Tutorial

Université d'Orléans U.F.R. Faculté des Sciences

Licence de Physique

Parcours Ingénierie Electrique

Introduction au langage C/C++

Jean-Philippe Grivet, 2005

Chapitre 1

Introduction

Un ordinateur est une machine capable d'effectuer très rapidement des opérations arithmétiques ou logiques sur des données afin de fournir des résultats. Cependant, cette machine ne dispose d'aucune connaissance : pour obtenir qu'elle réalise les opérations souhaitées, il faut lui fournir des instructions qu'elle suivra ensuite à la lettre.

En général, l'utilisateur dispose d'un «algorithme» qui décrit en détail comment on passe, au moyen d'un nombre fini d'opérations, des données aux résultats. L'algorithme (recette) est rédigé en langage humain, bien trop riche et complexe pour être compris par la machine. L'utilisateur devra tout d'abord transposer son algorithme en un jargon simpliste (un «langage de programmation ») pour obtenir un «programme» (on dit aussi «code» ou «code source») équivalent. Ce programme utilisateur sera traduit à son tour en une suite d'instructions primitives effectivement compréhensibles et exécutables par l'ordinateur. Cette traduction est réalisée par un programme spécialisé, le «compilateur». A chaque langage de programmation correspond un compilateur, qui porte le même nom ; ainsi ce cours est une introduction à la programmation en C++, qui utilisera pour les exercices pratiques un compilateur C++. Le programme source est rangé dans un fichier situé sur le disque dur ou sur une disquette. Sauf instructions spéciales, le compilateur ne connaît que le contenu de ce fichier source. Le programme en langage machine (repéré par le suffixe .exe

sous DOS et Windows) est lui aussi rangé dans un fichier (en général dans le même dossier que le source). Il peut alors être exécuté autant de fois que l'utilisateur le souhaite.

Pour certains langages de programmation (Basic à ses début, Maple, Scilab par exemple), on utilise un «interprèteur» qui traduit le «code utilisateur» en langage machine ligne par ligne à chaque exécution.

Il est important de se persuader du fait que l'ordinateur traite, selon les instructions qu'il a reçues, des chiffres binaires (des bits ou des octets) sans se préoccuper le moins du monde de leur signification ; les données peuvent être des résultats d'une mesure physique, un morceau de musique numérisé, une carte météo fournie par un satellite, un texte à traduire en anglais, c'est tout pareil pour la machine. C'est à l'utilisateur (à travers son programme) d'assurer l'interprétation des données et des résultats.

Chapitre 2

Les éléments du langage

Les langages de programmation, comme toute langue, utilisent des lettres pour former des mots ; ces mots peuvent être groupés en petites phrases simples que l'on appelle en général des instructions. Ces phrases doivent être construites en respectant un certain nombre de règles de syntaxe. Je présente dans ce chapitre les constituants de base des langages C/C++.

2.1 lettres et mots

Le langage C++ utilise 52 lettres : a, b, . . ., z et A, B, . . ., Z. Le compilateur C++ distingue parfaitement les majuscules des minuscules. On emploie également les signes usuels : =, +, -, *, /,(,),[,],{,},

> , < , &,%, ?,|, *n* , la virgule, le point, deux points :, le point-virgule, l'apostrophe (') et les guillemets anglais(", sous le 3). Certains de ces symboles peuvent être combinés entre eux.

Le langage C est formé de « mots réservés» ou «mots clés», qui doivent être utilisés comme

prévu par la définition du langage ; en voici la liste :

auto double int struct

break else long switch

case enum register typedef

char extern return union

const float short unsigned

continue for signed void

default goto sizeof volatile

do if static while

Le langage C++ ajoute les mots clés suivants

asm bool catch class const_cast

delete dynamic_cast explicit false friend

inline mutable namespace new operator

private protected public reinterpret_cast

static_cast template this throw true

try typeid typename using virtual

wchar_t

La signification de certains de ces mots sera expliquée par la suite. On emploie bien d'autres mots (toutes les fonctions mathématiques par exemple). Ces mots (identificateurs) ne font pas partie de la norme C/C++ et peuvent

en principe être redéfinis par l'utilisateur.

Dans le temps imparti pour ce cours, il ne sera pas possible d'examiner (et encore moins d'assimiler) tous les aspects du langage C++ : il s'agira plutôt d'une version simplifiée du langage, proche du C, du C+ en quelque sorte, ou encore du C amélioré.

2.2 Premiers exemples

Je présente maintenant quelques programmes très simples en C++, avec leurs équivalents en C pur (et en Pascal pour ceux qui connaissent ce langage). Ce premier exemple n'utilise aucune donnée et a pour seul résultat un affichage à l'écran.

2.2.1 le programme traditionnel

```
1
// bonjour_2 . cpp : a f f i c h e un message t r a d i t i o n n e l
2
#include <ios t ream>
3
#include <c s t d l i b >
4
int main ( void ){
5 s td : : cout << "Bonjour tout l e monde ! " << s td : : endl ;
6 system( " pause " ) ;
7
return 0 ;
8 }
```

La numérotation des lignes ne fait PAS partie du programme ou du langage, elle est là pour faciliter les explications.

Sur la ligne 1, j'ai placé un commentaire (une ligne commençant par //) : c'est du texte destiné aux utilisateurs, la machine n'en tient aucun compte. Un commentaire expliquant le rôle de chaque élément d'un programme permet au lecteur de comprendre sans trop de mal ce que fait ce programme. Le compilateur considère comme un commentaire tout ce qui se trouve entre // et une fin de ligne.

Sur la ligne 2 (et aussi 3), on rencontre une « directive du préprocesseur » laquelle commence toujours par # et dont le rôle est de réclamer l'utilisation d'une partie de la «librairie standard». Le préprocesseur agit comme une avant-garde du compilateur ; il va insérer (inclure) dans mon programme un fichier entête contenant des renseignements concernant les fonctions responsables

de la lecture des données et de l'écriture des résultats (

i nput- o utput). Celles-ci ne font pas partie

du langage C++ de base. Même chose pour la ligne 3, qui rend possible l'accès à la fonction

system

(communication avec le système d'exploitation).

La ligne 4 contient le nom de l'élément principal (

main) du programme. Ce nom doit figurer

une fois et une seule

. Les parenthèses indiquent au compilateur qu'il s'agit en fait d'une fonction ;

elles contiennent le mot-clé

void parce que la fonction main ne demande ici aucun argument.

Un programme plus complexe pourrait comporter de nombreuses fonctions, dont une seule devra

s'appeler

main . D'autre part, on indique au système d'exploitation (DOS, Windows,Linux,. . .), par

le mot réservé

int , que main « fournit un résultat», sous la forme d'un entier. Cet entier permet

en principe de vérifier la bonne fin du programme.

La ligne 4 contient une accolade ouvrante, qui marque le début de la fonction proprement dite ;

cette fonction se termine par l'accolade fermante de la ligne 8. Tout ce qui se trouve entre les deux

accolades s'appelle le «corps» de la fonction. On dit aussi que les instructions situées entre une

paire d'accolades forment un «bloc».

La ligne 5 contient la seule instruction effective du programme. Elle demande l'insertion, dans

le «flot de sortie», de la «chaîne de caractères»

Bonjour tout le monde ! , suivie d'une instruction

destinée au terminal (ici l'écran),

endl (à la ligne) ce qui, en français, signifie que l'on veut afficher

à l'écran la phrase

Bonjour tout le monde ! et positionner le curseur au début de la ligne suivante.

Comme l'opérateur

cout et le caractère endl font partie de la librairie standard, on préfixe ces

deux noms par

std:: .

La ligne 6 demande au système de marquer un temps d'arrêt, pour que l'on puisse lire le

résultat. Le déroulement normal (retour à l'écran d'accueil) reprend dès que l'on appuie sur une

touche. On voit que la fonction

system admet un argument, la «chaîne de caractères» pause .

On a enfin (ligne 7) une instruction

return 0 , qui envoie au système d'exploitation le «résultat

» de la fonction main, la valeur 0, indiquant que tout s'est bien passé (un résultat non nul

indiquerait une erreur). Le compilateur vérifiera que la déclaration de la fonction (

int main()),

de type entier, est bien cohérente avec la nature du résultat (l'entier 0).

On remarque que chaque instruction se termine (et DOIT se terminer) par un point-virgule.

Les éléments précédents sont toujours, ou presque toujours, présents dans un programme en C++.

Le programme qui vient d'être présenté respecte strictement la norme C++; les compilateurs

dont vous pourrez disposer respecte plus ou moins strictement cette norme. Il se peut que votre

compilateur admette ou préfère l'entête

#include <iostream.h> . Il est aussi possible (c'est le cas

du compilateur que j'utilise) qu'il ne soit pas regardant sur la présence des préfixes

std:: et que

main()

le satisfasse. Ce comportement voisin de l'anarchie est du à ce que la norme qui définit le

C++ est récente et continue d'évoluer.

2.2.2 le même en C

Voici maintenant le même programme en C pur ; il sera parfaitement compris par un compilateur C++, mais l'inverse n'est pas vrai : les mots-clés du C++ ne sont pas connus du C.

```
1 /*bonjour.c: Affiche un message traditionnel*/
2 #include <stdio.h>
3 int main()
4 {
5 printf("Bonjour tout le monde!\n");
6 return 0;
7 }
```

Examinons les quelques différences qui existent, à ce niveau, entre les deux dialectes.

Sur la ligne 1, un commentaire est indiqué par une ligne commençant par /* et finissant par */.

Sur la ligne 2, la « directive du préprocesseur» se réfère à un fichier entête (h eader) dont le nom est différent des précédents mais dont le rôle identique : insérer des renseignements concernant les fonctions responsables de la lecture des données et de l'écriture des résultats (i nput- o utput).

Celles-ci ne font pas partie, en toute rigueur, du langage C.

La ligne 3 appelle la fonction principale, et les parenthèses vides indiquent que cette fonction n'attend pas d'arguments.

La ligne 5 est une instruction d'écriture (appel à la fonction printf). Cette fonction reçoit un argument, la chaine de caractères (entre guillemets anglais) *Bonjour tout le monde !* . Cette chaîne est suivie d'un «caractère d'échappement» \n , qui demande de passer à la ligne.

2.2.3 le même en Pascal

Voici maintenant un programme équivalent, rédigé en Pascal.

Intro_C

5

1 {Premier programme}

2

3 PROGRAM bonjour;

4 BEGIN

5 WRITELN('Bonjour tout le monde!');

6

7 END.

Le mot clé

program doit être présent. Le corps du programme est compris entre BEGIN et

END. Les chaines de caractères sont écrites entre apostrophes et non entre guillemets. Bien que Pascal ne distingue pas entre majuscule et minuscule, il est commode de choisir l'une des casses pour les mots du langage, l'autre pour les variables définies par l'utilisateur.

2.2.4 format libre

Dans les trois langages, la position des mots sur la ligne est indifférente (on parle de format libre). Le compilateur s'y retrouve grace aux mots réservés (ou aux accolades) et aux points-virgules qui terminent chaque instruction (sauf les instructions destinées au préprocesseur, qui doivent être chacune seule sur sa ligne, peut-être avec un commentaire). Ainsi, j'aurais pu écrire le programme C++ précédent sous la forme

1 /*Premier programme horrible*/ #include <iostream>

2 int main(void){std::cout<<"Bonjour tout le monde!"<<std::endl;return 0;}

La lisibilité n'est pas la même. Il y a grand intérêt à mettre en évidence la logique du programme à l'aide de la mise en page du texte. Chaque programmeur doit doit créer sa propre présentation, combinant clarté, lisibilité et économie de place.

2.2.5 saisir des nombres au clavier

Le programme suivant (addition de deux entiers), va nous permettre de découvrir quelques caractéristiques supplémentaires du langage.

```
1
//somme . cpp : l i t deux e n t i e r s e t a f f i c h e l e u r somme
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using s td : : c in ; using s td : : cout ; using s td : : endl ;
5
int main ( void ){
6
int nb1 , nb2 , somme ;
7 cout << "donner l e premier e n t i e r : \n" ;
8 c in >> nb1 ;
9 cout << "donner l e deuxième e n t i e r : " << endl ;
10 c in >> nb2 ;
11 somme = nb1 + nb2 ;
12 cout << " l a somme vaut : " << somme << "\n" ;
13 system( " pause " ) ;
14
return 0 ;
15 }
```


J'ai fait savoir au compilateur (ligne 4) que j'allais utiliser les opérateurs standard

cin, cout

et

endl , ce qui va me dispenser de réécrire std:: à chaque utilisation.

Sur la ligne 6, j'ai déclaré les variables que je veux utiliser. Beaucoup de recettes de cuisine sont rédigées de cette façon : on énonce les ingrédients requis avant de donner les instructions pour confectionner le plat. Ces variables sont toutes des entiers. Un nom de variable (un «identificateur») ne doit pas commencer par un chiffre et peut comporter jusqu'à 31 lettres ou chiffres (le seul autre caractère permis est le souligné _). Le nom d'une variable doit être significatif et aider à la compréhension du programme (par exemple prix_par_kilo plutôt que pk3). En C, les déclarations sont placées avant toute instruction exécutable. Le C++ admet que les variables soient déclarées juste avant leur utilisation ; cela diminue les risques d'erreurs dans les grands programmes.

Les lignes 7 et 9 contiennent deux instructions (injection dans le flot de sortie) d'écriture à l'écran, comme précédemment. J'ai mélangé diverses forme du retour à la ligne, toutes compréhensibles par C++.

Les lignes 8 et 10 réalisent l'opération inverse : l'extraction de deux valeurs à partir du « flot d'entrée», ou encore la lecture de deux entiers tapés au clavier.

À l'exécution des lignes 7 et 8 par exemple, l'écran affichera

donner le premier entier:

et l'ordinateur attendra (curseur à gauche de l'écran) que l'utilisateur tape la valeur de nb1, suivie de

< return > ou < entrée > .

L'instruction de la ligne 10 est une «affectation» : la variable

somme reçoit une valeur, laquelle

est justement la somme nb1+nb2.

On se rend compte que les variables (ou les identificateurs) n'ont pas d'existence réelle ; ce sont des noms (plutôt des pseudonymes) pour des emplacements dans la mémoire de la machine. La déclaration d'une variable sert d'une part à réserver un emplacement, d'autre part à définir sa taille, car toutes les variables n'ont pas le même encombrement.

2.2.6 la version C

Je présente maintenant et pour la dernière fois, la version C pure.

```
1 /*somme.c: lecture de deux entiers et affichage de leur somme*/
2 #include <stdio.h>
3 #include <stdlib.h>
4 int main(void)
5 {
6 int nb1, nb2, somme;
7 printf("donner le premier entier: \n");
8 scanf("%d", &nb1);
9 printf("donner le deuxième entier: \n");
10 scanf("%d", &nb2);
11 somme = nb1 + nb2;
12 printf("la somme vaut %d\n", somme);
13 system("pause");
14 return 0;
15 }
```

Sur la ligne 6, j'ai déclaré les variables que je veux utiliser.

Les lignes 7 et 9 contiennent deux instructions (appels de la fonction printf) d'écriture à l'écran, comme précédemment.

À la ligne 8 (de même qu'en 10), on voit apparaître une nouvelle fonction,

scanf , chargée de

capter ce que l'utilisateur tape au clavier. Cette fonction utilise deux arguments. Le premier est une chaine de caractères qui décrit la nature de l'information à saisir, ici un entier (le symbole %d).

Le deuxième est le nom de la variable qui doit recueillir l'information, l'un des identificateurs

nb1

ou

nb2 , précédé du symbole magique & (la raison de cette construction deviendra claire plus tard).

Cette complication est une source assez fréquente d'erreurs chez les débutants en C.

2.2.7 la version Pascal

Terminons ce paragraphe en citant un programme équivalent en Pascal.

```
1 {Deuxième programme}
2
3 PROGRAM addition;
4 VAR nb1, nb2, somme: integer;
5 BEGIN
6 WRITE('donnez le premier entier: '); READLN(nb1);
7 WRITE('donnez le deuxième entier: '); READLN(nb2);
8 somme := nb1 + nb2;
9 WRITELN('voici leur somme':, somme);
10 END;
```

On remarque une nouvelle différence entre les deux langages : le « = » du C (affectation) est l'équivalent du « := » du Pascal. Je profite de l'occasion pour vous faire remarquer la différence fondamentale qui existe entre affectation et égalité. La phrase «si x = 3 alors . . .» effectue une comparaison entre x et 3, sans modifier la valeur de x ; par contre, «posons x = 3» modifie bien cette variable : elle lui affecte la valeur 3. Tous les langages de programmation modernes différencient ces deux significations du signe «égal».

2.3 Arithmétique en C

Le programme précédent vous a permis de voir comment on additionnait deux entiers. En fait, chaque opération arithmétique a son équivalent en C++ : on combine deux variables au moyen d'un « opérateur», comme indiqué dans le tableau suivant.

Opération arithmétique	Opérateur	Expression instruction C

Addition + a + x a + x

Soustraction - a - b a - b

Multiplication * bx b*x

Division / a/b a/b

reste modulo % r mod s r%s

2.3.1 priorités

Pour que le résultat d'une opération arithmétique comportant plusieurs « opérandes» soit bien défini, il faut établir des règles de priorité ; pour le C++, comme pour la plupart des langages de programmation, l'ordre de priorité décroissante est : parenthèses, multiplication ou division, addition ou soustraction. Une expression faisant intervenir des opérateurs de même priorité est

évaluée de gauche à droite. Les quelques exemples qui suivent montrent que l'arithmétique en C++ est en fait très simple.

