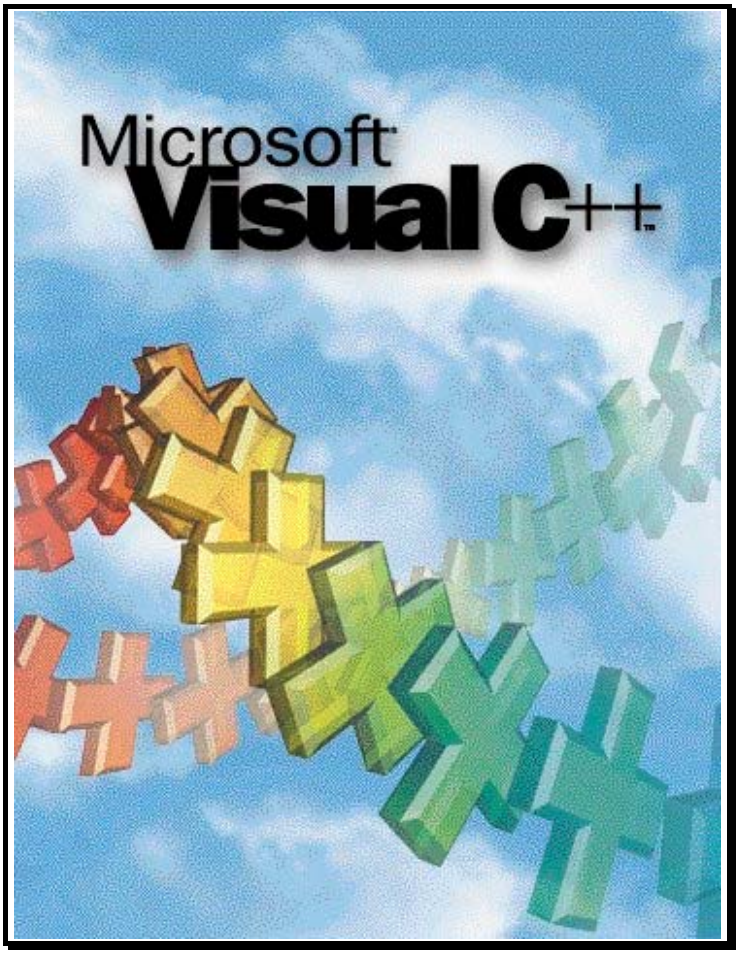
Section de Génie Optique option Photonique
5, rue Jean Mermoz
68300 SAINT-LOUIS
☎ : 03.89.70.22.70
✉ : bernard.kempf@wanadoo.fr
http://www.lyceemermoz.com, ou http://perso.orange.fr/bernard.kempf/photonique/



Nom de l'étudiant : _____

A. LES FONDEMENTS DE LA PROGRAMMATION

1. Documentation en ligne du logiciel Visual C++

Ce document ne pourra donner qu'un aperçu du langage C++ et de la programmation sous Visual C++. On pourra aussi consulter la documentation en ligne (Book Online). Celle-ci fait partie intégrante de l'environnement de développement. Elle couvre la totalité de l'environnement du langage (MFC, langage C++, etc…) et donne accès à de nombreuses applications exemples. Dès le lancement le lancement de l'application, l'écran est divisé en deux volets qui donnent accès à la documentation en ligne (voir *cadre 2*).

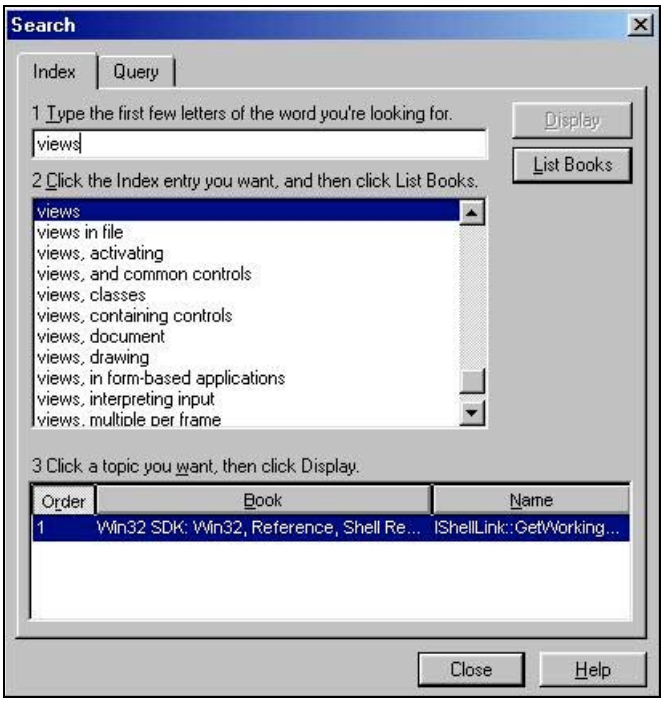
On peut aussi rechercher une rubrique par mot clé (voir *cadre 1*) ou appuyer sur F1 sur une fonction (voir *cadre 3*, fonction sinus).

