avec les expressions 1 et 3, par exemple, pour faire varier 2 compteurs en parallèle.
Concrètement, cela donne quelque chose comme :

```
for ( i = 0, j = n; i < j; i++, j-- ) ...
```

Conclusion

Désormais vous savez tout sur les tests et les boucles de C. J'espère que vous commencez maintenant à voir tout ce que l'on peut faire avec ce langage. Dans le prochain GS Infos, nous continuerons notre croisière à bord de C, avec tout ce qui concerne les sous-programmes, que l'on appelle fonctions.

Programme d'accompagnement

Je ne sais pas ce que vous en pensez, mais il me semble que l'on comprend mieux (notamment lorsqu'on aborde un nouveau domaine) quand on voit comment les concepts étudiés sont mis en œuvre dans la pratique, et dans notre cas, dans un vrai programme.

Notez que ce n'est pas une chose facile pour moi; il faut que je trouve une idée qui permette de développer un programme fonctionnel (et intéressant !), qui illustre tout ce qui a été abordé dans le texte de l'article, mais qui n'emploie pas des techniques que nous n'avons pas encore traité.

Je pense y être à peu près arrivé avec le programme de ce numéro, qui montre quasiment tous les différents types de boucles et de tests possibles en C, mais qui utilise aussi des tableaux, que nous ne verrons que dans 4 mois (en général, les boucles sont souvent employées pour travailler avec des tableaux; il m'était donc impossible de faire autrement).

Je ne sais pas si vous trouverez le programme que je vous propose intéressant (dans les 2 cas, faites le moi savoir), mais bien que l'utilisateur soit passif, il revêt un certain côté ludique.

Avez-vous déjà entendu parler des "cryptarithmes" ? Il s'agit d'un jeu de type casse-tête, dont le principe de base est de retrouver une opération arithmétique simple (addition, soustraction, multiplication ou division). Cette opération a subi une transformation littérale, c'est à dire que chacun de ses chiffres a été remplacé par une lettre. Les règles sont alors :

- Un chiffre donné est toujours remplacé par la même lettre;
- Une lettre donnée représente toujours le même chiffre (la substitution est donc bijective);
- Aucun nombre ne commence par 0, et les accents n'ont pas d'incidence.
- Bien entendu, une fois les chiffres retrouvés, l'opération est exacte !

Les lettres utilisées forment en général des mots, et souvent l'opération sous forme de texte a aussi un sens (les différents mots on un rapport entre eux).

Le programme, que je vous propose donc, s'appelle "Crypt". Il permet de résoudre les cryptarithmes représentant des additions de 2 opérandes. Par simple transformation, il peut aussi résoudre des soustractions (en effet, si le problème est a - b -> c, il suffit de soumettre au programme b + c -> a).

Je vous donne ci-après les 10 qui m'ont servi à mettre au point le programme (la plupart ont été puisés dans la défunte revue Jeux & Stratégie). Vous pouvez vous amuser à essayer de les résoudre à la main, puis vérifier le résultat avec le programme.

```
  SEND      SPEND      VACHE      SUISSE      DEUX
+ MORE    + LESS     + CHEVAL   + SUEDE     + NEUF
------    ------     -------    --------     -----
 MONEY     MONEY      OISEAU      BRESIL      ONZE

  ROMEO      TIGRE      JOYEUX      CALCUL     SOLEIL
+ JULIE    + LIONNE   +  NOEL     + MENTAL   + SABLE
-------    -------    -------     -------    -------
 AMOUR      TIGRON     LUCIEN      ELEVES     BIKINI
```

Il est possible que le programme ne marche pas avec d'autres cryptarithmes, mais il donne une solution correcte aux 10 précédents. Notez qu'il ne trouve qu'une seule solution, même lorsqu'il y en a plusieurs.

Plusieurs améliorations à ce programme sont possibles (je vous les laisse en exercice si le cœur vous en dit) :

- Trouvez toutes les solutions à un problème.
- Résoudre les problèmes avec plus de 2 opérandes.
- Résoudre les problèmes autres que l'addition.

Deux remarques pour terminer :

- Au début du programme, vous verrez 2 définitions de macro avec paramètres. Ceci permet de décrire une portion de code utilisée plusieurs fois, et de l'appeler comme si il s'agissait d'une fonction; cependant, l'appel de la macro est remplacé par sa définition, ce qui est plus économique, en termes de performances, qu'un appel à une fonction.

Chacune de ces macros se présente sous la forme d'un bloc, ce qui permet de déclarer une variable locale au bloc, et donc à la macro. Comme pour une véritable fonction, les arguments de la macro seront substitués par les paramètres utilisés lors de son appel. Enfin, vous noterez la présence du "\" à la fin de chacune des lignes définissant la macro; je vous rappelle qu'il s'agit du caractère de continuation de ligne. Il est rendu nécessaire, car la définition d'une macro doit être faite nécessairement sur une seule ligne.

• Les printf contiennent parfois des caractères exotiques; il s'agit en fait de caractères de contrôle pour le driver de la console (notamment la fonction "gotoxy" avec ces arguments), ce qui permet de réaliser un affichage plus joli.

La méthode employée n'est pas très élégante; j'aurais dû en effet utiliser des définitions de constante pour les codes de contrôle, et des macros pour les envoyer à l'écran. Allez, je vous laisse le faire ...

Nouvelles du front

Une nouvelle version d'ORCA/C, la v1.3, est disponible depuis le mois de novembre (il en est de même pour ORCA/Pascal qui passe en v1.4, et de ORCA/M dont la version 2.0 est désormais disponible). Les informations données dans l'article de présentation de ORCA/C, paru dans GS Infos 17, sont toujours valables. Cette version corrige avant tout certains des bugs des versions précédentes (il en reste encore !), notamment celui dont je vous avais parlé la dernière fois, semble t'il; depuis le mois d'août je suis malheureusement tombé sur plusieurs d'entre eux, dont un qui m'a fait perdre plusieurs heures, et qui est corrigé dans cette version.

Cette version comprend néanmoins 2 nouvelles fonctionnalités (ORCA/Pascal intègre les mêmes améliorations) :

• Une directive "cdev" permet de les réaliser directement (c'était déjà possible avant, en faisant quelque peu attention), de la même manière qu'un NDA ou un CDA.

• 2 drivers GS/OS (.PRINTER et .NULL) à installer dans le dossier *:SYSTEM:DRIVERS sont fournis sur la disquette système accompagnant les compilateurs. Le premier permet d'utiliser l'imprimante en mode texte, directement depuis un programme C ou Pascal. Il suffit de faire un appel à la fonction "open" de son langage ou de GS/OS en donnant comme nom de fichier .PRINTER; toutes les écritures suivantes sont alors redirigées vers l'imprimante. Notez que l'on n'a pas besoin d'être sous le shell pour l'utiliser. Un CDA et un CDEV permettent de configurer le driver.

Le driver .NULL ne présente d'intérêt qu'avec la fonction de redirection du shell, par exemple lorsqu'on n'est intéressé que par le résultat final d'une commande, et pas par les messages intermédiaires (ceci est notamment utile dans les fichiers EXEC).

chapitre 5

==========

Les Fonctions

Mea culpa

=========

Avant d'aborder le thème de cet article, je voudrais faire un petit rectificatif.
Dans le numéro 16 de GS Infos, j'ai parlé de la déclaration des variables en C (il s'agissait de la deuxième partie de l'initiation). J'ai commis une petite erreur dans la description des variables globales. J'ai en effet écrit qu'en C les variables globales n'étaient visibles qu'à partir de l'endroit où elles étaient définies dans le source. C'était vrai dans le langage C original, défini par Kernighan et Ritchie. Ca ne l'est plus en ANSI C qui indique qu'une variable globale est visible pour toutes les fonctions du source, quelque soit l'endroit où elle est déclarée dans ce source. En général, cela ne fait pas une grande différence, car, comme je vous l'avais déjà indiqué à cette époque, les variables globales sont souvent déclarées au début du fichier source, à des fins de lisibilité.

Je vous prie de bien vouloir m'excuser de cette petite erreur; le problème est que je programme en C depuis très longtemps, et que je n'ai pas relevé tous les changements que la standardisation a apportés. J'espère que je n'ai pas commis de bévue plus grave; en général, j'essaie de vérifier ce que j'ai écrit avec le manuel d'ORCA/C pour éviter de telles erreurs, bien que celle-ci m'avait échappé jusqu'à ce que je lise récemment un article sur ce sujet; si tel était malheureusement le cas, n'hésitez pas à me le dire, pour que j'apporte d'autres rectificatifs.

Venons en maintenant au sujet de cet article.

Introduction aux fonctions

==========

Tous les langages (même l'assembleur) permettent au programmeur de découper son programme en plusieurs morceaux, de façon à ce qu'il soit gérable, c'est à dire qu'on puisse notamment le faire évoluer et corriger les défauts le plus facilement possible. Ceci permet aussi de réutiliser ces morceaux d'un programme à un autre, et d'en masquer les détails au reste du programme.

Le terme général que l'on emploie pour désigner ce découpage est celui de routine ou de sous-routine. Une routine permet donc d'isoler un ensemble d'instructions, effectuant une tâche bien déterminée du programme. Il est souvent souhaitable que cette tâche puisse être exécutée plusieurs fois ou dans des contextes légérement différents. Une routine peut aussi nécessiter les services d'une ou plusieurs autres routines pour effectuer son travail, et ce de façon récurrente, c'est à dire avoir des sous-sous-routines.

Une telle routine va être identifiée par un nom. Le langage permet d'exécuter cette routine depuis n'importe quel point du programme, et donc, en quelque sorte, de transférer l'exécution du programme à cette routine; on désigne ce processus par le terme d'"appel". Le programme appelle la routine en spécifiant son nom, et dans certains langages, en utilisant une instruction spéciale pour effectuer cet appel (par exemple, en assembleur, on va utiliser l'instruction JSR ou JSL).

Lorsque celle-ci a terminé son travail, l'exécution du programme doit reprendre là où elle en était restée au moment de l'appel à la routine, comme si elle représentait en fait une instruction de base du langage; ce mécanisme est appelé le "retour à l'appelant", ou de façon abrégée le "retour".

Un programme est donc capable d'appeler une routine, qui à son tour peut appeler d'autres routines (et éventuellement elle-même, comme nous le verrons à la fin de cet article), et ce indéfiniment, la seule limite étant définie par l'ordinateur sur lequel est exécuté ce programme. Les retours permettent de revenir successivement dans chacune des routines appelantes jusqu'au point de départ.

Une routine peut être appelée dans des contextes légérement différents, et donc adapter son travail au contexte voulu; ceci est réalisé par un processus appelé "passage de paramètres", lesquels paramètres indiquent à la routine sur quelles données elle doit travailler. De la même manière, lorsque la routine retourne à son appelant, elle peut lui délivrer un "résultat" correspondant au travail effectué.

Chaque langage utilise un nom différent pour représenter ce concept de routine; certains langages définissent même 2 instructions selon que la routine retourne ou non un résultat à la fin de son exécution. Par exemple, en Pascal, une routine ne retournant pas de résultat est appelée "PROCEDURE", tandis que si elle retourne un résultat, on emploie le terme "FUNCTION".

En C, il n'y a que des "fonctions" qui, en théorie, retournent toujours une valeur. Un mécanisme a cependant été ajouté dans les compilateurs ANSI pour indiquer quand il n'y a pas de résultat retourné. Comme le C est un langage économe, il n'y a pas de mot clé indiquant qu'une fonction est définie, le contexte permettant de le déterminer automatiquement.

En C, il n'y a pas non plus de concept de "programme principal", comme c'est par exemple le cas en Pascal. A la place, il y a une fonction qui a un nom prédéterminé; en l'occurence, elle doit s'appeler "main".

En dehors de ce nom réservé, cette fonction se comporte comme toutes les autres fonctions d'un programme; la seule différence est que lorsqu'elle retourne à son appelant, le programme se termine, puisque l'appelant peut être vu comme étant le programme qui l'a lancé (par exemple le shell ORCA ou le Finder) ou plus généralement, on peut considérer que c'est le système.

D'un point de vue pratique, le compilateur fait en sorte que la fonction "main" soit la première exécutée. Par exemple, dans le cas d'ORCA/C, le compilateur génère systématiquement (sauf si on utilise la directive "noroot" dans le source considéré), un fichier ".root" qui initialise l'environnement, puis qui appelle la fonction "main"; lorsque celle-ci retourne à son appelant, le code écrit dans le fichier ".root" reprend la main et remet les choses au propre, avant de quitter vers le système. Si le source correspondant ne comporte pas de fonction "main", ce qui est le cas lorsque l'on découpe un programme en plusieurs fichiers sources, le fichier ".root" ne sert à rien; il est donc souhaitable d'employer la directive "noroot" pour éviter sa génération.

Pour les programmes 'spéciaux', tels que les accessoires, il n'y a pas de fonction "main"; à la place, une directive spécifique du compilateur permet de générer un fichier ".root" différent, ce fichier appelle cependant une ou plusieurs fonctions bien déterminées, définies par la forme particulière de ce programme.

Déclaration et définition

Lorsqu'on parle de fonctions (ou plus généralement de routines, ceci s'appliquant à tous les langages), on doit distinguer les termes "déclaration" et "définition".

Le second terme est utilisé lorsqu'on fait référence aux instructions la composant : on "définit" la fonction. En revanche, lorsqu'on indique au compilateur que l'on va utiliser une fonction, quelque soit l'endroit où elle est définie d'ailleurs (ce peut être dans le même source, soit avant, soit après l'endroit où on l'utilise, ou bien dans un autre fichier source du programme, ou encore dans une librairie d'usage général, comme par exemple la librairie standard), on "déclare" cette fonction.

La déclaration d'une fonction se présente de la même manière que la déclaration d'une variable ordinaire, sauf qu'elle peut être faite automatiquement si on fait reférence à une fonction dans une expression, et qu'on ne l'a pas déclarée préalablement; la valeur retournée est alors considérée comme étant du type "int".
Toutes les déclarations ultérieures, ainsi que la définition de cette fonction, devront retourner une valeur de ce type. Il est donc fortement recommandé de déclarer toutes les fonctions avant de les appeler, pour éviter ce genre de problème. De plus, les compilateurs ANSI permettent de vérifier la concordance entre la déclaration d'une fonction et sa définition, et donc d'éviter des erreurs.

Notez, que contrairement au Pascal, une fonction n'a pas besoin d'être définie avant d'être utilisée, ce qui fait qu'il y a une distinction entre définition et déclaration. Pascal nécessitant de définir toute procédure avant de l'utiliser, la déclaration est faite implicitement par la définition. Par conséquent aussi, C n'a donc pas besoin de directive "forward" pour indiquer au compilateur que le corps de la fonction est après sa déclaration.

Deux écoles ont tendance à s'affronter : celle disant que les fonctions doivent être définies avant d'être utilisées (comme en Pascal); dans ce cas, la fonction "main" se retrouve généralement à la fin du source. Les autres définissent les fonctions appelantes avant les fonctions appelées; la fonction "main" est alors la première définie. Personnellement, je n'ai pas de préférence particulière, et j'utilise indifféremment les 2, bien que j'ai une certaine tendance à utiliser de plus en plus la première méthode. L'intérêt est que dans ce cas la déclaration est faite implicitement par la définition, et que l'on peut bénéficier des contrôles effectués par le compilateur, lorsqu'on utilise la forme moderne, sans avoir à effectuer de déclaration explicite.

Définition d'une fonction

La définition d'une fonction peut avoir lieu n'importe où dans le source, à partir du moment où on n'est pas déjà en train de définir une autre fonction.
En d'autres termes, il n'est pas possible d'imbriquer la définition des fonctions.

La syntaxe générale est alors la suivante :

```
type-résultat  nom-fonction  ( paramètres )
{
        corps-de-la-fonction
}
```

La déclaration d'une fonction, similaire à celle d'une variable aurait la forme suivante :

```
type-resultat  nom-fonction  ( paramètres );
```

Notez la présence du ";" à la fin de la déclaration, alors qu'il n'y en a pas pour une définition. De plus, la définition comprend une instruction composée correspondant au corps de cette fonction.

Chacune des parties de la définition peut être absente, si bien que la définition minimale d'une fonction se réduit à :

```
nom-fonction ()
{
}
```

A priori, cela ne présente pas un grand intérêt, puisqu'il est évident qu'une telle fonction ne fait strictement rien. En fait, cela est tout de même utile lorsqu'on développe un gros programme, puisque cela permet de satisfaire le linker, et donc d'exécuter correctement le programme, par exemple pour tester d'autres parties, sans pour autant nécessiter l'écriture de toutes les fonctions.

Lorsqu'on ne spécifie pas le type du résultat, celui-ci est pris automatiquement comme étant "int", et non pas, comme on pourrait s'y attendre, pour indiquer qu'il n'y a pas de résultat, et donc correspondre à une procédure Pascal.

Traditionnellement, le fait de ne pas indiquer de type de résultat pouvait aussi bien correspondre à un résultat de type int que pas de résultat du tout.

Pour lever cette ambiguïté, les compilateurs modernes (et donc notamment ceux respectant la norme) définissent le type "void" pour indiquer qu'il n'y a pas de résultat. Il est aussi fortement conseillé de toujours spécifier le type du résultat, lorsqu'il y en a un, même si c'est "int". Le type du résultat peut être n'importe quel type valide en C (c'est à dire aussi bien un type prédéfini qu'un type spécifique), à l'exception d'une fonction ou d'un tableau, mais on peut retourner un pointeur sur ces 2 types (nous y reviendrons dans un article futur).

ORCA/C dispose d'options spéciales dans la définition d'une fonction :

"inline" permettant de définir l'interface avec la boîte à outils et "asm" permettant d'écrire des fonctions en assembleur directement dans le source C.

Je ne m'étendrai pas plus ces options, et je vous renvoie au manuel du compilateur pour tous les détails les concernant.

La syntaxe générale précédente correspond en fait à 2 manières de définir une fonction en C : la première correspond à la définition du langage par K&R et a tendance à devenir obsolète; la seconde est celle recommandée par la normalisation du langage par l'ANSI, et que l'on désigne sous le vocable de "prototype". ORCA/C permet d'utiliser les 2 formes.

Ces 2 formes ont un impact sur la façon dont on déclare les paramètres de la fonction. Nous allons maintenant voir ces 2 formes.

Déclaration des paramètres traditionnelle

Dans cette forme, on déclare les paramètres en 2 temps : on liste d'abord leur nom dans la définition de la fonction, puis on indique leur types avant d'écrire le corps de la fonction. Un exemple permettra d'y voir plus clair :

```
type-résultat nom-fonction ( paramètre1, paramètre2 )
char paramètre1;
int paramètre2;
{
        corps-de-la-fonction
}
```

Le typage des paramètres a donc lieu sous la forme d'une déclaration de variables qui seront considérées locales à la fonction. On ne peut cependant déclarer que des variables correspondant aux paramètres, c'est à dire ayant le même nom. Bien entendu, il ne doit y avoir qu'une seule déclaration de variable pour chaque paramètre. On ne peut pas initialiser ces variables (encore heureux !), puisque ces variables prennent comme valeur initiale celle passée en paramètre, ni les déclarer autrement qu'en automatique ou dans un registre. En revanche, il est possible de ne pas déclarer de variable pour un paramètre, indiquant que celui-ci est du type "int"; encore une fois, cela ne me paraît pas être une bonne idée.

Cette syntaxe permet aussi de déclarer des types avant de commencer le corps de la fonction, ce que je ne peux que vous déconseiller.

La déclaration d'une fonction définie de cette manière ne permet pas de spécifier les paramètres qu'elle attend. Par conséquent, le compilateur ne peut pas vérifier la concordance des types, ni même le nombre de paramètres passés à la fonction lors de son appel. Ceci n'est pas sans poser de problèmes; en effet, l'une des sources d'erreur les plus fréquentes en C (et donc de plantage des programmes) vient du fait qu'un programme peut passer de mauvais paramètres ou en nombre incorrect, sans que le compilateur ne voit rien. De plus, les conversions de type effectuées automatiquement sont prédéfinies, et pas forcément en accord avec ce qui est attendu : par exemple, K&R spécifie que tous les entiers (char, short et long) sont convertis en "int" (ce qui dans le cas du GS revient à être un "short"), à moins que le nombre ne tienne pas sur 16 bits, ou qu'on a utilisé un cas explicite. Si la fonction attendait effectivement un "long", vous imaginez ce qui va se passer !

Sur le GS, ce type de problème se pose avec les fonctions de la boîte à outils qui sont déclarées de cette manière (car elles sont fournies par Apple, dont le compilateur APW C ne dispose que de cette forme), et il est malheureusement très facile de se tromper, par exemple, en passant un entier sur 16 bits là où la fonction attend un entier sur 32 bits.

Cette forme est donc en train de devenir obsolète, au fur et à mesure de l'apparition des compilateurs respectant la norme ANSI C. Le plus gros apport de la normalisation par rapport au C original défini par K&R se situe au niveau de la définition/déclaration des fonctions. La nouvelle forme de déclaration des paramètres est désignée par le terme de "prototype".

Déclaration d'un prototype

Un prototype permet de définir très précisement le type des paramètres dans la déclaration de la fonction, et par la suite permet au compilateur de contrôler que les paramètres passés lors de l'appel à la fonction concordent avec ceux qu'elle attend.
Le prototype peut être utilisé à la fois pour déclarer une fonction et pour la définir. En fait, cette méthode est identique à celle requise par Pascal.

Voyons comment on déclare un prototype sur un exemple :

```
type-résultat nom-fonction ( type-param1 param1, type-param2 param2 )
{
}
```

Le typage des paramètres est donc effectué dans l'entête de la fonction, et non plus à part, comme dans la méthode traditionnelle. Le compilateur peut donc vérifier lors de l'appel à cette fonction que les paramètres passés ont effectivement le bon type, et éventuellement faire les conversions appropriées, et non plus se baser sur des règles prédéfinies. Il en profite aussi pour vérifier qu'il y a bien le nombre de paramètres requis.

Lorsque la fonction n'attend pas de paramètres, on doit utiliser le type void comme seul et unique paramètre. Ainsi, le compilateur pourra interdire toute tentative d'appel à cette fonction qui passerait des paramètres.

L'utilisation d'un prototype oblige à déclarer une fonction avant de l'appeler, ce qui peut être fait si celle-ci est définie avant d'être utilisée, ou en la déclarant explicitement. Dans le cas contraire, le compilateur va enregistrer la fonction appelée comme utilisant l'ancienne syntaxe, et va signaler une erreur lorsqu'il va rencontrer la définition de cette fonction avec un prototype.

ORCA/C dispose d'une directive "lint" que j'ai décrite dans le précédent GS Infos (dans l'article concernant la programmation de la calculatrice), et sur laquelle je ne reviendrais pas aujourd'hui. Cette directive permet de forcer l'utilisation des prototypes, de s'assurer qu'une fonction appelée a bien été déclarée avant et que le résultat de la fonction a bien un type explicite. Je ne saurais assez vous recommander d'utiliser cette directive systématiquement; cela permet de se simplifier sacrément la vie. Le seul problème est qu'elle doit être spécifiée après l'inclusion éventuelle des fichiers header de la boîte à outils, autrement la déclaration des outils provoquerait systématiquement une erreur.

Vous aurez sans doute compris que je vous engage à n'utiliser que la forme prototypée de déclaration des fonctions. Le plus simple est même d'oublier qu'il existe une autre forme, d'autant plus qu'elle a disparu dans le langage C++. Notez d'ailleurs que le prototypage des fonctions en C est issu de C++.

Lorsqu'on déclare un prototype de fonction, il est possible de ne spécifier que le type des paramètres, sans leur donner de nom, comme par exemple :

```
type-résultat nom-fonction ( type-param1, type param2 );
```

Cette forme peut aussi être utilisée dans la définition, mais cela ne présente aucun intérêt.

Passage des paramètres

Jusqu'à présent, nous avons vu comment déclarer une fonction et les paramètres qu'elle attend. Nous n'avons cependant pas défini comment ces paramètres étaient passés lors de l'appel à une fonction, ni la syntaxe particulière indiquant ce mécanisme de passage.

En général, on distingue 2 mécanismes de passage d'un paramètre à une fonction :

- Le passage par "valeur" : ce terme spécifie que l'on passe la valeur de la variable associée dans la fonction appelante. En quelque sorte, le paramètre est une copie de la variable initiale. Ainsi, toute modification de cette valeur n'entraine aucun changement pour le contenu de la variable de départ.

- Le passage par "référence" ou par "adresse" : ici, on passe la variable elle-même et non son contenu. En général, cela se traduit par le passage de l'adresse de la variable, c'est à dire que le paramètre et la variable pointent sur le même emplacement mémoire. Par conséquent, toute modification du paramètre change directement le contenu de la variable.

Selon les langages, une seule ou les 2 méthodes de passage de paramètre sont offertes. Par exemple, Pascal dispose des 2 : le mot clé VAR précédent la déclaration d'un paramètre indique que celui-ci sera passé par référence; son abscence stipule qu'il sera passé par valeur.

C ne dispose que du passage par valeur, sauf pour les tableaux qui sont toujours passés par référence, afin d'éviter la recopie de l'ensemble du tableau, ce qui pourrait coûter très cher en termes de temps et de consommation mémoire.

Ceci ne signifie pourtant pas qu'il n'existe pas la possibilité pour une fonction de modifier une variable d'une autre fonction (qui serait normalement passée par référence), sans pour autant recourir à une variable globale.

Puisque le passage par référence consiste à passer l'adresse d'une variable, pourquoi ne pas passer explicitement cette adresse ? C'est la technique retenue en C, c'est à dire que le paramètre en question devra être déclaré comme un pointeur sur la zone voulue lors de la définition de la fonction, et que lors de l'appel à cette fonction, l'adresse de la variable sera passée (par valeur d'ailleurs !) et non la variable elle-même.

C dispose de l'opérateur "&" (dont nous n'avons pas encore parlé) pour obtenir l'adresse d'une variable, et de l'opérateur "*" pour déréférencer un pointeur. Notez qu'à cause de ce mécanisme, il est quasiment impossible d'écrire un programme C sans utiliser de pointeur. Histoire de se compliquer encore un peu plus la vie, lorsqu'on veut passer un pointeur par référence, on doit aussi passer son adresse, créant ainsi une double indirection. On peut ainsi arriver jusqu'à des pointeurs triples; rassurez-vous, je n'ai encore jamais vu de quadruple indirection. On s'en sort bien souvent en utilisant des définissant des types sur les pointeurs intermédiaires.

Pour cet article, je n'en dirai pas plus sur les pointeurs, car ce sera le sujet de la septième partie de l'initiation.

Attention ! le manuel d'ORCA/C comporte une erreur : il indique en effet qu'une structure ou une union (nous les traiterons dans le prochain article) sont aussi passées par référence, ce qui est faux ! Elles sont passées par valeur.

D'ailleurs, bien souvent pour éviter leur recopie (notamment lorsqu'elles commencent à avoir une certaine taille), on préfére les passer par référence explicitement, même si on n'a pas l'intention de les modifier.

On peut indiquer ce cas au compilateur en préfixant la déclaration du paramètre par le mot clé "const". Par exemple :

```
type-résultat nom-fonction ( const type-param1 *param1 );
```

Cette syntaxe peut aussi être utilisée lorsque le paramètre est un tableau, qui, je vous le rappelle, est toujours passé par référence, lorsqu'on n'a pas l'intention de le modifier.

Valeur retournée

Lorsqu'une fonction souhaite retourner une valeur à son appelant, elle doit utiliser l'instruction "return" suivie de l'expression permettant d'obtenir la valeur désirée.

Normalement, il n'est pas nécessaire de mettre la valeur (ou l'expression) retournée entre parenthèses, contrairement aux autres instructions de contrôle.

J'ai néanmoins pris cette habitude depuis que je programme en C. Ne vous étonnez donc pas si vous trouvez toujours des parenthèses pour les "return" dans mes programmes, et pas dans dans ceux que vous pourrez voir par ailleurs.

Si la fonction ne retourne pas de valeur, la fin de la fonction (c'est à dire du bloc correspondant) constitue un "return" implicite. C'est vrai aussi si la fonction est censée retourner une valeur, mais cela devrait être indiqué comme une erreur.