2 + 3

¡ 4 = 1 ; 6 = 2 ? 3 = 9 ; 6 ? 2 = 3 = 4 ; 3 ¡ 2 ? 2 = ¡ 1 ; 1 + 9 = 3 ¡ 2 = 2

2

? (4 ¡ 1) = 6 = (1 + 2 + 3 + 4 + 5 + 6) = 3 ¡ 1

Évaluons un trinôme du second degré,

$y = ax^2 + bx + c$, avec $a = 2$, $b = 3$, $c = 7$ et

x

= 5 . Comme la fonction puissance n'est pas encore définie, l'instruction correspondante s'écrira

y = a*x*x + b*x +c

. Le diagramme montre l'ordre de opérations successives, indiquées par des

numéros entre parenthèses.

y = 2 * 5 * 5 + 3 * 5 + 7

(6) (1) (2) (4) (3) (5)

En d'autres termes, l'expression de

y prend les formes successives suivantes :

y

= 10 *?* 5 + 3 *?* 5 + 7 = 50 + 3 *?* 5 + 7 = 50 + 15 + 7 = 65 + 7 = 72 *:*

L'affectation d'une valeur à

y est la dernière opération effectuée.

2.4 Prendre des décisions : les opérateurs de relation

On peut comparer deux variables à l'aide d'un «opérateur de relation», comme dans l'expression

suivante :

x < *y* . Cette expression ne peut prendre que deux valeurs, vrai (si *y* est effectivement

plus grand que

x) ou faux (autres cas). On parle souvent «d'expression logique» pour désigner

ce type de construction. Le C++ reconnait 6 opérateurs de relation, rassemblés dans le tableau

ci-dessous.

Expression algébrique Expression en C signification

=

x == *y* *x* égal à *y*

6

= *x* ! = *y* *x* différent de *y*

< *x* < *y* *x*

plus petit que *y*

> *x* > *y* *x*

plus grand que *y*

.

x < = *y* *x* plus petit que ou égal à *y*

?

x > = *y* *x* plus grand que ou égal à *y*

Je le répète : tous les langages modernes représentent de manière différente l'affectation (= en C++, := en Pascal) et la comparaison, == en C++, = en Pascal.

2.4.1 petit exemple

La façon la plus courante d'utiliser une expression logique construite autour d'un opérateur de relation consiste à employer le mot réservé « if » (si) dans des instructions traduisant des phrases comme « si

a est plus grand que *b* , alors...», comme dans l'exemple rudimentaire qui suit.

```
1
//compar . cpp : l e c t u r e e t comparaison de deux e n t i e r s
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int nb1 , nb2 ;
7 cout << "donner moi deux e n t i e r s e t j e vous d i r a i \n" ;
8 cout << " q u e l l e r e l a t i o n e x i s t e ent r e eux : " << endl ;
9 c in >> nb1 >> nb2 ;
```



```
10
i f ( nb1 == nb2 ) cout << nb1 << " e s t é g a l à " << nb2 << endl ;
11
i f ( nb1 != nb2 )
12 cout << nb1 << " n ' e s t pas é g a l à " << nb2 << endl ;
13
```

```
i f ( nb1 < nb2 )
14 cout << nb1 << " e s t plus p e t i t que " << nb2 << endl ;
15
i f ( nb1> nb2 ) cout << nb1 << " e s t PLUS GRAND que " << nb2 << endl ;
16
i f ( nb1 <= nb2 )
17 cout << nb1 << " e s t plus p e t i t que ou é g a l à " << nb2 << endl ;
18
i f ( nb1 >= nb2 )
19 cout << nb1 << " e s t plus grand que ou é g a l à " << nb2 << endl ;
20 system( " pause " ) ;
21
return 0 ;
22 }
```

Le programme comporte (ligne 4) une abbréviation plus puissante que la précédente ; plutôt que d'écrire

std::cout à chaque instruction d'affichage, ou encore using std::cout et ses analogues pour chaque opérateur, je préviens une fois pour toute que je veux pouvoir utiliser tous les identificateurs qui font partie de la bibliothèque standard.

J'ai écrit (ligne 9) une instruction de lecture un peu plus compliquée, qui capte les valeurs de deux entiers. Les deux valeurs peuvent être séparées par un espace ou un passage à la ligne.

Viennent ensuite 6 instructions, allant chacune d'un

if à un point-virgule dont la signification est

la suivante : si l'affirmation entre parenthèses est vraie, alors on exécute l'affichage correspondant ; sinon, on passe à l'instruction suivante. J'ai joué avec les blancs et les passages à la ligne pour rendre le programme lisible et varié (?). Les parenthèses qui entourent la condition qui suit chaque

if

sont obligatoires.

2.5 Programmer par vous-même

Nous venons de passer en revue quelques éléments du langage C++; ils sont suffisants pour écrire des programmes simples. Comment cela se passe-t-il en pratique ? En gros, il y a cinq étapes.

- Concevoir l'algorithme. C'est la partie la plus importante, mais elle est souvent passée sous silence. Pour gagner du temps dans les cours de programmation, on fournit souvent l'algorithme en même temps que l'exercice à résoudre. Il est cependant très souhaitable de réfléchir quelques instants à la méthode que l'on va employer, aux variables que l'on devra utiliser, avant d'écrire la moindre ligne de programme. Tout informaticien devrait s'inspirer de cette phrase attribuée à Racine :

«Ma pièce, Phèdre, est terminée, je n'ai plus que les vers à écrire.»

- Rédiger le programme sur papier ou directement dans l'ordinateur, à l'aide d'un traitement de texte, le sauvegarder sur disque ou disquette.
- Compiler le programme.
- Corriger les erreurs détectées par le compilateur et revenir à l'étape précédente, jusqu'à disparition des messages d'erreur.
- Exécuter le programme.
- Corriger les erreurs, améliorer le programme.

Les étapes 2, 3, et 4 peuvent s'exécuter commodément à l'aide d'un «environnement de programmation intégré» qui permet d'alterner sans heurt rédaction et compilation. Les disques durs

des machines mises à votre disposition comportent un ou plusieurs compilateurs C++. par ailleurs, ces logiciels sont en général gratuits et téléchargeables par tous ceux qui ont la patience nécessaire. Ainsi, si vous travaillez sous Windows, vous pouvez vous procurer :

DJGPP, qui fonctionne sous DOS ou dans une fenêtre DOS sous Windows. Il est accompagné de l'interface graphique RHIDE, qui est une copie de celle de Turbo-Pascal (http : //www.delorie.com /djgpp/).

DevC++, qui fonctionne sous Windows et permet d'accéder à l'API Windows (http : // www.bloodshed.net/)

On peut trouver, sur le site de Borland, des versions gratuites, plus ou moins simplifiées, de Turbo-C++.

Les personnes qui travaillent sous Linux bénéficient des logiciels gratuits que l'on trouve naturellement pour ce système d'exploitation, en particulier le compilateur g++ et le traitement de texte emacs.

Il existe par ailleurs un très grand nombre de sites consacrés à l'enseignement du C ou du C++; vous trouverez ici : http ://www.developpez.com/c/ un bon point de départ.

Enfin, la plupart des questions que l'on peut se poser sur le langage C++ trouvent leur réponse dans la Foire Aaux Questions (FAQ) : http ://www.research.att.com/ austern/csc/faq.html

Dernière remarque : les fichiers de programme rédigés en C doivent comporter le suffixe .c , alors que ceux qui sont écrits en C++ doivent s'appeler nnnn.cpp pour ne pas troubler le compilateur.

Chapitre 3

Ça se complique : les structures de programmation

Les instructions d'un programme sont exécutées l'une après l'autre, dans l'ordre où elles sont écrites. Cette « exécution séquentielle» n'est pas toujours souhaitable. Ainsi, le dernier programme du chapitre précédent comportait des ordres d'impression qui n'étaient exécutés que lorsque certaines conditions étaient remplies. Dans la pratique, on rencontre souvent des enchaînements plus compliqués. Il arrive aussi que l'on souhaite répéter certaines instructions un grand nombre de fois ; il serait pénible d'avoir à écrire un nombre égal de lignes de programme. Les «structures de programmation» présentées dans ce paragraphe permettent justement de choisir avec précision

l'enchaînement et/ou la répétition des instructions d'un programme.

3.1 Choix

3.1.1 deux cas seulement

Je commence par compléter ce qui a été dit au paragraphe précédent concernant la construction « si. . .alors. . .». Je trouve commode de mettre en évidence la logique du programme en rédigeant d'abord une ébauche en «pseudo-code». À titre d'exemple, j'envisage d'imprimer des résultats d'examen ; l'une des actions à réaliser se résume en

si note

> = 10, alors imprimer reçu.

L'avantage de cette formulation est qu'elle se traduit immédiatement en C++ :

```
if (note >= 10)
cout << "Reçu\n";
```

Certains préfèrent une représentation graphique (organigramme), avec des flèches représentant les cas vrai ou faux ; ces représentations sont équivalentes, mais le pseudo-code se généralise plus facilement à un algorithme complexe.

Prenons maintenant en compte le cas des candidats ajournés, au moyen de la structure

si note

> = 10, alors imprimer reçu, sinon imprimer ajourné.

La version C++ s'écrit

```
if (note >= 10)
cout << "Reçu\n";
else
cout "Ajourné\n";
```


Il n'y a qu'une différence avec Pascal : dans ce langage, chaque

if doit être associé à un then

3.1.2 l'opérateur ternaire « ? »

On dispose de plus en C/C++ d'un «opérateur conditionnel» à

trois opérandes , qui permet de

résumer le code précédent en une ligne, comme ceci :

note >= 10 ? cout << "Reçu" : cout << "Ajourné";

Si la condition est vraie, on exécute la première instruction, si elle est fausse, la deuxième. Remarquez

les deux points qui servent à séparer les deux instructions possibles. On peut parvenir au

même résultat d'une autre façon, plus obscure :

cout << (note >= 10 ? "Reçu" : "Ajourné");

La parenthèse contient une expression logique (la condition

note >= 10), l'opérateur « ? » et deux

chaînes de caractères. Si la condition est vraie, l'ensemble prend une valeur égale au deuxième

argument (ici la chaîne

Reçu) ; dans le cas contraire, l'ensemble prend la valeur du troisième

argument (la chaîne

Ajourné). Dans l'un ou l'autre cas, le résultat du ? est injecté dans le flot

de sortie.

3.1.3 beaucoup de cas

Pour tenir compte de l'existence de mentions, je suis amené à compliquer le schéma précédent.

Je pars de l'ébauche

si note

> = 16, imprimer TB

sinon

si note

> = 14 imprimer B

sinon

si note

> = 12 imprimer AB

sinon

si note

> = 10 imprimer P

sinon imprimer Ajourné

dont la traduction en C++ peut s'écrire de plusieurs façons, comme par exemple

```
if (note >= 16)
cout << "TB\n";
else
if (note >= 14)
cout << "B\n";
else
if (note >= 12)
cout << "AB\n";
else
if (note >= 10)
cout << "P\n";
else
cout << "Ajourné\n";
```

```
if (note >= 16)
cout << "TB\n";
else if (note >= 14)
cout << "B\n";
else if (note >= 12)
cout << "AB\n";
else if (note >= 10)
cout << "P\n";
```

```
else
cout << "Ajourné\n";
```

La forme de droite est sans doute préférable car tout aussi compréhensible mais moins encombrante.

Dans la vie courante comme en programmation, un « si » peut très bien commander plusieurs actions. Pour réaliser cela en C++, je dois créer une « instruction composée » (ou un «bloc» d'instructions), formée de plusieurs instructions simples entourées d'accolades, comme ceci :


```
if (note >= 10)
cout << "Reçu\n");
else {
cout << "Ajourné" << endl;
cout << "Prenez rendez-vous avec votre enseignant" << endl;
}
```

Toutes les instructions entre { et } sont solidaires et seront exécutées en bloc (de même qu'en Pascal, tout ce qui se trouve entre BEGIN et END).

3.2 Répétitions

3.2.1 tant que

J'aborde maintenant un premier exemple de répétition programmée. Je vais utiliser une construction analogue au pseudo-code suivant

```
tant qu'on ne l'a pas fait dix fois
répéter l'action demandée
```

En C++ (et en anglais), « tant que » se dit « while ». Je me propose de lire au clavier 10 entiers et d'en calculer la somme. C'est le rôle du programme while1.cpp .

```
1
// while1.cpp : répétition simple
```

```
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int compteur , nombre , somme ;
7
/ ? i n i t i a l i s a t i o n ? /
8 somme = 0 ; compteur = 1 ;
9
/ ? r é p é t i t i o n ? /
10
while ( compteur <= 10) {
11 cout << " ent r e z un nombre : " ;
12 c in >> nombre ;
13 somme = somme + nombre ;
14 compteur = compteur + 1 ;
15 }
16
/ ? c onc lus i on ? /
17 cout << " l a somme des dix nombres e s t : " << somme << endl ;
18 system( " pause " ) ;
19
return 0 ;
20 }
```

Il faut faire attention à la phase d'initialisation. Certains langages mettent systématiquement à zéro toutes les variables avant d'exécuter la première instruction ; tel n'est pas le cas du C++. Sans

initialisation, le programme peut commencer avec une valeur de

compteur égale à ce qui traîne

dans la mémoire à cet emplacement : note de musique, morceau de texte. . .et ne jamais s'exécuter

ou ne jamais s'arrêter.

Il arrive souvent que je ne sache pas combien de fois il faut répéter une action ; c'est le cas pour

une liste de courses : je m'arrêterai de dépenser lorsque la liste sera épuisée, soit de façon plus

formelle

tant qu'il reste des objets sur ma liste

chercher et acheter l'objet suivant

rayer son nom sur la liste.

L'ordinateur ne sait pas (pas encore) interrompre la répétition quand il arrive au bout d'une

liste. Il revient au même de lui demander d'arrêter lorsqu'apparaît une valeur «anormale» dans

la liste ; on dit que la liste se termine par un «drapeau» ou une «sentinelle». Je me propose de

calculer la taille moyenne des élèves d'une classe ; il est commode de terminer la saisie des données

en fournissant une valeur négative ou nulle, comme dans

while2.cpp .

```
1
// whi l e2 . cpp : c a l c u l e l a moyenne de p l u s i e u r s nombres
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ) {
6
```

```
int compteur ;
7
f loat t a i l l e , moyenne , somme ;
8
/ ? i n i t i a l i s a t i o n ? /
9 somme = 0 ; compteur = 0 ;
10
/ ? t r a i t ement ? /
11 cout << " ent r e z une t a i l l e (m, <= 0 pour f i n i r ) : " ;
12 c in >> t a i l l e ;
13
while ( t a i l l e > 0) {
14 somme = somme + t a i l l e ;
15 compteur = compteur + 1 ;
16 cout << " ent r e z une t a i l l e (m, <= 0 pour f i n i r ) : " ;
17 c in >> t a i l l e ;
18 }
19
/ ? c onc lus i on ? /
20 moyenne = somme/ compteur ;
21 cout << " l a t a i l l e moyenne e s t : " << moyenne << endl ;
22 system( " pause " ) ;
23 }
```

Il y a plusieurs nouveautés dans ce programme. Sur la ligne 7, j'ai déclaré trois variables qui représentent des nombres fractionnaires (on dit parfois à virgule flottante, SINGLE en Pascal), ce qui est assez normal pour des tailles exprimées en mètres. Vous remarquerez que l'initialisation est différente de celle de l'exemple précédent (compteur = 0). D'autre part, pour que la condition d'arrêt (taille

< = 0) ait un sens, il faut que taille soit connue, ce qui m'impose de le lire une fois

AVANT la boucle, puis une fois à chaque tour de boucle.

3.2.2 opérateurs composés

Les auteurs du C/C++ et bien des amateurs de ces langages aiment les instructions concises ;

c'est peut-être pourquoi la ligne 15 peut aussi s'écrire

compteur += 1;

Plus généralement, toute modification d'une variable de la forme

variable = variable opérateur expression

peut aussi s'écrire

variable opérateur= expression

où opérateur est l'un des «opérateurs arithmétiques» définis au §2. Attention à ne pas introduire

de blanc entre opérateur et =. On peut compacter la ligne 14 de la même façon.

