Types de données numériques en C

Type	Domaine de valeurs	Taille
char	-128 à 127	8 bits
unsigned char	0 à 255	8 bits
int	-32768 à 32767	16 bits
unsigned int	0 à 65535	16 bits
long	-2147483648 à 2147483647	32 bits
unsigned long	0 à 4294967295	32 bits
float	3.4e-38 à 3.4e38	32 bits
double	1.7e-308 à 1.7e308	64 bits
long double	3.4e-4932 à 1.1e4932	80 bits

Types de données numériques en Pascal

Type	Domaine de valeurs	Taille
shortint	-128 à 127	8 bits
byte	0 à 255	8 bits
integer	-32768 à 32767	16 bits
word	0 à 65535	16 bits
longint	-2147483648 à 2147483647	32 bits
comp	-9.2e18 à 9.2e18	64 bits
single	1.5e-45 à 3.4e38	32 bits
real	2.9e-39 à 1.7e38	48 bits
Double	5.0e-324 à 1.7e308	64 bits
Extended	3.4e-4932 à 1.1e4932	80 bits

Pour programmer et faire des calculs dans de tels langages, on n'a pas besoin de savoir si les entiers (par exemple le type **byte** en Pascal) sont codés en compléments à deux sur 8 bits, ou si les réels (par exemple le type **float** en C) sont codés en norme IEEE 764, en effet, le compilateur se charge de toutes ces considérations à votre place. Ces questions de représentations/codages de nombres en machine, quoique fondamentales pour qui veut comprendre comment fonctionne un ordinateur, sont donc superflues au quotidien et "transparentes" pour l'utilisateur lambda.

En revanche, il est toujours utile de réfléchir sur le type de la variable que l'on va utiliser : par exemple il vaut mieux coder des températures météorologiques avec le type **char** en C plutôt qu'avec le type **unsigned char**, qui pourrait induire l'utilisateur en erreur puisqu'on n'aurait pas alors les températures négatives, par ailleurs si les températures sont codées non plus en degré Celsius mais en degré Fahrenheit, la plage -128 à 127 ne convient plus et il vaudrait mieux opter pour le type **int** en C. De plus, si l'on veut stocker des températures avec quelques chiffres après la virgule, il faut donc abandonner ces types entiers, pour avoir recours aux types réels (par exemple le type **float** en C).
Bien sûr, il ne faut pas perdre de vue qu'utiliser des variables codées sur 32 bits ou lieu de 8 bits augmentera la place de stockage sur le disque dur (imaginons une immense base de données de températures relevées partout en France) et diminuera certainement la rapidité du traitement de ces données par l'ordinateur.

La discussion précédente n'est pas sans rapport avec le *bug de l'an 2000*, c'est en effet à cause de considérations de ce type que beaucoup de programmes écrits dans les années 70-80 (à cette époque, on était encore avare sur la place dans le disque dur, etc.) avaient opté pour un codage de la date en prenant simplement, pour l'année, que les deux derniers chiffres ; les auteurs de ces programmes n'imaginaient que leurs produits seraient toujours utilisés 20 ans après ! Dans le même genre, mais il s'agit plus cette fois d'un problème plus matériel (hardware) que logicel (software), si vous avez un vieil ordinateur chez vous (de 1988 pour situer les idées), il est fort probable que son BIOS (=l'endroit où sont enregistrés la date, l'heure, le type de votre disque dur, etc.) ne passe pas 2010, en effet, sur ces vieux modèles, l'année était souvent codées sur 5 bits (équivaut à 32 ans, de 1968 à 2010 disons).
Toutes ces astuces (qu'on aurait tort de considérer comme des économies de bout de chandelles) deviennent dangereuses quant un programme (ou un appareil électronique) est utilisé plus longtemps que prévu mais aussi dans un autre cas, bien plus subtil, et c'est là que résidait le principal problème du bug de l'an 2000 : lorsque qu'un bout de programme est réutilisé, bien des années après sa conception, par un programmeur autre que son concepteur original. Là ou le problème devient assez subtil, c'est quand des bouts de codes sont repris d'une personne à l'autre (par exemple, une version de Windows ne passait pas l'an 2000, en fait, il s'agissait simplement d'un petit utilitaire antédiluvien, Notepad, créer avec les premières versions Windows et qui n'aurait certainement gêné personne !).