Les puristes de la programmation structurée disent qu'une fonction ne doit contenir qu'une seule entrée et qu'une seule sortie, c'est à dire qu'un seul "return" en fin de fonction. Le Pascal leur donne raison, puisqu'il n'y a pas moyen de faire autrement, sauf sous la forme d'une extension du langage, qui prend une forme différente selon les implémentations.

Je ne suis pas d'accord ! Bien entendu, il ne faut pas retourner à tort et à travers, mais il me semble préférable d'éliminer immédiatemment les cas pour lesquels les conditions d'appel de la fonction ne sont pas satisfaites, afin d'éviter une imbrication de tests, qui, à mon avis, alourdissent inutilement le code. C'est, par exemple, la méthode que j'ai employée dans la calculatrice, et elle ne me semble pas 'barbare' !

Sémantique de passage

C, contrairement à tous les autres langages, passe les paramètres dans l'ordre inverse de leur déclaration, c'est à dire de droite à gauche. En général, cela n'a pas d'importance, mais ca interdit cependant d'appeler une routine écrite dans un autre langage qui ne respecterait pas ces conventions. ORCA/C permet d'indiquer qu'une fonction doit utiliser les conventions de passage des paramètres des autres langages en préfixant la déclaration de la fonction par le mot clé "pascal".

Par exemple :

```
extern pascal type-résultat nom-fonction ( type-param1 param1, type-param2
param2 );
```

Bien que le mot clé soit "pascal", cela ne restreint pas le langage étranger à être le Pascal. Cette option doit aussi être utilisée lorsque la fonction C est appelée par de l'assembleur, par exemple par la boîte à outils, ou lorsqu'elle appelle de l'assembleur qui utilise les conventions d'appel de Pascal.

De plus, C est un langage qui fait la distinction entre minuscules et majuscules, ce qui n'est pas le cas des autres langages. Cette option permet de supprimer cette distinction pour les fonctions correspondantes.

Fonctions locales et globales

Vous aurez sans doute remarqué que j'ai utilisé le mot clé "extern" dans l'exemple précédent. En fait, cela n'est pas vraiment nécessaire, car il est présent implicitement dans toute déclaration de fonction. Dans le cas précis de l'exemple, il peut être utile de stipuler que cette fonction est localisée ailleurs, en plus du fait qu'elle utilise une sémantique différente.

Vous vous souvenez aussi que j'ai écrit plus haut qu'une fonction ne pouvait pas être imbriquée dans la définition d'une autre. Si vous connaissez le Pascal, vous savez qu'on utilise cette technique pour masquer une sous-routine aux niveaux supérieurs. Vous devez alors vous demander comment faire en C quelque chose de semblable, si jamais c'est possible ? Ca l'est ...

C est un langage modulaire. Le module pour C est le fichier source. Par conséquent, on va rassembler l'ensemble des fonctions utilisées de concert, dans un même fichier source. Certaines de ces fonctions peuvent être publiques (c'est à dire disponibles à l'extérieur du module), tandis que d'autres sont privées (réservées à l'usage exclusif des fonctions du module). Donc, ce qu'il nous faut, c'est un moyen de cacher les fonctions qui ne doivent pas être directement accédées depuis l'utilisateur de ce module.

Une fonction privée est préfixée par le mot clé "static", par exemple :

```
static type-résultat nom-fonction ( type-param1 );
```

Cette fonction est normalement appelable par toutes les autres fonctions du fichier source, et dans le cas d'ORCA/C par toutes les fonctions des fichiers sources inclus au moyen de la directive "append". En revanche, cette fonction ne sera marquée comme globale et sera donc inconnue pour le linker. Toute tentative d'appel depuis une fonction d'un autre module ne pourra être résolue, provoquant une erreur au moment du link.

Nombre de paramètres variable

C permet de façon standardisée, donc portable, de définir des fonctions ayant un nombre variable de paramètres. Par exemple, la fonction "printf" peut être écrite très facilement en C.

Cette méthode requiert l'utilisation d'un prototype. On indique que la fonction admet un nombre variable de paramètres en écrivant "..." à la place du type et du nom du paramètre.
En voici un exemple :

```
type-résultat nom-fonction ( type-param1 param1, ... )
```

Vous remarquerez que j'ai précisé un premier paramètre fixe. En fait, il s'agit d'une restriction de cette possibilité : une telle fonction doit en effet avoir au moins un paramètre fixe. De plus, les paramètres variables doivent être obligatoirement à la fin de la liste des paramètres. En général, le paramètre fixe indique le nombre de paramètres variables qui le suivent, ou, comme dans le cas de "printf", il permet de déterminer au cours de son traitement, les paramètres variables à exploiter.

L'accès à ces paramètres se fait par un ensemble de macros et de fonctions de la librairie standard, définies dans le fichier header "stdarg.h".
va_start permet de débuter la récupération des paramètres variables, va_arg est appelé successivement pour chacun des paramètres, et va_end est utilisé pour conclure la session. Je vous renvoie au manuel d'ORCA/C pour les explications concernant leur utilisation.

Macros

Dans la deuxième partie de l'initiation au C, je vous avais parlé du préprocesseur, et notamment de l'instruction "#define", utilisé pour définir des constantes.

Cette instruction est aussi utilisée pour définir ce que l'on appelle des "macros". D'un certain point de vue, une macro peut être vue comme une fonction. La différence principale est que le contenu de la macro sera substitué à la référence de la macro, au lieu de provoquer un appel à une fonction.

Cette possibilité permet donc d'améliorer les performances d'un programme tout en conservant un certain degré de lisibilité, par exemple lorsqu'une section de code est localisée dans une boucle exécutée un grand nombre de fois.

Les exemples les plus fréquents d'utilisation d'une macro concernent des fonctions relativement simples, mais très souvent employées, telles que abs() ou min() et max(). Pour que vous voyez bien ce que cela peut donner, quelques exemples :

```
#define abs(x)      ( (x) > 0 ? (x) : -(x) )
#define min(a,b)    ( (a) > (b) ? (b) : (a) )
```

Pourquoi tant de parenthèses ? Ce qu'il faut bien voir, c'est qu'une macro procède par substitution de ses arguments. Par conséquent, comme toute expression peut être passée en paramètre, il est préférable de l'encadrer par des parenthèses, pour éviter toute erreur d'évaluation.

En revanche, l'utilisation d'une macro n'est en général pas complètement transparente, et présente, comme c'est le cas ci-dessus, des possibilités d'effets de bord. Par exemple, les macros précédentes interdisent d'utiliser les opérateurs d'incrémentation et de décrementation, car les paramètres sont évalués 2 fois. Ainsi, si on effectue un abs(x++), on obtiendra la valeur absolue de x+1, ce qui n'est certainement pas ce qui est désiré; au cas où vous ne voyez pas pourquoi, faites la substitution à la main.

Malgré les restrictions précédentes, le concept de macro est très puissant.
Comme tous les concepts sophistiqués, il faut l'utiliser à bon escient. Le fait qu'une macro procède par substitution de ses arguments, on peut notamment passer en paramètre des mots clefs du langage.
Par exemple, la macro :

```
#define new(type)    (type *)   malloc ( sizeof(type) );
```

réalise un équivalent de l'instruction new() de Pascal. Si on l'appelle par new(long), un entier long sera alloué, et un pointeur sur la mémoire qu'il occupe sera retourné.

Récursivité

Un peu plus haut, j'ai évoqué le fait qu'une fonction en C pouvait s'appeler elle-même; on désigne ce mécanisme sous le nom de "récursivité". C'est un concept qui fait un peu peur lorsqu'on est débutant. Je vais tenter dans la suite de cet article de vous expliquer en quoi elle consiste, et dans quels contextes elle peut être utilisée.

La récursivité est un concept très important dans la définition et la mise en œuvre des algorithmes. Cela est très utile pour la résolution de certains problèmes, pour lesquels une solution itérative (donc basée sur des instructions du type for ou while) est beaucoup plus difficile à réaliser.

Il s'agit notamment des problèmes dans lesquels la solution complète est construite à partir de solutions partielles à des problèmes plus simples, ainsi que de la théorie des jeux lorsque le programme analyse la situation sur plusieurs niveaux de profondeur, ou plus généralement des programmes basés sur des structures arborescentes, comme par exemple les compilateurs.

Cependant, comme tout concept puissant, la récursivité doit être utilisée avec précaution, car elle peut produire des programmes difficiles à mettre au point et moins performants qu'une implémentation itérative (l'appel à une fonction coûte souvent plus cher qu'une simple boucle, bien que pas trop encore sur le GS); il existe d'ailleurs des techniques permettant de supprimer la récursivité, utilisant principalement le type pile décrit dans dans le précédent GS Infos.

Lorsqu'une fonction s'appelle elle-même (on dit qu'elle s'auto-référence), la récursivité mise en œuvre est dite directe. Il existe aussi des cas ou une fonction X fait référence à une fonction Y qui à son tour référence à la fonction X : on parle alors de récursivité indirecte. Notez que le nombre d'indirections, c'est à dire de fonctions intermédiaires, peut être quelconque.

Une des difficultés des techniques récursives est de définir la condition d'arrêt de la récursivité, c'est à dire qu'une fonction récursive devra toujours déterminer un cas élémentaire pour lequel elle a une réponse immédiate, sans devoir se rappeler elle-même pour traiter ce cas; ceci permet alors de retourner de l'ensemble des appels intermédiaires provoqués par la récursivité. Une récursivité infinie (donc sans condition d'arrêt) se soldera toujours par un débordement de pile et un plantage assuré; c'est pourquoi il s'agit vraiment d'un concept difficile à maîtriser qu'il faut de plus utiliser quand c'est nécessaire, pour éviter de consommer inutilement la ressource limitée qu'est la pile de l'ordinateur.

La structure de données pile décrite dans le précédent numéro de GS Infos accompagne souvent les algorithmes dont on a supprimé la récursivité de manière à imiter assez fidèlement le comportement de l'ordinateur lors d'un appel récursif : un nouvel appel d'une fonction récursive provoque en effet un nouveau passage de paramètres et la création d'un nouvel espace pour les variables locales qui sont en général allouées sur la pile. La notion de récursivité implique donc de pouvoir gérer des variables locales indépendantes d'un appel à l'autre de la fonction. Tous les langages ne sont donc pas récursifs, notamment les anciens comme Fortran ou Cobol; heureusement C et Pascal le sont, ainsi que l'assembleur si comme pour le reste on fait tout soi-même.

Prenons un exemple concret; la fonction mathématique factorielle se définit de la façon suivante :

```
n! = 1              si n = 0
n! = n * (n - 1)!   si n > 0
```

On a bien ici une fonction récursive puisque n! est défini en fonction de (n-1)! Par exemple, en appliquant cette formule, 3! se calcule ainsi :

$$3! = 3*(2!) = 3*(2*(1!)) = 3*(2*(1*(0!))) = 3*(2*(1*(1))) = 6$$

On voit bien que cette définition n'est pas circulaire, n'entraînant donc pas une récursivité infinie : lorsque la fonction s'auto-référence, la valeur de son argument diminue de 1 à chaque fois; on a de plus une condition d'arrêt sans auto-référence (0! = 1) permettant de stopper la série d'appels.

Cette fonction peut se programmer directement telle quelle, par exemple :

```
short factorielle ( short n )
{
     if ( n == 0 )
         return ( 1 );
    else
          return ( n * factorielle ( n - 1 ) );
}
```

La fonction factorielle ci-dessus emploie ce que l'on appelle une récursivité finale (en anglais tail recursion). C'est à dire que la dernière instruction de la fonction contient la seule auto-référence à la fonction. Ce type de récursivité peut toujours être supprimé et doit l'être dans la plupart des cas, car elle possède une équivalence itérative directe. Pour le cas de la factorielle, la fonction itérative est :

```
short factorielle ( short n )
{
     short i, fact;

     fact = 1;
     if ( n == 0 )
         return ( 1 );
    else
         for ( i = 2; i <= n; i++ )
             fact = fact * i;
      return (fact );
}
```

Les algorithmes récursifs sont souvent utilisés lorsque la solution d'un problème est établie en décomposant le problème en des problèmes plus simples par raffinements successifs, jusqu'à obtenir un cas suffisamment simple pour avoir une solution immédiate. La récursivité finale correspond ainsi à la résolution du cas n à partir du cas n-1.

Plus généralement, une fonction récursive est amenée à recombiner les solutions de plusieurs sous-problèmes (effectuant ainsi plusieurs appels récursifs) avant de générer la solution finale. Cette technique est désignée sous l'expression diviser pour régner (en anglais divide and conquer) ou parfois sous la forme moins militaire mais plus adaptée à l'informatique de diviser pour résoudre.

Prenons pour exemple le problème classique des tours de Hanoï : il s'agit d'un jeu consistant en 3 aiguilles et un ensemble de disques de tailles décroissantes (en partant du haut) que l'on a empilé sur la première aiguille; le but du jeu est de déplacer ces disques sur la troisième aiguille avec la double contrainte de ne déplacer qu'un seul disque à la fois et de ne placer un disque que sur un disque de plus grande taille; un disque peut cependant être déplacé de n'importe quelle aiguille vers n'importe quelle autre.

Si l'on n'a qu'un seul disque, la solution est immédiate : on déplace ce disque de l'aiguille A vers l'aiguille C. Si l'on a 2 disques, c'est à peine plus compliqué : on déplace le premier disque de A vers B, puis le deuxième de A vers C, et enfin le premier disque à nouveau de B vers C.

A partir de 3 disques, cela commence à se compliquer sérieusement. Pour trouver une solution générale au problème, il faut essayer de trouver un motif dans les solutions à 1 et 2 disques.

On s'aperçoit alors du principe suivant : pour pouvoir déplacer le plus grand disque de A vers C, il faut d'abord déplacer les disques plus petits qui sont au dessus sur l'aiguille B. Une fois que cela a été effectué, il faut déplacer la pile qui est maintenant sur B vers C. En réexprimant ce mécanisme de façon récursive, cela donne :

```
si nombre de disques > 0 alors
    déplacer le nombre - 1 de disques plus petits de A vers B
    déplacer le plus grand disque de A vers C
    déplacer le nombre - 1 de disques plus petits de B vers C
```

Par exemple, pour 3 disques, l'algorithme permet de déplacer les 2 petits disques de A vers B, puis le grand disque de A vers C, puis les 2 petits disques de B vers C. En décomposant encore la première et la dernière étapes de façon récursive, on s'aperçoit que pour déplacer les 2 petits disques de A vers B, il faut d'abord déplacer le plus petit des disques sur C, puis l'autre sur B, et enfin le petit disque que l'on a mis provisoirement sur C, sur B.

L'implémentation de cet algorithme est extrêmement courte, par exemple en C :

```
void hanoi ( short n, char a, char b, char c )
{
    if ( n > 0 ) {
        hanoi ( n-1, a, c, b );
        printf ( "Déplacement du disque %c vers %c\n", a, c );
        hanoi ( n-1, b, a, c );
    }
}
```

Cette fonction peut alors s'appeler ainsi : hanoi (3, 'a', 'b', 'c');

Vous trouverez une implémentation complète de cette fonction dans le programme Hanoi1.cc.

La suppression de la récursivité de cette fonction, bien que possible (en utilisant la structure de données pile), rend la fonction beaucoup plus incompréhensible (sans parler du fait que le code est nettement plus conséquent), alors que la forme ci-dessus est particulièrement élégante, comme vous pourrez le constater dans le programme Hanoi2.cc.

C'est typiquement le cas où la récursivité montre toute sa puissance et sa simplicité d'écriture, à défaut de pouvoir suivre facilement le déroulement des opérations.

Pour le fun, j'ai aussi écrit une version graphique que vous trouverez dans Hanoi3.cc. Cette version montre visuellement le déplacement des disques.

L'animation est simplifiée à l'extrême, mais pour un cas aussi simple que celui-ci, elle suffit largement.

Ces 3 programmes (ainsi que mes autres productions) constituent un bon ensemble d'exemples de fonctions.

Utilisation de la mémoire par un programme C

Sur le GS, qui est une machine 16 bits, l'ensemble des fonctions du programme, auxquelles s'ajoutent celles de la librairie C, appelées soit explicitement, soit implicitement, doivent occuper normalement au maximum 64 Ko de mémoire. Dans la plupart des cas, c'est largement suffisant. Toutefois, si vous veniez à écrire un programme énorme, nécessitant plus de 64 Ko de code, ORCA/C dispose de l'instruction "segment" qui permet de regrouper toutes les définitions de fonctions suivant son utilisation dans un autre segment, permettant ainsi de s'affranchir de la limite des 64 Ko.

On peut utiliser autant de fois que nécessaire cette instruction, créant autant de segments. On peut même regrouper des fonctions qui ne suivent pas dans le source, ou qui sont localisées dans un autre fichier source, dans un même segment, dès lors qu'on utilise le même nom de segment. Un segment peut aussi être dynamique, c'est à dire n'être chargé en mémoire que lorsqu'on appelle l'une des fonctions qu'il contient. Pour cela, on précise l'option "dynamic" lorsqu'on écrit l'instruction "segment". Avec ORCA/Pascal, les 2 directives "segment" et "dynamic" permettent de réaliser la même chose.

Dans la situation standard, les données globales sont stockées dans le même segment que le code, ce qui peut être aussi la cause de la saturation des 64 Ko. ORCA/C et ORCA/Pascal disposent de la directive "memorymodel" permettant de générer un segment de données spécifiques, indépendant de celui dans lequel est stocké le code. En fait, lorsqu'on utilise le modèle "large memory", ORCA/C et ORCA/Pascal génèrent 2 segments de données, l'un de 64 Ko pour les variables globales scalaires, et un de taille non limitée pour les tableaux.

Cette directive présente l'inconvénient de ralentir l'accès aux tableaux et de nécessiter l'emploi d'une autre librairie (ORCAGLIB) au lieu de la librairie standard ORCALIB. En revanche, la librairie Pascal PASLIB fonctionne avec les 2 modèles.

Je pense que l'on n'a jamais besoin d'utiliser cette directive, car il n'est pas raisonnable de déclarer des tableaux statiques de grande taille. En effet, une telle déclaration est souvent faite en prévision de la manipulation d'un grand volume de données. Or, dans la plupart des cas, on n'aura qu'à gérer une faible quantité de données, ce qui a pour conséquence de gâcher une partie de la mémoire, et d'éventuellement interdire l'accès simultané à d'autres programmes (accessoires ...).

Il est préférable d'allouer dynamiquement la mémoire nécessaire au cas qui se présente, ce qui ne nécessite pas d'utiliser la directive "memorymodel" (même si on alloue plus de 64 Ko, contrairement d'ailleurs à ce qui est indiqué dans le manuel), et permet de s'adapter à toutes les situations. Ainsi, si la mémoire disponible ne permet pas de traiter un volume de données trop important, le programme reste utilisable avec une quantité plus faible.

En fait, la taille du code et des données pose rarement un problème (à moins que vous ne réécriviez AppleWorks GS :-). La véritable limitation de mémoire se situe au niveau de la pile. En effet, celle-ci doit obligatoirement se situer dans les 64 premiers Ko de la mémoire (que l'on appelle le banc 0).

Or, pour assurer la compatibilité avec les Apple II 8 bits, toute cette mémoire n'est pas disponible; de plus, GS/OS et le loader en occupent une bonne partie. Si bien qu'il n'y a environ que 40 Ko qui sont utilisables.

Lorsqu'on lance un programme (par exemple, par l'intermédiaire du Finder™ ou du shell ORCA), GS/OS alloue automatiquement une pile de 4 Ko (sauf si on spécifie explicitement une autre taille). Le problème est que les programmes créés par ORCA/C et ORCA/Pascal n'utilisent pas cette pile; à la place, le code de démarrage situé dans le ".root" alloue une nouvelle pile de 8 Ko par défaut. C'est à dire que chaque programme C ou Pascal consomme 12 Ko de mémoire dans le banc 0. Il peut donc être impossible de pouvoir lancer un programme avec d'autres programmes déjà chargés, par exemple sous MultiSwitch, alors qu'il y a encore plein de mémoire disponible, uniquement parce que le système ne peut pas allouer la place demandée pour la pile.

La perte des 4 Ko de la pile allouée automatiquement par GS/OS peut être facilement annulée en linkant avec un segment de type "stack/direct page" qui doit être écrit en assembleur; en voici un exemple :

```
StackDP  START  ~StackDP        ; on donne un nom au load segment
           KIND   $12              ; crée un segment de type stack/direct-page
           DS     256              ; taille minimale de la pile = 1 page
         END
```

Pour vous éviter de la frappe, vous trouverez le source et l'objet de ce segment sur votre disquette GS Infos. Notez qu'on ne peut pas créer un segment de ce type de moins d'une page. Enfin, c'est toujours mieux que ce que font les compilateurs ORCA...

Contrairement à ce qui est écrit dans les manuels des compilateurs ORCA, il n'est que rarement nécessaire de disposer de 8 Ko de pile : ce besoin s'applique essentiellement aux programmes récursifs (et encore, cela dépend de la profondeur attendue de la récursivité), éventuellement aux programmes en mode bureau (la boîte à outils requiert pas mal d'espace sur la pile), et aux programmes déclarant des tableaux en variables locales, donc sur la pile. Dans ce dernier cas, on peut éviter d'utiliser la pile pour les grosses variables locales en précédant la déclaration du mot clé "static", si bien que ces variables se retrouvent allouées dans le même espace que les variables globales; elles restent néanmoins locales, sauf que leur valeur est préservée d'un appel à l'autre, mais on n'est pas obligé d'en tenir compte. Le seul cas où cela peut poser un problème, c'est lorsque la fonction est récursive, mais je ne connais pas d'exemple de telle fonction nécessitant de manipuler de gros volumes de données. Notez qu'en Pascal, on ne dispose pas de cette possibilité d'avoir des variables locales n'utilisant pas la pile.

Une fois que l'on a fait en sorte de ne pas consommer excessivement la pile, il suffit d'indiquer au compilateur de modérer sa boulimie avec la directive "stacksize". Il est raisonnable de définir une pile de 4 Ko pour un programme moyen et de descendre à 2 Ko pour un petit programme, ou un utilitaire pour le shell.

J'avoue n'avoir pas utilisé cette directive dans aucun des programmes que je vous ai fournis : la raison en est qu'il s'agit soit d'exemples qui ne sont pas destinés à être utilisés de façon intensive, soit que le programme n'en a pas besoin; par exemple le NDA calculatrice (comme tous les NDA) utilise la pile de l'application hôte. D'ailleurs dans le cas des accessoires, il faut consommer la pile avec le plus de modération possible, car on ne sait pas réellement combien de place est disponible dans l'application qui nous héberge; si l'on est un gros consommateur, il faut envisager de s'allouer sa propre pile au moment de l'ouverture.

Déclaration des prototypes

Lorsqu'on utilise le mécanisme de prototype, il est d'usage de déclarer toutes les fonctions constituant le programme dans un fichier "header" (de suffixe ".h"), qu'elles soient appelées avant ou après leur définition. Ainsi, on est sûr que le compilateur pourra détecter toutes les erreurs éventuelles. C'est ce qui est fait pour les fonctions de la librairie C, dans chacun des fichiers header correspondant à la famille d'appartenance de ces fonctions.

Je vous rappelle que la directive "lint" permettra de vérifier que toutes les fonctions définies dans un programme ont bien été déclarées. Ainsi, en les déclarant dans un fichier "header", on s'assurera d'un maximum de cohérence du programme. Ce sont en effet les erreurs de déclaration/définition/appel de fonction qui sont à la source de pratiquement tous les plantages des programmes C, puisqu'elles provoquent en général une corruption de la pile qui, sur le GS, est fatale.

Paramètres

J'avais volontairement omis de définir toute la terminologie se rapportant aux paramètres. Cependant, je me suis dit après coup que vous aurez forcément à faire à ces termes dans la littérature. Alors, voici un petit glossaire :

• On emploie le terme "paramètre formel" ou "argument" pour désigner le type et la position d'un paramètre lors de la définition d'une fonction. Ceci signifie simplement que cet argument sera substitué par le contenu des variables correspondantes de la fonction appelante lors de l'appel.

• Les variables passées à une fonction lors de l'appel à cette fonction sont désignées sous le nom de "paramètres actuels" ou plus simplement de "paramètres". En fait, il n'y a pas grand chose d'autre à dire de plus, par rapport à ce que nous avons vu dans le précédent article ...

Dans la littérature, les 2 termes "paramètre" et "argument" sont parfois utilisés de façon interchangeable ou à l'inverse de la définition précédente.
J'espère que vous voyez bien la différence entre les 2, et que cela ne vous pertubera pas outre mesure.

Comme en C, les arguments sont passés aux fonctions de droite à gauche, il peut s'ensuivre des effets de bords involontaires. Supposez par exemple que vous appeliez une fonction "f(j,k)" ainsi "f(i,i++)"; si "i" valait "1" avant l'appel, les valeurs des paramètres "j" et "k" dans la fonction "f" seront respectivement "2" et "1", ce qui n'est certainement pas ce que vous espériez !
Bien entendu, cet exemple est simplifié à l'extrême, mais ce type de surprise peut vraiment se produire dans des situations réelles.

Allez, histoire de s'amuser un peu, je vais donner du pain aux détracteurs de C ;-) Vous vous souvenez peut-être que C dispose d'un opérateur "," dit de "séquence"; je vous avais dit à l'époque qu'on l'utilisait essentiellement avec l'instruction "for". Or, la "," est aussi employée pour séparer les paramètres d'une fonction. C fait la distinction entre les 2 cas en fonction du contexte. Toutefois, chaque argument d'une fonction pouvant être en fait une expression, on pourrait envisager d'utiliser l'opérateur "," dans ladite expression. Comment le compilateur fait-il la différence ? Eh bien, il suffit de mettre l'expression en question entre "()". Voyons sur un exemple, histoire d'éclaircir un peu ce charabia :
"f(z,(z=y+x,w=z-t))". Hum ... est-ce que c'est vraiment plus clair ? En fait l'exemple est simple : les 2 paramètres sont "y+x" et "y+x-t", sauf que l'on a sauvé dans les variables "z" et "w" les résultats intermédiaires; rappelez-vous que les paramètres sont évalués de droite à gauche, d'où la valeur du premier. Je vous ai montré cet exemple juste pour le fun; dans la pratique, on n'a jamais besoin de faire des choses pareilles ...

L'ordre d'évaluation que je vous ai indiqué est le plus souvent l'ordre utilisé, mais la définition du langage C ne le garantit pas, c'est à dire que la supposition de la valeur de "z" dans l'exemple précédent peut s'avérer fausse avec d'autres machines ou d'autres compilateurs; alors, évitez tout effet de bord ! En tout cas, vous pouvez imaginer combien l'écriture d'un compilateur C est autrement plus compliquée que celle d'un compilateur Pascal.

Paramètres variables

La possibilité d'avoir des paramètres variables, ainsi que l'obligation d'avoir au moins un paramètre fixe au début de la liste des arguments sont liés au fait que les paramètres sont traités de droite à gauche. La plupart des machines et des compilateurs utilisent la pile pour le passage des paramètres. Sur presque tous les ordinateurs, les adresses dans la pile sont décroissantes. Donc le paramètre le plus à gauche de la liste se

retrouve empilé le dernier et est donc le plus proche du pointeur de pile (si l'on fait abstraction de l'adresse de retour). On peut donc facilement obtenir son adresse et de là, accéder à tous les autres paramètres avec les macros "va_start" et "va_arg".

La dernière fois, je vous avais renvoyé à la doc pour de plus amples détails.
Je vais quand même vous donner un exemple autre de celui qu'elle donne :

```
short max ( short n, ... )          /* calcul du max de n valeurs */
{
      va_list argp;                 /* déclaration ptr liste paramètres variables */
      short i, m;

       va_start ( argp, n );        /* initialisation du ptr sur paramètres variables */
       m = va_arg ( argp, short );  /* 1ère valeur donne le max provisoire */
       while ( --n > 0 ) {
            i = va_arg ( argp, short );  /* valeurs suivantes : doivent être des short */
          if ( i > m )
             m = i;
     }
      va_end ( argp );              /* nettoyage de la pile */
       return ( m );
}
```

Le seul inconvénient de ce mécanisme est qu'il interdit tout contrôle de type et de quantité des paramètres par le compilateur. Par exemple dans l'exemple ci-dessus, rien n'interdit d'appeler "max" avec des entiers longs ou des réels. Il faut donc utiliser cette possibilité avec prudence. D'un autre côté, c'est grâce à ce mécanisme que C dispose d'une fonction telle que "printf" qui n'a pas besoin d'utiliser d'astuces particulières, et peut donc être réécrite facilement, par exemple pour appeler QuickDraw. En Pascal, il est totalement impossible d'écrire une fonction similaire à "write".