3.2.3 Opérateurs d'incrémentation/décrémentation

Lorsque l'on manipule un compteur (ou un indice), on est souvent amené à augmenter cette

variable d'une unité ; le langage C++ propose une instruction compacte et assez commode pour

ce faire. En fait, les trois instructions ci-dessous sont équivalentes :

i = i+1 ; i += 1 ; i++ ;

Les choses se compliquent un peu lorsque la structure

i++ est incluse dans une expression. La

variable

i i est utilisée puis incrémentée. On emploie souvent le terme de post-incrémentation pour

désigner ce comportement. Il est aussi possible d'augmenter

i de un avant de l'utiliser, au moyen

d'un opérateur de pré-incrémentation,

++i; Vous ne serez pas surpris d'apprendre qu'il existe

aussi des opérateurs de pré- ou post-décrémentation, qui diminuent de un la variable à laquelle ils

s'appliquent, comme dans l'instruction

indice--;

Ces opérateurs sont rarement utilisés seuls, mais ils apparaissent très souvent dans l'écriture des répétitions, comme expliqué plus loin. En attendant, je vous propose d'observer le fonctionnement du programme

incr.cpp , qui met en oeuvre plusieurs opérateurs d'incrémentation.

```
1
/ ? inc r . cpp : pre ¡ e t pos t ¡ inc r ementat ion ? /
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int i ;
7 i = 5 ;
8 cout << i << endl ;
9 cout << i++ << endl ;
10 cout << i << endl << endl ;
11 i = 5 ;
12 cout << i << endl ;
13 cout << ++i << endl ;
14 cout << i << endl ;
15 system( " pause " ) ;
16
return 0 ;
17 }
```

3.2.4 Encore une répétition : la boucle «for»

Examinons d'abord une répétition utilisant un «while». Je me propose d'imprimer, au moyen d'un programme en C++, les 12 premiers nombres entiers, leur carré et leur cube. Le programme ci-dessous convient.

```
1
// whi l e3 . cpp b ouc l e " wh i l e "
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ) {
6
int compteur = 1 ;
7
while ( compteur <= 12) {
8 cout << compteur << ' \ t ' << compteur
? compteur << ' \ t '
9 << compteur
? compteur ? compteur << ' \n ' ;
10 compteur += 1 ;
11 }
12 system( " pause " ) ;
13
return 0 ;
```



```
14 }
```

J'ai utilisé (ligne 6) un nouveau raccourci ; la variable

compteur y est simultanément définie et

initialisée. De plus, j'ai introduit plusieurs caractères de tabulation,

\t ; il s'agit bien d'un seul

caractère, que l'on peut mettre entre apostrophes.

Une instruction «for » peut être considérée à peu près comme un condensé des lignes 6 à 11 du

programme précédent.

```
1
// f o r 1 . cpp : b ouc l e " f o r " e l ement a i r e
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int compteur ;
7
for ( compteur = 1 ; compteur <= 1 2 ; compteur++)
8 cout << compteur << ' \ t ' << compteur
? compteur << ' \ t '
9 << compteur
? compteur ? compteur << ' \n ' ;
10 system( " pause " ) ;
11
return 0 ;
12 }
```

La variable

compteur est déclarée ligne 6 et initialisée ligne 7. Malgré les apparences de la mise en

page, l'opérateur

for ne gouverne qu'une seule instruction, qui commence par cout et se répend

sur deux lignes. Pour répéter un bloc d'instructions, il faut les placer entre accolades.

3.2.5 équivalence «while»-«for»

La structure générale d'une instruction comme celle de la ligne 7 est

for (initialisation ; condition pour continuer ; in- ou dé-crémentation)

instruction ou bloc d'instructions

Plus généralement, on peut écrire

for(Expression_1,Expression_2,Expression_3)

instruction

ce qui est pratiquement équivalent à la construction

Expression_1 ; (initialisation)

while{Expression_2}{ (condition d'arrêt)

instruction

Expression_3 ; (incrémentation)

}

3.2.6 jusqu'à

Tous les exemples de répétition que je viens de présenter ont une caractéristique commune :

on vérifie si une condition est vraie avant d'exécuter la (ou les) instructions de la boucle. Si la

condition est fausse dès le début, la boucle n'est jamais parcourue. Il est parfois utile d'effectuer

cette vérification après avoir décrit au moins une fois le corps de boucle. Au lieu de : tant que

(condition), répéter, on voudrait : répéter, jusqu'à ce que (condition). Cette distinction existe en

Pascal (while et repeat. . .until) ; elle existe aussi, sous une forme un peu différente, en C++, comme

le montre le code suivant, une transposition du programme

while3.cpp .

```
1
// jus qua . cpp : b ouc l e r e p e t e r . . . jus qu ' a ce que
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ) {
6
int compteur = 1 ;
7
do {
8 cout << compteur << ' \ t ' << compteur
? compteur << ' \ t '
9 << compteur
? compteur ? compteur << endl ;
10 compteur += 1 ;
11 }
12
while ( compteur <= 1 2 ) ;
13 system( " pause " ) ;
14
return 0 ;
15 }
```

3.3 Choix entre plusieurs possibilités : switch

Il peut m'arriver d'avoir à choisir entre plusieurs possibilités d'égale importance. Dans ce cas,

plutôt que d'avoir recours à une enfilade de

if . . . else ,je peux utiliser la structure switch , comme

le montre l'exemple un peu artificiel

switch.cpp . Il s'agit cependant d'un programme nettement

plus compliqué que les précédents et il n'est pas nécessaire d'assimiler dès maintenant tous les détails.

L'utilisateur va taper au clavier un certain nombre de voyelles et le programme lui répondra

combien il y avait de a, de e,. . .

```
1
// swi t ch1 . cpp : s t r u c t u r e swi t c h pour compter des v o y e l l e s
2
#include <ios t r eam . h>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int nba = 0 , nbe = 0 , nbi = 0 , nbo = 0 ,
7 nbu = 0 , nby = 0 ;
8
int voyl ;
9 cout << " tape z des v o y e l l e s , en minuscule \n" ;
10 cout << " c a r a c t è r e f i n de f i c h i e r pour a r r ê t e r \n" ;
11
while ( ( voyl = c in . ge t ( ) ) != EOF){
12
switch ( voyl ) {
13
case ' a ' : ++nba ;
14
break ;
```

```
15
case ' e ' : ++nbe ;
16
break ;
17
case ' i ' : ++nbi ;
18
break ;
19
case ' o ' : ++nbo ;
20
break ;
21
case 'u ' : ++nbu ;
22
break ;
23
case ' y ' : ++nby ;
24
break ;
```



```
25
default :
26 cout << " ce n ' e s t pas une v o y e l l e \n" ;
27 cout << " tapez une v o y e l l e s . v . p . \ n" ;
28
break ;
29 }
30 }
31 cout << "nombre de a : " << nba << " nombre de e : " << nbe << endl ;
```

```
32 cout << "nombre de i : " << nbi << " nombre de o : " << nbo << endl ;
33 cout << "nombre de u : " << nbu << " nombre de y : " << nby << endl ;
34 system( " pause " ) ;
35
return 0 ;
36 }
```

Les variables entières

nba , nbe ,. . .sont des compteurs de voyelles. Surprise, le programme lit

des voyelles entreposées dans

voyl , qui est un entier. Ceci est correct et souvent plus commode

que la déclaration rigoureuse

char voyl; . En effet, les caractères sont codés dans la machine sous

forme d'un entier et peuvent être considérées comme des entiers sans signe.

Je trouve ensuite une boucle

while qui lit une série de lettres et s'arrète quand je frappe

une «marque de fin de fichier ».

EOF est la désignation conventionnelle de cette marque, qui est

différente pour chaque système d'exploitation (Apple, Microsoft, Linux) mais la bonne définition

se trouve dans les entêtes

<stdio.h> ou <iostream> . Chez Microsoft, c'est <control>-z .

La logique qui commande la boucle est concentrée dans l'instruction bizarre de la ligne 10

(bizarre mais caractéristique du C++). La fonction

cin.get() lit un caractère au clavier. L'affectation

attribue à

voyl cette même valeur. Enfin, en C++, une affectation a elle-même une valeur,

justement celle que vient de recevoir l'identificateur de gauche (

voyl ici). On peut donc comparer

cette valeur à

EOF et continuer si elle est différente. Ouf. En français, on aurait dit « si le prochain

caractère lu n'est pas

EOF , alors. . .». Enfin, l'écriture bizarre cin.get de la fonction sera expliquée

vers la fin du cours.

On entre ensuite dans la structure

switch . La variable qui régit le fonctionnement de switch

est citée entre parenthèses. On énumère ensuite chaque cas à l'aide du mot clé

case , puis on

énonce les instructions correspondantes. Il est ici inutile de mettre des accolades pour constituer

un groupe solidaire : tout ce qui se trouve entre deux

case est exécuté.

L'instruction

break provoque un saut à la première ligne qui suit la structure switch. Si

break

ne figure pas, toutes les instructions de la structure sont exécutées.

Le dernier cas (

default) est un fourre-tout qui récupère tout ce qui n'est pas une voyelle. On

voit que cette structure est semblable au CASE OF de Pascal, à part

break et default .

3.4 Opérateurs logiques

Jusqu'ici, je n'ai présenté que des prises de décision reposant sur des conditions simples. Mais

il est tout à fait possible de construire des expressions logiques composées ou complexes, à l'aide

d'opérateurs logiques, comme dans « si le coefficient de

x^2 est non nul ET si le déterminant est

positif, alors. . .» La traduction en C++ fait appel à l'un des opérateurs rassemblés dans le tableau

suivant.

Expression logique Expression en C++

négation !

et &&

ou

jj

Le premier opérateur n'attend qu'un seul argument (opérateur « unaire»), qui doit être vrai ou faux ; il produit la valeur opposée. Les deux autres doivent recevoir deux arguments logiques (opérateurs «binaires»), comme par exemple

if (note_moyenne >= 10 || note_examen >= 10)

cout << reçu\n";

Remarquez que la condition globale sera vraie si l'une ou l'autre ou les deux conditions partielles sont vraies (ou non-exclusif).

Chapitre 4

Les fonctions

Le meilleure façon de rédiger un programme complexe est de le construire à partir d'éléments ou de modules indépendants, qu'on assemble un peu comme des briques (On peut aussi penser au proverbe : diviser pour régner lorsqu'il s'agit de maîtriser une tâche complexe). Chaque langage de programmation a une façon à lui de désigner et d'organiser ces modules ; en C++, on ne dispose que d'une seule sorte de module, la fonction. Il existe des fonctions « standards », préprogrammées, qui économisent bien du travail, et les «fonctions utilisateur», écrite par le programmeur. Je rappelle que l'ensemble des modules ou fonctions présentes dans un programme est régi par une fonction principale, qui s'appelle justement « main».

4.1 Les fonctions mathématiques

Toutes les fonctions mathématiques courantes sont prédéfinies en C (un avantage certain sur

Pascal). Les plus courantes sont rassemblées dans le tableau suivant.

Fonction Description Exemple

sqrt(x) racine carrée sqrt(121.0)

¡! 11.0

exp(x) exponentielle,

e^x exp(2) ¡! 7.389056

log(x) logarithme à base

e log(7.389056) ¡! 2.0

log10(x) logarithme à base 10 log10(2.0)

¡! 0.30103

fabs(x) valeur absolue fabs(-2.0)

¡! 2.0

ceil(x) arrondi au plus petit ceil(3.2)

¡! 3.0

entier non inférieur à

x ceil(-4.7) ¡! -4.0

floor(x) arrondi au plus grand floor(7.6)

¡! 7.0

entier non supérieur à

x floor(-8.8) ¡! -9.0

pow(x,y) puissance,

x^y pow(2,10) ¡! 1024.0

pow(27.0,0.3333)

¡! 3.0

sin(x) sinus (argument en radian) sin(1.5707963)

¡! 1.0

cos(x) cosinus (idem) cos(1.5707963)

¡! 0.0

tan(x) tangente (idem) tan(0.0)

¡! 0.0

Comme pour la lecture et l'écriture, ces fonctions ne font pas partie du langage C++ au sens strict : il faut donc prévenir le compilateur que l'on souhaite les utiliser, en insérant au début la directive

#include <cmath> . Ceci n'est pas suffisant pour certains systèmes ; il faudra encore, au moment de la compilation, indiquer que l'on veut utiliser une librairie de fonctions mathématiques « -lm » pour Linux).

Les fonctions mathématiques convertissent automatiquement leur argument dans le «type double» (double précision, 13 chiffres significatifs, 8 octets) et renvoient un résultat de même

type. La notion de «type» sera détaillée un peu plus loin.

Pour C++, le générateur de nombres aléatoires n'est pas une fonction mathématique. La fonction correspondante s'appelle

rand() (sans arguments). Le résultat est un nombre entier aléatoire

compris entre 0 et

RAND_MAX (en majuscules). Ces deux entités sont définies dans un autre fichier

d'entête, « cstdlib».

4.2 Fonctions de l'utilisateur

J'aborde maintenant l'écriture de fonctions propres à l'utilisateur. Toute fonction doit en principe correspondre à une tâche précise et bien définie et son nom doit refléter cette tâche. De plus, une fonction bien conçue est appelée à être réutilisée souvent et donc à économiser du temps de programmeur. Voici un premier exemple simpliste, une fonction qui calcule le cube d'un entier. Elle fait partie d'un programme qui dresse la table des cubes des dix premiers entiers.

1

```
/ ? f_cube . cpp : exemple de f onc t i on ? /
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int cube ( int ) ;
6
int main ( void ){
7
int x ;
8
for ( x = 1 ; x <= 1 0 ; x++)
9 cout << x << "\ t " << cube ( x ) << endl ;
10 system( " pause " ) ;
11
return 0 ;
12 }
13
int cube ( int y ){
14
return y ? y ? y ;
15 }
```

On a vu, dans les chapitres précédents, que ce programme, grâce à l'entête

<iostream> , pouvait

utiliser les fonctions d'écriture comme

cout << . La ligne 5 joue un peu le même rôle : elle prévient

le compilateur que je vais utiliser une fonction recevant un argument entier, dont le résultat sera

aussi un entier et qui s'appellera cube. Le compilateur pourra ainsi vérifier que, tout au long du programme, mes instructions seront conformes à cette définition. La ligne 5 est le « prototype » de la fonction cube ou encore sa déclaration. On peut mentionner des noms de variables dans la déclaration ; le compilateur n'en tient aucun compte, mais cela peut aider à la compréhension du programme. Si la fonction ne renvoyait aucune information vers le programme principal (comme par exemple une fonction destinée à afficher un message), il faudrait la déclarer de type

void .

La fonction principale commence à la ligne 6 ; elle contient essentiellement une boucle « for» qui répète 10 fois la ligne 9, où se fait tout le travail. Cette ligne contient un «appel» de la fonction

cube

, avec son «argument effectif», x.

La fonction

cube elle-même est définie lignes 13-15 ; elle se présente de façon assez semblable à

main

: une entête où sont précisés le type de la fonction, son nom, le type et le nom de l'argument. Le corps de la fonction (entre accolades) est ici réduit à peu de chose : le résultat ligne 14. Remarquez que le prototype est suivi d'un point-virgule, alors que l'entête est suivie d'une accolade ouvrante qui marque le début du corps.

Le fonctionnement du programme est simple. L'ordinateur exécute l'une après l'autre les instructions

de

main . Lorsqu'il parvient à la ligne 9, il imprime la valeur de x , puis exécute les instructions contenues dans la fonction (une seule ici), en remplaçant « l'argument formel»

y par

la valeur courante de

x . L'identificateur cube contient la valeur de x^3 . L'ordinateur reprend la suite

des instructions de

main , c'est-à-dire qu'il imprime la valeur de x^3 , incrémente x et recommence

tout, tant que la condition de contrôle est vérifiée.

Voici un deuxième exemple, la recherche du maximum de trois nombres.

```
1
/ ? max3 . cpp : maximum de t r o i s nombres ? /
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int maximum( int x , int y , int z ){
6
int max = x ;
7
i f ( y > max) max = y ;
8
i f ( z > max) max = z ;
9
return max ;
10 }
11
int main ( void ){
12
int a , b , c ;
13 cout << "donnez t r o i s e n t i e r s : " ;
14 c in >> a >> b >> c ;
15 cout << "Le plus grand e s t : " << maximum( a , b , c ) << endl ;
16 system( " pause " ) ;
17
```

```
return 0 ;
18 }
```

J'ai adopté ici une présentation différente. De même que l'on peut déclarer et initialiser une variable en seule instruction (`int uvw = 321`), on peut déclarer et définir une fonction en une fois. Dans ce cas le prototype est inutile et disparaît. Il semble que la majorité des programmeurs préfère la première présentation (déclaration et définition séparées), peut-être parce que l'ensemble est plus lisible si le corps de la fonction est volumineux.

4.3 Types et conversion de type

4.3.1 des types différents

Toutes les variables d'un programme en C++ doivent avoir un type ; celui-ci peut être soit prédéfini par le langage soit défini par l'utilisateur. On peut comparer la mémoire de l'ordinateur à une bibliothèque, avec des casiers de tailles différentes : des petits casiers pour les livres de poche, des gros casiers pour les atlas. En ce qui concerne les variables numériques, il existe 8 types prédéfinis, auxquels on peut encore rattacher le type « char», très voisin d'un entier. Ils sont listés dans le tableau ci-dessous, par ordre de taille décroissante (et donc de nombre de chiffres significatifs décroissant).

spécification spécification

Type pour printf pour scanf

long double %Lf %Lf

double %f %lf

float %f %f

unsigned long int %lu %lu

long int %ld %ld

unsigned int %u %u

int %d %d

short %hd %hd

char %c %c

Tous les compilateurs ne reconnaissent pas tous ces types, en particulier « long double». J'ai indiqué dans le même tableau les codes de formattage utilisés en C pour les entrées-sorties ; remar

quez le piège classique : la spécification de format des nombres en double précision, très commun en calcul scientifique, est différente pour la lecture et l'écriture ! Les utilisateurs de C++ n'ont pas à tenir compte de cette remarque, puisque

cin et cout formattent automatiquement les données

qui leur sont soumises.