Ce type de problèmes, assez pernicieux, est aussi à l'origine du "bug" qui avait fait exploser la fusée Ariane en vol. En effet, une sous-routine d'un sous-programme d'une sous-librairie du programme d'une fusée précédente avait été réutilisée (apparemment sans une relecture suffisamment approfondie) et malheureusement, une variable a eu la mauvaise idée de valoir 0 (ce qui ne se produisait pas avec la fusée précédente), ce qui a dû provoquer une division par zéro avec au final panne générale et crash de la fusée.

Pour en revenir, aux types des variables, les types suivants sont également courant dans la plupart des langages (donnons l'exemple de la terminologie Pascal) : array (tableau) file (fichier) object (objet) ordinal (scalaire) pointer (pointeur) real (réel) record (structure) set (ensemble) string (chaîne).

Un langage de programmation offre alors un jeu d'instructions qui permet de “ manipuler ” ces variables.

4.2 Exercices

4.3
EXERCICE 0 : Les premiers mots…

Lancez le programme suivant plusieurs fois de suite (\n indique que l'on veut un “ retour à la ligne ”).

```
#include <stdio.h>  \* mon premier programme *\
main( )
{  printf(“ hello,  world\n ”) ;}
```

Retirez le \n et relancez le programme plusieurs fois de suite, que remarquez-vous ?

EXERCICE 1 : Le test “ tant que… ”

La conversion entre degré Celsius et degré Fahrenheit est donnée par C=(5/9) F-32. Prédire ce que fait le programme suivant. Exécutez-le pour vérifier.

```
#include <stdio.h>
main( )
{
int fahr, celsius ;
mini=0 ;
maxi=300;
pas=20;
fahr=mini ;
while(fahr<=maxi) {
celsius=5*(fahr-32)/9 ;
printf(“%d\t%d\n ”, fahr, celsius) ;
fahr=fahr+pas ;
}
}
```

EXERCICE 2 : La boucle for.

Sachant que %8d affiche au format entier, sur 8 caractères de long et que
%3.7f affiche au format flottant, sur 7 caractères de long dont 3 chiffres après la virgule,
prédire ce que fait le programme suivant. Exécutez-le pour vérifier.

```
#include <stdio.h>
main( )
{
int fahr ;
for (fahr=0 ;fahr<=300 ; fahr=fahr+20)
     printf(“%3d %6.1f\n ”, fahr, (5.0/9.0)*(fahr-32)) ;
}
```

Exercice 3 Un peu d'interactivité…

Nous allons maintenant faire un petit programme pour calculer la puissance n-ième d'un nombre.

```
#include <stdio.h>
int puis(int base, int n)
{ int i,p ;
p=1 ;
for (i=1 ;i<=n ;++i)
p=p*base ;
return p ;
}

main( )
{
int i ;
for (i=0 ;i<10 ;++i)     \*  ++i est un raccourci de i=i+1 *\
    printf("%d %d %d\n ", i, puis(2,i), puis(-3,i));
return 0 ;
}
```

Exercice 4 Un petit test

En utilisant " while " et l'instruction " if … else ", écrire un programme qui dise si un nombre entier est pair ou non. On pourra s'inspirer des lignes suivantes :
if (n<0) printf(" n est negatif ") else printf(" n est positif ") ;

Exercice 5 Une bien belle conjecture

Le problème suivant n'est pas encore résolu par les mathématiciens
" Prenez un nombre entier positif, s'il est pair, divisez-le par deux, s'il est impair, multipliez-le par 3 et ajoutez 1.
Est-il vrai qu'en répétant ce procédé, on finit toujours par tomber sur 1 ? "

Exemple : 3 – 10 – 5 - 16 – 8 – 4 - 2 –1 on finit bien par retomber sur 1.

Ce problème est connu sous le nom de " conjecture de Syracuse ", durant la guerre froide courrait la blague qu'il avait été inventé par les Russes pour que les Américains perdent leur temps sur des recherches pas sérieuses.

Faites un programme qui dise au bout de combien de temps, en partant d'un entier n, on atteint 1 (par exemple, pour n=3, votre programme doit afficher 7, car c'est au bout de 7 transformations que 3 est changé en 1, voir l'exemple ci-dessus).