Appel d'une fonction

En C, l'appel d'une fonction est effectué en indiquant le nom de la fonction suivi de la liste des arguments entre parenthèses. Comme en Pascal, il n'y a pas de mot clé précisant que l'on effectue un appel de fonction. Dans les anciens langages, tels Fortran ou Cobol, l'appel à une sous-routine se fait en utilisant l'instruction "CALL".

Il est très important de noter que C se sert des parenthèses pour distinguer l'appel d'une fonction de la référence à une variable quelconque, notamment à cause de l'historique du langage. Je vous rappelle en effet qu'on n'était pas obligé de déclarer une fonction avant de l'appeler; c'était le seul moyen pour le compilateur de faire la distinction.

Par conséquent, les parenthèses sont obligatoires, même lorsque la fonction n'a pas de paramètres. L'appel à une telle fonction sera alors de la forme "nom-fonction()".

Retour d'une fonction

L'instruction "return", que nous avons vue dans le précédent numéro, permet de retourner à la fonction appelante une valeur correspondant à l'évaluation de l'expression associée à l'instruction. La syntaxe est alors "return (expression)". Le résultat de l'évaluation de cette expression doit bien entendu avoir le même type que celui avec lequel la fonction a été déclarée.
Si l'expression est omise, la valeur retournée sera quelconque et ne donnera pas le résultat escompté; en général ce sera la valeur présente dans le registre "A" au moment du "return".

Si la fonction a été déclarée "void", c'est à dire qu'elle ne retourne rien et se comporte donc comme une procédure Pascal, l'instruction "return" est alors utilisée telle quelle, sans préciser d'expression, sinon une erreur sera signalée.

La fonction effectuera un "return" implicite lorsque son exécution atteindra l'"}" fermant le bloc principal de la fonction. Ce cas ne s'applique que lorsque la fonction ne retourne aucune valeur. Le comportement étant identique à celui du "return" simple, si la fonction doit retourner une valeur, celle ci sera quelconque (en fait ce sera la valeur du registre "A" à ce moment), et provoquera sans doute une erreur de fonctionnement du programme.

La fonction "main"

Dans mon précédent article, je vous ai indiqué que la fonction "main" était la première exécutée d'un programme C. Etant une fonction comme une autre, elle accepte des paramètres, sauf qu'ils sont prédéfinis par le langage. On les désigne habituellement sous les noms "argc" et "argv"; le deuxième est un tableau de chaînes de caractères, chacune de ces chaînes correspond aux mots listés après le nom du programme (que l'on retrouve d'ailleurs comme première chaîne du tableau); le premier paramètre est le nombre d'éléments du tableau. Ces paramètres n'ont un sens que lorsque le programme est lancé depuis le shell et est de type EXE ($B5). Lorsqu'un programme est lancé depuis le Finder™ ou Prosel-16 ou depuis le shell si il est de type S16 ($B3), argc et argv sont inutilisables; argc vaudra 0 et argv NULL.

Si vous vous rappelez de mon premier article, je vous avais présenté un programme qui imitait la commande "echo" du shell. En voici une nouvelle version bénéficiant de ce que nous avons appris depuis :

```
#include <stdio.h>
void main ( short argc, char *argv[] )
{
        short i;

        for ( i = 1; i < argc; i++ )
                printf ( "%s ", argv[i] );
        printf ( "\n" );
}
```

Si je compile et linke ce programme sous le nom "msg", et que je tape la commande "msg Bonjour !", il me répondra par "Bonjour !". Si "i" avait démarré à Ø, j'aurais aussi eu le nom du programme affiché, éventuellement avec le chemin si je l'avais précisé.

Un argument d'un programme peut comprendre plusieurs mots, si ceux-ci ont été entourés par des """. Toutefois, l'argument "argv[n]" correspondant ne contiendra pas les """.

Tout ceci est réalisé par le code du fichier ".root" généré par le compilateur ORCA/C. Ce code initialise l'environnement, c'est à dire qu'il alloue la pile comme indiqué au début de cet article, décode la ligne de commande dans les paramètres argc et argv, ouvre les fichiers standard de C, "stdin", "stdout" et "stderr", en tenant compte des redirections éventuelles, et appelle la fonction "main".

Lorsque la fonction "main" retourne à son appelant soit explicitement avec l'instruction "return", soit implicitement en atteignant l'} finale, le code du fichier ".root" reprend la main pour faire le ménage : fermer les fichiers, libérer la mémoire, et quitter vers le shell ou un autre lanceur de programmes. La fonction "main" peut retourner une valeur qui sera transmise au shell dans la variable d'environnement "{Status}" : par convention, on retourne Ø si tout c'est bien passé, et une autre valeur en cas d'erreur, la valeur "-1" étant utilisée pour indiquer une erreur générique.

Un programme peut se débrancher directement au code contenu dans le fichier ".root" suivant l'exécution de la fonction "main". Il dispose pour cela des fonctions de la librairie C "exit", "_exit" et "abort". La première effectue le même travail que lorsqu'un return est effectué depuis la fonction main. Le second rend la main au shell sans faire aucun ménage (toutefois, le shell ferme aussi les fichiers et libère la mémoire). Dans les 2 cas, on peut passer une valeur qui sera affectée à la variable shell "{Status}". Dans le cas d'ORCA/C, la troisième fonction est identique à _exit(-1).

Pour pouvoir bénéficier pleinement de ces fonctions d'arrêt, un programme peut aussi enregistrer une ou plusieurs fonctions qui seront exécutées avant de rendre la main au code du fichier ".root", grâce à la fonction de la librairie C "atexit()". On désigne de telles fonctions sous le vocable de "exit handler". Elles permettent de faire du ménage en plus de celui fait normalement par la librairie C via le fichier ".root", et donc de terminer le plus proprement possible.

Chapitre 6

Les Structures

Dans la deuxième partie de cette initiation (dans le numéro 16 de GS Infos, il y a déjà plus d'un an !), nous avons étudié les types de base, ainsi que la déclaration de variables de ces types. Les types de base correspondent en général (et cela est particulièrement vrai pour le C) aux objets manipulables directement par le micro-processeur. Ils permettent de représenter la plupart des données élémentaires requises par les problèmes les plus simples.

Toutefois, même des problèmes relativement simples nécessitent de manipuler des données plus complexes que ce que peuvent offrir les types de base d'un langage. Il est notamment souvent nécessaire de gérer des quantités de données qui ne peuvent être représentées par les types de base.

C, comme la plupart des autres langages, dispose de 2 constructions permettant d'établir de nouveaux types de données que l'on dit "composites" ou "composées" : les "tableaux" et les "structures", ces dernières correspondant aux "record" de Pascal.

Les tableaux permettent de regrouper sous un même nom de variable un ensemble de données de même type, tandis qu'une structure sera utilisée pour représenter une donnée comprenant des éléments qui peuvent être de différents types, en permettant de les manipuler comme une seule entité.

Contrairement à ce que j'ai pu annoncer dans les numéros précédents, nous n'étudierons les tableaux que dans le prochain article. Il est en effet difficile de les traiter de façon détaillée sans aborder les pointeurs, ce que nous ferons la prochaine fois.

Donc, dans cet article, nous allons présenter la définition d'une structure en C, ainsi qu'une partie des opérateurs spécifiques aux structures (les autres font appel aux pointeurs et seront donc traités à ce moment).

Définition d'une structure

Comme il a été dit précédemment, une structure permet de regrouper sous un même nom, un ensemble de données de types divers, qui peuvent être de base ou composites (c'est à dire que cela peut être d'autres structures ou des tableaux).

Voyons tout de suite un exemple pour clarifier les choses :

```
struct personne {
        string nom;
        string prenom;
       short   age;
        string numero_ss;
};
```

Cet exemple définit une structure (le mot clef "struct" indique au compilateur qu'on va effectuer une telle définition) intitulée "personne" et comprenant un certain nombre de rubriques : le nom, le prénom, l'âge et le numéro de sécurité sociale de cette personne. Notez que j'ai utilisé un type "string" qui n'existe pas en C; pour cet article, nous supposerons qu'il a été défini quelque part comme étant par exemple un tableau de caractères; en fait cela n'a pas d'importance.

La définition d'une structure consiste donc à lister les déclarations des données constituant cette structure. Ces données sont appelées des "champs" ("fields" en anglais) ou encore des "membres". Chaque champ est une variable qui est donc nommée (il existe une exception que nous verrons plus loin) et typée; on peut utiliser tous les types disponibles en C, ainsi que des types définis précédemment dans le programme, à l'exception du type "void". On ne peut pas non plus définir ni déclarer une fonction à l'intérieur d'une structure (mais on peut le faire en C++, c'est d'ailleurs le principe de base d'un langage orienté objet); en revanche, on peut déclarer un pointeur sur une fonction, comme nous aurons l'occasion de le voir dans un prochain article.

Un membre ne peut pas non plus être du type de la structure (c'est à dire que la structure ne peut pas s'auto-référencer, sinon cela créerait une récursivité infinie et rendrait impossible le calcul de sa taille), sauf si c'est un pointeur, ce qui permet de créer des structures chaînées. Nous y reviendrons dans un prochain article.

Chacun des membres est stocké en mémoire dans le même ordre que celui dans lequel il a été listé. Avec la structure précédente, on trouvera d'abord le nom puis le prénom, l'âge et enfin le numéro de sécurité sociale.

Le nom que l'on donne à la structure afin de l'identifier ultérieurement est désigné par le terme de "tag" (intraduisible en français, j'ai cependant utilisé dans la suite de cet article le terme "identifiant"); on s'en sert dans la suite du programme pour déclarer des variables du type de cette structure. Par exemple :

```
struct personne moi, lui;
```

Cependant, ce "tag" est optionnel; on peut tout à fait définir la structure et déclarer des variables du type ainsi défini en même temps, comme par exemple :

```
struct  {
        short  jour;
        string mois;
       short  an;
} date, hier;
```

Cette structure déclare 2 variables "date" et "hier" composées d'un "jour" et d'un "an" numériques et d'un "mois" alphabétique. Cette syntaxe est équivalente à celle que l'on utilise pour déclarer une variable simple, et va donc demander au compilateur d'allouer la mémoire nécessaire au stockage des variables déclarées.

En revanche, la définition d'une structure avec simplement son identifiant (son "tag") correspond à la description d'un modèle, et ne provoque donc pas d'allocation de mémoire pour le représenter (il n'est défini qu'au niveau du compilateur). Ce n'est qu'en déclarant des variables du type de cette structure qu'on allouera la mémoire nécessaire à leur stockage.

On peut aussi bien entendu donner un nom à la structure et déclarer des variables en même temps, auquel cas on allouera la mémoire nécessaire aux variables, et on se donnera la possibilité de déclarer de nouvelles variables du type de cette structure. Il est tout aussi évident qu'il faut soit donner un identifiant à la structure, soit déclarer des variables.

La syntaxe de l'exemple présente toutefois un inconvénient : puisque la structure n'est pas nommée, on ne pourra pas déclarer de nouvelles variables de ce type, sans redéfinir le contenu de la structure, ce qui n'est pas une très bonne idée pour la clarté et la maintenabilité du programme. De plus, certains compilateurs peuvent refuser une telle redéfinition.

Pour pallier à cet inconvénient et éviter de repréciser le mot clef "struct" à chaque fois que l'on veut déclarer une variable, on utilise très souvent le mot clef "typedef" (permettant de définir un nouveau type) avec les structures. Notre premier exemple pourrait alors être :

```
typedef struct {
        string nom;
        string prenom;
        short  age;
        string numero_ss;
} personne;
```

Notez que le nom du type est donné à la position de la variable et non pas à celle du nom de la structure. Cela vient du fait que "typedef" est considéré par le compilateur comme une "classe de stockage"; il se réfère donc à une variable qui est alors considérée comme un nom de type. La raison de cette classification est que "typedef" ne définit pas réellement de type, mais permet d'associer un synonyme (ou une abréviation à une définition complexe) à un type défini par ailleurs. Dans notre exemple, le type est défini par "struct", "typedef" servant uniquement à donner le synonyme "personne" au type "struct" (qui n'est pas nommé dans l'exemple). En quelque sorte, "typedef" est équivalent au "#define", excepté qu'il est traité par le compilateur et non par le préprocesseur.

Une fois la structure définie, on déclare une variable de ce type ainsi :

```
personne moi, lui;
```

Il n'est alors plus nécessaire d'indiquer qu'il s'agit d'une structure, "typedef" ayant associé le nouveau type "personne" à la définition de la structure.

On peut bien entendu imbriquer les structures; par exemple, pour notre personne, on pourrait avoir :

```
typedef struct {
        string nom;
        string prenom;
        date   naissance;
} personne;
```

en ayant préalablement défini le type date, par exemple :

```
typedef struct {
       short jour;
       string mois;
       short an;
} date;
```

Si la structure référencée ne sert qu'à la structure y faisant appel, on peut la définir à l'intérieur de la première, ce qui donne en reprenant l'exemple précédent :

```
typedef struct {
        string nom;
        string prenom;
        struct {
              short jour;
              string mois;
              short an;
        } naissance;
} personne;
```

Dans ce cas, la structure date n'existe que dans le contexte de la structure personne, et ses champs ne peuvent pas être accédés en dehors de ce contexte.

La déclaration des membres d'une structure correspondant à la déclaration d'une variable ordinaire, elle respecte la même syntaxe, excepté qu'on ne peut spécifier aucun attribut, que l'on désigne en C par "classe de stockage"; en revanche, ces attributs sont utilisables avec une variable de type struct. Par exemple, on peut déclarer plusieurs membres pour un même type, en les séparant par des ",", ou indiquer qu'une variable de type structure est "static", ce qui rend tous ces membres "static"; en revanche, cela n'a aucun sens (et c'est donc interdit) de déclarer que l'un des membres est "static".

Différentes structures peuvent avoir des membres ayant le même nom, puisque ceux-ci ne seront accessibles qu'en les qualifiant par le nom de la structure. De même, les noms de ces membres pourront être identiques à des noms de variables simples. Enfin, l'identifiant d'une structure (le "tag") peut aussi avoir un nom identique à des noms de membres ou de variables simples, car il n'est pas utilisé dans le même contexte. Toutefois, il est d'usage de n'utiliser le même nom pour 2 entités différentes que si elles représentent des objets similaires.

Initialisation d'une structure

Une variable de type struct peut être initialisée de la même façon qu'une variable simple en faisant suivre sa déclaration du signe "=" et de la liste des valeurs à donner à chacun des membres, séparées par des "," et entourée par des "{}", par exemple :

```
struct  {
        string nom;
         string prenom;
        struct {
              short jour;
              string mois;
             short an;
        } naissance;
} bird  =  {  "Parker",  "Charlie",  20,  "août",  1920  };
```

Lorsqu'une structure est incluse dans une autre, il n'est pas obligatoire d'entourer les valeurs de ses membres par de nouvelles "{}", à condition de donner une valeur à l'ensemble des membres, ce qui est généralement le cas.

Dans le cas où on n'initialise que les premiers membres d'une structure, les autres se verront attribuer la valeur Ø.

Si la variable de type struct est déclarée comme une variable globale ou "static" et qu'elle n'est pas initialisée, l'ensemble de ses membres seront égaux à Ø; les variables locales "auto" auront une valeur quelconque (en fait ce qu'il y a sur la pile au moment de l'appel à la fonction dans laquelle elles sont déclarées). Il n'est bien sûr pas possible d'initialiser une structure étant déclarée "extern", comme cela est d'ailleurs le cas pour toutes les variables, quel que soit leur type.

Règle d'alignement

Tous les exemples que je vous présentés jusqu'à présent sont corrects. En particulier, l'ordre des membres que j'ai utilisé correspond à une certaine logique d'organisation des informations, qui, à priori, semble appropriée.

Toutefois, cet ordre présente un défaut : il ne tient pas compte de la règle d'alignement. En fait, sur le GS, cela n'est pas trop gênant, mais sur des machines plus sophistiquées, et notamment sur des processeurs RISC, il y a de fortes chances que ces définitions soient rejetées par le compilateur.

Commençons par définir ce qu'est cette fameuse règle d'alignement : une donnée est dite "naturellement alignée" (ou plus simplement "alignée") lorsque son adresse est un multiple de sa longueur.

En d'autres termes, cela signifie qu'un mot de 16 bits est aligné si il a une adresse paire, un mot de 32 bits l'est si son adresse est un multiple de 4 et un caractère (ou une chaîne) est aligné quelle que soit son adresse.

Pour améliorer les performances d'un ordinateur, on a mis au point de nouveaux processeurs que l'on a appelé "RISC" ("Reduced Instruction Set Computer", c'est à dire "Ordinateur à jeu d'instructions réduit"), et qui d'ailleurs n'est plus tellement réduit de nos jours, mais plutôt simplifié.
Par extension, on a aussi appelé les processeurs les ayant précédé, "CISC" ("Complex Instruction Set Computer").

A part les toutes dernières générations, ces processeurs sont tous 32 bits (les nouveaux sont 64 bits, mais il n'existe pas encore réellement d'ordinateurs les exploitant), c'est à dire que l'unité d'accès à la mémoire est un mot long de 32 bits.

Donc, pour obtenir une performance maximale, le processeur ne sait accéder qu'à un mot de 32 bits aligné, c'est à dire qu'il sait accéder à l'adresse Ø, 4, 8 ... à l'exclusion de toute autre adresse; si la donnée ne fait que 16 ou 8 bits, le processeur lira le mot long l'incluant, masquera la partie inutile et fera un décalage pour replacer la partie utile au bon endroit.
On peut remarquer que la plupart des processeurs CISC procèdent de la même manière; c'est notamment le cas des 680x0 qui ne savent pas accéder à une adresse impaire.

Si un entier long se trouve à une adresse non alignée, par exemple à l'adresse 2, une tentative d'accès à ce long avec un processeur RISC provoquera une violation d'accès qui sera récupérée par le système, lequel remplacera cet accès par 2 accès, l'un à l'adresse Ø et l'autre à l'adresse 4, puis isolera les 2 mots de 16 bits qu'il réintégrera ensuite dans le mot de 32 bits demandé.

Vous pouvez imaginer que cela est une action assez coûteuse, suffisamment en tout cas pour annuler le gain de performances pouvant être apporté par le processeur. Il n'est donc pas rare de voir des machines RISC tourner apparemment plus lentement que des machines CISC, car malheureusement, beaucoup d'applications ne tiennent pas compte de la règle d'alignement (en fait beaucoup ont été portées depuis un environnement CISC sans être réadaptées). Si le système ne possède pas une telle fonction de réalignement (ce qui est rarement le cas, sinon presque aucune application n'aurait tourné sans nécessiter sa réécriture), le programme sera tout simplement rejeté par le compilateur, puisqu'il ne pourra pas s'exécuter. Dans le cas contraire, le compilateur se contente de signaler le problème d'alignement.
Si on le souhaite, on peut demander au compilateur de forcer l'alignement, mais cela peut avoir des conséquences fâcheuses, comme nous allons le voir un peu plus loin.

Ne croyez pas pour autant que le problème n'existe que sur les machines RISC; la plupart des machines CISC ne savent pas non plus accéder à des données non alignées. Seulement, comme le processeur est "complexe", il effectue de lui-même le double accès, ainsi que la reconstitution de la donnée demandée, de façon transparente. Ceci implique cependant que les capacités de l'ordinateur sont loin d'être exploitées à leur maximum.
J'ai déjà fait l'expérience, qu'en alignant les données correctement, on pouvait obtenir un gain de performances de pratiquement 100% pour un programme quelconque sur une machine donnée (c'est à dire qu'il va 2 fois plus vite !)

Je ne sais pas trop comment se comporte le GS; à priori, il me semble que le 65C816 est capable d'accéder directement à n'importe quelle adresse, et qu'il lui est donc possible de lire directement un mot de 16 bits situé à une adresse impaire. Mais, il est tout aussi possible qu'il accède à chacun des 2 mots contenant l'un des octets du mot désiré pour les recombiner, de façon totalement transarente; par exemple, si notre mot est à l'adresse 1, il doit lire les mots aux adresses Ø et 2 et isoler puis combiner les octets utiles.

Comment faire pour résoudre ce problème ? Pour les variables simples, il est du rôle du compilateur de garantir qu'elles sont alignées à une frontière multiple de leur longueur. Il est à noter que les compilateurs sur GS ne respectent pas cette règle, et que les variables simples sont donc situées à n'importe quelle adresse. Il ne reste donc qu'à espérer que le 65C816 sache réellement accéder directement à n'importe quelle adresse, sinon on peut faire son deuil de la performance optimale d'un programme en langage évolué sur GS. C'est aussi vrai d'ailleurs en assembleur, si l'on ne fait pas attention, d'autant plus que l'on est obligé d'avoir des paramètres non alignés sur la pile, puisque l'adresse de retour d'une sous-routine est sur 3 octets.

Le problème ne se pose donc que pour les structures, puisque le compilateur place normalement chacun des membres à l'adresse suivant immédiatemment celle du membre précédent dans la structure, ce qui peut donc provoquer des fautes d'alignement. Certains compilateurs permettent de forcer l'alignement des membres. Ceci présente toutefois l'inconvénient de gonfler la taille de la structure, et de compliquer l'interchangeabilité des données avec une autre machine qui ne disposerait pas d'un compilateur effectuant la même opération. En gros, cela interdit l'enregistrement direct d'une structure dans un fichier, si on compte réutiliser ce fichier sur une autre machine. De toute façon, ce n'est pas une très bonne idée de stocker directement une structure dans un fichier, notamment à cause des problèmes d'ordre des octets dans un mot ou un mot long.

Heureusement, il existe une solution simple à tous ces problèmes, dès lors que l'on respecte la règle suivante : les membres d'une structure doivent être spécifiés dans l'ordre décroissant de leur taille.

Par exemple, si je dois créer une structure comprenant 2 longs et 2 shorts, je la définirai ainsi :

```
struct  {
        long   l1, l2;
        short s1, s2;
};
```

Pour les structures, les compilateurs des machines concernées garantissent qu'elles seront alignées sur une fonctière de 32 bits. Par conséquent, les membres de la structure précédente seront bien alignés sur un multiple de leur longueur.

Les membres d'une structure devront donc respecter l'ordre suivant : "double" (64 bits), "long" et "float" (32 bits), "short" (16 bits) et "extended" (80 bits, soit 5 short), "char" et "string". Si la structure comprend un tableau, on le mettra de préférence à la fin. Si il y a en plusieurs, on essayera de faire en sorte que la taille totale de chaque tableau soit un multiple de 16 ou 32 bits (ie pour une chaîne, un nombre pair de caractères), et on les mettra dans l'ordre décroissant de l'alignement (c'est à dire d'abord les multiples de 32 bits puis ceux de 16 bits).

Cette règle est bien sûr aménageable, de façon à prendre en compte le désir d'organiser les membres 'logiquement' en fonction de ce qu'il représentent : il suffit de les compacter dans un "long". Par exemple, l'exemple précédent peut aussi être écrit des 2 façons correctes suivantes :

```
struct {
     long    l1;
     short   s1, s2;
     long    l2;
};
```

```
struct {
     short s1, s2;
     long   l1, l2;
};
```

En revanche, il faut absolument éviter d'utiliser les 2 formes :

```
struct {
     long   l1;
     short  s1;
     long   l2;
     short  s2;
};
```

```
struct {
     short s1;
     long  l1;
     short s2;
     long  l2;
 };
```

Utilisation des structures

On ne peut réaliser que très peu d'opérations sur la totalité d'une structure : affecter une variable de type structure à une autre variable du même type structure, obtenir l'adresse de cette variable structure en mémoire (nous y reviendrons lorsque nous parlerons des pointeurs), passer une structure en paramètre à une fonction (ce qui revient en quelque sorte à une affectation), et retourner une structure comme résultat d'une fonction.

Toutefois, le passage d'une structure en paramètre à une fonction par valeur (ce qui est la seule possibilité en C) correspondant au passage de chacun de ses membres, on a tendance à plutôt passer l'adresse de cette structure à la fonction (même si on n'a pas l'intention de la modifier), notamment lorsqu'elle comprend beaucoup de membres. Dans le cas contraire, chacun des membres doit être empilé un par un, ce qui peut coûter cher en temps et en place sur la pile.

On peut utiliser le même principe pour retourner une structure, c'est à dire qu'on retourne un pointeur sur la structure plutôt que la structure elle-même; ce cas pose cependant un problème : si la structure est déclarée localement à la fonction, elle a été allouée sur la pile, et l'adresse n'est plus valide après le retour (plus précisemment, elle correspond à une partie de la pile qui se situe avant son sommet); on déclarera donc une telle structure en "static", mais cela oblige à la recopier après l'appel, car elle sera ecrasée au prochain appel à la fonction. De toute façon, c'est ce qui aurait été fait si on avait retourné la structure directement.

L'opération que l'on réalise le plus avec les structures est bien évidemment l'accès à ses différents champs. On dispose pour cela de l'opérateur spécifique "." (le point, comme en Pascal). Un champ est alors représenté ainsi : variable.champ, la variable désignant une structure. Si le champ est lui même une structure, on continue jusqu'à accéder à un champ de type élémentaire ou tableau. Avec la structure "personne" définie précédemment, si je veux accéder au jour de la naissance d'un individu, j'écrirai :

individu.naissance.jour

Cette opération consiste à qualifier le membre par la structure à laquelle il appartient. A partir de là, le membre s'utilise comme une variable scalaire du même type.

L'opérateur "." est évalué de gauche à droite (on dit qu'il a une associativité gauche-droite). L'exemple précédent correspond donc à "(individu.naissance).jour". L'opérateur "." est aussi celui qui a la plus forte précédence (priorité), avec notamment les "()".

Il est à noter que C ne dispose pas d'instruction "WITH" à la Pascal. Ceci signifie que l'on doit toujours qualifier tous les membres par le nom des structures auxquelles ils appartiennent.

L'opérateur "sizeof()" permet d'obtenir la taille d'une structure en octets. Cet opérateur n'est pas spécifique aux structures, mais il est souvent utilisé avec elles. Il est plus que recommandé d'utiliser cet

opérateur pour évaluer la taille d'une structure (d'autant que cette évaluation est effectuée au moment de la compilation et non pendant l'exécution), et non de fixer la taille en cumulant celles des membres, car la structure peut avoir des trous entre les membres, notamment lorsque l'alignement a été forcé par le compilateur.

La macro "offsetof()" définie dans "stddef.h" permet d'obtenir le décalage en octets entre un membre et le début de la structure qui le contient.
Par exemple, dans la structure :

```
struct  {
      long   m1;
       short m2;
      char   m3;
} x;
```

offsetof(x,m1) vaudra Ø, offsetof(x,m2) vaudra 4 et offsetof(x,m3) vaudra 6.

Champs de bits

Il est classique de regrouper des flags n'utilisant que deux valeurs (Ø et 1 pour faux et vrai) à l'intérieur d'un même mot. Pour cela, on définit des masques, comme par exemple :

```
#define F1      0x01
#define F2      0x02
#define F3      0xC0
```

On utilise alors les instructions bit à bit pour les manipuler (bien entendu chacun des flags doit correspondre à des puissances de 2 de façon à occuper des bits différents). Tout ceci ne présente pas de difficulté particulière.

Cependant, C permet d'accéder à des bits quelconques d'un mot sans requérir l'utilisation des instructions de manipulation de bits. Pour cela, on utilise des "champs de bits" ("bit fields" en anglais) qui permettent de représenter un ensemble contigü de bits, et dont l'unité d'alignement est le bit et non plus l'octet. ORCA/C permet de définir des champs ayant jusqu'à 32 bits (sur d'autres machines, la limite peut être de 16 bits). Ainsi nos flags précédents peuvent être déclarés ainsi :

```
struct  {
      unsigned f3 : 2;
     unsigned      : 4;
      unsigned f2 : 1;
      unsigned f1 : 1;
} flags;
```

La syntaxe est la suivante : la définition des champs de bits doit se faire à l'intérieur d'une structure; pour chaque champ on lui donne éventuellement un nom et une taille en bits séparés par un ":". Le mot clef "unsigned" indique que le champ est non signé (ORCA/C autorise des champs de bits signés, mais je ne vois pas tellement l'intérêt). Si on ne donne pas de nom, cela permet simplement de marquer des bits non utilisés (on crée un "filler" en anglais), que l'on ne pourra donc pas accéder par la suite.