4.3.2 promotion

J'ai déjà dit que les fonctions de la bibliothèque mathématique attendaient un argument de type « double» ; je peux quand même calculer la racine carrée de 4 par l'expression

sqrt(4) . Dans

ce cas, il y a « promotion (conversion) automatique» de l'argument en son équivalent

double , soit

4.00, cela sans perte d'information (je peux ranger une livre de poche dans le casier destiné à une encyclopédie). Le résultat sera donné en double précision.

La même opération de promotion a lieu chaque fois que j'écrit une expression en mélangeant des types ; tous les arguments sont temporairement convertis dans le type le plus précis. Ainsi, si

x = 1.0

et y sont des float , alors que a = 2 est un int , l'affectation y = x + a + 1; donnera

à

y la valeur 4.00. Il est toutefois peu prudent de faire aveuglément confiance à ce mécanisme. De plus, le compilateur va protester si j'écris simplement

y = a .

D'autre part, il faut faire attention aux divisions entre entiers. Ainsi, le fragment de programme

int a = 2, b = 3, c = 7;

cout << a/b << '\t' << c/a << endl;

affichera

0 3 Pour obtenir des résultats plus précis (mais fractionnaires), il faut convertir l'un

(au moins) des facteurs en un nombre fractionnaire. Ceci se fait proprement par un «transtypage»

(«cast» en anglais) :

int a = 2, b = 3, c = 7;

cout << (double)a/b << '\t' << (double)c/(double)a << endl;

Cette écriture est conforme à la norme C et acceptée en C++. En C++, il faut en principe écrire

static_cast<double>(a)/b

.

4.3.3 dégradation

La conversion vers un type moins précis est en fait une dégradation (comme si je voulais

absolument faire pénétrer un dictionnaire dans le casier d'un livre de poche). Si j'appelle la fonction

cube

du paragraphe précédent (qui attend un argument entier) avec un argument fractionnaire,

cube(2.7)

, celui-ci sera tronqué à sa partie entière et j'obtiendrais le résultat 8 au lieu de 19.683.

Pour prendre volontairement la partie entière, il existe les fonctions

floor et ceil , décrites plus

haut.

4.4 Passage des arguments

Il y a en principe deux façons simples de transmettre un (ou des) argument(s) à une fonction :

le « passage par valeur» et le « passage par référence» (on dit aussi « passage par adresse »). En

C++, sauf recours à un formalisme spécial, les arguments simples (type entier, flottant, double,

caractère) sont passés par valeur, ce qui veut dire qu'une copie de l'argument d'appel est transmise

à la fonction. Ceci a un avantage évident : si l'argument est modifié dans le corps de la fonction, cela n'affecte pas la variable du programme principal. Dans d'autres langages (Fortran), on pratique l'appel par référence : toute modification de l'argument dans la fonction appelée est répercutée dans le programme principal. La convention du C++ a aussi un inconvénient : comme la fonction ne peut (par l'intermédiaire du mot réservé

return) renvoyer qu'une valeur unique, comment pourrais-je

construire une fonction dont le résultat serait un ensemble de valeurs (composantes d'un vecteur par exemple) ? La solution sera abordée dans un prochain chapitre.

4.5 portée des variables

À partir du moment on l'on commence à décomposer un programme en blocs et en fonctions, on doit se demander quel est le domaine de validité de chaque variable ou dans quelle portion du programme chaque variable est définie.

Si un programme se compose de plusieurs blocs, il est possible de définir des variables à l'intérieur de chaque bloc, ces variables étant invisibles à l'extérieur de leur bloc de définition, comme dans l'exemple un peu artificiel qui suit.

```
1
// por t e e1 . cpp : p o r t é e des v a r i a b l e s
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
void fonc ( int ) ;
6
int nb = 1000 , val = 128;
```

```
7

int main ( void ){

8 cout << " d i v e r s e s v a l e ur s des v a r i a b l e s : \n" ;

9 cout << "\t

¡ nb dans main : " << nb << "\n\n" ;

10 cout << "\t

¡ val dans main : " << val << "\n\n" ;

11 fonc ( 1 0 0 ) ;

12 cout << "\t

¡ nb apr e s fonc : " << nb << "\n" ;

13 system( " pause " ) ;

14

return 0 ;

15 }

16

void fonc ( int nb){

17 cout << "nb au debut de fonc : " << nb << endl << endl ;

18 cout << " val au debut de fonc : " << val << endl << endl ;

19 {

20 nb = 2 0 ;

21 cout << "nb dans l e premier bloc de fonc : " << nb << "\n\n" ;

22 }

23 {

24 nb = 3 0 ;

25 cout << "nb dans l e deuxieme bloc de fonc : " << nb <<endl << endl ;

26 cout << " val dans l e deuxieme bloc de fonc : " << val << "\n\n" ;

27 }

28 }
```

J'ai défini « au niveau global» ou encore « à la profondeur 0», c'est à dire en dehors de toute fonction ou bloc, des entiers

nb = 1000, val = 128 . Ces objets (identificateurs) sont visibles de tout le

programme, sauf s'ils sont masqués par une définition ultérieure (voir plus loin). On peut définir des variables dans une fonction ou à l'intérieur d'un bloc (entre accolades). Ces définitions sont locales au bloc ou à la fonction. Les variables déclarées dans un bloc (fonction) ne sont visibles que dans ce bloc (fonction) et dans les sous-blocs qu'il (ou elle) contient. D'autre part, une déclaration masque toutes les déclarations d'une variable de même nom à une profondeur inférieure.

L'exécution de portée1.exe donne comme résultat

diverses valeurs des variables:

- nb dans main: 1000

- val dans main: 128

nb au debut de fonc: 100

val au debut de fonc: 128

nb dans le premier bloc de fonc: 20

nb dans le deuxieme bloc de fonc: 30

val dans le deuxieme bloc de fonc: 128

- nb apres fonc: 1000

La logique de ce programme peut être représentée par le dessin ci-contre. Le programme est contenu dans le fichier

portee1.cpp , analogue à une grande boite et qui constitue l'espace de travail du compilateur. À l'intérieur, on trouve quatre objets,

nb, val,main et fonc , dont deux sont aussi des boites. Ces déclarations sont globales. Dans main , on s'intéresse aux deux variables nb et val .

Comme on ne trouve pas de déclaration dans main , ce sont les initialisations précédentes qui sont valables.

main appelle fonc avec l'argument nb = 100 et c'est cette valeur qui a cours à l'intérieur

de la boite

fonc . Seulement, fonc contient deux autres boites, anonymes et qui contiennent chacune

une définition de

nb . C'est cette définition qui est la bonne dans la boite où elle se trouve.

Intro_C

portee1.cpp

nb = 1000 ; val = 128

main

cout

<< nb ; cout << val ;

fonc(100) ;

cout

<< nb ;

fonc(nb)

cout

<< nb ; cout << val ;

nb = 20 ; cout

<< nb ;

nb = 30 ; cout

<< nb ; cout << val

Les déclarations de fonctions sont toujours globales (à la différence de Pascal, où l'on peut définir

une fonction à l'intérieur d'une autre fonction. Voici encore un exemple, à peine plus compliqué.

Pour gagner de la place, j'ai fait figurer plusieurs instructions sur la même ligne, ce qui ne favorise

pas la lisibilité ; elles sont bien sûr lues et exécutées de gauche à droite.

```
1
// por t e e2 . cpp : aut r e exemple de p o r t e e des v a r i a b l e s
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
void a ( void ) ; void b( void ) ; void c ( void ) ;
6
int x = 1 ;
7
int main ( void ){
8
int x = 8 ;
9 cout << "x , d e f i n i dans main , vaut : " << x << endl ;
10 a ( ) ; b ( ) ; c ( ) ;
11 a ( ) ; b ( ) ; c ( ) ;
12 cout << "x , d e f i n i dans main , vaut : " << x << endl ;
13 system( " pause " ) ;
14
return 0 ;
15 }
16
void a ( void ){
17
int x = 2 5 ;
18 cout << "x au debut de a : " << x << endl ;
19 x++;
20 cout << "x a l a f i n de a : "<< x << endl ;
```

```
21 }
22
void b( void ){
23
s tat ic int x = 5 0 ;
24 cout << "x ( s t a t i c ) au debut de b : "<< x << endl ;
25 x++;
26 cout << "x ( s t a t i c ) a l a f i n de b : " << x << endl ;
27 }
28
void c ( void ){
29 cout << "x ( g l o b a l ) au debut de c : " << x << endl ;
30 x
? = 1 0 ;
31 cout << "x ( g l o b a l ) a l a f i n de c : " << x << endl ;
32 }
```

J'ai introduit une nouveauté, le type de variable

static ; une telle variable conserve sa valeur

entre deux appels de la fonction. Essayez de prévoir ce que fera ce programme avant de regarder

le résultat.


```
x, defini dans main, vaut : 8
x au debut de a: 25
x a la fin de a: 26
x(static) au debut de b: 50
x(static) a la fin de b: 51
x(global) au debut de c: 1
x(global) a la fin de c: 10
```

x au debut de a: 25

x a la fin de a: 26

x(static) au debut de b: 51

x(static) a la fin de b: 52

x(global) au debut de c: 10

x(global) a la fin de c: 100

x, defini dans main, vaut : 8

4.6 La récurrence

Beaucoup d'objets mathématiques peuvent être définis de manière récursive, par une relation de récurrence. Le langage C++ permet de même des définitions de fonctions par récurrence. La fonction factorielle est l'exemple traditionnel dans ce domaine. Sa définition mathématique explicite est

n

! = *n* ¢ (*n* ¡ 1) ¢ (*n* ¡ 2) ¢ ¢ ¢ 2 ¢ 1 *:*

Je peux aussi utiliser la définition récursive équivalente

n

! = *n* ¢ (*n* ¡ 1)!

En C++, j'écrirai soit un fragment de programme itératif :

```
fact = 1;
for( cptr = n; cptr >= 1; cptr--)
fact *= cptr;
```

soit un programme appelant une fonction définie par récurrence :

```
1
/ ? facto_r . cpp : f a c t o r i e l l e , forme r é c u r s i v e ? /
2
#include <ios t ream>
3
#include <c s t d l i b >
```

```
4
using namespace s td ;
5
int f a c t ( int ) ;
6
int main ( void ){
7
int i ;
8
for ( i = 1 ; i <= 1 0 ; i++)
9 cout << i << " ! = " << f a c t ( i ) << endl ;
10
return 0 ;
11 }
12
int f a c t ( int x ){
13
i f ( x <= 1)
14
return 1 ;
15
el se
16
return ( x  ? f a c t (x  ¡  1) ) ;
17 }
```

La programmation récursive peut être très élégante et concise. Elle souffre de deux inconvénients.

Les risques d'erreur spectaculaire sont grands. Si je me trompe dans la condition d'arrêt ou

dans la définition de la fonction, la récurrence peut devenir infinie : il n'y a plus qu'à arracher la

prise de courant. De plus, le temps de calcul est souvent élevé, car l'ordinateur doit effectivement évaluer toutes les valeurs intermédiaires de la fonction récurrente,

fact ici.

4.7 Passage d'arguments par référence

On a vu que les arguments «simples» d'une fonction (nombres, caractères) étaient transmis «par valeur» : la fonction reçoit une copie de l'argument. Si cette pratique accroît la sécurité de la programmation (il est impossible de modifier, depuis la fonction, une variable du programme principal), elle ne facilite pas les communications entre fonctions. Si une fonction calcule des valeurs, comment renvoyer ces données dans le programme principal ? C'est possible pour une valeur unique : l'intruction

return permet justement d'affecter à l'identificateur de la fonction une valeur

visible du programme appelant.

Les tableaux sont traités de façon diamètralement opposée : ils sont transmis «par référence» ou «par adresse». Tout se passe comme si la fonction et le programme appelant partageait les mêmes données : toute modification apportée à un élément du tableau dans la fonction est immédiatement répercutée dans

main . L'utilisation de tableaux comme arguments de fonctions est détaillée dans

le chapitre 6.

Il serait commode de pouvoir transmettre plusieurs valeurs de nature différente d'une fonction à une autre : comme elles sont de nature différente, elles ne peuvent pas être des éléments d'un tableau. C++ offre cependant cette possibilité, appelée «transmission par référence». Un paramètre par référence est un pseudonyme (un alias) de l'argument correspondant. Pour passer un paramètre par référence, il suffit de faire suivre le type du paramètre (dans l'entête et dans le prototype de la fonction) par une esperluette (&). Les connaisseurs du C peuvent considérer que ce mécanisme est une version simplifiée du passage par adresse, à l'aide d'un pointeur. Le programme suivant met en oeuvre le passage normal (par valeur) et le passage par référence.

```
// pas sag e . cpp : pas sag e d ' argument par v a l e u r e t par r e f e r enc e
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int cubeVal ( int ) ;
6
void cubeRef ( int &);
7
int main ( void ){
8
int a = 3 , b = ¡ 5;
9 cout << "a avant cubeVal : " << a << endl ;
10 cout << " r e s u l t a t de cubeVal : " << cubeVal ( a ) << endl ;
11 cout << "a apr e s cubeVal : " << a << endl ;
12 cout << "b avant cubeRef : " << b << endl ;
13 cubeRef (b ) ;
14 cout << "b apr e s cubeRef : " << b << endl ;
15 system( " pause " ) ;
16
return 0 ;
17 }
18
int cubeVal ( int aval ){
19
return aval ? = aval ? aval ;
20 }
21
```

```
void cubeRef ( int & aRef ){
22 aRef
? = aRef ? aRef ;
23 }
```

avec le résultat :

Intro_C

30

a avant cubeVal: 3

resultat de cubeVal: 27

a apres cubeVal: 3

b avant cubeRef: -5

b apres cubeRef: -125

Remarquez, ligne 21, que la manipulation d'un paramètre passé par référence est identique à celle d'un paramètre normal.

Il est possible (encore que d'un intérêt faible) d'utiliser un alias dans le corps d'une fonction, comme dans l'extrait ci-dessous.

```
int i = 1;
int &iRef = i;
++iRef; //i est incrémenté par l'intermédiaire de son pseudo
```

Toute variable qui en référence une autre doit être initialisée au moment de sa déclaration ; on peut comprendre cette contrainte en remarquant que

iRef , par exemple est déclarée comme une

copie, mais une copie de quoi demande le compilateur ? On doit répondre immédiatement à cette question.

Chapitre 5

Entrées et sorties

5.1 Généralités

Un programme, écrit par exemple en C++, doit souvent lire des données pour pouvoir fonctionner ; il écrit fréquemment des résultats.

On dit que l'échange de données entre un programme et l'extérieur fait intervenir un « flot », c'est à dire une suite d'octets. Pour l'utilisateur (mais pas pour la machine), chacun de ces octets a une signification : lettre, nombre, pixel, échantillon de son... Pendant une opération de lecture (plus généralement d'entrée de données), les octets passent d'un dispositif extérieur (clavier, disquette, modem) à la mémoire centrale. Pendant une opération d'écriture (sortie de données), les octets passent de la mémoire centrale vers un dispositif extérieur (écran, imprimante, disque).

On distingue les E/S (entrées/sorties) de bas niveau (non-formattées), où l'on se contente de spécifier le nombre d'octets à transmettre entre tel et tel partenaire, et les E/S de haut niveau (formattées), où les octets sont regroupés en ensembles significatifs pour l'utilisateur (nombres entiers, fractionnaires, chaînes de caractères). Je ne parlerai pas des premières, les secondes suffisant à toutes les applications habituelles.

Les manuels emploient souvent les expressions de «dispositif standard de sortie», l'écran, et de «dispositif standard d'entrée», le clavier. Vous avez déjà constaté que l'on pouvait capter des données tapées au clavier et afficher des résultats à l'écran très simplement à l'aide des «opérateurs» cin

et cout . Dans ce chapitre, je vais détailler les propriétés de ces opérateurs, je montrerai comment on peut personnaliser les affichages et j'expliquerai comment on peut lire et écrire dans un fichier sur disque ou sur disquette. Ce dernier point est évidemment important pour les applications, mais aussi dans le cadre de l'enseignement. Pendant la mise au point d'un programme, il est fastidieux de retaper les données à chaque essai et bien plus commode de les lire sur le disque dur.

Une dernière remarque générale. Les concepteurs du C++ se sont donné du mal pour que ces opérations de lecture et d'écriture soient «robustes», ce qui signifie que tous les types classiques de données sont lus ou écrits correctement, sans précaution particulière (ce qui est loin d'être le cas en C pur).

5.2 Les opérateurs d'insertion et d'extraction

L'écriture à l'écran se fait, comme nous le savons, à l'aide de « l'opérateur»

<< , qui insère les

éléments à afficher dans le flot de sortie représenté par

cout (dans le sens des flèches). Je rappelle

que l'on peut écrire indifféremment

cout << "Bienvenue à tous" << endl;

\\

cout << "Bienvenue à tous\n";

32

\\

cout << "Bienvenue";

cout << " à";

cout << " tous";

cout << endl;

\\

cout << "Bienvenue" << " à"

<< " tous" << '\n';

(associativité de gauche à droite et équivalence du caractère d'échappement

\n avec le « manipulateur

de flot»,

endl).

Les mêmes règles s'appliquent à l'affichage des nombres, entiers ou fractionnaires ; l'opérateur

<<

est « assez malin » pour savoir à quel type de donnée il a affaire et pour agir en conséquence.

Symmétriquement, la lecture des données se fait au moyen de « l'opérateur d'extraction du flot d'entrée»,

>> , dans la direction des flèches.

Les opérateurs

<< et >> ont une priorité élevée : il ne faut donc pas hésiter à utiliser des

parenthèses, comme dans le morceau de code suivant, pour être sûr de l'interprétation.

```
cout << "donnez deux entiers: ";
cin >> x >> y;
cout << x << (x == y ? "est " : "n'est pas ") << "égal à" << y ;
```

qui n'est pas correctement compilé si l'on enlève les parenthèses.

Lorsque l'on veut lire une série de données en nombre inconnu, on peut utiliser une boucle « while», comme ceci

```
cout << "entrez un nombre (fin-de-fichier pour arrêter): ";
while( cin >> nb){
.....
cout << "entrez un nombre (fin-de-fichier pour arrêter): ";
}
```

La lecture s'interrompra lorsque l'utilisateur tapera « ctrl-Z» (fin de fichier). L'extraction fournit un résultat nul, interprété comme faux.

On parvient au même résultat en examinant un par un les caractères entrés et en interrompant la lecture dès qu'on détecte le caractère fin-de-fichier (EOF) :

```
char c;
while ( (c = cin.get()) != EOF) {
.....
}
```

5.3 Opérateurs de mise en forme des nombres

Les entêtes de la plupart des fonctions décrites dans ce paragraphe se trouvent dans le fichier

« iomanip», qu'il faut appeler par

#include <iomanip> ; cette bibliothèque contient <iostream> ,

il est donc en principe inutile d'appeler cette dernière, mais tous les compilateurs ne sont pas au courant.

5.3.1 nombre de chiffres significatifs

Pour les applications scientifiques et techniques, il est commode de pouvoir choisir le nombre de chiffres après la virgule ; il existe deux méthodes pratiques de le faire, montrées dans l'exemple ci-dessous.


```
1
// decimal . cpp : nombre de de c imal e s
2
#include <iomanip>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void )
6 {
7
double r2 = s q r t ( 2 ) ;
8
int nbchs ;
9 cout << " spontanement : " << r2 << endl ;
10 cout << " avec l a f o n c t i o n cout . p r e c i s i o n : " << endl ;
11
for ( nbchs = 1 ; nbchs <= 1 0 ; nbchs++){
12 cout . p r e c i s i o n ( nbchs ) ;
```

```
13 cout << r2 << endl ;

14 }

15 cout << " spontanement : " << r2 << endl ;

16 cout << " avec l e manipulateur s e t p r e c i s i o n : " << endl ;

17

for ( nbchs = 1 ; nbchs <= 1 0 ; nbchs++)

18 cout << s e t p r e c i s i o n ( nbchs ) << r2 << endl ;

19 cout << " spontanement : " << r2 << endl ;

20 system( " pause " ) ;

21

return 0 ;

22 }
```

avec le résultat

spontanement: 1.41421

avec la fonction cout.precision:

1

1.4

1.41

1.414

1.4142

1.41421

1.414214

1.4142136

1.41421356

1.414213562

spontanement: 1.414213562

avec le manipulateur setprecision:

1

1.4

1.41

1.414

1.4142

1.41421

1.414214

1.4142136

1.41421356

1.414213562

spontanement: 1.414213562

Spontanément, C++ affiche

p 2 avec 5 chiffres après la virgule ; j'ai modifié ce comportement

d'abord à l'aide de la fonction

cout.precision(n) , dont l'argument est le nombre de chiffres puis

avec le « manipulateur»

setprecision(n) . Remarquez que l'effet produit sur le nombre de chiffres

est permanent, tant qu'une nouvelle instruction ne vient pas le modifier.

5.3.2 largeur de champ

Il est possible de choisir la largeur de la zone (champ) où va apparaître une donnée, grâce à la

fonction

cout.width ou en insérant le manipulateur setwidth(n) .

```
1
//l_champ . cpp
2
#include <iomanip>
3
#include <c s t d l i b >
```

```
4
using namespace s td ;
5
int main ( void ){
6
double mi l l e = 1000 , c e n tmi l l e = 1 0 0 0 0 0 . 5 ;
7 cout << " spontanement : " << endl
8 << mi l l e << '
? ' << c e n tmi l l e << ' ? ' <<endl << endl ;
9
for ( int nbchs = 3 ; nbchs <= 8 ; nbchs++){
10 cout . width ( nbchs ) ;
11 cout << mi l l e << '
? '<< c e n tmi l l e << ' ? ' << endl ;
12 }
13 cout << endl ;
14 cout << " spontanement : " << endl
15 << mi l l e << '
? '<< c e n tmi l l e << ' ? ' << endl << endl ;
16
for ( int nbchs = 3 ; nbchs <= 8 ; nbchs++)
17 cout << setw ( nbchs ) << mi l l e << '
? ' << c e n tmi l l e << ' ? ' << endl ;
18 cout << endl ;
19 cout << " spontanement : " << endl
20 << mi l l e << '
? ' << c e n tmi l l e << ' ? ' << endl ;
21 system( " pause " ) ;
22 }
```

Avec le compilateur que j'utilise, j'obtiens le même résultat dans les deux cas. Au contraire des modifications de précision, ces instructions n'ont pas d'effet permanent : il faut les renouveler pour

chaque objet à imprimer.

spontanement:

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

spontanement:

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

1000*100000*

spontanement:

1000*100000*

C++ est très économe sur les zéros qu'il affiche : le nombre 1.2 sera affiché tel quel, même

si la précision demandée est de 6 chiffres après la virgule. D'autre part, C++ fait de son mieux

pour corriger les erreurs de programmation :

mille et centmille seront affichés tant bien que mal,

même si vous ne réservez que deux caractères pour le faire. Le cas de

centmille (100000.5) est

particulier : si on n'indique pas la précision, C++ imprime 6 chiffres (100000), la largeur du champ

ne fait rien à l'affaire. Enfin, vous remarquez que le nombre 1000 apparaît à droite du champ. Cous trouverez dans les manuels plus détaillés des instructions pour modifier ce cadrage.

5.3.3 notation scientifique

Pour les nombres très grands ou très petits, il est préférable d'employer la notation scientifique. C++ sait le faire, à condition de lui demander.

```
1
// s c i e n t i f . cpp : format s pour l e s nombres
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
double r2 = s q r t ( 2 ) , micro = 1.234 e ¡ 9, mega = 1.234 e9 ;
7
int nb ;
8 cout << " spontanement : " << '\ t ' << '\ t ' << r2 << " " << micro
9 << " " << mega << endl ;
10 cout . s e t f ( i o s : : s c i e n t i f i c ) ;
11 cout << " no t a t i on s c i e n t i f i q u e : " << '\ t ' << r2 << '\ t '
12 << micro << '\ t ' << mega << endl ;
13 cout . uns e t f ( i o s : : s c i e n t i f i c ) ;
14 cout . s e t f ( i o s : : f i x e d ) ;
15 cout << " v i r g u l e f i x e : " << '\ t ' << '\ t ' << r2 << " " << micro
16 << " " << mega << endl ;
17 cout . uns e t f ( i o s : : f i x e d ) ;
18 cout << " spontanement : " << '\ t ' << '\ t ' << r2 << " "<< micro
```

```
19 << " " << mega << endl ;
20 system( " pause " ) ;
21 }
```

Ce programme utilise des «drapeaux» (flags) que l'on peut installer (set) ou désinstaller (unset).

Ils se trouvent dans la bibliothèque ios ; plutôt que de lire celle-ci en entier, avec un include , je vais chercher les objets qui m'intéressent, avec l'opérateur de résolution de portée ::

```
spontanement: 1.41421 1.234e-09 1.234e+09
notation scientifique: 1.414214e+00 1.234000e-09 1.234000e+09
virgule fixe: 1.414214 0.000000 1234000000.000000
spontanement: 1.41421 1.234e-09 1.234e+09
```

Toutes les fonctions et manipulateurs précédents peuvent se combiner dans un même programme, avec des résultats que je vous laisse le soin de découvrir.

5.3.4 un parfum de classe

Vous avez du remarquer que les identificateurs de nombreuses fonctions spécialisées de C++ sont de la forme nom1.nom2(), comme cin.get() ou cout.width() . On voit apparaître ici une trace de ce qui fait l'originalité du C++, la possibilité de définir des classes (ou des objets). Je peux donner une idée simpliste de ce dont il s'agit par analogie avec le type « record (enregistrement)» du Pascal. Cette structure de donnée est commode lorsque l'on veut rassembler des données de nature différente ayant un point commun (nom, prénom, âge, taille et sexe d'un même individu par exemple). On définit alors un enregistrement à plusieurs champs, auxquels on accède par des identificateurs comme rec1.nom , rec1.prenom , rec1.age . La construction correspondante

existe en C/C++, elle s'appelle une

struct , tout simplement. Le C++ pousse l'idée un cran

plus loin : une classe (en simplifiant) est une sorte d'enregistrement qui contient non seulement

des données mais encore des définitions de fonctions, capable d'opérer sur ces données. Ainsi, la

fonction

cout.width() est un « membre» de la classe cout .

5.4 Les fichiers

Les programmes traitent souvent de grandes quantités de données, et peuvent aussi produire énormément de résultats. Il serait tout à fait impossible de saisir à la main ces monceaux d'octets ou de les analyser en temps réel sur l'écran. Pour assembler, archiver ou analyser beaucoup de données, on a recours à des fichiers. Un fichier réside sur un organe de stockage, disque, disquette, CD, etc. Je donne ici quelques éléments d'un vaste sujet.

Nous allons nous intéresser aux fichiers « séquentiels», où les octets sont rangés en file, sans structure spéciale autre que celle prévue par le programmeur. Il existe aussi des fichiers « à accès aléatoire», où l'on peut aller chercher une information connaissant son rang (comme les plages d'un disque). Ils ne sont pas abordés ici.

La manipulation d'un fichier en C++ ressemble beaucoup à celle des entrées-sorties habituelles, et c'est normal : tout est fait pour qu'un quelconque périphérique se présente comme un fichier.

L'ensemble des fonctions utiles est déclaré dans le fichier

fstream qu'il faut donc inclure au début

du programme.

Le processus de déclaration d'un fichier est à peu près le même quelque soit le langage. On commence par établir une relation entre le nom du fichier tel qu'il est ou sera connu sur le périphérique (disque par exemple) et une variable qui le représente dans le programme (

ASSIGN en Pascal). On

précise ensuite s'il s'agit de lire, créer ou ajouter dans un fichier. Il ne reste plus qu'à lire, écrire ou rajouter. Les choses vont paraître un peu mystérieuses parce que je ne veux pas entrer dans

le détail des opérations sur les classes et que je ne présente que ce qui ressemble à des fonctions tordues.

5.4.1 Création d'un fichier

Examinons le petit programme qui suit.

```
1
// c r e_f i ch . cpp : c r é a t i on d 'un f i c h i e r s é q u e n t i e l .
2
#include <ios t ream>
3
#include <f s t ream>
4
#include <c s t d l i b >
5
using namespace s td ;
6
int main ( void ){
7
int nb1 = 1234; double nb2 = 5 . 6 7 8 9 ;
8
char msg [ ] = " bi en l e bonjour chez vous ! " ;
9 of s t r eam f i c h ( "D: / CoursC/ g l o b a l / e s s a i . dta " , i o s : : out ) ;
10 f i c h << msg << endl ;
11 f i c h << nb1 << ' \ t ' << nb2 << endl ;
12 cout << " f i n " << endl ;
13 system( " pause " ) ;
14
return 0 ;
15 }
```

Il crée un fichier et y inscrit une phrase et deux nombres ; on peut vérifier que le contenu de

essai.dta

est bien

bien le bonjour chez vous!

1234 5.6789

À la ligne 9 je définis l'objet

fich et je l'initialise pour qu'il corresponde au fichier extérieur

D:\CoursC\global\essai.dta

. J'ajoute ios::out pour indiquer que je veux écrire dans le fichier

(cette mention est facultative). Si le fichier n'existe pas, il sera créé ; s'il existe, son contenu sera

écrasé. On peut éviter cette issue fâcheuse en remplaçant

ios::out par ios::app , qui demande

que les données soient ajoutées à la fin du fichier existant. On pourrait procéder en deux étapes :

établir la relation d'abord et ouvrir le fichier plus tard, par les lignes suivantes.

```
ofstream fich;
.....
fich.open("D:\CoursC\global\essai.dta", ios::out);
```

Le fichier ainsi créé est, comme on dit, un fichier texte : il peut être lu par n'importe quel

éditeur de texte, ce qui est commode. Le programme précédent pourrait être amélioré en prévoyant

la conduite à tenir si on ne parvient pas à ouvrir le fichier.

Le fichier

fich sera fermé automatiquement lorsque le programme se terminera (à la différence

du C et du Pascal). On peut fermer explicitement les fichiers dont on n'a plus l'usage par

```
fich.close();
```

L'imprimante est considérée comme un fichier dans lequel on peut écrire ; il est donc possible

d'imprimer en remplaçant dans le programme

"D:\CoursC\global\essai.dta" par le nom de

fichier de l'imprimante, souvent

LPT1: . Ceci est vrai pour un fonctionnement dans une fenêtre

DOS, mais pas sous Windows ni sans doute lorsque l'imprimante est accessible par l'intermédiaire

d'un réseau.

5.4.2 Lecture d'un fichier

Le procédé, très voisin du précédent, est illustré ci-dessous. Avec un éditeur de texte ou avec

un programme, j'ai créé le fichier param.txt dont le contenu figure ci-dessous.

mardi

123

0.0258

jeudi

654

10.98

Le programme

lec_fich.cpp

```
1
// l e c_f i c h . cpp : l e c t u r e d 'un f i c h i e r s é q u e n t i e l .
2
#include <ios t ream>
3
#include <f s t ream>
4
#include <c s t d l i b >
5
using namespace s td ;
6
int main ( void ){
7
```

```
int nb1 ; double nb2 ; char j our [ 2 0 ] ;
8 i f s t r e am f i c h ( "D: / CoursC/ g l o b a l /param. txt " , i o s : : in ) ;
9
for ( int i = 1 ; i <= 2 ; i++){
10 f i c h >> j our >> nb1 >> nb2 ;
11 cout << j our << ' \ t ' << nb1 << ' \ t ' << nb2 << endl ;
12 }
13 cout << " f i n " << endl ;
14 system( " pause " ) ;
15
return 0 ;
16 }
```

produit alors le résultat

mardi 123 0.0258

jeudi 654 10.98

fin

Le fichier est fermé automatiquement en fin de programme, mais il pourrait l'être sur demande

(

fich.close()).

En conclusion de cette brève introduction aux fichiers, il faut retenir que ces objets sont extrêmement commodes et que leur emploi n'est pas plus compliqué que celui d'une imprimante.

5.5 «string»

Vous avez remarqué que j'utilisais beaucoup de mots ou de phrases dans les programmes qui illustrent ce cours. Nous venons de voir une application (les fichiers) où les noms jouaient un rôle plus important que celui d'un simple exemple. Dans le jargon de l'informatique, un mot ou une phrase constituent une «chaîne de caractères». Les chaînes se manipulent facilement en Pascal et de façon plus tortueuse en C (comme décrit dans un chapitre suivant). C++ (pas C) comporte cependant une bibliothèque de fonctions spécialisées dans la manipulation aisée de chaînes particulières que je vais décrire sommairement. Pour éviter des confusions, j'emploierai le mot

«string» plutôt que «chaîne» qui sera réservé aux chaînes de caractères de style C. D'autre part, les strings sont des objets que l'on doit manipuler avec les règles de la programmation objet ; pour

simplifier, je vais masquer cet aspect des choses. Pour utiliser ces ressources, il faut appeler la bibliothèque :

```
# include <string>
```

5.5.1 déclaration et initialisation

La déclaration d'une string est banale :

```
string s, str, nom_fich, titre;
```

Une string peut être initialisée de façon normale

```
s = "programmation";
titre = 'X';
str = s;
```

Il existe plusieurs méthodes de déclaration et initialisation couplées :

```
string nom1("Dupont");
string nom2 = "Durand";
string etoiles (10,'*'); // une rangée de 10 étoiles
```

Vous avez du remarquer qu'à aucun moment je n'ai précisé la longueur de la «string». Celle-ci est arbitraire et n'est limitée que par la taille de la mémoire de l'ordinateur, un avantage considérable par rapport aux chaînes de caactères du C ou du Pascal.

5.5.2 lecture et écriture

La lecture et l'écriture de strings sont aussi conventionnelles

```
cout << s << endl;
cin >> nom_fich;
```

La lecture s'arrête au premier blanc. Pour lire tout jusqu'au passage à la ligne :

```
getline (cin,titre);
```

On pourrait lire ou écrire dans un fichier en remplaçant simplement

cin par le nom complet du

fichier.