Je suis parti du dernier flag, car ORCA/C alloue les bits de gauche à droite, c'est à dire en partant du plus significatif dans un octet, et de l'octet de poids faible (qui est le premier en mémoire) pour un mot de 16 ou 32 bits.

Ce n'est pas le cas de toutes les machines (cela peut être exactement le contraire, notamment sur les 680x0). Il faut donc être prudent quand on enregistre un champ de bit dans un fichier; il est souvent souhaitable de l'étendre à un octet dans ce cas.

Avec cette définition, f1 peut avoir la valeur Ø ou 1 et correspond au bit Ø de l'octet, f2 est le bit 1, et f3 les bits 6 et 7; il peut prendre les valeurs de Ø à 3.

On accède à ces champs de la même manière que les autres membres d'une structure, par exemple flags.f1 ... Ils sont considérés comme des entiers à part entière et peuvent donc être employés dans une expression arithmétique.
Je peux donc écrire :

```
flag.f3 = 1 + flag.f1;
```

Si le résultat est plus grand que ce que peut représenter le champ de bits, il sera tronqué.

Les champs de bits peuvent être mélangés avec la définition d'autres membres d'une structure. Il est en effet fréquent de trouver des définitions similaires à :

```
struct {
        string          nom, prenom;
        unsigned        celibataire : 1;
        unsigned        marie : 1;
        unsigned        divorce : 1;
        unsigned        veuf : 1;
        unsigned        enfants : 4;
        unsigned        char age;
} individu;
```

Dans cette structure, j'ai utilisé un bit différent pour représenter la situation sociale d'un individu (c'est soit loin d'être la meilleure solution car ces situations sont exclusives, et donc 2 bits suffisent au lieu de 4; mais bon, c'est un exemple !), 4 bits pour indiquer le nombre d'enfants (ce qui lui permet d'en avoir jusqu'à 15) et un octet pour l'âge.
Cette définition permet de compacter toutes ces informations dans un même mot de 16 bits tout en donnant un nom à chacune d'entre elles. Un champ de bits peut donc être utilisé aussi pour représenter un petit nombre entier, ou un nombre n'occupant pas un nombre entier d'octets (par exemple un nombre ne nécessitant que 10 bits), et pas uniquement des flags.

Si mes champs de bits n'avaient pas occupé un octet entier, l'âge aurait quand même été aligné sur une frontière d'octet (c'est l'alignement minimum quelque soit la machine), mais j'aurai aussi pu le déclarer en "unsigned age : 8" pour forcer le compactage.

Les autres membres d'une structure étant alignés sur une frontière d'octet, il ne faut pas intercaler les champs de bits et les autres membres (par exemple bit, short, bit), sinon on perd tout le bénéfice du compactage.

Les champs de bits doivent aussi respecter la règle d'alignement que j'ai indiqué ci-dessus; il est donc d'usage de les regrouper tous ensemble et de les faire voisiner avec des chars ou des short de façon à former un long.

Si un champ de bits ne peut être inclus dans le mot long courant, c'est à dire que sa taille le force à chevaucher 2 mots longs, il est réaligné à la frontière du mot long suivant. On peut forcer ce changement de mot en codant un champ de bits de taille Ø.

Un champ de bits n'a pas d'adresse (on ne peut pas utiliser l'opérateur "&" avec ces membres, et par conséquent la macro "offsetof()" non plus), et nepeut être représenté autrement que par une structure. Enfin, un champ de bits n'est pas un tableau de bits.

Il est bien évident que l'accès à des champs de bits est assez coûteuse, puisqu'ils ne servent qu'à masquer les manipulations de bits qu'on devrait faire par ailleurs si on avait utilisé des entiers et des masques. Mais ils ont quand même leur utilité.

Unions

Dans une structure, chacun des membres est stocké immédiatemment après le membre précédent. Il est parfois utile d'associer plusieurs types à une même adresse. Dans ce cas, on utilise une "union".

Une "union" est similaire à la partie variable d'un record Pascal, que l'on déclare avec un "CASE OF". Elle n'en a toutefois aucune des restrictions.

La syntaxe d'une "union" est identique à celle d'une "structure", excepté qu'elle ne peut contenir de champs de bits. Les autres règles stipulées précédemment pour les structures s'appliquent pareillement aux unions. De même l'accès aux membres d'une union est identique à ceux d'une structure, c'est à dire qu'on utilise l'opérateur ".". Une structure peut inclure une union (c'est le cas le plus fréquent) et une union peut aussi inclure une structure.

La seule différence entre une structure et une union se situe au niveau de la manière dont la mémoire est allouée aux membres. Dans une structure, chaque membre occupe une adresse consécutive, et la taille de la structure est la somme des tailles des membres; dans une union, chaque membre occupe la même adresse, et la taille de l'union est celle du plus grand de ses membres. A un instant donné ou pour une variable du type de cette union, un seul des types des membres est valide.

Voyons tout ça sur un exemple :

```
struct {                        union {
    long    l;                      long    l;
    float   f;                      float   f;
    short   s;                      short   s;
} s;                            } u;
```

Dans la structure, le membre "l" est à l'adresse Ø, "f" est l'adresse 4 et "s" à l'adresse 8; la taille de la structure est de 10 octets. Pour l'union, "l", "f" et "s" sont tous à l'adresse Ø et la taille de l'union de 4 octets.

Un exemple plus concret pourrait être :

```
typedef struct {
        enum { entier, reel } type_nombre;
    union {
            long    val_entiere;
            long    val_reelle;
    } val;
} nombre;
```

Cette structure permet de représenter un nombre entier ou réel. Le membre "type_nombre" indique le type du nombre et, par conséquent, quelle valeur utiliser dans l'union. Si on n'avait pas ce premier membre, on ne pourrait déterminer si le nombre est un entier ou un réel, ni quel membre de l'union est utilisable, à moins de disposer de cette information par ailleurs.

Un dernier exemple :

```
union {
    long                                    l;
     struct {
          short  n;
          unsigned x : 4;
          unsigned f1 : 1;
          unsigned f2 : 2;
          unsigned : 1;
          char c;
    } s;
} u;
```

Ce type de définition permet de représenter un ensemble d'informations occupant 32 bits et de lui associer un long. Les opérations de copie, passage de paramètre ... seront beaucoup plus efficaces en utilisant le long plutôt que la structure associée. En même temps, les accès aux différents membres de la structure restent simplifiés, en ne nécessitant pas de masques ni de manipulation de bits.

On peut initialiser le premier membre d'une union en faisant suivre la déclaration de la variable du type de cette union, du signe "=" et la valeur désirée. Ceci requiert que le premier membre soit initialisable.
Dans l'exemple précédent, je pourrai écrire "union ... u = 0"; si j'avais inversé le "long" et la "structure", j'aurais du initialiser chacun des membres de la structure.

Chapitre 7

Les Tableaux et les Pointeurs

Comme je l'ai laissé sous-entendre dans mon précédent article, il y a une très forte relation en C entre les tableaux et les pointeurs. Il était donc naturel d'aborder ces 2 aspects du langage simultanément.

Cependant, et j'ai déjà eu plusieurs fois l'occasion de l'écrire, les pointeurs jouent un rôle très important dans le langage C; il n'est donc pas possible de décrire tout ce que l'on peut faire avec les pointeurs dans un seul article. Par conséquent, nous nous contenterons de ne faire qu'une première approche des pointeurs dans cet article, tandis que nous en ferons une étude plus approfondie dans le prochain.

Définition d'un tableau

A priori, les tableaux en C semblent assez rudimentaires, notamment par rapport aux possibilités offertes par Pascal. Ainsi, lorsqu'on déclare un tableau, la seule chose que l'on doive indiquer, c'est sa taille, comme par exemple :

```
int tableau[10];
```

Cette définition va réserver en mémoire un bloc permettant de stocker 10 entiers (de 2 octets chacun) les uns à la suite des autres, c'est à dire que ce tableau occupera 20 octets de mémoire. Dans cet exemple, j'ai déclaré un tableau d'entiers, mais, bien entendu, tout autre type, qu'il soit prédéfini ou bien défini préalablement à la déclaration du tableau, peut être utilisé, et notamment un type structure ou union, tels que nous les avons vus dans le précédent GS Infos. Le seul type qu'on ne peut pas utiliser est le type fonction; on ne peut donc pas définir de tableaux de fonctions.

On déclare donc un tableau de la même manière qu'une variable simple, sauf qu'on spécifie en plus le nombre d'éléments qu'il doit comporter, cette taille étant indiquée entre crochets ("[]"). La taille du tableau doit être obligatoirement une constante entière, ou une expression pouvant être évaluée par le compilateur et dont le résultat est donc constant. Ceci implique que tous les éléments d'un tableau sont du même type.

On peut aussi définir un nouveau type tableau avec "typedef". Par exemple :

```
typedef int inttab10[10];
```

définit un type "inttab10" permettant ensuite de déclarer des tableaux de 10 entiers, comme :

```
inttab10 t1, t2;
```

On peut constituer des tableaux de structures, par exemple :

```
struct {
    int x, y;
} coords[100];
```

et déclarer un tableau à l'intérieur d'une structure :

```
struct  {
        int tab[10];
} s;
```

Attention : il faut veiller à la taille des tableaux que vous allouez, car, par défaut, la taille maximale d'un programme est de 64K octets; si vous avez besoin de plus de mémoire, il vous faut soit faire des allocations dynamiques (nous en reparlerons dans le prochain article), soit utiliser un autre modèle de mémoire (voir le manuel d'ORCA/C à ce sujet).

Il faut être extrêmement prudent pour les tableaux déclarés en variables locales, car ils sont alors alloués sur la pile, qui est une ressource très limitée sur le GS.

Pour éviter tout problème, je vous conseille de déclarer vos tableaux soit en global, soit en utilisant l'attribut "static" si vous les déclarez localement à une fonction.

Initialisation d'un tableau

Lors de la déclaration d'un tableau, il est possible, comme avec toutes les autres variables, d'affecter une valeur à chacun des éléments. La syntaxe est identique à celle que nous avons vu avec les structures, c'est à dire une liste de constantes (ou d'expressions constantes) séparées par des virgules et encadrées par des accolades ("{}"), par exemple :

```
int tab[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Toutefois, l'initialisation d'un tableau connaît quelques règles particulières :

• Il n'est pas nécessaire de donner une valeur à chacun des éléments. Dans ce cas, les éléments non initialisés se verront affecter la valeur Ø par le compilateur.

• Lorsqu'on initialise un tableau, il n'est pas obligatoire d'indiquer explicitement sa taille, le compilateur la calculant automatiquement en fonction du nombre de constantes d'initialisation spécifiées. Ainsi, l'exemple précédent peut aussi être écrit :

```
int tab[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Les crochets restent obligatoires pour indiquer au compilateur qu'il s'agit d'un tableau; il déterminera toutefois tout seul qu'il y a 10 éléments dans ce tableau.

Bien entendu, cette règle ne peut pas être appliquée en même temps que la précédente; je vous laisse chercher pourquoi si cela ne vous paraît pas évident.

• Les tableaux de caractères constituent un cas particulier. En C, il n'y a pas de type "string" (chaîne de caractères) à proprement parler. A la place, on utilise des tableaux de caractères. Toutefois, le langage permet d'écrire une chaîne constante d'une façon spéciale en la délimitant par des """. On peut donc initialiser un tableau de caractères avec une chaine, par exemple :

```
char str[] = "Ceci est une chaîne";
```

Le compilateur rajoute automatiquement un octet à "Ø" à la fin puisque c'est une chaîne de caractères. Le tableau comprend donc un élémént de plus que le nombre de caractères de la chaîne. Il est donc souhaitable de ne pas dimensionner le tableau explicitement, afin de ne pas se tromper (voir ci-après pour plus de détails sur ce point).

On peut cependant aussi vouloir initialiser un tableau de caractères sans avoir cet octet à "Ø" final. Il suffit alors d'employer la même syntaxe que pour les autres types :

```
char tabc[] = { 'a', 'b', 'c' };
```

Il n'est pas non plus obligatoire de donner la taille d'un tableau lorsqu'on déclare ce tableau sans le définir (c'est à dire que l'on ne réserve pas de mémoire).
C'est le cas lorsqu'on effectue une référence externe :

```
extern int tab[];
```

ou lorsqu'on passe un tableau en paramètre à une fonction :

```
void f ( int tab[] )
{
.
.
.
}
```

Comme le compilateur ne fait aucun contrôle par lui-même (voir plus loin), le fait d'indiquer ou non la taille du tableau externe ne change pas grand chose. Dans un cas comme dans l'autre, c'est à vous de faire attention.

Accès aux éléments d'un tableau

Une fois que l'on dispose d'un tableau, il nous faut pouvoir accéder à chacun des éléments qui le composent. Pour cela, on utilise un mécanisme que l'on désigne sous le nom d'index ou d'indice. Ainsi, chacun des éléments d'un tableau sera accédé par son numéro d'index, qui, en C, démarre TOUJOURS de Ø. On utilise la même syntaxe que pour déclarer le tableau, c'est à dire que l'on donne le nom du tableau suivi de l'indice de l'élément voulu entre "[]". Avec l'exemple ci-dessus, on aura donc tableau[Ø], tableau[1], ..., tableau[9]. En fait, l'index représente la position de l'élément par rapport au début du tableau. Un index est obligatoirement une constante, une variable ou une expression entière; on peut toutefois utiliser des entiers courts ou longs en fonction de la taille du tableau.

Il est très important de bien comprendre ce principe d'indexation d'un tableau en C, car il est souvent cause d'erreurs et par conséquent, de plantages du programme : puisque l'index d'un tableau commence toujours à Ø, l'index du dernier élément du tableau est donc égal au nombre d'éléments - 1. Ainsi, dans notre exemple, il n'y a pas d'élément d'index 10. Vous pouvez vous en convaincre en vérifiant que de Ø à 9, il y a bien déjà les 10 éléments demandés lors de la déclaration du tableau. Donc, tableau[10] n'existe pas ! Toute tentative de modification de cet élément va en fait écrire dans une zone mémoire inconnue : cela peut être une variable adjacente dans votre programme, mais cela peut aussi être l'adresse de retour dans la pile ou une zone mémoire appartenant à un autre programme.

Malheureusement, c'est un piège dans lequel on tombe très facilement. En effet, en regardant la taille du tableau, on se dit que l'index va jusqu'à 10 alors qu'il ne va réellement que jusqu'à 9. Il n'y a pas de remède miracle, si ce n'est qu'il faut faire très attention.

C'est d'autant plus frustrant que C n'offre aucun mécanisme de validation des index lors de l'exécution du programme, et que le plantage peut avoir lieu ou pas selon la zone mémoire écrasée.

En général, un tableau est manipulé dans une boucle, et il faut alors être prudent sur la façon de faire varier l'index dans cette boucle. Avec notre exemple précédent, je peux initialiser mon tableau de la manière suivante :

```
for ( i = 0; i < 10; i++ )
        tableau[i] = i;
```

Notez bien la condition d'arrêt de la boucle : l'instruction "tableau[i] = i" est exécutée tant que "i" est strictement inférieur à 10. L'erreur la plus classique est d'écrire la boucle de cette manière :

```
for ( i = 0; i <= 10; i++ )
        tableau[i] = i;
```

J'espère que vous avez bien compris le problème. Une autre cause d'erreur potentielle vient de la mauvaise utilisation des opérateurs d'incrémentation et de décrémentation. Par exemple :

```
for ( i = 0; i < 10; tableau[i] = i++ );
```

Ici, vous ne savez pas qui sera évalué en premier entre "tableau[i]" et "i++"; si c'est le deuxième membre (ce qui est fort probable), lorsque "i" vaudra 9, l'expression sera alors interprétée comme "tableau[10] = 9", et vous aurez un problème : le "i" utilisé comme index sera évalué après avoir été incrémenté.

Si vous connaissez le langage Pascal, vous devez vous dire que pour une fois C n'est pas très sophistiqué. Vous avez en partie raison. D'un autre côté, il faut bien voir qu'une fois traduit en langage machine, un tableau est tel que C le représente : une zone mémoire contiguë dont le premier élément est à la même adresse que le début du tableau, et donc à la position Ø. C'est à dire qu'il y a une transcription directe entre la représentation C et la représentation machine du tableau, ce qui est cohérent avec les autres concepts du langage. C'est aussi à cause de cette limitation apparente qu'il y a une équivalence directe entre les tableaux et les pointeurs.

Lorsqu'en Pascal, on déclare un tableau d'indice quelconque commençant à n'importe qu'elle valeur, le compilateur doit générer du code supplémentaire pour effectuer la transcription entre la forme évoluée et l'adresse en mémoire, ce qui a un coût non négligeable en général. De la même manière, les contrôles d'indices effectués pendant l'exécution d'un programme Pascal sont le fait d'instructions rajoutées au programme, ce qui a aussi un impact sur la taille et la vitesse du programme (bien entendu on peut toujours les enlever).

Dans un prochain article de ma série "Programmation" que je consacrerai aux techniques de mise au point (debugging), je vous montrerai notamment comment effectuer des contrôles d'indices en C équivalents à ce que fait le compilateur Pascal.

Tableaux à plusieurs dimensions

C ne permet pas vraiment de déclarer des tableaux à plusieurs dimensions; à la place, on peut déclarer des tableaux de tableaux, c'est à dire que chaque élément du premier tableau est un tableau, les éléments de ce second tableau étant les données elles-mêmes. Ce qui se fait de la manière suivante :

```
int tab[10][10];
```

C'est à dire que l'on spécifie chacune des dimensions en leur donnant une taille stipulée entre crochets. Cette manière de procéder revient à peu près à la même chose qu'un tableau à plusieurs dimensions, et à part la différence de syntaxe, un tableau de tableau s'utilise de la même manière qu'un tableau à plusieurs dimensions. Il n'y a pas de limites particulières au nombre de dimensions que l'on peut déclarer, mais au delà de 3 ou 4, cela devient difficilement gérable pour le programmeur.

Le tableau défini ci-dessus occupera 10 * 10 * 2 = 200 octets.

On accède aux éléments de ces tableaux en indiquant chacun des index, eux aussi encadrés par des crochets. Le tableau précédent varie donc de tab[0][0] à tab[9][9].
Je peux donc l'initialiser avec la boucle imbriquée suivante :

```
for ( i = 0; i < 10; i++ )
        for ( j = 0; j < 10; j++ )
                tab[i][j] = i * j;
```

Un tel tableau est stocké en mémoire de telle sorte que les éléments identifiés par le dernier indice soient consécutifs, ce qui est cohérent avec la notion de tableau de tableau; on dit que le tableau est organisé par lignes. Avec mon exemple, j'aurai donc en mémoire successivement tab[0][0], tab[0][1], ... tab[0][9], tab[1][0] et ainsi de suite.

On peut aussi initialiser un tableau à plusieurs dimensions; on procède alors de la manière suivante :

```
int  tab[2][5]  =  {  {  1,2,3,4,5 },  {  6,7,8,9,10 }  };
```

Avec cette méthode, chaque sous-tableau est bien initialisé par les valeurs, le tableau de tableau étant lui initialisé par les listes de valeurs de chacun des sous-tableaux.

Les mêmes règles que celles listées précédemment s'appliquent, on y ajoute néanmoins une quatrième règle :

• On peut supprimer les accolades délimitant les sous-tableaux si l'ensemble des éléments sont initialisés. L'exemple précédent devient alors :

```
int  tab[2][5]  =  {  1,2,3,4,5,6,7,8,9,10 };
```

Dans le cas où on ne spécifie pas les dimensions du tableau explicitement, il faudra bien entendu utiliser les accolades séparant chaque sous-tableau afin que le compilateur sache comment accéder à chacun des

éléments.

Lorsqu'on référence un tableau déclaré par ailleurs ou qu'on déclare un paramètre formel, il est possible d'omettre la taille de la première dimension (c'est à dire celle du tableau englobant les sous-tableaux), car le compilateur n'en a pas besoin pour calculer l'adresse de l'élément en mémoire, c'est à dire que je peux écrire :

```
extern int tab[][10]
```

mais pas

```
extern int tab [10][]
```

En effet, pour l'élément

```
tab[i][j]
```

l'adresse sera obtenue parle calcul suivant :

i * nombre de colonnes (éléments du tableau de droite) * taille d'un élément en octets + j * taille élément en octets. Avec l'exemple précédent, tab[1][3] aura l'adresse (relativement au début du tableau) : 1 * 5 *2 + 3 * 2 = 16.

Il est important de connaître l'ordre de progression des indices dans un tableau à plusieurs dimensions, même si cela n'a pas tellement d'importance sur le GS ou sur d'autres micro-ordinateurs avec des systèmes d'exploitation simples. En revanche, sur des machines et des systèmes plus sophistiqués implémentant un mécanisme appelé 'mémoire virtuelle', cela est extrêmement important.

En effet, l'accès à deux éléments du tableau dans le mauvais ordre peut provoquer, si le tableau est grand, ce que l'on appelle une 'faute de page', c'est à dire que la page contenant l'adresse en question n'est pas en mémoire, et qu'il faut aller la chercher sur disque (ce phénomène s'appelle la 'pagination'), en général au détriment d'une page qui est elle en mémoire et que l'on va sauver sur disque à la place de celle dont on a besoin. Le problème est que lorsqu'on va accéder à l'élément voisin du premier, la page le contenant a peut être été supprimée, d'où une nouvelle faute de page ... Les performances du programme s'en ressentent très significativement. Sur un cas vécu, cela était dans un rapport 10, c'est à dire que l'accès à un tableau d'un mégaoctet dans le mauvais ordre des indices était 10 fois plus lent que dans le bon ordre.

En C, cela signifie qu'il faut d'abord faire progresser l'indice le plus à droite dans la boucle la plus imbriquée. En effet, tab[i][j+1] est consécutif en mémoire à tab[i][j] tandis que tab[i+1][j] est distant de tab[i][j] du nombre d'éléments de la deuxième dimension multiplié par la taille de chaque élément en octets.

Si vous voulez étudier un exemple concret d'utilisation des tableaux, je vous recommende de vous repencher sur le programme "Crypt" que j'avais écrit pour accompagner la quatrième partie de cette initiation. Vous le trouverez sur le numéro 19 de GS Infos.

Pointeurs

Un pointeur est une variable dont le contenu n'est pas une donnée (comme les autres types de variables), mais l'adresse d'une autre variable, ou plus généralement l'adresse d'une donnée quelque part en mémoire. Dans le cas du GS, qui a un adressage sur 24 bits, un pointeur est représenté par un entier long non signé

sur 32 bits, car le GS ne manipule pas de données sur 24 bits (mis à part l'adresse de retour d'une sous-routine). Cet entier long représente donc une adresse dans l'espace adressable du GS; l'octet de poids le plus fort est donc toujours égal à Ø.

Cependant, un pointeur n'est pas un entier et il faut donc éviter de faire des conversions de types entre les 2, même si cela ne change en général rien à la valeur du pointeur.

Les pointeurs jouent un rôle très important en C, et il difficile d'écrire un programme pouvant s'en affranchir; d'ailleurs, j'ai déjà dû en dire quelques mots dans mes articles précédents, notamment pour le passage de paramètres par référence.

Les pointeurs jouissent par ailleurs d'une assez mauvaise réputation, car ils permettent de réaliser des programmes incompréhensibles. C'est vrai si on ne les utilise pas avec un minimum de soins; il est notamment très facile de créer des pointeurs qui ne pointent nulle part, et donc de planter le programme lorsqu'on les manipulent.

En revanche, ils permettent souvent d'écrire des programmes plus efficaces et plus simples, voire même parfois plus clairs que si on ne les utilisait pas. Je vais donc essayer de vous apprendre les bonnes manières pour les utiliser.

Déclaration et utilisation d'une variable pointeur

Un pointeur sur une variable d'un type donné se déclare en précédant le nom de la variable par le symbole "*"; par exemple :

```
int  *p;
```

déclare un pointeur p sur une variable entière. Pour l'instant ce pointeur ne pointe sur rien; il faut donc lui affecter l'adresse de l'objet sur lequel il pointe, ce qui se fait par l'opérateur unaire "&", soit par exemple :

```
int i, *p;
p = &i;
```

En fait, comme pour toutes les autres variables, un pointeur doit être initialisé avant de pouvoir être utilisé. La différence principale est que l'utilisation d'un pointeur non initialisé est plus risquée que celle d'une variable non initialisée, puisque cela va accéder à une zone mémoire inconnue, et peut donc planter le programme selon la zone référencée ou modifiée inconsidérément, notamment si l'adresse contenue dans le pointeur correspond à l'espace d'entrées-sorties.

L'opérateur "&" ne peut être utilisé qu'avec un objet ayant une adresse, c'est à dire en gros une variable; dans un précédent article, un tel objet était désigné sous le nom de "valeur-g". Je ne peux donc pas prendre l'adresse d'une constante ou d'une expression. En général, il n'est pas non plus possible de prendre l'adresse d'une variable affectée à un registre; comme le 65C816 ne dispose pas d'assez de registres, ORCA/C ignore cet attribut, que de toute façon, je vous conseille de ne pas utiliser, les compilateurs actuels déterminant eux-mêmes ce qui doit être mis dans des registres ou pas. ORCA/C permet donc d'obtenir l'adresse d'une variable déclarée comme devant être stockée dans un registre.

Il existe une adresse spéciale en C, c'est l'adresse Ø. Un pointeur ayant cette valeur ne pointe sur rien, ce qui n'est pas la même chose qu'un pointeur non initialisé, qui lui pointe sur quelque chose, sauf que l'on ne sait pas quoi. Cette adresse est l'équivalent de NIL en Pascal, et on a l'habitude en C de lui associer un nom avec le préprocesseur : NULL. Ce nom est défini dans les fichiers header standards du C : stddef.h et stdio.h, ainsi que dans le header spécifique au GS : types.h. Toutefois, il faut bien faire attention que contrairement

au Pascal où NIL est un mot clef du langage, NULL est simplement la définition d'un symbole associé à l'adresse Ø. Ceci étant dit, on l'utilise exactement de la même manière.

Le symbole "*" utilisé précédemment pour déclarer un pointeur est aussi employé comme opérateur unaire pour accéder à la valeur de la variable pointée. Ainsi :

```
*p = Ø;
```
est identique à :
```
i = Ø:
```

Cette opération d'accès à une variable pointée s'appelle le 'déréférencement'. De la même manière que l'on ne peut prendre l'adresse que d'une "valeur-g", on ne peut déréférencer un objet que s'il a une adresse, c'est à dire qu'il doit s'agir d'une "valeur-g".

L'opérateur "*" doit être utilisé avec une variable de type pointeur et contenant l'adresse d'une autre variable qui peut être d'un type quelconque mis à part un champ de bit. Cela peut notamment être aussi une variable de type pointeur, créant ainsi une indirection multiple. On accède alors à la donnée réelle en indiquant autant d'"*" que d'indirections, par exemple :

```
**p = Ø;
```

Lorsque le type de base utilisé pour déclarer la variable pointée est "void", comme dans :

```
void *p;
```

on définit ce que l'on appelle un pointeur générique, c'est à dire qu'il ne pointe pas sur une donnée précise. Pour pouvoir le déréférencer, il faudra effectuer un "type cast" afin de déterminer le type précis de la donnée à accéder. En revanche, on peut affecter à ce type de pointeur n'importe quel autre type de pointeur et vice-versa : il est universel.

Un pointeur étant une variable comme les autres, il peut être affecté à un autre pointeur, à condition qu'ils soient de même type ou que le pointeur cible soit universel. Dans le cas contraire, on peut recourir à un cast, mais cela n'est pas sans risque, car le type de la variable pointée indique comment la mémoire sera accédée, notamment pour le nombre d'octets à traiter. Par exemple :

```
int *p1, *p2, i;

p1 = &i;
p2 = p1;
*p2 = Ø;
```

Ici, "i", "p1" et "p2" ont tous la même valeur. L'affectation de pointeurs est donc identique à l'affectation de variables de tout autre type.

Une variable pointée s'utilise exactement de la même manière qu'une variable simple, et peut donc participer à la constitution d'une expression quelconque valide pour le type de la donnée pointée. Par exemple :

```
j = *p * 3 + i;
```

La seule difficulté est de bien différencier les usages de l'opérateur "*" qui sert à la fois à la multiplication et au déréférencement d'un pointeur. Un autre problème peut se poser à cause de la priorité (précédence) des opérateurs "*" et "&" : ils ont une priorité supérieure à celle des opérateurs

arithmétiques. Ainsi, dans :

```
j = *p + 1;
```

le pointeur "p" est d'abord déréférencé, puis on ajoute un à la variable sur laquelle il pointe avant d'affecter le résultat final à "j". L'expression

```
j = *(p + 1)
```

a aussi un sens en C, bien que totalement différent, comme nous allons le voir un peu plus loin.

Il est possible d'utiliser avec les variables pointées l'ensemble des opérateurs que nous avons vu dans les articles précédents, comme par exemple :

```
*p += 2;
```

ou

```
(*p)++;
```

Les parenthèses autour du déréférencement viennent du fait que les opérateurs d'incrémentation et de décrémentation ont une priorité identique à celle de l'opérateur "*", et que tous ces opérateurs sont évalués de droite à gauche. Dans l'exemple précédent, cela signifie que "++" est évalué avant "*" les parenthèses permettent de changer l'ordre d'évaluation. Comme tout à l'heure, l'expression

```
*p++;
```

a un sens en C, bien que cela ne réalise pas du tout l'opération désirée ici.

Pointeurs et tableaux

Nous avons écrit au début de cet article qu'il y avait une équivalence entre les tableaux et les pointeurs. Voyons maintenant en détail ce dont il s'agit.

L'opérateur "&" permet d'obtenir l'adresse d'une variable. Dans le cas d'un tableau, il permet donc d'accéder à l'adresse de l'un des éléments. Je peux donc écrire :

```
int tab[10], *pt, i;
pt = &tab[0];
```

Dans cet exemple, "pt" est l'adresse du premier élément du tableau. Par conséquent, l'instruction :

```
i = *pt;
```

copie dans "i" la première valeur du tableau.

Il serait intéressant maintenant que "pt" pointe sur les autres éléments de mon tableau. Bien sûr, je pourrais écrire une boucle telle que :

```
for ( i = 0; i < 10; i++ ) {
        pt = &tab[i];
```

```
        traitement impliquant *pt;
}
```

mais cela n'apporterait pas grand chose à l'utilisation du tableau.

Pour résoudre notre problème, C définit la règle suivante qui est fondamentale pour l'utilisation des pointeurs : si un pointeur "pt" pointe sur un élément quelconque d'un tableau alors "pt+1" pointe sur l'élément suivant de ce tableau, et plus généralement "pt+i" pointe sur l'élément "i" après "pt" et "pt-j" pointe sur l'élément "j" avant "pt", et ce quelque soit le type des éléments du tableau, et par conséquent la taille de ces éléments.

Donc, si "pt" pointe sur "tab[0]", alors

```
*(pt+1)
```

correspond à tab[1], "pt+i" est l'adresse de "tab[i]" et *(pt + i) est le contenu de "tab[i]".

La définition de l'addition (et par extension toute l'arithmétique des pointeurs) d'un entier à un pointeur est telle que la valeur de l'entier est ajustée automatiquement par la taille d'un élément du type pointé. Ainsi, dans l'expression "pt+i", "i" est multiplié implicitement par la taille des éléments pointés par "pt"; si il s'agit d'entiers courts, le multiplicateur sera 2, tandis que pour une structure quelconque, le multiplicateur sera la taille de la structure.

Il est bien évident que l'arithmétique sur les pointeurs représente, à mon avis, l'avantage majeur de C sur tous les autres langages (de même que la correspondance entre les tableaux et les pointeurs). Par exemple, en Pascal, où il n'y a pas de telle arithmétique, pour réaliser l'équivalent, on est obligé de convertir le pointeur en entier, de lui ajouter l'incrément en tenant compte de la taille de l'élément, puis de reconvertir le tout en pointeur. Une expression aussi simple que "pt+i" en C devient en Pascal :

```
pointer ( ord4 ( pt ) + i *sizeof(integer) );
```

ce qui est, avouez le, nettement plus lourd.

J'espère que vous voyez bien maintenant qu'il y a une équivalence totale entre les tableaux et les pointeurs. En fait, la plupart des compilateurs C convertissent toutes les références à un tableau par une référence à un pointeur sur le début de ce tableau. Ainsi, le nom d'un tableau est en fait un pointeur sur son premier élément, et plutôt que d'écrire :

```
pt = &tab[0];
```

je peux simplifier en écrivant :

```
pt = tab;
```

Cette équivalence a des implications très importantes; en effet, il est possible d'utiliser les deux syntaxes "tab[i]" et "*(tab + i)" de façon interchangeable. Le choix de l'un ou l'autre est plutôt une affaire de goût. A mon avis, la première forme est plus lisible, et exprime bien que l'on utilise un tableau.

De la même manière, si "pt" est un pointeur sur une zone allouée dynamiquement (nous verrons comment dans le prochain article), je peux accéder à cette zone, soit par un pointeur, soit en l'utilisant comme un tableau.

Il y a tout de même une petite différence : "tab" est effectivement un pointeur, mais celui-ci est constant; on ne peut donc pas affecter une nouvelle adresse à "tab", ce qui est, en fin de compte, assez logique. Je ne peux donc pas écrire :

```
tab = pt;
ni pt = &tab;
```

ni aucune autre instruction modifiant "tab".

Une autre conséquence de cette équivalence est que lorsqu'on passe un tableau en paramètre à une fonction comme dans "f(tab)", on passe en fait son adresse. Dans la fonction, je peux déclarer ce paramètre indifféremment comme un tableau ou comme un pointeur; c'est aussi pour cela qu'il n'est pas nécessaire de réindiquer la taille du tableau dans la fonction.

Arithmétique sur les pointeurs

Vous devez vous demander à quoi peut bien être utile cette équivalence entre tableaux et pointeurs ?

Et bien, elle prend tout son sens lorsqu'on utilise l'arithmétique sur les pointeurs que j'ai introduite tout à l'heure.

Si "p" est un pointeur, alors "p++" incrémente "p" de telle sorte qu'il pointe sur l'élément suivant en mémoire quel que soit le type de cet élément. De la même manière, "p+=i" fait pointer "p" sur le ième élément suivant la position courante en mémoire; "p--" pointe sur l'élément précédent et "p-i" pointe sur le ième élément avant.

Reprenons notre boucle d'initialisation d'un tableau :

```
for ( i = 0; i < 10; i++ )
        tab[i] = i;
```

Grâce à l'arithmétique sur les pointeurs, je peux tout aussi bien écrire :

```
for ( i = 0; p = tab; i < 10; p++, i++ )
      *p = i;
```

Evidemment, sur un exemple aussi simple, l'avantage des pointeurs n'apparaît pas vraiment. Toutefois, il est suffisant pour expliquer l'intérêt de cette technique.

Dans le premier cas, à chaque tour de boucle, l'adresse de l'élément à modifier "tab[i]" doit être calculée; pour ce faire, on multiplie "i" par la taille de l'élément et on ajoute ce résultat à l'adresse de début du tableau.

Dans le deuxième cas, la seule chose à faire est d'ajouter la taille d'un élément à "p", qui pointe en permanence sur l'élément à accéder, une seule fois à la fin de la boucle, afin de pouvoir le faire pointer sur l'élément suivant.

Vous imaginez le gain de performance que cela peut représenter lorsqu'on manipule un tableau de taille beaucoup plus importante que celui-ci, sans compter que dans un programme réel, il est fort probable qu'on accède plusieurs fois à chaque élément à chaque passage dans la boucle; il est presque certain que l'adresse de l'élément devra être recalculée à chaque fois qu'on y fait référence.

L'arithmétique sur les pointeurs permet aussi de faire des comparaisons entre 2 pointeurs, mais uniquement dans le cas où les 2 pointeurs pointent sur le même tableau; dans le cas contraire, C ne dira rien, mais le résultat n'aura aucun sens.

Donc si j'ai 2 pointeurs "p1" et "p2" pointant sur 2 éléments d'un tableau "tab", je peux déterminer par exemple si "p1" pointe sur un élément précédant celui pointé par "p2" ou si les 2 pointeurs sont identiques. La boucle précédente peut, par exemple, être écrite ainsi :

```
for ( p = tab; p < tab + 10; p++ )
      *p = Ø;
```

"tab+10" donnerait l'adresse d'un dixième élément s'il existait; en l'occurence, cela correspond à l'adresse immédiatement après le tableau. Par conséquent tant que "p" ne pointe pas sur cette adresse, il correspond à un élément du tableau.

On peut donc notamment comparer un pointeur avec la valeur NULL (Ø) pour déterminer si ce pointeur pointe effectivement sur une zone mémoire valide ou pas.

C permet de soustraire 2 pointeurs à condition qu'ils pointent tous deux sur 2 éléments d'un même tableau. Avec les mêmes "p1" et "p2" que tout à l'heure et "p1" > "p2", "p1-p2" donne le nombre d'éléments entre les 2 pointeurs. A titre d'exemple, voici la fonction de la librairie C standard "strlen()" écrite en C :

```
short strlen ( char *s )
{
char  *p;

for ( p = s; *p != '\0'; p++ );
return ( p - s);
}
```

A la fin de la boucle, "p" pointe sur l'octet à Ø terminant la chaîne de caractères. La différence entre "p" et "s" donne par conséquent le nombre de caractères de la chaîne. Pour vous en convaincre, exécutez cette fonction à la main sur une chaîne quelconque, en partant du principe qu'elle est à l'adresse Ø pour simplifier.

La forme la plus classique d'arithmétique sur les pointeurs est souvent représentée par une construction du type :

```
*p++
```

Comme l'ordre d'évaluation de ces opérateurs est de droite à gauche, l'expression précédente signifie : déréférencer le pointeur, puis l'incrémenter. Si vous voulez incrémenter votre pointeur avant de le déréférencer, vous écrirez alors :

```
*++p
```

En résumé, voici toutes les opérations que l'on peut effectuer avec des pointeurs :

- Addition ou soustraction d'un entier, y compris auto incrémentation et décrémentation.
- Soustraction de 2 pointeurs.
- Comparaisons entre 2 pointeurs.

Toutes les autres opérations sont illégales : on ne peut ajouter 2 pointeurs entre eux, ni effectuer de multiplication ou de division, ni d'opération bit à bit ...

Conclusion

Voilà qui conclut notre étude des tableaux et la première partie concernant les pointeurs. Dans le prochain article, nous aborderons des concepts plus avancés sur les pointeurs, notamment leur utilisation avec les structures et les fonctions, ainsi que l'allocation dynamique de mémoire. D'ici là, bonne lecture ...

Chapitre 8

Les pointeurs et la mémoire dynamique

Avant d'entrer dans le vif du sujet, voici la solution au petit exercice que je vous avais proposé dans le précédent GS Infos : si vous avez exécuté le programme, vous vous êtes immédiatement aperçus qu'il calculait une valeur approchée de PI. Je dois avouer que je vous ai simplifié la tâche par rapport au programme original que j'avais obtenu (d'ailleurs il comportait un bug empêchant sa compilation); en effet, celui-ci affichait 0.26 comme valeur approchée de PI et il fallait donc être devin pour trouver la bonne solution. J'ai seulement modifié le coefficient dans l'instruction comportant le printf : j'ai changé le 4 en 44; comme le programme de départ était erroné, j'ai pensé qu'il y avait d'autres erreurs, et c'est pourquoi j'ai modifié ce coefficient.

En dehors de son aspect ludique, ce type de programme a aussi un intérêt pratique : beaucoup d'auteurs s'en servent pour valider leurs compilateurs C; ce genre de programmes a en effet tendance à pousser les compilateurs dans leurs derniers retranchements, même si celui que je vous ai proposé est relativement simple. Si cela vous intéresse, j'ai quelques autres programmes du même style (en fait, il font partie du même concours), dont certains sont nettement plus tordus ! Le seul problème est que je ne les ai que sur papier, et leur saisie est plus que fastidieuse; de plus, rien ne dit que ORCA/C les digérera. J'essaierai néanmoins d'en mettre un de temps à autre dans GS Infos.

Pour ceux qui n'étaient pas avec nous l'année dernière, et aussi pour ceux qui ont déjà oublié mon précédent article, je vous rappelle que celui-ci décrivait les tableaux et les pointeurs du C, en insistant sur le fait que ces 2 concepts présentent des équivalences. Dans cet article, nous allons continuer notre étude des pointeurs, et aborder des mécanismes plus sophistiqués.

Tableaux de pointeurs et pointeurs de tableaux

Nous savons maintenant qu'un pointeur désigne l'adresse d'une zone mémoire, et que cette dernière peut avoir été déclarée comme un tableau.

Nous avons aussi vu comment, grâce à l'arithmétique sur les pointeurs, accéder aux différents éléments d'un tableau.

Dans la discussion du précédent article, je m'étais toutefois cantonné à des tableaux à une dimension. Voyons maintenant ce qui se passe avec un tableau à 2 dimensions (ou plus généralement à N dimensions) ou, comme je l'avais écrit un tableau de tableaux.

Soit donc un tableau tab[m][n] et un pointeur *pt; je peux alors utiliser l'expression pt = &tab[i][j] pour pointer sur un élément donné du tableau, de la même manière que j'écris par exemple p = &t[k] pour pointer sur un élément quelconque d'un tableau à 1 dimension. x = *pt me permet alors d'accéder à l'élément d'indices i et j dans l'exemple précédent. Si j'écris pt = tab, alors *pt donne le premier élément du tableau, soit tab[0][0]. Jusque là, rien de bien nouveau.

En revanche, comme tab est un tableau de tableaux, tab[i] (avec un seul indice) donne l'adresse de début du ième sous-tableau. S'agissant d'une adresse, je peux affecter cette valeur à un pointeur. Je peux donc écrire pt = tab[i]; pour accéder aux éléments définis par la seconde dimension, je peux ensuite utiliser mon pointeur comme tableau à une dimension et donc écrire pt[j], ce qui revient alors exactement au même que tab[i][j].

Vous me direz que tout cela prête facilement à confusion, car on peut s'y perdre rapidement. Malgré tout, il y a un avantage énorme à cette écriture : comme les tableaux sont en général manipulés dans des boucles, et dans le cas d'un tableau à plusieurs dimensions, dans des boucles imbriquées, l'écriture pt[j] simplifie le calcul des adresses des éléments j, et donc améliore la performance, puisque le calcul de l'adresse du ième sous-tableau tab[i] est faite en dehors de la boucle interne. Sachez néanmoins que la plupart des compilateurs modernes (dont ORCA/C dans sa version 2.0) effectuent ce genre d'optimisation eux-mêmes, et que l'on peut s'en passer si l'on ne la trouve pas claire (cela fait partie ce que l'on désigne sous le nom d'élimination des invariants).

Lorsqu'on fait référence à un tableau déclaré par ailleurs, je vous avais dit dans le précédent article qu'il n'était pas nécessaire d'indiquer la taille des premières dimensions; ainsi, on peut par exemple déclarer un tableau comme paramètre formel d'une fonction comme tab[][n]. Mais, puisque la première dimension correspond à un tableau de tableaux, c'est à dire à l'adresse de chacun des sous-tableaux, je peux définir ce paramètre comme étant un pointeur sur un tableau, soit (*tab)[n]. Les parenthèses sont importantes, car les crochets "[]" ont une précédence supérieure à l'opérateur de déréférencement "*". La syntaxe *tab[n] correspond donc à un tableau de pointeurs, et nous verrons son utilisation un peu plus loin.

L'utilisation des pointeurs a tendance à compliquer un peu les déclarations, en obligeant souvent à mettre des tas de parenthèses un peu partout de façon à prendre en compte la précédence des différents opérateurs impliqués. En fait, il faut voir la déclaration d'une variable en C comme un modèle de l'utilisation de cette variable, c'est à dire que si j'utilise exactement la même syntaxe dans une déclaration et dans une expression, j'obtiendrai un objet du type déclaré. Dans notre exemple précédent, en déclarant (*tab)[n] un pointeur sur un tableau de n éléments, j'accéderai à chacun de ces éléments par un accès à (*tab)[i], ce qui signifie déréférencer le pointeur sur le tableau, ce qui donne bien le tableau en question, puis accéder à l'élément i de ce tableau.

Il me suffira donc de faire varier le pointeur (*tab) dans l'expression précédente, par exemple en utilisant l'arithmétique sur les pointeurs, pour accéder à chacun des sous-tableaux. Cependant, cette écriture est un peu lourde et l'utilisation d'un tableau à plusieurs dimensions sous cette forme n'apporte pas grand chose par rapport à la syntaxe complète tab[m][n].

Tableaux de pointeurs

Par ailleurs, je vous avais dit dans le précédent article que l'on n'avait pas le droit d'écrire tab[m][] puisque l'on n'était alors pas capable de calculer les adresses de chacun des éléments. En revanche, comme tab[i] est un pointeur sur des sous-tableaux, je pourrai à la place dire que mon tableau à 2 dimensions est un tableau de pointeurs, c'est à dire écrire *tab[m].

Cette forme est beaucoup plus intéressante que celle décrite dans la section précédente, puisqu'elle permet par exemple d'avoir des sous tableaux de taille différente, alors que l'utilisation d'un tableau à 2 dimensions définit obligatoirement un rectangle dans lequel chaque ligne et chaque colonne a le même nombre d'éléments.

Les pointeurs de chaque élément de ce tableau correspondent à une zone mémoire contiguë qui aura été en général allouée dynamiquement, mais qui peut s'utiliser comme un tableau, grâce à l'interchangeabilité des 2 concepts.

Par conséquent, je peux continuer à accéder à chacun des éléments des zones pointées comme s'il s'agissait d'un tableau à 2 dimensions, soit tab[i][j].

Il y a quand même une différence : si j'ai déclaré par exemple un tableau de 10 lignes et de 10 colonnes d'entiers, j'ai réservé quelque part en mémoire 100 entiers soit 200 octets contigüs et le compilateur C fera tous les calculs d'adresse nécessaires à l'accès de chacun des éléments lorsque j'utiliserai tab[i][j]. En revanche, si le tableau a été déclaré comme un tableau de 10 pointeurs sur des entiers *tab[10], le

compilateur C ne réservera la place nécessaire qu'aux 10 pointeurs, soit 40 octets. Il sera de ma responsabilité de programmeur d'allouer la mémoire aux 10 tableaux de 10 entiers et d'initialiser les pointeurs avec les adresses de ces 10 tableaux; au final, j'aurai toujours bien les 100 entiers désirés (et les 200 octets qu'ils occupent), mais ils ne seront pas forcément contigüs (cela dépendra de la façon dont fonctionne l'allocateur de mémoire); de plus, cette solution occupera 40 octets supplémentaires à la précédente. Il est clair que cet exemple (avec un tableau de cette taille) ne démontre pas réellement l'utilité d'une telle approche.

Elle a toutefois plusieurs avantages : même si l'on accède à chaque élément par un tableau à 2 dimensions, cet accès se fera par un déréférencement de pointeur et non par une multiplication pour chaque calcul d'adresse, ce qui est nettement plus performant; ensuite, comme je l'ai dit plus haut, chacun des sous tableaux n'a pas besoin d'avoir la même taille que les autres, ce qui permet de n'utiliser que la mémoire strictement nécessaire à une exécution de mon application plutôt que de réserver systématiquement d'avance une grande quantité de mémoire en prévision d'un cas particulier; de plus, je peux démarrer avec des tableaux de petite taille et les redimensionner selon les besoins, ce qui implique que je n'ai pas besoin de connaître la taille de mes tableaux lors de l'écriture de mon programme.

Pour se compliquer la vie, on peut aussi définir un pointeur sur un tableau de pointeurs, ce qui s'écrirait *(*tab)[n]. Ainsi, même le premier tableau peut être alloué dynamiquement lors de l'exécution du programme. On pourra néanmoins préférer la syntaxe **tab pour parvenir au même résultat, puisqu'au fond, on ne s'intéresse pas tellement à la taille maximale "n".

Tableaux de chaînes de caractères

Cette utilisation de tableaux de pointeurs trouve tout son sens avec les chaînes de caractères. En effet, en C, il n'y a pas de notion de chaînes de caractères, mais plutôt celle de tableaux de caractères. Un tableau de caractères est aussi, de part l'équivalence entre pointeurs et tableaux, un pointeur sur le premier caractère de la chaîne. En particulier, une chaîne constante telle que "Ceci est une chaîne" est en fait un pointeur sur le premier caractère de cette chaîne. Par conséquent, on peut affecter une telle chaîne à un pointeur, comme char *str = "Ceci est une chaîne"; il faut bien comprendre que la chaîne n'est pas recopiée, mais simplement que le pointeur sur cette chaîne est affecté à la variable "str" qui pointe donc aussi sur la chaîne en question.

C ne dispose d'aucun opérateur pour manipuler des chaînes de caractères, et il faut recourir à des fonctions de la librairie pour les traiter; celle-ci est assez riche et il est très facile de la compléter grâce aux opérations que l'on peut effectuer avec les pointeurs. La plupart des fonctions fournies en standard peuvent d'ailleurs être écrites en C de façon très simple (en fait elles le sont dans la plupart des implémentations); nous avons vu dans le précédent article la fonction strlen(); pour vous donner un autre exemple, voici les fonctions strcpy() et strcmp() :

```
void strcpy ( char *s1, char* s2 )          /* copie s2 dans s1 */
{
        while ( *s1++ = *s2++ );
}

short strcmp ( char *s1, char *s2 )       /* retourne 0 si s1 == s2, < 0 si < et > 0
si > */
{
        for ( ; *s1 == *s2; s1++, s2++ )
             if ( *s1 == '\0' )
                return ( 0 );
        return ( *s - *t ):
}
```

Vous constatez encore une fois la concision du langage C, notamment dans la fonction strcpy() : on copie chacun des caractères de s2 dans s1 en utilisant la post-incrémentation pour accéder à chacun des caractères, le fait que les paramètres sont passés par valeur pour pouvoir les modifier sans risques, ainsi que le fait qu'une comparaison se fait toujours implicitement par rapport à Ø; on bénéficie alors du principe que toutes les chaînes C sont terminées par '\0' (c'est à dire Ø) pour terminer la boucle, une fois que le Ø en question a été recopié. La fonction strcmp() utilise les mêmes principes et utilise en plus l'ordre du code ASCII pour indiquer si les 2 chaînes sont identiques ou laquelle est inférieure à l'autre dès qu'elles divergent.

Bien entendu, si cette écriture vous perturbe, vous pouvez expliciter chacune des opérations, et exploiter l'équivalence entre les pointeurs et les tableaux.
Par exemple, la fonction strcpy() peut aussi être programmée de la façon suivante :

```
void strcpy ( char *s1, char* s2 )          /* copie s2 dans s1 */
{
        short i;

          for ( i = 0; s2[i] != '\0'; i++ )
              s1[i] = s2[i];
         s1[i] = '\0';
}
```

Cette forme est peut-être plus claire, encore qu'une fois qu'on est habitué à C, ce ne soit pas si sûr, mais elle est en tout cas nettement moins efficace.

La notion de tableaux de pointeurs est donc très intéressante lorsqu'on les pointeurs en question correspondent à des chaînes de caractères. Un tel tableau est défini par :

```
char *tab[n];
```

Chaque élément d'un tel tableau est alors un pointeur sur des caractères, autrement dit une chaîne de caractères. Il est bien évident que dans ce cas, chacun des tableaux pointé a une longueur différente, ce qui prouve, si besoin était, l'intérêt de cette notion.

Ce tableau peut être initialisé comme tout autre tableau, par exemple :

```
char *jours[] = { "Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi",
"Dimanche" };
```

Comme d'habitude, le compilateur calculera la taille de ce tableau, en l'occurence, il allouera un tableau de 7 pointeurs, chacun d'entre eux pointant sur l'une des constantes chaînes de caractères correspondant à l'un des jours de la semaine. Si je veux afficher le nom du jour correspondant à un jour donné, je pourrai alors écrire l'instruction :

```
printf ( "Jour   = %s\n", jours[j] );
```

j donnant le numéro du jour désiré.

Comme un tableau est aussi une adresse à laquelle je peux associer un pointeur, je peux afficher dans l'exemple précédent tous les jours de la semaine ainsi :

```
char  **j;

for ( j = jours; j < jours + 7; j++ )
        printf ( "Jour = %s\n", *j );
```

Dans cet exemple "j" est un pointeur sur un pointeur sur un caractère, ou, dit autrement, un pointeur sur une chaîne de caractères; je peux donc faire parcourir le tableau en incrémentant "j" et en m'assurant que je ne déborde pas de ce tableau, l'affichage des chaînes se faisant lui en déréférençant le pointeur "j".

Je ne sais pas si vous vous en rappelez, mais dans le premier article de cette initiation, je vous avais présenté un programme affichant ses arguments, appelé "echoc". Je vous le redonne ici au cas où vous ne vous en souviendriez plus :

```
void main ( int argc, char argv[][] )
{
   int i;

   for ( i = 1; i < argc; i++ ) {
        printf ( argv[i] );
      if ( i < argc - 1 )
          printf ( " " );
     else
          printf ( "\n" );
  }
}
```

Ce programme déclarait "argv" comme étant un tableau de tableaux de caractères. Avec nos connaissances fraîchement acquises, nous pouvons maintenant réécrire ce programme en utilisant des pointeurs :

```
void main ( int argc, char **argv )
{
    while ( --argc > 0 )
          printf ( "%s%c", *++argv, ( argc > 1 ) ? ' ' : '\n' );
}
```

ce qui, encore une fois, est beaucoup plus concis donc plus efficace. De plus, je ne trouve pas que cette forme nuise à la lisibilité du programme.

Pointeurs et structures

Les pointeurs peuvent bien entendu être utilisés avec les structures; c'est même là qu'ils expriment toute leur puissance.

Comme pour toute autre variable de n'importe quel type, on peut prendre l'adresse d'une variable de type "struct" (ou "union") par l'opérateur "&" et affecter cette adresse à une variable de type pointeur sur cette structure.

Ainsi si j'ai déclaré :

```
struct  {
   ...
} s, *p;
```

je peux écrire p = &s;

Pour accéder à la structure et à chacun de ses membres, je vais utiliser les 2 opérateurs unaires "*" de déréférencement et "." d'accès aux membres, soit :

```
(*p).membre
```

Vous noterez que j'ai dû encadrer le déréférencement du pointeur par des parenthèses, car il se trouve que l'opérateur "." a une précédence supérieure à celle de "*". Je dois avouer que cette syntaxe est plutôt lourde, et que l'on peut se demander ce qui a poussé les concepteurs du langage à définir des règles de priorité imposant une telle construction.

Heureusement, ils ont eux aussi estimé que l'on pouvait mieux faire, et ont défini un opérateur spécifique, démontrant ainsi la relation très forte qu'il y a en C entre les pointeurs et les structures; vous ne verrez en effet aucun programme C concret qui n'utilise pas cette construction. Cet opérateur est représenté par le symbole "->" montrant ainsi la relation du pointeur pointant sur le membre de la structure. L'expression précédente s'écrit alors :

```
p->membre
```

Cet opérateur a une associativité gauche droite, ce qui implique que si une structure comporte un pointeur sur une autre structure, je peux accéder à un des membres de cette deuxième structure en suivant les liens, par exemple p->q->membre. Cette syntaxe montre bien le chemin que l'on doit suivre pour obtenir l'information recherchée.

Cet opérateur "->", de même que l'opérateur "." auquel il correspond directement, a une priorité très élévée, si bien qu'une expression de type ++p->membre sera interprétée comme ++(p->membre), c'est à dire que c'est le membre qui sera incrémenté et non le pointeur. Si p est en fait un pointeur à l'intérieur d'un tableau de structures, et qu'on veuille le faire pointer sur la structure suivante du tableau, il faudra explicitement l'isoler par des parenthèses, par exemple (++p)->membre, qui a pour effet d'incrémenter p avant d'avoir accédé au membre, tandis que (p++)->membre incrémentera p après y avoir accédé. Notez d'ailleurs que dans ce dernier cas les parenthèses sont inutiles, c'est à dire que l'expression p++->membre est correcte par rapport au résultat désiré, car il n'y a pas d'ambiguïté sur la variable à incrémenter; si j'avais voulu incrémenter le membre, j'aurais dû écrire p->membre++. Comme le cas de post-incrémentation d'un pointeur est plus fréquent que celui de la pré-incrémentation, on s'aperçoit qu'en général, il n'y a pas trop besoin de rajouter des parenthèses un peu partout.