5.5.3 quelques opérations

Il est très facile de concaténer des strings, à l'aide des opérateurs + et +=. Ainsi

string s1 = "Du", s2 = "pont ", s3 = "de Nemours", s;

s = s1 + s2;

s += s3;

cout << s;

affichera le résultat

Dupont de Nemours .

À chaque string, on peut associer les fonctions

size et length qui renvoient la longueur de

l'objet ; le fragment

int n1 = s1.length();

int n2 = s2.size();

associé aux déclarations précédentes crée deux entiers de valeurs respectives 2 et 4.

Les caractères individuels d'une «string» sont accessibles, comme les éléments d'un tableau

(voir chapitre suivant), ce qui ne signifie pas qu'une string est représentée par un tableau. La numérotation

commence à zéro.

L'instruction cout << s3[3] produit N et l'affectation s2[3] = 'd'

crée la chaîne

pond .

5.5.4 un exemple

Le programme qui suit crée un fichier et y dépose une phrase ; l'utilisateur n'est pas obligé de

taper le nom complet du fichier, le répertoire et le suffixe sont ajoutés par le programme. Certains logiciels anciens redoutent les noms de fichiers comportant plus de 8 caractères, d'où l'affichage de la longueur du nom.

```
12
// f i c h_s t r . cpp : s t r i n g comme nom de f i c h i e r
3
#include <f s t ream>
4
#include <s t r ing>
5
#include <c s t d l i b >
6
using namespace s td ;
7
int main ( void ){
8 s t r i n g nom_fich , base = "D: / CoursC/ g l o b a l /" ;
9 cout << "Nom du Fi c h i e r a c r e e r : " ; c in >> nom_fich ;
10 cout << " l e nom de vot r e f i c h i e r comporte " << nom_fich . s i z e ( )
11 << " c a r a c t è r e s \n" ;
12 nom_fich = base + nom_fich + " . dta " ;
13 of s t r eam f i c h ( nom_fich . c_str ( ) , i o s : : out ) ;
14 f i c h << " que l beau programme ! " << endl ;
15 system( " pause " ) ;
16 }
```

Attention :

Une «string» ne peut pas être utilisée telle quelle dans une déclaration de fichier ;

il faut la convertir en chaîne de style C; c'est ce que fait la fonction

c_str() affectée à l'objet

nom_fich

.

Chapitre 6

Structures de données

Une structure de données est une construction qui essaie de rassembler de façon commode et fiable des données ayant un rapport entre elles et que l'on est souvent amené à manipuler « en bloc ». Il s'agit souvent d'une image informatique d'un objet ou d'un concept du monde réel. Un vecteur, comme une accélération ou un champ électrique, constitue un bon exemple ; avec les connaissances acquises dans les chapitres précédents, je peux représenter les trois composantes d'un champ électrique comme trois nombres
Ex, Ey, Ez . Cette représentation est peu commode
du point de vue du programmeur. Une carte d'identité est un autre objet que je peux souhaiter informatiser ; elle contient des données relative à une même personne : nom, prénom, date de naissance, taille, couleur des yeux. . .Pour l'instant, je ne sais pas manipuler une carte d'identité autrement qu'en lui associant une série d'identificateurs.
Les tableaux sont des « structures de données». Ils constituent en fait la catégorie la plus importante de structure de données en programmation scientifique et technique car ils sont parfaitement adaptés à la représentation des vecteurs et des matrices. Ces structures sont « rigides», c'est-à-dire qu'elles ne changent pas de taille au cours de l'exécution d'un programme ; on connaît en informatique des structures de données (listes, arbres) dont la taille peut croître ou décroître pendant l'exécution.

6.1 tableau : définition

Un tableau est un ensemble d'éléments qui portent tous le même nom et qui appartiennent tous au même type de variable (
int, float, char,...). Ces données occupent en général des cases
successives dans la mémoire, si bien que l'on a tendance à confondre élément du tableau et case mémoire correspondante. Pour désigner un emplacement particulier, on utilise le nom collectif et le

rang ou l'indice de l'élément. La figure ci-dessous représente schématiquement un tableau de nom

c .

c[0] c[1] c[2] c[3] c[4] c[5] c[6] c[7] c[8] c[9] c[10] c[11]

6 -68 123 6874 0 -852 951 -1 528 6 1 7896

Sur la première ligne, j'ai écrit le nom en C++ de chacune des douze variables

qui sont

numérotées de 0 à 11

. La deuxième ligne montre le contenu des douze emplacements mémoire

correspondants : ces variables sont manifestement de type

int . Le nom d'un élément, le sixième par

exemple, est

c[5] . C'est une variable comme une autre, que je peux manipuler comme d'habitude ;

les expressions suivantes ont leur sens habituel :

x = c[2] + c[3]; cout << c[8] << c[11]/c[0] << endl; x = c[1]/3;

L'indice est un entier, soumis aux règles de calcul ordinaire, comme dans

a = 3; b = 4; c[a + b];

Il y a cependant une règle

fondamentale à respecter : l'indice qui désigne l'un des éléments d'un

tableaux

doit rester à l'intérieur de l'intervalle défini lors de la déclaration du tableau (voir plus

loin). Contrairement à ce que l'on peut voir en Pascal, il n'y a aucun mécanisme pour «surveiller» les

indices et aucun message d'erreur si un indice déborde de l'intervalle permis. La seule conséquence,

mais elle est en général spectaculaire, est que l'on lit comme valeur d'une variable une syllabe de

code ou les données d'un autre utilisateur ; de même, on peut écrire sur un fragment de programme

ou sur des données à conserver. Ainsi, le compilateur ne protestera pas si j'écris

c[12] = 22 en

croyant initialiser le dernier élément du tableau précédent.

6.2 Déclaration et initialisation de tableaux

Comme toute variable, un tableau doit être déclaré avant utilisation, mais il faut préciser sa taille, pour que le compilateur réserve l'espace mémoire correspondant. Ainsi, pour le tableau précédent

int c[12]; ou encore

float x[1024]; int a[10], b[10]; char texte[128];

Je peux fixer la valeur des éléments du tableau en même temps que je le déclare :

int c[8] = {1,6,-4,123,-21,0,478,360};

Si la liste entre accolades contient moins de valeurs qu'il n'y a d'éléments, les derniers éléments sont initialisés à zéro. Le cas contraire (plus de valeurs que d'éléments) provoque une erreur de compilation. Il est bien préférable d'écrire

int c[] = {1,6,-4,123,-21,0,478,360};

pour laisser le compilateur compter lui-même.

6.3 Exemples élémentaires

Voici un programme simpliste qui rempli un tableau avec des entiers consécutifs et en calcule la somme.

```
1
// t a b l e a u 1 . cpp : r emp l i s s a g e d 'un t a b l e a u
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
```

```
6
int t [ 1 0 ] , s [ 1 0 ] , i ;
7
for ( i = 0 ; i < 1 0 ; i++)
8 t [ i ] = i ;
9 s [ 0 ] = 0 ;
10
for ( i = 1 ; i < 1 0 ; i++)
11 s [ i ] = s [ i
¡ 1] + t [ i ] ;
12 cout << " element somme cumulee \n" ;
13
for ( i = 0 ; i < 1 0 ; i++)
14 cout << "\ t " << t [ i ] << "\ t \ t " << s [ i ] << "\n" ;
15 system( " pause " ) ;
16 }
```

Je définis deux tableaux (ligne 6) de 10 éléments chacun. J'utilise ensuite une boucle « for » pour initialiser le tableau

t (lignes 7,8). La boucle suivante garnit le tableau s : l'élément s_i est la somme

des éléments de

t , depuis 0 jusqu'à i .

Dans l'exemple suivant, j'imagine que les réponses à un questionnaire ont été codées de 1 à 10.

À partir des résultats de l'enquête, je dois déterminer la fréquence d'apparition de chaque réponse.

J'ai supposé que les données étaient assez peu nombreuses pour permettre une initialisation du tableau dans le corps du programme.

```
1
// t a b l e a u 2 . cpp : t a b l e a u de f r é quenc e s ou histogramme
2
```

```
#include <ios t r eam . h>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int nb_reps , nb_code = 1 0 ;
7
int code , i , f r e q s [ 1 0 ] ;
8
int r eps [ 4 0 ] = { 1 , 3 , 8 , 8 , 4 , 2 , 3 , 1 , 2 , 6 ,
9 4 , 8 , 3 , 7 , 2 , 1 , 3 , 3 , 6 , 7 ,
10 5 , 8 , 2 , 3 , 1 , 9 , 7 , 8 , 6 , 1 ,
11 7 , 4 , 1 , 2 , 3 , 6 , 5 , 4 , 7 , 8 } ;
12
for ( code = 1 ; code <= nb_code ; code++)
13 f r e q s [ code ] = 0 ;
14
for ( i = 0 ; i < 4 0 ; i++)
15 ++f r e q s [ r eps [ i ] ] ;
16 cout << " code f r equenc e " << endl ;
17
for ( code = 1 ; code <= 1 0 ; code++)
18 cout << code << "\ t " << f r e q s [ code ] << endl ;
19 system( " pause " ) ;
20
return 0 ;
21 }
```

Une première boucle (lignes 12,13) me sert à initialiser à zéro le tableau des fréquences. Chaque

élément de ce tableau va en fait jouer le rôle d'un compteur, qui sera incrémenté de un à chaque apparition du code correspondant. Remarquez que je numérote ces éléments/compteurs de 1 à 10, comme les codes correspondants ; cela revient à ignorer

freqs[0] , qui peut contenir n'importe quoi.

Il serait en effet maladroit et malcommode de numéroter différemment le code et son compteur.

La deuxième boucle (lignes 14,15) parcourt les éléments du tableau des réponses ; si par exemple

reps[12]

vaut 4, l'élément freqs[4] = freqs[reps[12]] est incrémenté.

Les lignes suivantes impriment les résultats du décompte. J'insiste encore sur la nécessité de contrôler la validité des données. Si le tableau des réponses contient la valeur 13, le programme essaiera d'incrémenter freqs[13], qui n'existe pas, avec un résultat imprévisible ou catastrophique.

Voici ce qu'imprime le programme :

code frequence

1 6

2 5

3 7

4 4

5 2

6 4

7 5

8 6

9 1

10 0

Le programme précédent fonctionne, mais il est mal écrit ! En effet, il est difficile à modifier. Pour changer un détail comme le nombre de réponses ou le nombre de codes, il faut parcourir tout le programme, sans oublier aucune des apparitions de ces paramètres. C'est assez facile ici, ça le serait moins pour un programme de 10000 lignes. En général, il faut «paramétrer» son programme,

pour permettre des modifications faciles, qui affectent une ou deux lignes ; cela est particulièrement vrai des programmes qui utilisent des tableaux. C'est ce que j'ai déjà fait de façon très partielle, en définissant l'entier

nb_code .

Le langage C offre la possibilité de définir et d'initialiser facilement des paramètres constants grâce au préprocesseur. Il suffit d'introduire au début les définitions

#define NB_REPS 40

#define NB_CODE 10

Remarquez l'absence de ponctuation ! Lorsque le préprocesseur rencontre le nom

NB_CODE dans

le programme, il le remplace

mécaniquement par 10. Si l'utilisateur veut changer le nombre de

réponses, il lui suffit de le faire dans la seule ligne de définition. De même pour

NB_REPS et 40.

L'écriture en majuscules est une convention universellement respectée : elle signe une définition destinée au préprocesseur.

Si ce mécanisme est tout à fait admis en C++, il est cependant peu utilisé, car on dispose d'un autre procédé aussi simple et plus puissant. Il suffit de déclarer et d'initialiser (au niveau global ou à tout autre niveau commode) des constantes entières :

const int nb_reps = 40, nb_code = 10;

Ces «variables» ne sont pas modifiables dans la suite du programme : toute tentative d'affectation d'une nouvelle valeur à

nb_reps ou à nb_code provoquera une erreur de compilation.

Les deux méthodes précédentes offrent un autre avantage intéressant : les symboles ainsi définis peuvent servir lors de la déclaration de la taille d'un tableau. On ne peut pas écrire

int reps[nb_reps];

..... // impossible

cin << nb_reps;

Les tableaux de « dimension variable», définie au moment de l'exécution, sont, en effet, interdits en C comme en C++ sauf appel au mécanisme d'allocation dynamique, qui ne sera pas abordé dans ce cours. Par contre,

const int nb_reps = 40;

int reps[nb_reps];

est licite, et permet d'adapter la taille du tableau aux circonstances, en changeant la valeur d'une constante sur une seule ligne (ou d'un

#define).

Voici une nouvelle version du même programme, tenant compte de ces perfectionnements. J'en profite pour y ajouter le tracé (grossier) d'un histogramme des fréquences. Je vais imprimer, pour chaque valeur de

code , une barre horizontale de longueur proportionnelle à freqs[code] . Auparavant,

il me faudra normaliser les valeurs des éléments de

freqs pour que les barres tiennent dans

la page.

```
1
// h i s t o g . cpp : c ons t r u c t i on amé l ior é e d ' histogramme
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
const int nb_reps = 40 , //nombre de r épons e s
6 nb_code = 10 ,
//nombre de codes ¡ v a l e u r s
7 l_lg = 6 0 ;
```

```
// longueur d ' une l i g n e
8
int main ( void ){
9
int code , i , j , f r e q s [ nb_code+1] , fmax ;
10
int r eps [ nb_reps ] = { 1 , 3 , 8 , 8 , 4 , 2 , 3 , 1 , 2 , 6 ,
11 4 , 8 , 3 , 7 , 2 , 1 , 3 , 3 , 6 , 7 ,
```



```
12 5 , 8 , 2 , 3 , 1 , 9 , 7 , 8 , 6 , 1 ,
13 7 , 4 , 1 , 2 , 3 , 6 , 5 , 4 , 7 , 8 } ;
14
for ( code = 1 ; code <= nb_code ; code++)
15 f r e q s [ code ] = 0 ;
16
// c onf e c t i on de l ' histogramme
17
for ( i = 0 ; i < nb_reps ; i++)
18 ++f r e q s [ r eps [ i ] ] ;
19 cout << " code f r equenc e " << endl ;
20
for ( code = 1 ; code <= nb_code ; code++)
21 cout << code << "\ t " << f r e q s [ code ] << endl ;
22 cout << endl ;
23
// r e che r che de l a f r é quenc e max
24 fmax = f r e q s [ 1 ] ;
25
for ( code = 2 ; code <= nb_code ; code++)
26
```

```
i f ( f r e q s [ code ] > fmax ) fmax = f r e q s [ code ] ;

27

// normal i s at i on

28

for ( code = 1 ; code <= nb_code ; code++)

29 f r e q s [ code ]

? = l_lg /fmax ;

30

// impr e s s ion

31

for ( code = 1 ; code <= nb_code ; code++){

32 cout << code << "\ t " << f r e q s [ code ] << "\ t " ;

33

for ( j = 1 ; j <= f r e q s [ code ] ; j++)

34 cout << '

?' ;

35 cout << "\n" ;

36 }

37 system( " pause " ) ;

38

return 0 ;

39 }
```

Voici le résultat :

```
1 48 ************************************************

2 40 ****************************************

3 56 ********************************************************

4 32 ********************************

5 16 ****************

6 32 ********************************

7 40 ****************************************
```

8 48 **********

9 8 ********

10 0

Vous constatez que les valeurs affichées tiennent compte de la nouvelle normalisation, mais de façon pas tout à fait exacte. Cela est due à la ligne 29, où se produit une erreur d'arrondi. Comment faut-il modifier le programme ?

6.4 Tableau comme argument d'une fonction

Étant donné la déclaration

int temperature[24]; , qui fait référence à un tableau de 24 entiers,

je peux appeler une fonction qui utilise les éléments de ce tableau par l'instruction

calc(temperature, 24).

Il est souvent nécessaire d'informer la fonction de la taille du tableau (24 ici) pour qu'elle traite effectivement tous les éléments.

Attention :

à la différence des arguments « simples », un tableau argument d'une fonction est toujours transmis par référence (on dit aussi par adresse). Cela signifie que toute modification

des éléments de

temperature dans le sous-programme calc se fera sentir dans le programme principal ! La raison profonde est que le nom du tableau contient en fait l'adresse du premier élément ; à l'aide de cette information, la fonction peut faire ce qu'elle veut de chaque élément du tableau. Au contraire, un élément isolé est traité comme une variable ordinaire (passage par valeur). On peut justifier ce comportement par le fait que copier un gros tableau pour le passer par valeur coûterait beaucoup de temps et de place (cf le qualificatif

VAR en Pascal).