De la même manière, l'opérateur "->" a une priorité plus élevée que celle de l'opérateur de déréférencement "*", c'est à dire que dans l'expression *p->membre, on accédera à la valeur pointée par le membre, lui même accédé en déréférençant p. En combinant ces différents opérateurs, on écrira *p->q++ pour incrémenter le pointeur q après l'avoir déréférencé, (*p->q)++ pour incrémenter l'objet pointé par q et *p++->q pour incrémenter p après avoir accédé à l'objet pointé par q.
Bien entendu, tout ce que je viens de dire s'applique indifféremment aux structures et aux unions.

Auto-références

Un des aspects les plus importants de l'utilisation des pointeurs avec les structures concerne la possibilité pour une structure de se référencer elle-même, comme par exemple dans la définition suivante :

```
struct s {
    ...
      struct s *next;
    ...
} *head;
```

Bien évidemment, cette auto-référence ne peut se faire qu'au travers d'un pointeur; autrement la structure s'incluerait elle-même à l'infini, ce qu'interdit le compilateur (et le bon sens). En revanche, dans la définition précédente, la structure comporte un membre qui est un pointeur vers une autre structure du même type. Ce type de définition permet de réaliser ce que l'on appelle des structures chaînées; je ne détaillerai pas leur utilisation, et je vous renvoie à ma série sur la programmation qui en donne des exemples quasiment à chaque article.

Ceci étant dit, sachez que ce type de déclaration est fondamental, car il est à la base de la majorité des programmes concrets en C. On accède à chacun desmembres de la manière que l'on a vue précédemment, y compris pour le membre "next", par exemple, avec les déclarations précédentes, head->next donne un nouveau pointeur sur un objet du type de la structure. Si j'ai aussi déclaré un pointeur "p" sur cette structure, je peux parcourir l'ensemble des objets de ce type chaînés entre eux par une construction telle que :

```
for ( p = head; p != NULL; p = p->next ) { ... }
```

Le pointeur "head" est utilisé pour pointer sur le premier élément de la série, tandis que le pointeur "next" du dernier contient NULL par convention, ce qui permet de déterminer quand on a atteint la fin de la série; "p" pointe alors tour à tour sur chacun des objets constituant la série.

En général, on combine ce type de définition avec la directive "typedef" de façon à améliorer la lisibilité du programme. On peut donc écrire l'exemple précédent sous la forme :

```
typedef struct s {
    ...
      struct s *next;
    ...
} *NODE;
```

ce qui définit un nouveau type "NODE" comme étant un pointeur sur la structure chaînée. Cependant, le pointeur vers le noeud suivant doit utiliser la déclaration complète (struct s) car NODE n'est pas encore défini. Dans ce cas précis, il est possible de faire une référence en avant, et donc de déclarer la structure sous cette forme :

```
typedef struct s *NODE;
struct s {
    ...
     NODE next;
};
```

Table de précédence des opérateurs complète

Maintenant que nous avons vu l'ensemble des opérateurs du langage C, je peux vous présenter à nouveau la table des priorités de ces opérateurs, dont je vous avais fourni une version réduite dans la troisième partie de cette initiation :

Opérateur	Associativité
() [] -> .	gauche-droite
! ~ ++ -- -unaire +unaire (type) * & sizeof	droite-gauche
* / %	gauche-droite
+ -	gauche-droite
<<>>	gauche-droite
< <= > >=	gauche-droite
== !=	gauche-droite
&	gauche-droite
^	gauche-droite
|	gauche-droite
&&	gauche-droite
||	gauche-droite
?:	droite-gauche
= += -= *= /= %= <<= >>= &= ^= |=	droite-gauche
,	gauche-droite

Je vous rappelle que la colonne associativité indique dans quel ordre sont évalués ces opérateurs, c'est à dire de gauche à droite dans la plupart des cas, et parfois de droite à gauche, notamment pour les opérateurs unaires.

Cette table est dans l'ordre des précédences décroissantes, c'est à dire que les premiers opérateurs cités sont ceux ayant la plus forte priorité.

Allocation dynamique de mémoire

Jusqu'à présent, toutes les définitions d'objets que nous avons vu, que ce soit des variables simples, des structures, des tableaux ou des pointeurs, correspondent en fait à des zones mémoire allouées par le compilateur de façon statique en fonction de la taille de ces objets; le programmeur se contente alors de donner un nom et une taille à chacun de ces objets en fonction de ses besoins.

Dans la vie réelle, il est très fréquent que l'on ne sache pas à priori la taille des données que l'on va manipuler dans un programme. On a alors 2 possibilités : dans la première, on prévoit une taille maximale et on dimensionne ses tableaux à cette taille; si la quantité des données à traiter est inférieure à cette taille, c'est tant mieux, sauf que l'on gaspille éventuellement de la mémoire, si on a prévu trop large par rapport à ce que l'on sera amené à traiter. En revanche, si l'on a prévu trop juste, le programme devra indiquer à l'utilisateur qu'il ne peut traiter le jeu de données fourni; si l'on dispose du source du programme, on peut éventuellement augmenter la taille de ses tableaux pour permettre le traitement des données désiré; dans le cas contraire, il faut se retourner vers le fournisseur du programme, et espérer qu'il veuille bien satisfaire la demande. Les anciens langages (tels que Fortran, Cobol ou Basic) n'offrent pas d'autre possibilité directement; il est parfois possible d'utiliser certains services du système d'exploitation mais ce n'est pas toujours évident lorsque le langage ne vous aide pas.

La deuxième possibilité est offerte par des langages plus récents tels que Pascal ou C : le programme s'alloue la mémoire en fonction de ces besoins, et ce au fur et à mesure; ainsi, il n'y a pas de gaspillage inutile, et la seule limite imposée est celle de la mémoire disponible. En revanche, ce type de programme est un peu plus complexe à réaliser et à mettre au point que celui utilisant des tableaux pré-dimensionnés, car il va faire généralement appel à des structures chaînées telles que nous venons de les voir. Il peut aussi s'allouer des tableaux d'une taille suffisante en fonction des données fournies, et les utiliser comme tels, cela ne posant aucun problème en C.

L'allocation dynamique de mémoire, puisque c'est comme cela qu'on appelle ce principe, est réalisée de façon différente selon les langages. Pascal, par exemple, dispose de l'opérateur NEW pour le mettre en œuvre; on lui donne l'objet que l'on veut matérialiser, et il se débrouille pour allouer la mémoire nécessaire pour le stocker, notamment en prenant en compte sa taille. D'un autre côté, de par son fonctionnement, NEW ne peut allouer qu'un seul objet à la fois, et il est impossible d'envisager de l'utiliser pour allouer une zone mémoire quelconque que l'on va utiliser ensuite comme un tableau (Pascal permet d'allouer un tableau dynamiquement, mais comme on doit préciser sa taille au compilateur et même définir ce tableau complètement, on retombe sur le problème du départ).

C, qui, comme je l'ai déjà dit, est un langage de plus bas niveau, ne fournit aucune primitive pour réaliser l'allocation dynamique de mémoire. De la même manière qu'avec les autres opérations 'avancées', on doit recourir à la librairie fournie avec le compilateur pour effectuer une telle allocation. Cette librairie fournit 2 fonctions d'allocation de mémoire : "malloc" et "calloc".

La fonction "malloc" admet un paramètre qui est la taille de la zone mémoire à allouer; il est de la responsabilité du programmeur de spécifier une taille suffisante pour stocker l'objet qu'il souhaite matérialiser; si ce n'est pas le cas, l'initialisation de l'objet pourra aller écraser la mémoire adjacente, avec toutes les conséquences que cela implique (c'est notamment souvent le cas avec les chaînes de caractères, car on oublie fréquemment de tenir compte du '\0' final).

Heureusement, C dispose de opérateur "sizeof" (dont nous avons déjà parlé), évalué lors de la compilation, qui permet d'obtenir la taille d'une variable ou d'un type. La fonction "malloc" sera donc généralement employée dans une construction telle que :

```
ptr = (type *) malloc ( sizeof ( type ) )
```

"malloc" retourne un pointeur sur le premier octet de la zone allouée. Comme cette fonction va être utilisée pour allouer de la mémoire pour des types d'objets divers et variés, elle retourne un pointeur générique (c'est à dire void *). Par conséquent, avant de pouvoir affecter son résultat à la variable qui pointera sur la zone allouée, il faut faire un cast explicite sur le type du pointeur.

Si le système ne peut fournir la quantité de mémoire demandée, "malloc" retournera NULL; il est donc de bon goût de tester la valeur retournée et d'effectuer un traitement approprié à ce cas, par exemple d'indiquer à l'utilisateur que ses données ne peuvent être chargées faute de mémoire suffisante.

La taille fournie à malloc est sur 32 bits, ce qui permet théoriquement d'allouer jusqu'à 4 giga-octets, bien plus que ce que vous aurez jamais besoin. En tout cas, vous pouvez allouer plus de 64 kilo-octets d'un seul tenant sans problème, et parcourir cette zone avec un pointeur, même en utilisant le modèle de mémoire dit petit (voyez votre manuel ORCA/C pour comprendre ce dont il s'agit), avec quelques restrictions toutefois, mais qui sont levées dans la version 2.0 du compilateur.

Notez que c'est le cas sur toutes les machines (du moins toutes celles que je connais) sauf sur PC où vous avez 2 fonctions d'allocation de mémoire, une si vous voulez allouer moins de 64 Ko, et une pour des tailles plus grandes; ceci est dû à l'architecture segmentée des processeurs Intel x86 (et au bricolage qu'est MS-DOS), par opposition à l'architecture linéaire du 65816 ou des Motorola 680x0. Et encore, ce n'est pas le pire, car vous devez indiquer si vos pointeurs pointent dans les 64 Ko courants ou pas avec des mots clefs spéciaux, totalement non portables, et sources de plantages divers et variés, sans parler des problèmes de compatibilité.

La fonction "calloc" a aussi pour but d'allouer de la mémoire. Elle présente 2 différences par rapport à la fonction "malloc". D'abord, elle admet 2 paramètres, au lieu d'un seul : le premier indique le nombre d'éléments à allouer tandis que le second donne la taille de chaque élément. Si par exemple, vous souhaitez allouer un tableau de N entiers, au lieu d'écrire :

```
tab = (short *) malloc ( n * sizeof ( short ) );
```

vous écrirez :

```
tab = (short *) calloc ( n, sizeof ( short ) );
```

C'est vrai que cela ne change pas grand chose. La seconde différence est plus importante : la zone mémoire allouée par "calloc" est initialisée à Ø alors que celle allouée par "malloc" contient n'importe quoi (en fait ce qu'il y avait à cet endroit avant l'allocation).

Selon les besoins, on utilisera l'une ou l'autre des fonctions; il est bien évident que l'initialisation d'une zone mémoire à Ø peut être pénalisante si cette zone a une taille conséquente et que cette initialisation n'est pas absolument nécessaire au bon fonctionnement du programme. Dans le cas contraire, on peut supposer que l'initialisation effectuée par "calloc" sera plus performante que celle que l'on devrait faire par soi-même; ceci est particulièrement utile avec les structures et il n'est donc pas rare de voir des constructions du style :

```
p = (struct s *) calloc ( 1, sizeof ( struct s ) );
```

c'est à dire l'allocation d'un seul élément d'un type structure donné, simplement pour l'avoir initialisé à Ø.

Une fois qu'un pointeur a été initialisé avec l'adresse d'une zone mémoire allouée dynamiquement, il s'utilise exactement de la même manière qu'un pointeur initialisé avec une variable statique. On peut alors utiliser toutes les possibilités que je vous ai présentées dans cet article et le précédent, notamment exploiter l'équivalence existant entre les pointeurs et les tableaux, ou bien les structures chaînées.

Par exemple, dans le cas d'un tableau, vous pourriez écrire :

```
tab = (short *) malloc ( 1000 );
```

ce qui alloue un tableau de 500 entiers de 16 bits. Vous pourriez l'utiliser ensuite comme un tableau "tab[i]" ou comme une zone pointée "*p++".

Lorsqu'on n'a plus besoin de cette mémoire allouée dynamiquement, il faut la libérer; la librairie C fournit pour cela la fonction "free". Celle-ci accepte un paramètre qui est un pointeur sur une zone mémoire allouée précédemment.

Il est extrêmement important de noter que ce pointeur doit correspondre à une adresse retournée par "malloc" ou "calloc" à l'exclusion de toute autre adresse, sinon vous allez droit au devant de nombreux problèmes, dont potentiellement le crash. Bien évidemment, cela signifie que l'on ne peut pas libérer un pointeur correspondant à une variable statique. Mais le type de problème auquel on peut être exposé se produit lorsqu'on a fait varier le pointeur retourné par malloc ou calloc, sans préserver cette adresse. Par exemple, avec le tableau tab alloué précédemment :

```
tab++;
free ( tab);
```

on est à peu près certain d'avoir des problèmes sérieux. Il est donc nécessaire de préserver tab, et d'utiliser un pointeur indépendant pour le faire parcourir la zone mémoire allouée.

Vous êtes peut-être en train de vous demander si il est réellement indispensable de libérer la mémoire qu'on n'utilise plus, et je vous réponds immédiatement que OUI. Il faut en effet impérativement libérer la mémoire qui ne devient plus nécessaire, essentiellement parce qu'il s'agit d'une ressource limitée, et qu'il est donc important de la consommer avec parcimonie : il est frustrant pour un utilisateur de voir

apparaître un message lui signalant qu'il n'y a plus assez de mémoire pour satisfaire la tâche qu'il veut effectuer, alors qu'il y a plein de mémoire inutilisée à ce moment. Imaginez par exemple que votre traitement de textes ne libérerait pas la mémoire occupée par chaque document; bien que vous ayez fermé toutes les fenêtres, il arrivera un moment où ce programme vous signalera qu'il ne peut ouvrir de nouveau document faute de mémoire; je pense que vous aurez du mal à accepter cette situation.

Même avec des systèmes d'exploitation plus sophistiqués, mettant en œuvre des mécanismes de mémoire virtuelle, la libération de la mémoire inutilisée est primordiale; dans le cas contraire, on risque aussi d'atteindre la limite autorisée, mais surtout les performances du programme ont tendance à se dégrader quand la mémoire n'est pas utilisée de façon optimale.

Ce phénomène de non libération de mémoire au fur et à mesure que l'on n'en a plus besoin s'appelle "fuites de mémoire" (en anglais "memory leaks") et c'est l'un des problèmes les plus classiques des programmes C, même commerciaux. La plupart des logiciels Mac ou PC et même ceux tournant sous Unix en sont d'ailleurs victimes. Ce problème semble moins important en Pascal, mais je pense que c'est dû d'une part au fait que l'allocation de mémoire en Pascal est très limitée, et par conséquent nettement moins utilisée qu'en C; d'autre part, il n'y a quasiment pas de programmes Pascal vraiment conséquents, en particulier dans les logiciels commerciaux.

Sachez toutefois que le fait de quitter un programme libère implicitement toute la mémoire qu'il a allouée au cours de son exécution, et que par conséquent, la libération explicite de la mémoire est surtout importante pour les programmes interactifs, avec lesquels un utilisateur peut travailler pendant des heures sans le quitter, tout en changeant plusieurs fois de document de travail.

La librairie C standard comporte une dernière fonction d'allocation de mémoire dynamique : "realloc". Cette fonction permet d'agrandir ou de réduire une zone mémoire précédemment allouée par "malloc" ou "calloc". Elle accepte 2 paramètres qui sont l'ancienne valeur du pointeur et la nouvelle taille de la zone. Si l'ancienne valeur du pointeur est NULL, "realloc" effectue un "malloc", tandis que si la nouvelle taille est Ø, cette fonction se comporte comme un "free". Lorsque la nouvelle taille demandée est plus grande que la taille actuelle, il est possible que l'on ne puisse pas agrandir la zone mémoire à l'endroit où elle est allouée; dans ce cas, "realloc" alloue une nouvelle zone mémoire, et recopie l'ancienne dans la nouvelle, puis la libère.
Bien entendu, l'ancienne valeur du pointeur ne correspondra alors plus à une zone allouée au programme, et il ne faudra donc plus l'utiliser; dans la pratique, il n'y a pas de risques, car on emploiera une expression telle que p = realloc (p, nouvelle_taille) et le pointeur correspondra directement à la nouvelle zone.

Pointeurs et fonctions

==========

En C, il est possible de déclarer un pointeur sur une fonction, c'est à dire dont la valeur est l'adresse de l'une des fonctions du programme. Ce pointeur peut ensuite être manipulé comme tout autre pointeur, par exemple constituer un élément d'un tableau ou d'une structure, ou bien être passé en paramètre à une autre fonction. Cette possibilité permet ainsi de développer des bibliothèques de fonctions génériques. Par exemple, on peut envisager d'écrire une fonction de tri d'un tableau; un des paramètres de cette fonction sera un pointeur sur une fonction de comparaison; la fonction de tri pourrait alors être générique, et le programme qui l'utilise se chargerait de lui fournir une fonction annexe de comparaison tenant compte du type des éléments du tableau à trier.

Pour pouvoir affecter l'adresse d'une fonction à un pointeur, cette fonction doit avoir été déclarée au préalable, afin que le compilateur sache à quoi il a affaire; ceci étant fait, il suffit d'affecter le nom de la fonction au pointeur (sans les parenthèses qui sont utilisées pour appeler la fonction, même si elle n'a pas de paramètres), de la même manière qu'on donne juste le nom d'un tableau pour obtenir son adresse. Ainsi, si j'ai déclaré void func (...) et void (*p)(), l'expression p = func; donne à p l'adresse de la fonction func.

La syntaxe de déclaration du pointeur p semble un peu lourde à priori.

Encore une fois, il s'agit d'un modèle montrant la syntaxe qu'il faudra utiliser pour effectuer un appel indirect à cette fonction, ce qui donnera donc : (*p)();. Les parenthèses sont indispensables sinon p serait interprété comme étant une fonction retournant un pointeur sur void, ce qui n'est pas du tout le but recherché. Vous vous apercevrez que l'appel à la fonction est cohérent avec la déclaration : p est un pointeur sur une fonction, *p est la fonction elle-même, et (*p)() est l'appel à cette fonction, de la même manière que func() appelle la fonction func.

Comme la syntaxe d'appel à une fonction via un pointeur est un peu lourde, la norme ANSI C, et donc les compilateurs qui la respectent, permet d'utiliser la syntaxe classique d'appel. Ainsi, dans l'exemple précédent, je pourrai appeler la fonction pointée par p en écrivant p() au lieu de (*p)().

Personnellement, je préfère employer la syntaxe explicite de déréférencement du pointeur, afin de bien voir qu'il s'agit d'un appel indirect, mais c'est juste une affaire de goût.

La possibilité de faire des tables de fonctions et de passer en paramètre l'adresse d'une fonction constitue, vous vous en doutez, l'un des points les plus forts du langage C, car il permet, comme je l'ai dit un peu plus haut de se constituer des bibliothèques de fonctions génériques. C'est aussi ce type de mécanisme qui est une des bases des langages orientés objets; on peut mettre en œuvre ces concepts directement avec le langage C, même si ils sont alors explicites, au lieu d'être implicites comme avec un véritable langage orienté objets. Pour vous donner un exemple concret, les interfaces utilisateurs graphiques des stations Unix, basées sur le système X Windows sont quasiment toutes écrites en C, mais elles utilisent des tonnes de pointeurs sur des fonctions pour implémenter les mécanismes liés aux objets.

Je vous recommande d'étudier les librairies accompagnant mes articles sur la programmation; elles représentent différents exemples d'utilisation des pointeurs avec les fonctions, mais aussi illustrent l'ensemble des concepts que nous avons étudiés dans ces 2 derniers articles.

Conclusion

==========

Avec cet article, nous avons terminé l'étude de l'ensemble des concepts du langage C. Toutefois, comme vous avez pu le constater à plusieurs reprises, la librairie C standard est indissociable du langage. Nous conclurons donc cette initiation avec le prochain article en étudiant les différentes fonctions qui la composent, notamment au niveau des entrées/sorties.

Chapitre 9

La librairie C

Toutes les bonnes choses ont une fin : voici donc la dernière partie de mon initiation au langage C. J'espère que vous avez pu commencer à programmer dans ce langage grâce à cette initiation.

Comme vous avez pu le constater à plusieurs reprises lors de cette initiation, j'ai fait plusieurs fois référence à la librairie associée au langage. En effet, un certain nombre d'opérations ne sont pas réalisées par des instructions du langage (c'est le cas par exemple des manipulations de chaînes de caractères ou des entrées/sorties), mais par des fonctions présentes dans une librairie qui est fournie avec le compilateur.

Ces fonctions sont si importantes pour le développement de programmes en C que les comités de normalisation ANSI puis ISO ont défini une librairie minimale devant accompagner chaque compilateur se targuant d'implémenter le standard C.
Il est bien évident que le but de cette normalisation est de faciliter le portage des programmes d'un environnement à un autre.

On s'aperçoit en effet dans la pratique que la difficulté de portage d'un programme, dans le cas du C, ne vient pas tant des différences du langage lui-même (il est suffisamment complet pour ne pas nécessiter d'extensions, ou du moins on peut s'en passer si on a l'intention d'écrire un programme portable), que des différences dans les librairies. Si vous regardez le source d'un programme C destiné à être compilé dans plusieurs environnements, vous découvrirez une grande quantité de directives de compilation conditionnelle, nécessitées par la prise en compte de librairies C non standards.

La librairie minimale définie par le C standard est toutefois assez complète (ce qui ne veut pas dire que tous les compilateurs C, censés respecter la norme, fournissent une librairie conforme), notamment par rapport à ce que l'on peut trouver dans d'autres langages, comme le Pascal par exemple. Il suffit de comparer les chapitres décrivant les librairies d'ORCA/C et d'ORCA/Pascal pour s'en convaincre, sachant que la moitié des routines de cette dernière sont des extensions.

Les fonctions que l'on trouve dans la librairie C standard sont issues pour la plupart de la tradition, c'est à dire de ce que l'on trouvait habituellement dans les librairies C principalement venues du monde Unix, sachant qu'il y a un certain nombre de différences entre la version System V de AT&T et la version de l'université de Berkeley dite BSD. Ce sont ces différences qui posent souvent le plus de problèmes lors du portage, car la plupart des utilitaires du domaine public (que l'on pourrait avoir envie de porter sur le GS ou sur une autre plateforme) ont été développés dans les universités américaines, sous BSD, qui propose une librairie beaucoup plus riche que celle d'AT&T. La librairie C standard est elle plutôt basée sur celle de System V, plus répandu au moment de la normalisation (à moins qu'AT&T n'ait fait plus de pression !). Le comité de standardisation a aussi inventé plusieurs nouvelles fonctions qui n'existaient pas dans les librairies C de l'époque.

Mon but n'est pas de vous détailler l'ensemble des fonctions de la librairie fournie avec le compilateur ORCA/C (je vous renvoie au manuel pour cela), mais plutôt de vous indiquer les domaines couverts par cette librairie. Il est intéressant de noter qu'ORCA/C ne peut pas être considéré comme un compilateur à la norme à part entière puisque sa librairie n'est pas complète. Il est vrai que certaines des fonctions manquantes sont difficiles à réaliser sur le GS, car elles s'appuient sur des services système qui n'existent pas. D'autres en revanche seraient les bienvenues, comme celles permettant l'internationalisation des programmes.

Comme je fréquente pas mal d'environnements (plusieurs variétés d'Unix, MS-DOS, VAX/VMS), je me suis attaché à compléter la librairie d'ORCA/C par des fonctions utiles que j'ai rencontré dans l'un ou l'autre de ces environnements.

J'en ai profité pour réaliser une implémentation de certaines fonctions définies par la norme et absentes de la librairie fournie par ByteWorks. Enfin, lorsque j'ai eu besoin d'une fonction et que je l'ai considéré être suffisamment générale, je l'ai ajouté à l'une de mes librairies, en partant du principe que je pourrais en avoir besoin à un autre moment. Je me suis donc constitué au cours du temps plusieurs librairies complémentaires de celle d'ORCA/C.

Si je vous en parle ici, c'est non seulement pour vous encourager à vous constituer des librairies de fonctions que vous réutiliserez dans tous vos programmes, mais aussi parce que j'ai inclus 3 de mes librairies sur cette disquette GS Infos. La documentation des fonctions présentes dans ces librairies n'est malheureusement constituée que des commentaires en tête de chacune d'entre-elles; la plupart de ces fonctions sont cependant triviales à utiliser et leurs noms et leurs paramètres doivent suffire à déterminer leur syntaxe d'appel. J'ai toutefois placé sur la disquette des programmes d'exemple illustrant l'essentiel de ces fonctions; certains de ces programmes sont des petites commandes shell que vous placerez à l'endroit approprié (c'est à dire dans le sous-catalogue "Utilities" du shell ORCA), tandis que les autres sont purement des démonstrations. Je détaillerai toutefois un peu plus les fonctions présentes dans ces librairies, lorsque je traiterai dans la suite de cet article les domaines auxquels elles s'appliquent.

Ces librairies sont :

• **StrLib** : comme son nom l'indique, cette librairie offre des fonctions de manipulation de chaînes de caractères. Elle comporte plus de 30 fonctions permettant de réaliser toutes les opérations imaginables sur des chaînes C, c'est à dire terminées par un Ø.

• **Direct** : cette librairie permet de travailler avec des directories; elle contient des fonctions de récupération et de changement des préfixes, des fonctions de création et de suppression de directory, et surtout des fonctions de parcours de directory.

• **MiscLib** : cette librairie contient un ensemble de fonctions et de macros diverses et variées, qui ne trouvaient pas leur place dans les autres librairies. Vous y trouverez notamment des fonctions permettant de traiter les options d'une commande, de faire l'expansion d'un nom de fichier comportant les wildcards standard du shell ORCA, d'obtenir les attributs d'un fichier, de lire et d'écrire n'importe où en mémoire, d'interroger le clavier, ...

Un aspect important de l'utilisation de librairies par un programme C, que ce soit les miennes ou la librairie standard, ou n'importe quelle autre librairie, est l'interfaçage entre votre programme et ces librairies. Cette interface est réalisée par l'intermédiaire d'un fichier "header" identifié par son suffixe ".h" que l'on inclut au début du programme. Dans le cas de la librairie standard, vous trouverez un fichier header par domaine d'application. Tous ces fichiers headers sont rassemblés dans un répertoire défini par l'environnement de développement que vous utilisez (dans le cas d'ORCA/C, il s'agit de .../Libraries/OrcaCDefs); une syntaxe spéciale de la directive #include indique au compilateur qu'il doit exploiter ces fichiers headers et non ceux présents dans le répertoire du source compilé.

Pour indiquer au compilateur que vous souhaitez utiliser les fichiers header d'interface avec les librairies, vous utiliserez la syntaxe #include <header.h> tandis que pour inclure les fichiers header propres à votre programme, vous utiliserez la syntaxe #include "header.h".

La librairie C standard recouvre plusieurs grands domaines :

• Des fonctions permettant de réaliser des entrées/sorties aussi bien avec des fichiers, qu'avec le clavier et l'écran.

• Un ensemble assez complet, mais enrichissable, de fonctions de manipulation de chaînes de caractères.

• Des fonctions de classification des caractères et de conversion entre des nombres et des chaînes.

• Les fonctions mathématiques les plus usuelles ainsi que quelques unes que l'on trouve plus rarement.

• Des fonctions permettant de contrôler l'exécution du programme ainsi que des fonctions permettant d'interagir avec son environnement.

• Des fonctions d'allocation et de libération de mémoire dynamique.

• Des fonctions assez complètes de manipulation du temps.

• Des fonctions diverses n'entrant pas dans les catégories précédentes.

Voyons donc maintenant un peu plus en détail ce qu'offre la librairie d'ORCA/C dans chacun de ces domaines.

Les entrées/sorties

Le modèle d'entrées/sorties proposé par la librairie C est directement inspiré de celui d'Unix, c'est à dire qu'un fichier est vu comme une séquence continue d'octets, sans aucune structure interne prédéfinie. C'est donc à la charge de chaque application de définir le format et la longueur de ses enregistrements. Ce modèle est adéquat pour le GS, puisque c'est aussi de cette manière que GS/OS traite les fichiers.