Voyons un petit exemple de ces notions.

```
1
/ ? tab_arg . cpp : pas sag e de t a b l e a u e t d ' élément de t a b l e a u en argument ? /
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
#define TAILLE 5
6
void modifTab ( int [ ] , int ) ;
7
void modifElem( int ) ;
8
int main ( void ){
9
int a [TAILLE] = { 0 , 1 , 2 , 3 , 4 , } , i ;
10 cout << " e l ement s du tabl eau de depar t : " << endl ;
11
for ( i = 0 ; i < TAILLE; i++)
12 cout << a [ i ] << "\ t " ;
13 cout << endl ;
14 modifTab ( a ,TAILLE) ;
15 cout << " e l ement s du tabl eau apr e s modifTab : " << endl ;
16
for ( i = 0 ; i < TAILLE; i++)
17 cout << a [ i ] << "\ t " ;
18 cout << endl ;
19 cout << " va l eur de a [ 3 ] : " << a [ 3 ] << endl ;
20 modifElem( a [ 3 ] ) ;
```

```
21 cout << " va l eur de a [ 3 ] apr e s modifElem : " << a [ 3 ] << endl ;
22 system( " pause " ) ;
23
return 0 ;
24 }
25
void modifTab ( int b [ ] , int t a i l l e ){
26
int i ;
27
for ( i = 0 ; i < t a i l l e ; i++)
28 b [ i ]
? = 2 ;
29 }
30
void modifElem( int x ){
31 x
? = 5 ;
32 cout << " va l eur l o c a l e de x modi f i e e : " << x << endl ;
33 }
```

dont le résultat est

elements du tableau de depart:

0 1 2 3 4

elements du tableau apres modifTab:

0 2 4 6 8

valeur de a[3]: 6

valeur locale de x modifiee: 30

valeur de a[3] apres modifElem: 6

6.4.1 Une sage précaution : la déclaration « const »

On a vu que tout tableau fourni comme argument à une fonction était modifiable par celle-ci.

Pour éviter toute modification involontaire, on peut qualifier cet argument de « constant», comme

dans l'exemple suivant.

```
//tab_ct.cpp: argument tableau constant
1 #include <iostream>
2 void modif ( const int []);
3 int main(){
4 int a[] = {1,2,3,4};
5 modif(a);
6 for (int i = 0; i < 4; i++)
7 cout << a[i] << "\t" ;
8 cout << endl;
9 return 0;
10 }
11 void modif (const int b[]){
12 for (int i = 0; i < 4; i++)
13 b[i] += 5; // impossible!
14 }
```

Ce programme, d'apparence correcte, ne sera pas compilé : la fonction

modif tente de modifier

les éléments du tableau

a qui est déclaré comme constant, aussi bien dans l'entête (ligne 2) que

dans la déclaration de la fonction (ligne 12).

6.5 Tableaux à plusieurs dimensions

Dans le jargon de l'informatique, le « nombre de dimensions » d'un tableau est le nombre de

ses indices. Dans la pratique, on rencontre très souvent des tableaux à deux indices (lignes et colonnes),

qu'il s'agisse de représenter des dépenses par rubriques et par mois, des notes par étudiant et par matière ou les cases d'un jeu d'échec. La figure ci-dessous montre un tableau à 3 lignes et quatre colonnes.

col. 0 col. 1 col. 2 col. 3

ligne 0 a[0][0] a[0][1] a[0][2] a[0][3]

ligne 1 a[1][0] a[1][1] a[1][2] a[1][3]

ligne 2 a[2][0] a[2][1] a[2][2] a[2][3]

Dans cet exemple, j'ai suivi la convention du langage C++ : les indices commencent en zéro et donc le premier élément, en haut à gauche, est a[0][0] . J'ai aussi suivi la convention de l'algèbre linéaire : le premier indice d'un tableau désigne la ligne, le deuxième la colonne. Ceci n'est qu'une convention (presque universelle) mais il importe de retenir que cela ne préjuge en rien de la façon dont les éléments sont effectivement rangés en mémoire (ils sont en fait rangés à la queue-leu-leu, ligne par ligne ; le compilateur s'y retrouve, grâce à la déclaration du tableau, qui précise le nombre de lignes et de colonnes).

Un élément quelconque, a[2][3] par exemple, est une variable ordinaire, qui peut être lue, écrite ou manipulée. La surveillance des indices s'impose ; un indice trop grand peu correspondre à un autre élément du tableau, à une autre zone mémoire, ou à n'importe quoi.

La déclaration du tableau précédent se fait comme ceci :

```
int a[3][4];
```

Le nombre d'éléments par ligne ou colonne peut aussi être défini dans une directive du préprocesseur ou comme une constante entière :


```
const int nligne = 6, ncol = 5;
.....
double T[nligne][ncol];
```

L'initialisation d'un tableau peut se faire au moment de la déclaration, comme dans le cas d'un seul indice : on donne les éléments ligne par ligne, groupés au besoin à l'intérieur d'accolades. Le compilateur initialise à zéro les éléments pour lesquels on ne fournit pas de valeur. Voici un exemple simple de création et de manipulation de tableaux.

```
1
// tab2d . cpp : a f f i c h a g e de t a b l e a u x d i v e r s
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
const int NLG = 2 ;
6
const int NCL = 3 ;
7
void Af f i c h e ( char [ ] , const int [ ] [NCL] ) ;
8
int main ( ) {
9
int tab1 [NLG] [NCL] = {{ 1 , 2 , 3 } , { 7 , 8 , 9} } ,
10 tab2 [NLG] [NCL] = { 1 , 2 , 3 , 4 , 5 } ,
11 tab3 [NLG] [NCL] = {{4 ,5} ,{8}} ,
12 tab4 [NLG] [NCL] ;
13 Af f i c h e ( " tabl eau 1" , tab1 ) ;
14 Af f i c h e ( " tabl eau 2" , tab2 ) ;
15 Af f i c h e ( " tabl eau 3" , tab3 ) ;
16
for ( int l = 0 ; l < NLG; l++)
17
```

```
for ( int c = 0 ; c < NCL; c++)
18 tab4 [ l ] [ c ] = tab1 [ l ] [ c ] + tab2 [ l ] [ c ] + tab3 [ l ] [ c ] ;
19 Af f i c h e ( " tabl eau 4" , tab4 ) ;
20 system( " pause " ) ;
21
return 0 ;
22 }
23
void Af f i c h e ( char ph [ ] , const int a [ ] [NCL] ) {
24 cout << "\ t " << ph << endl ;
25 cout << "\ t " ;
26
for ( int c = 0 ; c < NCL; c++)
27 cout << c << "\ t " ;
28 cout << endl << endl ;
29
for ( int l = 0 ; l < NLG; l++){
30 cout << l << "\ t " ;
31
for ( int c = 0 ; c < NCL; c++)
32 cout << a [ l ] [ c ] <<"\ t " ;
33 cout << endl ;
34 }
35 cout << endl << endl ;
36 }
```

J'ai défini quatre tableaux, dont trois ont été plus ou moins complètement initialisés au même moment. La fonction

Affiche ne fait que cela : afficher à l'écran le contenu de ces tableaux. Elle

utilise deux boucles emboîtées pour écrire les éléments, ligne par ligne. Les lignes 14,15,16 du programme principal réalisent la somme, élément par élément, de trois tableaux, encore à l'aide

d'une double boucle. Plus surprenant peut-être, l'entête et la déclaration de la fonction

Affiche ne

contiennent pas l'indice (unique) de

ph , un tableau de caractères ni le *premier* indice du tableau à

deux dimensions : le compilateur peut reconstituer cette information (grâce aux variables globales

NCL, NLG

et la fonction est un peu plus générale comme cela et un peu plus compacte, un avantage

appréciable aux yeux des fanatiques.

6.6 Chaînes

6.6.1 caractères

Je rappelle que C++ définit le type caractère comme dans

char a = 'z'; qui définit une

variable

a de type char et l'initialise à la valeur de la constante 'z' . En fait, en interne, ce 'z'

est représenté (codé) par l'entier 122. Ceci fait que la distinction entre caractères et entiers sans

signe (compris entre 0 et 255) est assez ténue.

La correspondance entre caractères et codes est explicitée dans la «table ASCII». Cette table

représente le système de codage le plus fréquent, mais pas le seul (EBCDIC est utilisé par IBM).

Dans le système ASCII, les chiffres sont codés de 48 à 57, les majuscules de 65 à 90 et les minuscules

de 97 à 122. Divers signes d'imprimerie et des caractères non imprimables remplissent les autres

cases. De 128 à 255, on est dans un no man's land où toutes les conventions sont permises.

L'ordinateur peut donc classer des caractères (et aussi des chaines)par ordre alphabétique : il

lui suffit de classer par ordre de code croissant. Remarquez que A (majuscule) viendra avant a

(minuscule).

6.6.2 Définition de chaines

Une ensemble de caractères considérés comme un tout constitue une «constante chaine de caractères». Elle s'écrit entre guillements. Les chaines de caractères sont en C/C++ des êtres un peu hybrides, des tableaux avec des caractéristiques particulières. Les éléments d'un tableau doivent être tous de même type, mais ce type peut être quelconque, par exemple le type char . En fait, une chaîne de caractères comme bonjour ! * est considérée par C/C++ comme un tableau de caractères TERMINÉ PAR LE CARACTÈRE «nul», '\0' (pas l'entier zéro mais le caractère de code ASCII 0). La taille réelle de ce tableau est donc nombre-de-caractères + 1.

Attention :

Une chaine telle qu'elle vient d'etre définie est héritée du C. Sa construction est complètement différente celle de la structure «string» décrite au chapitre précédent. Seules les chaines de style C sont permises comme noms de fichiers. Rappel : On convertit une «string» S en une «chaine» par l'instruction

s = S.c_str()

6.6.3 déclaration et initialisation

Je peux déclarer et initialiser un tableau de caractères comme ceci :

char string[] = "bonjour !"

Ce tableau comporte dix éléments . En effet, le compilateur ajoute automatiquement à la fin de cette chaîne le « caractère nul ». Je peux tout aussi bien considérer la chaîne comme un tableau de caractères individuels, que j'initialise comme un tableau, en fournissant moi-même le caractère nul :

char string[] = {'b','o','n','j','o','u','r',' ','!','\0'}

Vous voyez que, dans ce formalisme, un caractère ne peut être confondu avec une chaîne de longueur

un (ce que l'on fait en Pascal).

6.6.4 exemple

On dispose encore d'autres méthodes pour initialiser une variable chaine. Certaines sont identiques à ce que l'on a vu pour les tableaux standards. Voici quelques exemple de manipulation de chaînes.


```
1
// chaine1 . cpp : d e f i n i t i o n s de chaine s
2
#include <ios t ream>
3
#include <cctype>
4
#include <c s t d l i b >
5
using namespace s td ;
6
int main ( void ){
7
char ch1 [ ] = " Li c enc e " , ch2 [ 4 0 ] ;
8
char ch3 [ ] = { 'm' , ' a ' , ' i ' , ' t ' , ' r ' , ' i ' , ' s ' , ' e ' , ' \0 ' } ;
9
char ch4 [ 1 0 ] ; char ? ch5 = " t ruc " ;
10 cout << " e n t r e r un mot \n" ;
11 c in >> ch2 ;
12
for ( char i = 0 ; i < 9 ; i++)
13 ch4 [ i ] = i + 4 8 ;
```

```
14 ch4 [ 9 ] = ' \0 ' ;

15 cout << ch1 << "

??? " << endl ;

16 cout << ch2 << "

??? " << endl ;

17 cout << ch3 << "

??? " << endl ;

18 cout << ch4 << "

??? " << endl ;

19 ch1 [ 1 ] = ' y ' ;

20 cout << ch1 << endl ;

21 ch1 [1]++;

22 cout << ch1 << endl ;

23

for ( int i = 0 ; i <= 8 ; i++) // imprime l e c a r a c t e r e nul

24 cout << ch3 [ i ] ;

25 cout << "

??? " << endl ;

26

for ( int i = 0 ; i < 7 ; i++)

27 ch1 [ i ] = toupper ( ch1 [ i ] ) ;

28 cout << ch1 << endl ;

29 cout << s t r l e n ( ch1 ) << ' \ t ' << s t r l e n ( ch2 )

30 << ' \ t ' << s t r l e n ( ch3 ) << ' \ t ' << s t r l e n ( ch4 ) << endl ;

31 ch1 [ 1 ] = ' i ' ;

32

for ( int i = 0 ; i < 7 ; i++)

33 ch1 [ i ] = tolowe r ( ch1 [ i ] ) ;

34 cout << ch1 << endl ;

35 cout << ch5 << endl ;

36
```

```
// ch5 [ 0 ] = 'T ' ; mo d i f i c a t i on d ' une chaine cons tant e
37
// ===> p l ant e Dev ¡ C++
38
// cout << ch5 << endl ;
39 system( " pause " ) ;
40 }
```

Ligne 3, j'appelle une nouvelle bibliothèque héritée du C, qui contient les fonctions

toupper()

(conversion en majuscules) et

tolower() (conversion en minuscules). La ligne 7 contient deux

définitions de chaines, comme tableaux. L'une est initialisée au moment de la déclaration (c'est le compilateur qui compte le nombre d'éléments), l'autre est lue au clavier (la chaine se termine au premier caractère blanc ou à la ligne). Dans ces deux cas, le compilateur introduit lui-même le caractère final

'\0' .

Je définis ensuite (ligne 8) une autre chaine, élément par élément : il m'incombe de placer le caractère nul final.

ch4[10] est un tableau de caractères de 10 éléments, numérotés de 0 à 9. Ils

sont initialisés un par un lignes 12-14. Ici, je définis des caractères par leur code ASCII : ce sont en fait les entiers de 0 à 8. C'est toujours à moi d'insérer

'\0' .

Enfin,

ch5 est définie par l'intermédiaire d'un pointeur, ce qui sera expliqué bientôt. Attention : beaucoup de compilateurs considèrent qu'une chaine ainsi définie est une constante inviolable et la range dans une zone spéciale de la mémoire. Toute tentative de modification (comme je le fais ligne 36) provoque une erreur d'exécution.

J'affiche ensuite ces chaines, suivies d'étoiles, pour que l'on voit bien où elles s'arrètent. Je me livre après à quelques manipulations élémentaires : modification d'un caractère (ligne 19), incrémentation d'un caractère (ligne 21) (incrémentation de son code ASCII en fait, donc passage au caractère suivant). Ligne 22, je frole la catastrophe, en dépassant de un la longueur de la chaine.

On montre ensuite l'effet des deux fonctions

tolower et toupper , puis l'usage de strlen() , qui

renvoie la longueur de la chaine passée en argument, SANS COMPTER LE CARACTÈRE NUL.

```
entrer un mot
programme
Licence***
programme***
maitrise***
012345678***
Lycence
Lzcence
maitrise ***
LZCENCE
7 9 8 9
licence
truc1
```

C++ comprend (dans la bibliothèque

<cstring>) un grand nombre d'autres fonctions de manipulation

de chaines, qui généralement font intervenir des pointeurs et seront abordées plus tard.

6.7 enregistrement

Un enregistrement est une structure de données qui permet de rassembler des variables de types différents mais ayant un point commun, comme les renseignements figurant sur une carte d'identité

ou une fiche de bibliothèque. En Pascal, on parle de

RECORD , en C/C++ il s'agit de struct .

6.7.1 définition

Avant d'utiliser une de ces structutres, il faut la définir : il s'agit d'un type nouveau, sorti

de l'imagination du programmeur, mais tout aussi valable qu'un

float ou un char . Voici une

définition d'une «struct» destinée à représenter l'heure

```
struct HMS {
int heure;
int minute;
int seconde;
};
```

Un enregistrement de type

HMS contient trois «champs» (ou trois «membres») de type entier ; le

nombre et la nature des champs ne sont pas limités, sauf qu'une

struct ne peut pas contenir une

struct

de même nature. Notez le point-virgule final de la déclaration.

Ayant défini un type, je peux maintenant déclarer des variables (des objets ou des identificateurs)

de ce type, comme ceci :

```
HMS epoque, lediner, tabHMS[10];
```

qui réserve de la place en mémoire pour 12 enregistrements de type

HMS , dont 10 sont groupés dans

un tableau.

6.7.2 accès aux champs

Pour initialiser les champs des

struct que l'on vient de définir, on utilise le nom de l'enregistrement

suivi d'un point et du nom du champ ; on fait de même pour lire, écrire ou calculer avec le

contenu d'un champ. L'initialisation peut aussi se faire comme pour un tableau, avec des valeurs

entre accolades.

lediner.heure = 19; lediner.minute = 30;

epoque = {6,25,45};

epoque.heure = lediner.heure - 12;

cin >> epoque.minute;

cout << "heure du diner: " << lediner.heure <<" heure "

<< lediner.minute << endl;

6.7.3 exemple

Dans le programme qui suit, je mets en oeuvre quelques opérations d'arithmétique en nombres

complexes ; chaque nombre est représenté par une

struct , avec deux champs, une partie réelle et

une partie complexe. Chaque fonction a le type

COMPLEXE et renvoie donc DEUX valeurs vers le

programme appelant.

```
1
#include <ios t ream>
2
#include <c s t d l i b >
3
struct COMPLEXE{
4
double r e ;
5
double im;
6 } ;
```

```
7 COMPLEXE add (COMPLEXE a , COMPLEXE b){
8 COMPLEXE somme ;
9 somme . r e = a . r e + b . r e ;
10 somme . im = a . im + b . im;
11
return somme ;
12 }
13 COMPLEXE mult (COMPLEXE a , COMPLEXE b){
14 COMPLEXE prod ;
15 prod . r e = a . r e
? b . r e ¡ a . im ? b . im;
16 prod . im = a . r e
? b . im + a . im ? b . r e ;
17
return prod ;
18 }
19
void modif (COMPLEXE a ){
20 a . r e
? = 1 0 ; a . im ? = 1 0 ;
21 }
22
int main ( ) {
23 COMPLEXE u = {1 ,0} , v = {0 ,1} , s , p ;
24 s = add (u , v ) ;
25 cout << s . r e << ' \ t ' << s . im << endl ;
26 p = mult (u , v ) ;
27 cout << p . r e << ' \ t ' << p . im << endl ;
28 modif (u ) ;
29 cout << u . r e << ' \ t ' << u . im << endl ;
30 system( " pause " ) ;
```

31

return 0 ;

32 }

Ce programme affiche

1 1

Intro_C

53

0 1

1 0

Vous remarquez que la fonction

\modif n'a pas eu d'effet sur la variable u : les arguments de type

struct

, contrairement aux tableaux, sont transmis pas valeur.