Si vous étudiez attentivement la documentation, vous découvrirez qu'il y a 2 ensembles de fonctions permettant de lire ou d'écrire dans un fichier : les premières sont précédées par la lettre "f" comme "file" ou "fichier" et travaillent avec un pointeur sur une structure "FILE" sur laquelle nous allons revenir, tandis que le second jeu de fonctions utilise un entier pour identifier le fichier; de plus, le nombre de fonctions est beaucoup plus limité.

L'existence de ces 2 jeux de fonctions est historique : sous Unix, le deuxième ensemble de fonctions décrit ci-dessus ne fait pas partie de la librairie C, mais correspond à des services système, et sont donc équivalents aux appels que vous pouvez faire à GS/OS; sous Unix, le premier jeu s'appuie sur ces fonctions pour fournir des services plus évolués. Comme ces services de bas niveau sont parfois utilisés, la plupart des librairies C les émule en tant que fonctions de ces librairies.

En dehors de cet aspect historique, il y a dans la plupart des implémentations (c'est notamment le cas sur le GS) une différence importante entre ces 2 jeux de fonctions : les fonctions émulant les services système d'Unix (ainsi que ces services eux-mêmes dans le cas d'Unix) effectuent des entrées/sorties non bufferisées tandis que les fonctions travaillant sur une structure FILE maintiennent un buffer sur le fichier.

La structure FILE associe, entre autres, l'adresse du buffer à l'identifiant du fichier (dans le cas du GS, il s'agit du numéro de référence GS/OS) ainsi que la position dans le buffer.

Ainsi, lorsque vous demandez à lire par exemple 20 octets, la librairie lit d'office 1024 octets (par défaut), puis, lors des lectures suivantes, vous retourne les octets suivants du buffer jusqu'à son épuisement, auquel cas elle réeffectue une nouvelle lecture du disque. De la même manière, le buffer n'est écrit sur disque que lorsqu'il est plein. La librairie C vous permet de changer la taille de ce buffer, voire même de l'annuler complètement, et même de vider le buffer lorsque vous écrivez un caractère de fin de ligne (en C, c'est le fameux "\n") grâce aux fonctions "setvbuf" et "setbuf" (la deuxième est la forme ancienne et est à considérer comme obsolète). Vous pouvez aussi vider le buffer quand vous le souhaitez en appelant la fonction "fflush". Bien évidemment le buffer sera aussi vidé lors de la fermeture du fichier.

Les fonctions émulant les services système d'Unix effectuent donc des entrées/sorties directes, c'est à dire que dans le cas du GS, elles se contentent d'appeler GS/OS. Ces fonctions sont peu nombreuses, puisqu'on ne trouve que "creat", "open", "read", "write", "lseek" et "close", ainsi que "fcntl" qui permet de contrôler certains aspects du fichier; la version de la librairie d'ORCA/C ne permet que de dupliquer le numéro d'identification du fichier (qui n'est pas le numéro de référence GS/OS, mais un index dans une table interne contenant le numéro de référence), et donc d'avoir 2 chemins d'accès au même fichier. Cette tâche est aussi assurée par la fonction "dup"; les 2 sont présentes uniquement pour assurer la compatibilité avec des programmes venant d'Unix. Bien évidemment, toutes ces fonctions ne font pas partie de la librairie normalisée qui ne définit que le jeu de fonctions d'entrées/sorties bufferisées, et qui sont donc appelées 'standard'.

Les fonctions de la librairie d'entrées/sorties standard ont bien entendu les mêmes noms que leurs consœurs, mais leurs noms sont précédés par un "f" : vous trouverez donc un "fopen", un "fread", un "fwrite" et un "fclose"; en revanche, il n'y a pas de "fcreat", ceci étant effectué par une option de "fopen" (d'ailleurs "creat" appelle aussi "open" avec des options particulières).

La librairie permet de distinguer entre des fichiers texte, dans lesquels sont définies des lignes séparées selon les environnements par un CR et/ou un LF, et des fichiers binaires qui ne contiennent qu'une suite d'octets sans signification particulière (du moins pour la librairie). Vous devrez indiquer quel type de fichier vous voulez traiter lors de l'ouverture. Cependant, les fonctions "fread" et "fwrite" lisent une suite d'octets sans tenir compte de leur valeur, quelque soit le type du fichier. Lorsque vous lisez ou écrivez un fichier texte, vous utiliserez donc plutôt les fonctions "fgets" et "fputs" qui s'arrêtent dès qu'elles rencontrent un caractère de fin de ligne.

Vous pouvez aussi utiliser les fonctions "fgetc" et "fputc" qui vous permettent de lire et d'écrire caractère par caractère et donc de les traiter au fur et à mesure que ceux-ci sont lus ou écrits. La librairie dispose aussi d'une fonction "ungetc" qui permet de renvoyer le caractère qui vient d'être lu (en fait on peut renvoyer n'importe quel caractère puisqu'on indique lequel) dans le fichier; ceci est utile par exemple lorsque vous décodez un fichier caractère par caractère et que vous vous apercevez que vous venez de lire un caractère de trop, vous le renvoyez dans le fichier afin que la prochaine lecture vous donne à nouveau ce caractère. ORCA/C ne permet de renvoyer qu'un seul caractère. En fait, celui-ci n'est pas réellement renvoyé, mais est stocké dans un des champs de la structure FILE.

Histoire de vous compliquer un peu la vie, certaines de ces fonctions (les "fputx" et les "fgetx") existent aussi sans le "f" de tête; ces fonctions appartiennent quand même à la même famille, ce sont simplement des raccourcis. Dans le cas de "getc" et de "putc", il s'agit de macros tandis que les "fgetc" et "fputc" correspondant sont de vraies fonctions; 2 autres macros sont définies dans le fichier header "stdio.h" correspondant à toutes ces fonctions : "getchar" et "putchar" permettent de lire et d'écrire un caractère directement au clavier et à l'écran. Toutefois dans le cas de "getchar", il ne s'agit pas d'une saisie au vol puisqu'il faut taper un retour chariot pour terminer la saisie; en fait "getchar" retourne successivement chacun des caractères tapés avant le caractère de fin de ligne.

Certaines librairies proposent une fonction "getch" de saisie d'un caractère au vol; cette fonction est souvent accompagnée par la fonction "kbhit" qui permet de savoir si il y a un caractère saisi en attente ou non. Vous trouverez une implémentation de ces 2 fonctions dans ma librairie "MiscLib". J'ai aussi écrit les fonctions "clrkb" réinitialisant l'état du clavier, et "getmod" récupérant l'état des touches de modification (majuscule, commande, option, contrôle, ...) de façon à prendre en compte les spécificités du GS. Des symboles définis dans "MiscLib.h" sont associés à chacune des touches de modification.

Pour "gets" et "puts", il s'agit de fonctions de lecture d'une ligne au clavier et d'écriture d'une chaîne à l'écran; attention : ce ne sont pas des macros indiquant que le fichier est "stdin" ou "stdout", mais de véritables fonctions, qui, en plus, ont une sémantique légèrement différentes de leurs consœurs : dans le cas de "fgets", le caractère de fin de ligne est placé en fin de chaîne tandis qu'il ne l'est pas avec "gets"; pour "fputs", il n'y a pas d'ajout automatiquement de caractère de fin de ligne après la chaîne (sur la base qu'il y en a déjà un provenant de "fgets"), alors que "puts" lui en envoie un systématiquement.

Vous pouvez enfin utilisez les fonctions "fscanf" et "fprintf" qui vous permettent de contrôler ce qui est lu ou écrit, en définissant précisemment le format de chacune des variables que vous souhaitez lire ou écrire dans le fichier. Ces fonctions ont chacune 2 variantes : les fonctions "scanf" et "printf" sont utilisées pour lire au clavier et écrire sur l'écran; les fonctions "sprintf" et "sscanf" permettent elles d'écrire le résultat formaté dans une chaîne de caractères, et de décoder une chaîne dans différentes variables. La norme définit aussi pour les fonctions "xprintf" une version acceptant une référence sur une liste d'arguments variables; ces fonctions sont identiques aux autres de la famille "printf", elles sont juste précédées par un "v" : on a donc "vprintf", "vsprintf" et "vfprintf". Ces fonctions étaient absentes de la librairie d'ORCA/C v1.x, mais font leur apparition avec ORCA/C v2.0. En attendant, vous en trouverez une implémentation triviale bien que complète dans ma librairie "MiscLib" (j'ai d'ailleurs ajouté un mécanisme de compilation conditionnelle pour ne pas les inclure si vous avez ORCA/C 2.0; toutefois, la librairie déjà générée sur la disquette les contient). L'intérêt de ces fonctions est de pouvoir offrir les possibilités de formatage de "printf" dans une fonction propre au programme, par exemple une fonction d'affichage de message d'erreur ou d'écriture dans une fenêtre via la Toolbox.

La librairie comporte plusieurs fonctions permettant d'obtenir la position dans le fichier et de se repositionner : "rewind", "ftell" et "fseek" sont destinées à être utilisées ensemble, de même que "fgetpos" et "fsetpos"; les premières sont les fonctions originales, tandis que les secondes ont été inventées par la norme. Les macros "ferror" et "feof" permettent de savoir si une erreur s'est produite (bien que chaque fonction indique si elle s'est déroulée correctement ou pas), et si on a atteint la fin du fichier. La macro "clearerr" permet d'annuler l'indicateur d'erreur.

La librairie contient enfin quelques fonctions annexes telles que "remove" pour supprimer un fichier ou "rename" pour le renommer.

Bien qu'elle soit déjà fort riche, plusieurs fonctions utiles manquent à l'appel et surtout existent ailleurs, et sont donc utilisées dans pas mal de programmes; je les ai donc écrites et ajoutées dans ma librairie "MiscLib" : la macro "fileno" permet d'obtenir le numéro de référence GS/OS associé à une structure FILE. La fonction "access" permet de savoir si un fichier existe ou non; de plus un flag indique si l'on veut aussi vérifier si on peut lire ou écrire le fichier en fonction de ses droits d'accès. La fonction "fgetname" retourne le nom complet d'un fichier ouvert; ceci est utile si le fichier a été donné avec un chemin partiel.

Enfin, les fonctions "stat" et "fstat" permettent de récupérer les attributs d'un fichier; ces 2 fonctions sont identiques : la première est utilisée avec un nom de fichier tandis que la seconde attend une structure FILE sur un fichier ouvert. Il est à noter que la fonction "fstat" s'utilise en principe avec un numéro de fichier, bien qu'elle commence par un "f", mais comme les fonctions correspondantes sont marginalement utilisées sur le GS, j'ai préféré faire une légère entorse à la compatibilité. Ces 2 fonctions sont basées sur la primitive "GetFileInfo" de GS/OS, et elles réorganisent les informations dans une structure C compatible avec d'autres implémentations de "stat"; de plus, les dates de création et de modification sont converties dans le format de la librairie C.

Au dela de ces fonctions de manipulation de fichiers, certaines librairies, notamment sous Unix, offrent un ensemble de fonctions permettant de travailler avec des directories. Je me suis donc attaché à développer une librairie (appelée "Direct") implémentant toutes les fonctions que l'on peut trouver dans ces librairies étrangères; j'ai essayé d'être le plus compatible possible avec ces librairies, aussi bien au niveau de l'interface (nombre et type des paramètres) que de la sémantique des fonctions (elles se comportent de la même manière et renvoient les mêmes erreurs). Mon implémentation diffère parfois de ces librairies, soit parce que GS/OS ne permettait pas d'émuler complètement la fonction correspondante, soit aussi qu'il avait plus à offrir, et j'ai alors exploité les possibilités supplémentaires.

La librairie "Direct" comporte 4 catégories de fonctions :

• Des fonctions permettant d'obtenir et de changer le préfixe courant, ainsi que n'importe lequel des préfixes gérés par GS/OS. Ces fonctions sont "getcwd", "getwd" (il s'agit d'une forme obsolète de la précédente, qu'il vaut mieux donc éviter d'employer) et "getdir_gs" pour la récupération de la valeur d'un préfixe GS/OS; les 2 premières obtiennent le directory par défaut représenté par le préfixe 8 (et/ou 0), tandis que la dernière permet de récupérer la valeur de n'importe quel préfixe GS/OS. Les fonctions "chdir" et "chdir_gs" permettent elles de changer le directory par défaut (encore une fois, il correspond au préfixe 8 et 0) ainsi que n'importe quel autre préfixe, à l'exception des préfixes 10, 11 et 12 réservés pour les entrées/sorties standard sur la console.

• Des fonctions permettant de créer et d'effacer des directories; dans ce dernier cas, le directory doit être vide, c'est à dire que les fichiers qu'il contient doivent être supprimés au préalable. Ces fonctions sont respectivement "mkdir" et "rmdir".

• Des fonctions permettant de lire un directory, nom de fichier par nom de fichier.
Je dirais que ces fonctions sont de bas niveau. La lecture d'un directory est réalisée par les 3 fonctions "opendir", "readdir" et "closedir"; ces fonctions travaillent avec un pointeur sur une structure DIR de la même manière que leurs équivalentes pour les fichiers utilisaient un pointeur sur une structure FILE. Trois fonctions de repositionnement complètent cette catégorie : "telldir" pour connaître la position actuelle et "seekdir" et "rewinddir" pour se repositionner, la première sur une entrée retournée par "telldir" et la seconde au début du directory.

• 2 fonctions de parcours de directory de plus haut niveau, s'appuyant sur les fonctions de la catégorie précédente. La fonction "scandir" retourne une liste triée (en principe dans l'ordre alphabétique) de tout ou partie des noms de fichiers d'un directory; cette fonction accepte comme paramètre l'adresse d'une routine de sélection qui indique en retour si le fichier doit être mis dans la liste ou pas; il est possible de ne pas employer cette possibilité et donc de sélectionner tous les fichiers. On utilisera la fonction "stat" décrite précédemment pour obtenir les attributs de chaque fichier, si on veut les utiliser comme critères de sélection. Un autre paramètre définit une fonction de comparaison pour le tri; la librairie fournit une fonction de comparaison alphabétique ne tenant pas compte des majuscules et des minuscules, si on souhaite utiliser ce défaut. Cette fonction, dénommée "alphasort" peut être utilisée dans d'autres contextes.

La fonction "ftw" (pour file-tree-walk) permet de parcourir l'arborescence de directories à partir d'un directory donné et sur une certaine profondeur; elle appelle une fonction utilisateur (dont l'adresse constitue un des paramètres) pour chacun des noms de fichiers trouvés. Elle a déjà appelé la fonction "stat" et elle passe les informations correspondantes à la fonction utilisateur, ainsi qu'un flag indiquant si le nom correspond à un fichier ou à un directory.

Toutes ces fonctions de manipulation de directories sont illustrées par les 2 programmes d'exemples "dtree" pour "ftw" et "testdir" pour toutes les autres.

Les chaînes de caractères

Comme je l'ai déjà écrit plusieurs fois, il n'y a pas de type "chaîne de caractères" en C, et par conséquent aucune instruction ne permet de les manipuler. Toutefois, il est évident qu'un grand nombre de programmes est amené à travailler avec de telles données. En C, cela se fait au travers d'un ensemble de fonctions de la librairie.

La librairie C comporte 2 catégories de fonctions de manipulation de chaînes de caractères : la première est identifiée par le préfixe "str" et la seconde par le préfixe "mem". Les fonctions dont le nom commence par "str" travaillent avec des chaînes de caractères dites C; elles sont caractérisées par une séquence d'octets quelconques, et de longueur quelconque aussi, et terminées par un Ø. Les fonctions dont le nom commence par "mem" travaillent elles sur une séquence d'octets quelconques et de longueur quelconque, mais sans terminateur particulier; en l'occurence, la séquence d'octets en question peut contenir des Ø n'importe où.

Avant de détailler les fonctions de la librairie, il me semble utile d'apporter quelques remarques préliminaires sur les chaînes C. Elles ont fait couler beaucoup d'encre, notamment de la part des supporters d'un véritable type chaîne de caractères, dans lequel est maintenu séparément la longueur de la chaîne, comme cela est le cas avec les chaînes dites Pascal, précédées par un octet de longueur (ce qui a pour inconvénient de limiter la longueur de ces chaînes à 255 caractères), ainsi que les chaînes GS/OS qui sont elles précédées par un mot de 16 bits pour indiquer la longueur.

En effet, certaines opérations aussi triviales que l'évaluation de la longueur d'une chaîne ou la concaténation de 2 chaînes demandent à parcourir la chaîne dans son entier et de compter les caractères jusqu'à ce que l'on rencontre le fameux Ø final, ce qui, bien évidemment, coûte assez cher en termes de performances. La conséquence de ceci est qu'il faut maintenir la longueur de la chaîne à côté de cette dernière, afin de minimiser ce type d'opération.

En revanche, cette représentation permet de faire toutes sortes de manipulations à l'intérieur même de la chaîne, sans nécessiter de la recopier dans une autre chaîne.

Supposez par exemple que vous vouliez séparer tous les mots d'une chaîne et que chaque mot forme une nouvelle chaîne; au lieu de recopier chaque mot dans une nouvelle chaîne, vous pouvez remplacer les blancs par des Ø et retourner un pointeur sur le premier caractère de chacun des mots. Je vous conseille d'étudier n'importe quel programme C travaillant sur des chaînes pour avoir une idée plus précise de tout ce qu'il est possible de faire.

Ceci dit, la facilité de manipulation de chaînes de caractères n'est toutefois pas sans risques. Le reproche le plus sérieux qui est fait aux chaînes C est que l'on a aucune idée de la longueur réelle de la chaîne, dans le sens où on ne sait pas très bien si la mémoire après le Ø terminateur appartient à la chaîne ou pas. Un des bugs classiques des programmes C est justement d'aller écraser des zones mémoires qui ne leur appartiennent pas lors de la manipulation de chaînes, d'autant que la librairie ne fait strictement aucun contrôle.

Un autre problème fréquent se produit lorsque les chaînes ont été allouées de façon dynamique; comme n'importe quel caractère de la chaîne est le début d'une nouvelle chaîne, sous-chaîne de la chaîne initiale, on a tendance a écrire des fonctions retournant des pointeurs sur le milieu d'une chaîne. Je vous l'ai dit dans mon précédent article, il est formellement interdit d'appeler "free" sur un pointeur non retourné par "malloc", "calloc" ou "realloc"; malheureusement, il arrive très fréquemment que "free" soit appelé avec des pointeurs correspondant à des milieux de chaîne, et ce y compris dans des programmes commerciaux parmi les plus connus. Je vous laisse imaginer les problèmes souvent obscurs qui peuvent en découler. La morale de cette histoire est qu'il faut bien faire attention à sauver le pointeur retourné par l'une des fonctions d'allocation, quelles que soient les manipulations faites par la suite. Il est classique de voir des séquences de code C écrites ainsi :

```
str = s = malloc ( longueur );
manipulations sur s (du type incrémentation du pointeur)
free ( str );
```

Si vous écrivez vos programmes C de cette manière, vous n'aurez jamais de problèmes. Dans mon article sur le debugging toujours à venir, je vous montrerai une librairie au dessus des routines d'allocation permettant de s'assurer qu'on libère bien ce qu'on a alloué.

Nous pouvons maintenant voir ce que la librairie C offre pour manipuler des chaînes de caractères :

• Puiqu'on ne connaît pas à priori la longueur d'une chaîne, il nous faut donc une fonction permettant de la calculer; ceci est fait grâce à la fonction "strlen".
Comme je l'ai écrit précédemment, cette fonction compte les caractères de la chaîne jusqu'au Ø final, et retourne cette valeur. Il est donc habile de sauver ce résultat dans une variable intermédiaire et éventuellement réajuster cette variable si la chaîne est modifiée, plutôt que de recalculer la longueur de la chaîne à chaque fois qu'on en a besoin.

• Plusieurs fonctions permettent de copier une chaîne dans une autre : "strcpy" est la fonction de base. Elle a une variante appelée "strncpy" qui permet de ne copier qu'une sous-chaîne d'une longueur spécifiée en paramètre; si cette longueur est plus grande que la longueur totale de la chaîne, cette fonction est strictement identique à sa sœur. Pour être complet, je dois préciser que cette fonction comporte un piège : si on ne copie pas tous les caractères de la chaîne initiale, la chaîne réceptrice n'aura pas de Ø final après cette fonction; si vous le souhaitez, vous pouvez l'ajouter vous-mêmes, ce qui donne par exemple :

```
strncpy ( dest, source, n );
dest[n] = '\0';
```

Dans le cas où toute la chaîne est copiée, le Ø final sera bien présent; notez qu'ORCA/C complète la chaîne destinatrice par une série de Ø si elle a rencontré le Ø final dans la chaîne source avant d'avoir copié les N caractères, mais cela n'est pas nécessairement le cas dans d'autres librairies; vous ne devez pas utiliser ce fait (d'ailleurs, ça ne me paraît pas très utile).
Les fonctions "memcpy" et "memmove" fonctionnent sur le même principe que "strncpy", sauf, comme je l'ai écrit plus haut, qu'elles copient systématiquement les N octets demandés, sans tenir compte de la présence éventuelle d'octets à Ø.

Pour toutes les fonctions "xxxcpy", les 2 zones mémoire correspondant aux chaînes de caractères ne doivent pas se recouvrir, sinon le résultat est totalement imprévisible. En revanche, la fonction "memmove" est conçue spécialement pour des zones mémoires qui se recouvrent. Supposez par exemple que vous vouliez supprimer les 3 premiers caractères d'une chaîne, vous pouvez alors écrire :

```
memmove ( str, str + 3, n + 1 - 3 );
```

vous êtes garantis que le résulat sera correct, même si la copie est faite dans l'autre sens (memmove (str+3, str, n-2) permet de faire de la place pour 3 caractères au début de la chaîne). Le +1 permet de copier aussi le Ø final.

Il faut bien faire attention avec toutes ces fonctions à ce que la chaîne destinatrice soit associée à une zone mémoire suffisante pour accueillir la chaîne initiale, en n'oubliant pas de compter le Ø final. Un bug classique se présente sous la forme suivante :

```
dest = malloc ( strlen ( source ) );
strcpy ( dest, source );
```

Ecrit ainsi, vous avez écrasé un octet après la zone mémoire allouée. Comme c'est là en général que la librairie maintient un pointeur sur la zone allouée suivante ou précédente, vous allez avoir de sacrés problèmes lors d'une prochaine allocation ou libération de mémoire. Il vous aurait fallu écrire :

```
dest = malloc ( strlen ( source ) + 1 );
strcpy ( dest, source );
```

Comme cette séquence d'instructions est très fréquente, et que d'autres librairies proposent une fonction pour la réaliser, je l'ai définie dans ma propre librairie "StrLib" en utilisant le même nom que ces autres librairies, à savoir "strdup". J'ai aussi défini la fonction "strndup" correspondant à la fonction "strncpy"; dans les 2 cas la mémoire nécessaire au stockage de la chaîne cible est allouée dynamiquement. On trouve parfois ces 2 fonctions sous les noms "strsave" et "strnsave", notamment dans le "K&R"; j'ai donc défini 2 macros permettant d'utiliser ces synonymes.

Sous Unix BSD, la fonction "memmove" est connue sous le nom "bcopy"; j'ai défini une macro dans "StrLib.h" assurant l'équivalence; elle présente toutefois un petit piège, car ses paramètres sont inversés par rapport à toutes les autres fonctions de copie. Ce n'est pas trop gênant, car cette macro est surtout destinée à faciliter le portage, et non à être utilisée dans de nouveaux programmes.

Toujours sous Unix BSD, il existe une autre fonction de copie, appelée "swab", ce qui signifie "swap bytes"; comme son nom l'indique, les octets sont inversés 2 à 2 dans la chaîne cible par rapport à la chaîne source. Cette fonction est pratique lorsqu'on veut traiter des données provenant de machines ayant un ordre des octets différents, comme c'est le cas par exemple du Macintosh. Allez savoir pourquoi, j'ai mis cette fonction dans "MiscLib" et non dans "StrLib".

Les librairies Unix comportent une dernière fonction de copie appelée "memccpy" : cette fonction, dont vous trouverez une implémentation dans "StrLib" fonctionne de la même manière que "memcpy", sauf qu'elle prend un paramètre supplémentaire qui est un caractère de fin, c'est à dire qu'elle copie la chaîne source dans la chaîne cible jusqu'à rencontrer ce caractère ou après avoir recopié les N octets demandés.

Ma librairie comporte une fonction de copie que je n'ai pas vu ailleurs et qui relève plus du gadget, bien qu'elle peut avoir son utilité quelquefois : la fonction "strrev" effectue une copie en inversant la chaîne de caractères; le premier caractère de la chaîne source se retrouvera à la fin de la chaîne cible, et ainsi de suite.

• Une chaîne peut être initialisée avec un caractère donné grâce à la fonction "memset". Par exemple pour initialiser une chaîne à blanc, vous écrirez :

```
memset ( str, ' ', len );
```

J'ai complété cette fonction par les fonctions "strset" et "strnset" qui effectuent cette initialisation jusqu'à rencontrer un Ø final dans la chaîne de départ, qui donc avoir déjà été initialisée, pour "strset" et jusqu'à soit un Ø final, soit jusqu'à ce que N caractères aient été initialisés, selon la première condition qui se produit.
Ces 2 fonctions ont chacune un alias sous la forme d'une macro : "strfill" et "strnfill".

"StrLib" contient 2 autres fonctions d'initialisation d'une chaîne avec un caractère donné jusqu'à concurrence d'un certain nombre : "strpad" et "strnpad" permettent de compléter une chaîne par N caractères (par exemple un espace). La première fonction effectue l'initialisation à la fin de la chaîne originale, tandis que la seconde démarre d'une position donnée.

Unix BSD dispose d'une version réduite de la fonction "memset" appelée "bzero" effectuant la mise à zéro d'une zone mémoire donnée. J'ai toutefois défini une macro réalisant l'équivalence entre les 2.

• La librairie C dispose aussi de fonctions de concaténation de chaînes de caractères : "strcat" et "strncat" construites sur le même principe que les fonctions de copie que nous venons de voir. Il est à noter que ces fonctions sont assez coûteuses, car il leur faut se positionner à la fin de la chaîne cible avant de copier la chaîne source. Si l'on connaît la longueur initiale de la chaîne cible, il est habile de remplacer l'instruction :

strcat (dest, source);

par

strcpy (dest + len, source);

car on ne fait qu'une addition entre un pointeur et un entier au lieu d'une lecture octet par octet de la chaîne.

Encore une fois, la mémoire allouée par la chaîne cible (qu'elle soit dynamique ou statique) doit être suffisante pour recevoir la chaîne source en plus de la chaîne cible déjà présente.

La concaténation de plusieurs chaînes étant une opération relativement fréquente, j'ai implémenté une fonction "strvcat" qui admet une chaîne cible et une liste variable de chaînes à concaténer terminée par un paramètre à NULL; un aspect important de cette fonction est que la chaîne cible est complètement écrasée, c'est à dire que son contenu initial n'est pas considéré dans la liste de chaînes à concaténer. Elle s'utilise de la façon suivante :

strvcat (dest, src1, src2, src3, NULL);

Notez bien qu'il n'y a pas de limite au nombre de chaînes concaténées, hormis celles imposées par la taille de la pile affectée au programme.

• Toujours baties sur le même principe, les fonctions de comparaison "strcmp" et "strncmp" permettent de savoir si 2 chaînes sont identiques ou si la première est 'supérieure' ou 'inférieure' à la seconde; la relation d'inégalité est basée sur l'ordre des caractères ASCII. Comme pour les autres opérations sur les chaînes, "strncmp" permet de ne comparer que les N premiers caractères d'une chaîne. Il existe aussi bien entendu une fonction "memcmp" ne s'arrêtant pas sur un quelconque octet à Ø. Cette dernière fonction s'appelle "bcmp" sous Unix BSD; j'ai donc défini la macro équivalente.

Toutes ces fonctions font la distinction entre majuscules et minuscules. Il était donc naturel d'ajouter des fonctions ne faisant pas une telle distinction, puisque la librairie standard ne les propose pas. C'est le cas grâce aux fonctions "stricmp" et "strincmp" dont les noms correspondent à leurs homologues sous MS-DOS, ainsi que "strcasecmp" et "strncasecmp" qui sont des synonymes pour les fonctions que l'on rencontre sous Unix.

"StrLib" comporte une autre fonction de comparaison de chaînes, inspirée par Unix, mais originale : "strmatch". Cette fonction permet de comparer une chaîne donnée avec un motif, pouvant comporter des 'wildcards' en anglais ou des 'jokers' en français. Les jokers reconnus sont des plus classiques; il s'agit du '?' qui correspond à n'importe quel caractère, de l'"*' qui correspond à une séquence de 0 à N caractères de la chaîne, de '[]' qui permet de définir un sous-ensemble de caractères, éventuellement négatif si le premier caractère après le '[' est un '!'; le sous-ensemble peut être un intervalle si l'on indique les caractères extrêmes et qu'on les sépare par un '-'; enfin, tous ces jokers peuvent être annulés si ils sont précédés par un '\'. Un exemple valant mieux qu'un long discours, je vous recommande d'exécuter le programme "match" qui vous montre comme tout cela fonctionne.

• Les fonctions "strchr" et "strpos" permettent de localiser un caractère donné dans une chaîne; la première fonction retourne un pointeur sur la position de ce caractère dans la chaîne, tandis que la seconde retourne un index entier sur cette même position. Les fonctions "strrchr" et "strrpos" localisent la dernière occurence d'un caractère dans une chaîne au lieu de la première. La fonction "memchr" recherche un octet dans une zone mémoire d'une longueur donnée, sans s'arrêter sur un quelconque octet à Ø.

Sous Unix BSD, les fonctions "strchr" et "strrchr" sont connues sous les noms de "index" et de "rindex" que j'ai donc définies en tant que macros dans "StrLib.h".
Elles sont en tout point identiques à leurs homologues.

Les fonctions "strspn", "strcspn","strpbrk", "strrpbrk" et "strtok" permettent de trouver un caractère de la chaîne parmi un ensemble de caractères. Ces fonctions sont relativement puissantes, car elles permettent d'extraire très facilement des sous-chaînes délimitées par un ensemble de séparateurs. Il serait trop long de les détailler ici; je vous renvoie donc au manuel pour toutes les explications les concernant.

J'ai développé des variantes plus simples de ces fonctions, répondant à des besoins moins sophistiqués; certaines de mes fonctions sont d'ailleurs directement inspirées de ce que j'ai pu trouver ici ou là. Les fonctions "strskip" et "strskipblanks" retournent respectivement la position du premier caractère différent de celui indiqué pour la première fonction, et non blanc (un blanc pouvant être un espace, une tabulation ou une fin de ligne) pour la seconde. Cette dernière fonction a un synonyme via la macro "strpblnks". La fonction "strsep" est quasiment identique à "strtok" du moins dans son but; ses paramètres et sa sémantique sont cependant différents.
La fonction "strstr" permet elle de localiser une sous-chaîne dans une chaîne. "StrLib" ajoute la fonction "strrstr" permettant de trouver la dernière occurence d'une sous-chaîne dans une chaîne.

• J'ai écrit plusieurs fonctions permettant de supprimer les blancs d'une chaîne (un blanc étant défini comme pouvant être un espace, une tabulation ou une fin de ligne) : "strtrim" supprime les blancs de queue; cette fonction est parfois appelée "strrpblnks" par analogie avec celle décrite précédemment; j'ai donc défini un alias par l'intermédiaire d'une macro. "strcompress" réduit tous les blancs consécutifs d'une chaîne à un seul tandis que "strcollapse" supprime tous les blancs d'une chaîne.

• Si vous avez programmé en Basic, vous avez sans doute apprécié les fonctions d'extraction de sous-chaînes. Qu'à cela ne tienne, voici une version en C des fonctions "strleft", "strmid" et "strright" qui extraient respectivement les N premiers caractères d'une chaîne, les N caractères) à partir d'une position P, et les N derniers caractères.

La fonction "strelement" permet elle d'extraire l'enième sous-chaîne d'une chaîne, chacune de ses sous-chaînes étant séparés par le délimiteur donné, qui peut d'ailleurs être différent d'un appel à l'autre.

Remarque importante : toutes ces fonctions allouent la mémoire nécessaire au stockage de la sous-chaîne extraite. Elles ne sont donc pas exposées aux problèmes d'écrasement évoqué plus haut; en revanche, elles présentent des risques de fuite de mémoire, car il est de la responsabilité du programmeur les utilisant de libérer la mémoire que les chaînes retournées occupent, lorsqu'il n'en a plus besoin.

• De la même façon, si vous avez programmé en Pascal, vous avez sans doute utilisé les 2 fonctions "insert" et "delete". Elles vous manquaient en C; ce ne sera plus le cas grâce à "strins" et "strdel". L'interface est un peu différent de leurs homologues Pascal, puisqu'elles utilisent des pointeurs et non des indices pour définir la position d'insertion ou de suppression dans la chaîne.

Dans la foulée, j'ai créé les fonctions "strrepl" et "strmrepl" qui remplacent une sous-chaîne par une autre. La deuxième fonction remplace toutes les occurences de la sous-chaîne, tandis que la première ne remplace que la première occurence.
De la manière dont "strmrepl" est implémentée, la sous-chaîne de remplacement ne doit pas inclure la sous-chaîne remplacée, sinon la fonction va boucler indéfiniment.

Comme les fonctions précédentes, ces fonctions allouent la mémoire nécessaire à la chaîne résultante de l'opération. La chaîne initiale n'est elle modifiée en aucune façon.

Toutes ces fonctions, ainsi que d'autres décrites plus loin, sont démontrées par le programme "testlib" que je vous recommande d'étudier avant d'utiliser cette librairie.

Classifications et conversions

La librairie C standard définit un ensemble de macros de classification d'un caractère. Ces macros commencent toutes par "is" et le nom de la macro indique la catégorie à laquelle elle se rapporte. La macro retourne un résultat booléen pour indiquer si le caractère fait partie ou non de la catégorie testée. Je ne vais pas vous lister toutes ces macros ici (je vous renvoie au manuel pour cela), mais sachez simplement que ces macros permettent de savoir si un caractère est alphabétique ou est un chiffre (dans les différentes bases gérés par le C), si c'est un caractère de contrôle, un blanc, ... Des macros définissent des catégories plus larges ou plus restreintes telles que caractère imprimable ou caractère pouvant être utilisé dans un symbole C par exemple.

Au niveau des conversions, la librairie C permet d'effectuer toutes sortes de conversions, soit entre caractères, soit entre des chaînes de caractères et les nombres binaires qu'elles représentent.

Les conversions d'un caractère sont réalisées par des fonctions ou des macros dont le nom débute par "to". Nous trouvons donc 2 fonctions "tolower" et "toupper" permettant de transformer une lettre en minuscule ou en majuscule. Si le caractère passé en paramètre n'est pas une lettre, il est retourné inchangé. Ces 2 fonctions existent aussi sous forme des macros "_tolower" et "_toupper" qui sont plus rapides, mais qui ne fonctionnent pas si le caractère n'est pas une lettre; d'autres implémentations n'ont pas cette restriction, mais présentent des effets de bord car l'argument est référencé plusieurs fois (l'implémentation d'ORCA/C est conforme au standard si mes souvenirs sont exacts).

En revanche, la librairie ne permet pas de faire une conversion d'une chaîne de caractères soit en minuscules, soit en majuscules. Comme ce sont des opérations assez fréquentes, je les ai implémenté dans "StrLib" grâce aux fonctions "strupper" et "strlower" (qui ont pour synonymes "strupr" et "strlwr"

que l'on rencontre parfois dans certains environnements); ces 2 fonctions font la conversion directement sur la chaîne originale qui est donc remplacée. Pour être tout à fait complet, j'ai aussi écrit une fonction "strcap" qui met la première lettre en majuscule, ainsi que toute lettre suivant un signe de ponctuation ou un espace, tandis toutes les autres lettres sont converties en minuscules.

La macro "toascii" définie par la librairie C permet de forcer le caractère passé en paramètre à être dans l'intervalle dit ASCII soit entre 0 et 127; on s'aperçoit d'ailleurs que ces fonctions de classification de la librairie C ne sont pas "8 bits clean" comme on dit, c'est à dire qu'elles ne reconnaissent pas les caractères étendus des autres langues que l'anglais. La norme a cependant défini toute une autre catégorie de fonctions permettant de travailler avec des caractères étendus, et même très étendus puisqu'ils sont représentés par plusieurs octets (en général 2) et non plus un seul, afin de prendre en compte les langues orientales (on peut d'ailleurs aussi définir le sens de la lecture de ces chaînes, de gauche à droite ou de droite à gauche). Malheureusement, ces fonctions ne sont pas implémentées dans la librairie d'ORCA/C; certaines d'entre elles sont très complexes, c'est peut être la raison pour laquelle elles sont absentes. Il est amuser de noter qu'il y a quand même une référence à ces caractères étendus, puisque le type "wchar_t", utilisé par ces fonctions, est défini dans "stddef.h".

La fonction "toint" convertit un caractère représentant un chiffre dans son équivalent binaire. L'aspect intéressant de cette fonction est que le caractère peut correspondre à un chiffre décimal ou hexadécimal, c'est à dire à l'une des lettres 'a' ou 'A' à 'f' ou 'F'.

Les autres fonctions de conversion de la librairie C agissent sur une chaîne de caractères représentant un nombre et donnent comme résultat la valeur binaire de ce nombre. Les fonctions "atoi", "atol" et "atof" convertissent une chaîne respectivement en entier sur 16 bits, en entier sur 32 bits et en nombre flottant.
Ces fonctions sont maintenant remplacées par les fonctions plus sophistiquées "strtod", "strtol" et "strtoul" qui retournent respectivement un nombre flottant et un long signé et non signé. En plus de la conversion, ces 3 fonctions peuvent retourner un pointeur sur le premier caractère suivant ceux exploités pour la conversion. De plus les fonctions de conversion en entier acceptent un paramètre indiquant la base dans laquelle le nombre inscrit dans la chaîne doit être interprété; cette base peut aller de 2 à 36, elle peut aussi prendre la valeur Ø, ce qui indique que c'est la chaîne elle-même qui indique la base en utilisant les règles du C (si le premier chiffre est Ø, le nombre est interprété en octal, si la chaîne commence par Øx, on fera une conversion en hexadécimal, dans les autres cas, le nombre sera converti dans la base décimale).

La librairie d'ORCA/C ne dispose pas de fonctions inverses des précédentes, c'est à dire qu'il n'y a pas de fonction convertissant un nombre binaire dans sa représentation par une chaîne de caractères. Certaines librairies ont une fonction "itoa" et d'autres fonctions de conversion pour les nombres flottants. Je n'ai pas implémenté ces fonctions, car on peut réaliser simplement ces opérations grâce à la fonction de formatage général "sprintf". On peut d'ailleurs faire les conversions décrites précédemment par la fonction "sscanf" sans grande difficulté.

Sur le GS, le système et la boîte à outils font grand usage de chaînes de caractères ayant une représentation différente de celle définie par le C. La librairie d'ORCA/C définit 2 fonctions permettant de faire des conversions entre le format C et le format Pascal "c2pstr" et "p2cstr". La seule remarque sur ces fonctions est qu'elles utilisent un buffer statique propre à la librairie, qui est écrasé à chaque appel; il est donc de la responsabilité du programmeur les utilisant de les sauver quelque part ailleurs.

J'ai complété ces 2 fonctions par "c2wstr" et "w2cstr" qui font la même chose avec des chaînes GS/OS, dont la longueur est représentée par un mot de 16 bits (d'où le 'w' pour 'word'). Mes fonctions allouent la mémoire nécessaire au stockage du résultat; il n'y a donc plus de problème d'écrasement potentiel, par contre, il ne faut oublier de libérer cette mémoire lorsqu'on n'en a plus besoin.

Fonctions mathématiques

Les fonctions mathématiques présentes dans la librairie C standard sont très complètes. On trouve bien entendu les fonctions les plus usuelles telles que l'élévation à une puissance quelconque ("pow"), le calcul de la racine carrée ("sqrt"), les fonctions trigonométriques ("sin", "cos", "tan") et leurs inverses ("asin", "acos", "atan" et "atan2"), les fonctions logarithmiques et exponentielles ("log", "log10" et "exp") ou le calcul de la valeur absolue d'un nombre entier ou flottant ("abs", "labs" et "fabs").

Les fonctions moins usuelles que l'on trouve dans la librairie C standard sont les fonctions hyperboliques ("sinh", "cosh" et "tanh"; en revanche, il n'y a pas les fonctions inverses dans la librairie ORCA/C, mais on les trouve ailleurs; je ne les ai toutefois pas implémentées, n'en ayant jamais eu besoin), des fonctions d'arrondi flottant ("ceil" et "floor"), des fonctions de calcul du reste d'une division flottante ("fmod" et "modf"), ainsi qu'une fonction de division entière retournant simultanément le quotient et le reste ("div" et "ldiv"). La fonction "frexp" permet de décomposer un nombre flottant en mantisse et exposant, tandis que la fonction "ldexp" effectue l'opération inverse.

J'ai ajouté quelques macros (définies dans "MiscLib.h") à toutes ces fonctions : "abs", "min", "max", "sgn" donnent respectivement la valeur absolue d'un nombre (ma version est une macro au lieu d'une fonction dans la librairie standard; c'est plus rapide, mais c'est aussi sujet à des effets de bord car l'argument est évalué 2 fois), le minimum et le maximum de 2 nombres, et le signe d'un nombre. La macro "sqr" retourne le carré de l'expression passée en paramètre. La macro "round" donne l'entier le plus proche d'un nombre flottant; l'arrondi est fait par excès dans le cas d'un nombre positif et par défaut dans le cas d'un nombre négatif. Les macros "hypot" et "cabs" calculent l'hypoténuse d'un triangle rectangle et le module d'un nombre complexe (le calcul est le même, seul les arguments changent); ces 2 fonctions font partie de toutes les librairies C que je connais, mais sont curieusement absentes de celle d'ORCA/C.

Enfin, la macro "isnan" indique si son argument est un nombre ou un NAN; si vous ne savez pas ce que c'est, je vous renvoie à la documentation de SANE.

Contrôle exécution et environnement

La librairie C dispose de plusieurs fonctions permettant d'aller au delà des structures de contrôles du langage. Nous en avons d'ailleurs déjà vu quelques unes dans la quatrième partie de cette initiation; ce sont les fonctions "setjmp" et "longjmp" sur lesquelles je ne reviendrai pas ici.

Lorsqu'on veut terminer un programme C ailleurs qu'à la fin de la fonction "main" (par exemple, lorsqu'on a détecté une erreur fatale), on peut appeler l'une des fonctions "exit", "_exit" ou "abort". La première est utilisée pour terminer proprement un programme, en faisant le ménage, tandis que la seconde rend directement la main au shell ou au lanceur (par exemple le Finder™) qui assurera ce ménage (fermeture des fichiers, libération de la mémoire); il est quand même souhaitable d'éviter d'appeler cette fonction dans la plupart des cas. La fonction "abort" est implémentée dans ORCA/C comme étant équivalente à "_exit(-1)", mais ce n'est pas nécessairement le cas dans d'autres environnements. Vous pouvez constater que les fonctions "exit" acceptent un paramètre qui donnera le résultat final de l'exécution du programme; cette valeur est surtout utile pour les commandes shell, puisqu'elle indique si l'éxecution doit continuer lorsqu'elle la commande est insérée dans un script. Dans le cas du shell ORCA/C, la valeur Ø indique un succès et toute autre valeur correspond à un échec; pour simplifier le portage, le fichier header "stdlib.h" définit 2 symboles "EXIT_SUCCESS" et "EXIT_FAILURE" qu'il est fortemment recommandé d'utiliser comme paramètres des fonctions "exit".

Une des actions du ménage effectué par "exit" consiste à appeler une ou plusieurs fonctions du programme que l'on aura enregistré au préalable par la fonction "atexit". Je vous renvoie au manuel pour voir ce que l'on peut faire avec cette fonction.

Un programme C peut être informé lorsqu'un événement extérieur se produit en utilisant la fonction "signal"; celle-ci permet de déclarer une fonction qui sera exécutée lorsque l'exception associée se produira, ou d'indiquer au système que l'événement en question doit être ignoré, ou qu'une action par défaut doit être effectuée (en général, le défaut est d'interrompre l'exécution du programme). Les mécanismes mis en œuvre par cette fonction ne sont pas disponibles sur le GS, sauf si l'on utilise GNO/ME; par conséquent, cette fonction n'est que de peu d'utilité. Il est toutefois possible à un programme de simuler une exception en utilisant la fonction "raise" qui permettra d'exécuter la fonction correspondant au signal déclenché.

La gestion des erreurs de la librairie C est plus que rudimentaire, mais elle a le mérite d'exister : elle se fait au travers d'une variable globale "errno" qui est positionnée par les routines de la librairie lorsqu'elles détectent une erreur. Ces routines retournent par ailleurs une valeur particulière pour indiquer qu'une erreur s'est produite. Un piège à éviter est que la librairie ne met jamais "errno" à Ø elle-même; c'est de la responsabilité du programmeur de l'application de le faire avant d'appeler une fonction dont il veut tester la réussite ou l'échec. La variable "errno" peut prendre un très petit nombre de valeurs correspondant à des domaines d'erreur assez vastes. Le libellé associé au code d'erreur peut être affiché à l'écran (en fait sur stderr) par la fonction "perror" et retourné dans une chaîne de caractères par la fonction "strerror". Je me suis attaché dans mes librairies à positionner "errno" à l'une des valeurs prédéfinies en fonction de l'erreur qui s'est produite.

Pour les erreurs générées par GS/OS ou la Toolbox, ORCA/C a une fonction "toolerror()" retournant le code de la dernière erreur rencontrée.

La librairie C définit aussi un mécanisme d'assertion qui permet de vérifier qu'une condition est vraie, et sinon d'interrompre l'exécution du programme. Ce mécanisme est mis en œuvre par la macro "assert" que nous étudierons plus en détail dans mon article sur le débugging.

La fonction "getenv" permet d'accéder à l'environnement, ce qui dans le cas d'ORCA correspond à la récupération de la valeur d'une variable du shell. La norme ne définit pas de fonction de création ou de modification d'une telle variable, bien que ce soit des fonctions que l'on rencontre fréquemment; j'ai donc développé les fonctions "putenv", "setenv" et "unsetenv", complémentaires à la précédente. Les 2 premières sont identiques mais ont une interface légèrement différente; la dernière permet de supprimer une variable d'environnement.

La fonction "system" permet d'exécuter une commande quelconque du shell ORCA ou GNO depuis un programme C; dans un système multitâches, et donc notamment sous GNO, cette commande est exécutée dans le contexte d'un sous-process, tandis que dans le cas d'ORCA, la fonction "system" correspond à l'utilisation d'une routine interne du shell.

La fonction "commandline" est destinée à récupérer la ligne de commande complète, sans bénéficier du découpage des arguments effectués via les paramètres "argc" et "argv" de la fonction "main".

Toutes ces fonctions interagissant avec l'environnement d'exécution ne sont opérationnelles que si le programme a été exécuté dans le contexte d'un shell. Si le programme peut aussi être lancé directement depuis le Finder™ ou tout autre lanceur de programme, il ne faudra bien entendu pas les utiliser. La fonction "shellid" permet de savoir dans quel contexte le programme est en train de s'exécuter, car la valeur NULL est retournée si l'on n'est pas sous le contrôle d'un shell.

La fonction "userid" permet elle de connaître l'identifiant qui a été affecté au programme par le système. C'est bien entendu une fonction propre au GS et à ORCA/C.

Les paramètres "argc" et "argv" d'une commande shell permettent d'obtenir les valeurs passées à cette commande; ces valeurs peuvent être des options ou des noms de fichiers, utilisant éventuellement des wildcards, ou d'autres types d'arguments.

Le décodage des options d'un programme est souvent une tâche fastidieuse et répétitive d'un programme à l'autre; la fonction "getopt" présente dans les librairies sous Unix, et dans "MiscLib.h" permet d'automatiser ce processus. La version que je vous propose contient plusieurs extensions par rapport à son homologue Unix, avec laquelle elle est toutefois compatible à 100%. Je vous recommande d'étudier les explications fournies dans les commentaires pour l'utiliser correctement. Vous trouverez aussi une démonstration de son utilisation dans le programme "opt", ainsi que dans les commandes shell "size" et "dtree" de votre disquette "GS Infos".

L'expansion des noms de fichiers avec wildcard est réalisée par les 2 fonctions "firstfile" et "nextfile" dérivées de leurs homologues MS-DOS. Ces fonctions sont assez triviales à utiliser et je vous recommande de regarder le programme "size.cc" pour avoir un exemple.

Allocation/Libération de mémoire

J'ai traité toutes ces fonctions dans mon précédent article, je n'y reviens donc pas ici.

Gestion du temps

La librairie C standard dispose de fonctions très complètes pour manipuler le temps, qu'elle représente sous 3 formats différents : un format littéral (une chaîne de caractères ASCII), un entier long représentant en général le nombre de secondes écoulées depuis un temps de référence, et une structure détaillant chacun des éléments d'une date et d'une heure.

La fonction "time" permet d'obtenir la date et l'heure actuelle sous la forme d'un entier long. Ensuite, les fonctions "localtime" et "gmtime" permettent de le convertir dans la structure détaillée évoquée ci-dessus. Pour les systèmes gérant les fuseaux horaires, la fonction "localtime" tient compte du décalage horaire de façon à fournir l'heure locale, tandis que "gmtime" donnne l'heure GMT. Le GS ne gérant pas cette notion, ces 2 fonctions sont identiques. La fonction "mktime" effectue la conversion inverse, c'est à dire qu'elle retourne un entier long à partir d'une structure détaillée.

La fonction "difftime" permet d'obtenir le temps écoulé en secondes entre 2 temps exprimés dans le format entier long.

La fonction "clock" est un peu à part; dans un système multi-tâches, elle est censée retourner le temps pris par l'exécution d'un programme dans une unité qui dépend de la résolution interne de l'horloge de la machine hôte. Sur le GS, ces concepts ne sont pas mis en œuvre; à la place la fonction "clock" retourne le nombre de ticks d'horloge tels que générés par le sous-système du heartbeat, mis en action notamment par l'outil "Event Manager".

Les fonctions "asctime" et "ctime" retournent toutes deux la forme littérale d'une date et heure donnée (en anglais); la première effectue la conversion à partir de la structure détaillée, tandis que la seconde opère sur l'entier long. Notez qu'il n'y a pas de fonction effectuant la conversion inverse, c'est à dire décodant une chaîne de caractères et retournant le temps correspondant dans l'une des 2 formes décrites précédemment.

Ces 2 fonctions ont tendance à devenir obsolètes, car elles ont été remplacées lors de la normalisation par une fonction de formatage du temps beaucoup plus sophistiquée, et prenant en compte les spécificités de chaque pays (langue pour les jours et les mois, ordre des informations, ...). Cette fonction, "strftime", même si elle n'était pas implémentable dans son intégralité, aurait dû faire partie de la librairie d'ORCA/C. Ce n'est malheureusement pas le cas, y compris dans la version 2.0 du compilateur. J'en ai donc réalisé une implémentation, que vous trouverez dans la librairie "StrLib". Cette fonction dispose d'un très grand nombre d'options de formatage, qu'il serait vain de détailler ici; je vous renvoie au commentaire du début de la fonction qui liste toutes les possibilités, et il y en a un très grand nombre (d'ailleurs, bien que cette fonction soit assez triviale — et a été fastidieuse à écrire malgré le coupé/collé —, l'optimiseur d'ORCA/C v1.x n'arrive pas à la traiter correctement; je n'ai pas réessayé avec la v2.0, suite aux problèmes que j'ai rencontré avec cette version, et que j'ai déjà évoqué). Le programme "date" illustre une partie des options possibles en affichant la date et l'heure en clair et en français. Selon les options choisies, mon implémentation utilise les noms de mois et de jours en anglais ou en français.

Fonctions diverses

La librairie C comporte quelques fonctions qui n'entrent pas dans les catégories précédentes :

• Une fonction implémentant l'algorithme de tri rapide : "qsort". Cette fonction peut trier n'importe quel tableau de n'importe quel type d'objet sur n'importe quel critère, grâce à ses paramètres que je vous laisse étudier dans le manuel.
Nous étudierons cet algorithme de tri, ainsi que quelques autres, dans un prochain article de ma série sur la programmation.

• Une fonction de recherche dichotomique : "bsearch" fonctionnant avec le même type de paramètres que "qsort". Encore une fois, je vous conseille de vous reporter à la documentation. Certaines librairies proposent d'autres fonctions de recherche implémentant d'autres algorithmes. Nous étudierons cela dans un prochain article de ma série sur la programmation.

• La librairie d'ORCA/C contient les fonctions "startdesk", "enddesk", "startgraph" et "endgraph", facilitant l'initialisation et la terminaison de programmes en mode bureau ou simplement graphiques. Ces fonctions peuvent être utilisées pour des programmes simples, écrits vite fait sur un coin de table.
Pour des programmes plus sérieux, je vous recommande d'utiliser les fonctions "StartUpTools" et "ShutDownTools" de la boîte à outils.

• La macro "offsetof" permet d'évaluer facilement le décalage en octets d'un champ d'une structure, ce qui peut être parfois utile. Jetez un coup d'œil au manuel pour plus de détails.

• Les fonctions "rand" et "srand" mettent en œuvre un générateur de nombres pseudo-aléatoires assez simple, mais relativement correct. La deuxième fonction permet d'initialiser le générateur; si on lui passe toujours la même valeur, la série de nombres retournée par la première fonction sera toujours la même. En général, on récupera donc l'heure actuelle grâce aux fonctions présentées ci-dessus, et on utilisera par exemple les secondes comme graine du générateur.

• La macro "va_start" et la fonction "va_end" ainsi que le type "va_arg" permettent de réaliser des fonctions ayant une liste d'arguments variable, comme c'est le cas par exemple de la fonction "strvcat" décrite plus haut. J'ai déjà expliqué ce mécanisme dans mon article sur les fonctions auquel je vous renvoie (c'était la cinquième partie de mon initiation).

La librairie "MiscLib" contient aussi quelques fonctions et macros diverses dont je n'ai pas encore parlé :

• Pour les nostalgiques du Basic, j'ai réalisé une version des fonctions "peek" et "poke" qui permettent de lire et d'écrire n'importe où en mémoire, et notamment d'accéder facilement aux softswitches. Ces fonctions existent pour des données stockées sur des octets, des mots de 16 bits et des mots de 32 bits : leur nom est suivi des lettres "b", "w" ou "l" pour indiquer le type manipulé.

• Les macros "new" et "newstruct" émulent la primitive "new" du Pascal et simplifient l'allocation de mémoire pour une structure; la première est à utiliser quand un type a été défini via "typedef", la seconde correspondant au cas d'une structure n'ayant pas de nom de type associé par "typedef".

• J'ai déjà présenté la macro "swap" : elle effectue la permutation de 2 nombres sans passer par une variable intermédiaire.

• La fonction "isatty" permet de savoir si l'entrée, la sortie ou la sortie des erreurs standards ont été redirigés du clavier ou de l'écran vers un fichier, au niveau du shell. Cela permet par exemple de n'utiliser les séquences de contrôle du driver de la console ou les caractères MouseText que si on affiche sur l'écran.

Conclusion

Voilà ! Nous avons fait un tour d'horizon assez complet de ce qu'offre la librairie C standard, complétée par les trois librairies qui accompagnent cet article.

Comme vous avez pu le constater, cette librairie est assez riche; de plus, il est très facile de la compléter par des fonctions qui peuvent garder le même niveau de généralité. Un aspect important du C est que sa librairie n'utilise aucun mécanisme spécifique, et peut donc être complètement écrite en C elle-même (celle d'ORCA/C est programmée en assembleur), contrairement à celle de Pascal par exemple (je vous mets au défi d'écrire en Pascal des fonctions à arguments variables telles que WRITE ou READ, ou un allocateur de mémoire ayant la sémantique de NEW).

J'espère encore une fois que cette série vous a intéressé, et qu'elle vous a permis de débuter en C. Je vous conseille maintenant de prendre l'un des livres que j'ai présenté dans mon premier article, ou le cours de ByteWorks si vous maîtrisez l'anglais.