Chapitre 7

Les Pointeurs

Le pointeur est un type défini en C/C++, tout comme en Pascal (et en Fortran depuis 1990). C'est une caractéristique à la fois puissante et difficile du langage. Un pointeur permet de simuler l'appel par référence, de créer et de manipuler des structures dynamiques et est largement utilisé dans la programmation par objet. Ici, je ne fais qu'introduire le sujet et montrer les liens qui existent entre pointeur, tableau et chaîne de caractères. Une variable ordinaire contient une valeur ; un pointeur, au contraire, contient l'adresse d'une autre variable qui, elle, contient une valeur. On dit que la variable habituelle est une référence directe à une valeur alors que le pointeur est une référence indirecte.

7.1 Déclaration et initialisation

7.1.1 déclaration et notation

Comme toute autre variable, un pointeur doit être déclaré avant usage.

int a, *Pb, c = 3;

déclare un entier

a , un pointeur vers un entier Pb et un entier c initialisé à 3. Cette déclaration

se lit comme «a est un entier, Pb est un pointeur vers un entier, c est un entier de valeur 3» (la

parti pointeur se lit «à l'envers»). L'opérateur * n'est pas distributif, il faut l'écrire pour chaque

variable de type pointeur :

double x, *Py, *Pz;

déclare deux pointeurs (

Py, Pz) vers des nombres en double précision. Pour limiter les risques

d'erreur, il est commode de donner aux pointeurs des noms aisément reconnaissables, comportant

par exemple les caractères

p ou ptr . On peut dire qu'il existe deux opérateurs homonymes :

l'opérateur de multiplication et l'opérateur d'indirection, représentés tous deux par une étoile.

7.1.2 Opérateur adresse et initialisation d'un pointeur

L'opérateur «adresse» & en C++, renvoie l'adresse de son unique opérande. Le fragment de

code suivant affecte à un pointeur l'adresse d'une variable :

int x = 5;

int *xP;

xP = &x;

J'ai déclaré et initialisé un entier

x , puis un pointeur (adresse) vers un entier xP et j'ai affecté à

xP

une valeur, l'adresse de x . On dit que xP «pointe» vers x . Si on pouvait lire le contenu de la

mémoire au moment de l'exécution de ce code, on verrait quelque chose comme

adresse contenu nom de la variable

320000 530000

xP

530000 5

x

Attention :

ces adresses, imaginaires et données à titre d'exemple, ne sont, de toute façon, pas

immuables : elles vont varier d'une machine à l'autre, d'une exécution à l'autre. Et c'est très bien

ainsi ! Vous verrez que la valeur absolue d'une adresse n'a pas d'intérêt pour le programmeur.

Attention :

déclarer un pointeur n'est en rien équivalent à réserver en mémoire de la place pour

l'objet vers lequel il pointe (pensez à un hôtelier qui vous indique un numéro de chambre dans une

aile de bâtiment qui sera construite dans deux ans).

7.1.3 l'opérateur «*»

Je répète ce que j'ai dit au paragraphe précédent, de façon plus abstraite (ou prétentieuse).

L'opérateur

* (opérateur d'indirection ou de «déréférencement») appliqué à un argument de type

pointeur, renvoie un synonyme de la variable désignée par ce pointeur.

cout << *xP << endl;

affiche la valeur de

x (si l'association entre x et xP est comme déclarée plus haut), pratiquement

comme le fait

cout << x << endl; . J'ai «déréférencé» le pointeur xP . Attention : déréférencer un

pointeur qui ne pointe sur rien provoque le plantage du programme.

On peut donc lire

*xP comme «contenu de l'adresse désignée par xP » et cette interprétation

est confirmée par l'utilisation de l'opérateur que je fais maintenant :

*xP = -100;

cout << *xP;

J'affecte à

x (la variable pointée par xP) la valeur -100 et je l'imprime.

Quand on les applique à une variable de type pointeur, les opérateurs * et & sont inverses l'un

de l'autre et commutent entre eux, comme je le montre ci-dessous,

```
1
// point eur1 . cpp : p r o p r i é t é s de ? e t &
2
#include <ios t ream>
3
#include <c s t d l i b >
4
using namespace s td ;
5
int main ( void ){
6
int a , ? aP;
7 a = 7 ;
8 aP = &a ;
9 cout << " adr e s s e de a : " << &a ;
10 cout << "\ nval eur de aP: " << aP;
11 cout << "\ nval eur de a : " << a ;
12 cout << "\ nval eur de
? aP: " << ? aP;
13
? aP = ¡ 36;
14 cout << "\ nval eur de a : " << a ;
15 cout << "\ nval eur de
? aP: " << ? aP;
```

```
16 cout << "\n&
? aP: " << & ? aP;
17 cout << "\n
? &aP: " << ? &aP;
18 system( " pause " ) ;
19
return 0 ;
20 }
```


avec les résultats

```
adresse de a: 0x22ff7c
valeur de aP: 0x22ff7c
valeur de a: 7
valeur de *aP: 7
valeur de a: -36
valeur de *aP: -36
&*aP: 0x22ff7c
*&aP: 0x22ff7c
```

7.2 Passage d'argument par adresse

Nous avons rencontré quatre façons de faire circuler de l'information entre une fonction «appelée » et un programme principal «appelant» : déclarer des variables communes au niveau global, renvoyer une valeur unique par l'instruction return , « partager » un tableau avec le programme et enfin, mettre à disposition de la fonction des variables de main sous forme de référence. Une autre méthode, plus générale, de transmission par adresse (référence) fait appel à un pointeur (méthode commune à C et C++). L'exemple suivant met en oeuvre les trois dernières méthodes

de transmission d'une valeur.

```
1

#include <ios t ream>

2

#include <c s t d l i b >

3

int cubeVal ( int x ){

4

return x ? x ? x ;

5 }

6

void cubeRef ( int & x ){

7 x = x

? x ? x ;

8 }

9

void cubePtr ( int ? xP){

10

? xP = ? xP ? ? xP ? ? xP;

11 }

12

using namespace s td ;

13

int main ( void ){

14

int nb = ¡ 5;

15 cout << "nb au début : " << nb << endl ;

16 nb = cubeVal (nb ) ;

17 cout << "nb apr è s cubeVal : " << nb << endl ;

18 nb =

¡ 5;
```

```
19 cubeRef (nb ) ;
20 cout << "nb apr è s cubeRef : " << nb << endl ;
21 nb =
¡ 5;
22 cubePtr (&nb ) ;
23 cout << "nb apr è s cubePtr : " << nb << endl ;
24 system( " pause " ) ;
25
return 0 ;
26 }
```

La seule nouveauté est le passage par un pointeur (lignes 9-10, 22). Le calcul du cube(ligne 10) est délibérément obscur ; il fonctionne parce que l'opérateur de déréférencement (*) a une priorité beaucoup plus élevée que l'opérateur de multiplication (*) : il est donc évalué en premier. En pratique, on mettrait assez de parenthèses pour lever toute ambiguïté.

7.3 Pointeur constant et pointeur sur une constante

Une variable qualifiée de

const ne peut pas être modifiée par le programme : cela constitue

une bonne méthode pour sécuriser le programme. Ce qualificatif peut également s'appliquer à un pointeur et aussi à une variable désignée par un pointeur.

On dispose en fait de quatre possibilités : pointeur vers une variable, pointeur vers une constante, pointeur constant vers une variable, pointeur constant vers une constante. La première n'apporte aucune restriction, les autres sont plus contraignantes. Les déclarations pourraient être les suivantes.

```
char * cP; //pointeur vers un caractère
const int *Pi // Pi est un pointeur sur un entier constant
float * const Px // Px est un pointeur constant
// vers un nombre fractionnaire
```

const double * const Ptruc // pointeur constant vers un

// réel double précision constant

Il appartient au programmeur de faire le meilleur usage de ces possibilités.

7.4 Arithmétique sur les pointeurs

On ne peut faire sur les pointeurs que quelques opérations arithmétiques (ce qui est assez évident

si l'on se souvient qu'il s'agit d'adresses) : incrémentation (

++), décrémentation (--), addition

(

+,+=) ou soustraction (-, -=) d'un entier. L'intérêt de telles opérations apparaîtra au paragraphe

suivant. Il ne s'agit pas d'une addition (soustraction) simple. En effet, supposons que

aP soit un

pointeur d'entier de valeur 40096, où 40096 désigne l'emplacement d'un OCTET en mémoire.

aP+1

désigne l'emplacement suivant pour un entier, soit 40100 parce que, en général, un entier occupe

4 octets. Ces considérations n'ont de sens que si

*aP et *(aP+1) sont tous les deux des entiers.

7.5 Pointeurs et tableaux

En C/C++, tableaux et pointeurs sont des entités très proches. En fait,

le nom d'un tableau

est un pointeur constant vers le premier élément du tableau (indice 0)

; autrement dit, le nom du

tableau est l'adresse (constante) de son premier élément.

Soient les déclarations

int b[5]; int * bP; . Le nom du tableau est l'adresse (un pointeur

vers) le premier élément. Je peut donc écrire

bP = b;

ce qui est équivalent à

bP = &b[0];

L'élément d'indice 3 peut maintenant être désigné comme

*(bP+3);

L'entier 3 est un «décalage» («offset») pour le pointeur. Les parenthèses sont rendues nécessaires

par la priorité de l'opérateur * ; l'expression

*bP + 3 ajoute 3 à la valeur désignée par bP (et donc

calcule

b[0] + 3).

L'adresse de l'élément d'indice 3 peut s'écrire

&b[3] ou bP + 3 . Le nom du tableau étant un

pointeur, on peut écrire, symétriquement,

*(b+3) pour désigner ce même élément.

Enfin, un pointeur peut être muni d'un indice, comme

pB[1] qui désigne le deuxième élément du

tableau, tout comme

b[1] . Vous voyez pourquoi, dans le jargon du C/C++, on parle «d'opérateur

crochet» : le brave pointeur

b , affublé de crochets, est transformé aussitôt en tableau.

Attention :

une écriture comme b += 2; est impossible, puisqu'elle tend à modifier un pointeur

constant.

En général, un programme qui manipule des tableaux est plus lisible lorsqu'il utilise des indices

plutôt que des pointeurs.

Le programme qui suit illustre les différentes méthodes d'accès aux éléments d'un tableau.

```
1

// tab_pt r . cpp : d i f f é r e n t e s méthodes pour accéde r
```

```
2
// aux é l ément s d 'un t a b l e a u
3
#include <ios t ream>
4
#include <c s t d l i b >
5
using namespace s td ;
6
int main ( void ){
7
int a [ ] = { ¡ 40 , ¡ 30 , ¡ 20 , ¡ 10 ,0}; //a e s t un t a b l e a u d ' e n t i e r s
8
int ? Pa = a ; //Pa e s t un p o int eur d ' ent i e r , qui d é s i gne a
9
int i , de c l g ;
10 cout << "\navec un i n d i c e " << endl ;
11
for ( i = 0 ; i < 5 ; i++)
12 cout << "a [ " << i << " ] = "<< a [ i ] << ' \ t ' ;
13 cout << "\navec un po int eur (nom du tabl eau ) e t un dé c a l a g e \n" ;
14
for ( de c l g = 0 ; de c l g < 5 ; de c l g++)
15 cout << "
? ( a+" << de c l g << " ) = "<< ? ( a+de c l g ) << ' \ t ' ;
16 cout << "\navec un po int eur e t un i n d i c e \n" ;
17
for ( i = 0 ; i < 5 ; i++)
18 cout << "Pa [ " << i << " ] = "<< Pa [ i ] << ' \ t ' ;
19 cout << "\navec un po int eur e t un dé c a l a g e \n" ;
20
```

```
for ( de c l g = 0 ; de c l g < 5 ; de c l g++)
21 cout << "
? (Pa+" << de c l g << " ) = "<< ? (Pa+de c l g ) << ' \ t ' ;
22 cout << endl ;
23 system( " pause " ) ;
24
return 0 ;
25 }
```

7.6 Pointeurs et chaînes

Une chaîne de caractères est un tableau : son nom est donc un pointeur constant vers l'adresse du premier caractère. Les deux représentations sont en gros équivalentes, sauf qu'une chaîne désignée par un pointeur est souvent considérée par le compilateur comme une constante non modifiable.

Rappel de déclarations possibles :

```
char vin[] = "bordeaux";
char *Pvin = "bourgogne";
char vin[] = {'c','h','i','n','o','n','\0'};
```

Je cite maintenant un certain nombre de fonctions de la bibliothèque

cstring> permettant la

manipulation de chaînes ; je donne d'abord le prototype, puis une description.

```
char * strcpy(char *s1, const char *s2)
```

copie la chaîne s2 dans le tableau s1 et renvoie la

valeur de

s1 .

```
char * strncpy(char *s1, const char *s2, int n)
```

copie au plus n caractères de s2 dans s1 et

renvoie

s1 .

```
chr * strcat(char *s1, const char *s2)
```

ajoute s2 à la fin de s1 ; le premier caractère de s2

écrase le 0 terminal de

s1 et renvoie la valeur de s1 .

chr * strncat(char *s1, const char *s2, int n)

ajoute au plus n caractères de s2 à la fin de

s1

; le premier caractère de s2 écrase le 0 terminal de s1 et renvoie la valeur de s1 .

int strcomp(char *s1, const char *s2)

compare s1 à s2 et renvoie une valeur nulle, négative

ou positive selon que

s1 est égale, plus petite ou plus grande que s2 .

int strncomp(char *s1, char *s2, int n)

compare au plus n caractères de s1 à s2 et renvoie

une valeur nulle, négative ou positive selon que

s1 est égale, plus petite ou plus grande que

s2

.

int strlen(const char *s)

renvoie le nombre de caractères non-nuls de l'argument.

Attention aux longueurs de chaînes lorsque vous utilisez

strcpy ; cette fonction copie le second

argument (avec son zéro) dans le premier, qui doit être assez grand pour recevoir le tout. Dans le

cas de

strncpy , il faut encore s'assurer que n est assez grand pour que le zéro terminal de s2 soit

recopié.

7.7 Et les «strings» ?

Soit la déclaration/initialisation

string ch1 = "que voici une belle petite chaine";

La norme C++ impose les propriétés de

ch1 mais ne dit rien sur la façon de les réaliser «à

l'intérieur» du compilateur. Une string n'est pas un vecteur, ne se termine pas par le caractère

nul. Son nom n'est pas un pointeur, et l'écriture

&ch1[0] n'a pas de signification. Il reste vrai que

ch1[10]

est le caractère «u».

Toutes les fonctions citées au paragraphe précédent ont leur équivalent pour les «strings».

Si je déclare

ch2 = "bonjour" , je peux comparer deux «strings» ; la condition ch1 > ch2 est

vraie (ordre lexicographique plus élevé) et

ch1 == ch2 est fausse.

cout << ch1.substr(10,3)

imprime une (la sous-chaîne qui commence au caractère de rang

10 et comporte 3 lettres).

J'emploie

ch1.find("petite") pour découvrir à quel rang commence la sous-chaine «petite».

Enfin, il est facile d'insérer des caractères dans une «string» ; l'instruction

ch1.insert(9,"voila ")

injectera «voila » après voici (après le caractère blanc de rang 9).

7.8 pointeurs et «struct»

Si j'ai défini un type d'enregistrement et si j'ai déclaré des variables de ce type, je peux déclarer

des pointeurs vers ces variables. De plus, un (ou plusieurs) des champs de l'enregistrement peut

être lui-meme un pointeur vers un objet quelconque (en particulier un objet de même type). Ces

remarques rendent possibles la contruction de structures de données extrèmement riches et variées (listes, piles, queues, . . .) qui ne seront pas décrites ici. Je me contente de préciser quelques points de syntaxe. Examinons les déclarations suivantes.

```
struct cpx {double re; double im;};
cpx u,v, *Pcpx, &Refcpx = u, Tcpx[123];
```

J'ai défini une
struct qui mime un nombre complexe, u et v sont deux exemplaires de cette
structure,
Pcpx est un pointeur vers un complexe, Refcpx est un pseudonyme de u (défini par
référence) et
Tcpx est un tableau de «nombres» complexes.
Pour le moment, le pointeur ne pointe sur rien. Je l'initialise et j'imprime ses deux champs
comme ceci par exemple

```
Pcpx = &v;
cout << (*Pcpx).re << '\t' << Pcpx->im << endl;
```


Les deux manières d'accéder aux champs de l'enregistrement pointé par
Pcpx sont équivalentes.
Les parenthèses de la première sont obligatoires parce que l'opérateur «point» a une priorité plus élevée que l'opérateur de déréférencement ; la seconde est peut-être plus claire et plus concise : elle a la faveur des spécialistes.