2. Avant de programmer : Faire l'analyse du problème

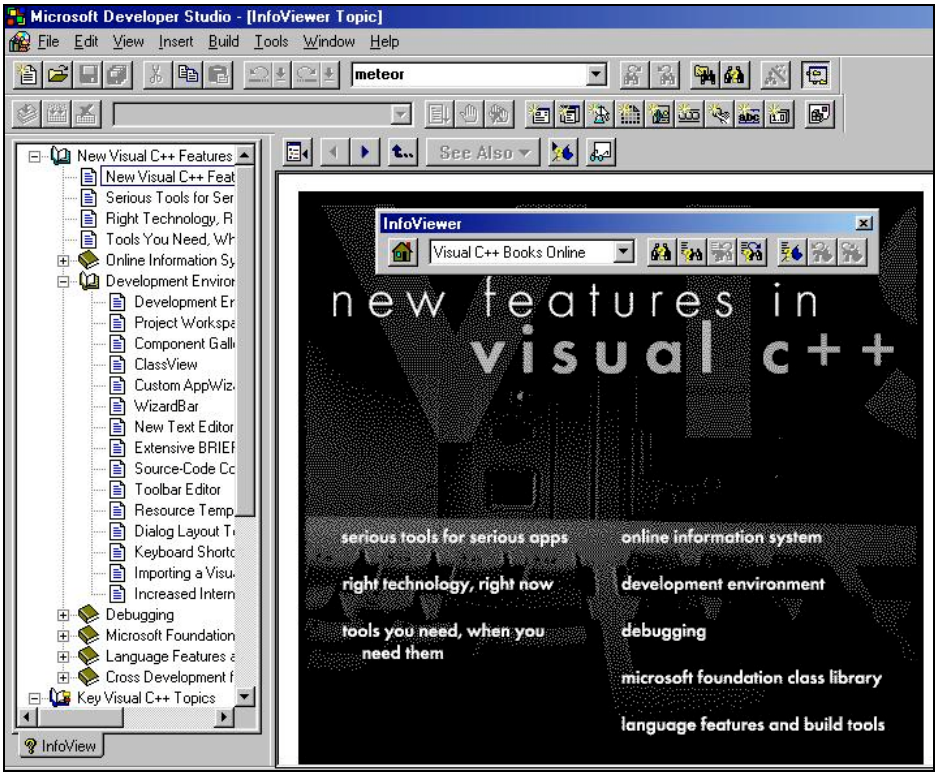
Il faut tout d'abord souligner qu'un programme est réalisé dans le but d'effectuer un ensemble de traitements. La première tâche est donc toujours de spécifier précisément la nature des éléments à traiter ainsi que des opérations à leur faire subir. Il faudra spécifier :

- les paramètres d'entrée du programme. Exemple : un fichier de données en provenance du disque, une tension lue sur une carte analogique/numérique, une image digitalisée à la caméra, un spectre oscilloscope dont on fait l'acquisition par liaison série, …
- les paramètres de sortie du programme. Exemple : un graphe imprimé qui explique les résultats d'un T.P., un fichier texte qui sera repris par *Excel*, des impulsions qui vont déplacer un moteur électrique, une communication série avec un modem…
- la façon dont on souhaite passer des paramètres d'entrée vers les paramètres de sortie. Exemple : L'acquisition de la tension devra se faire toutes les 50 ms avec une moyenne de 256 valeurs par acquisition afin d'atténuer le bruit. Le graphe U = f(t) est à tracer en fin d'acquisition.

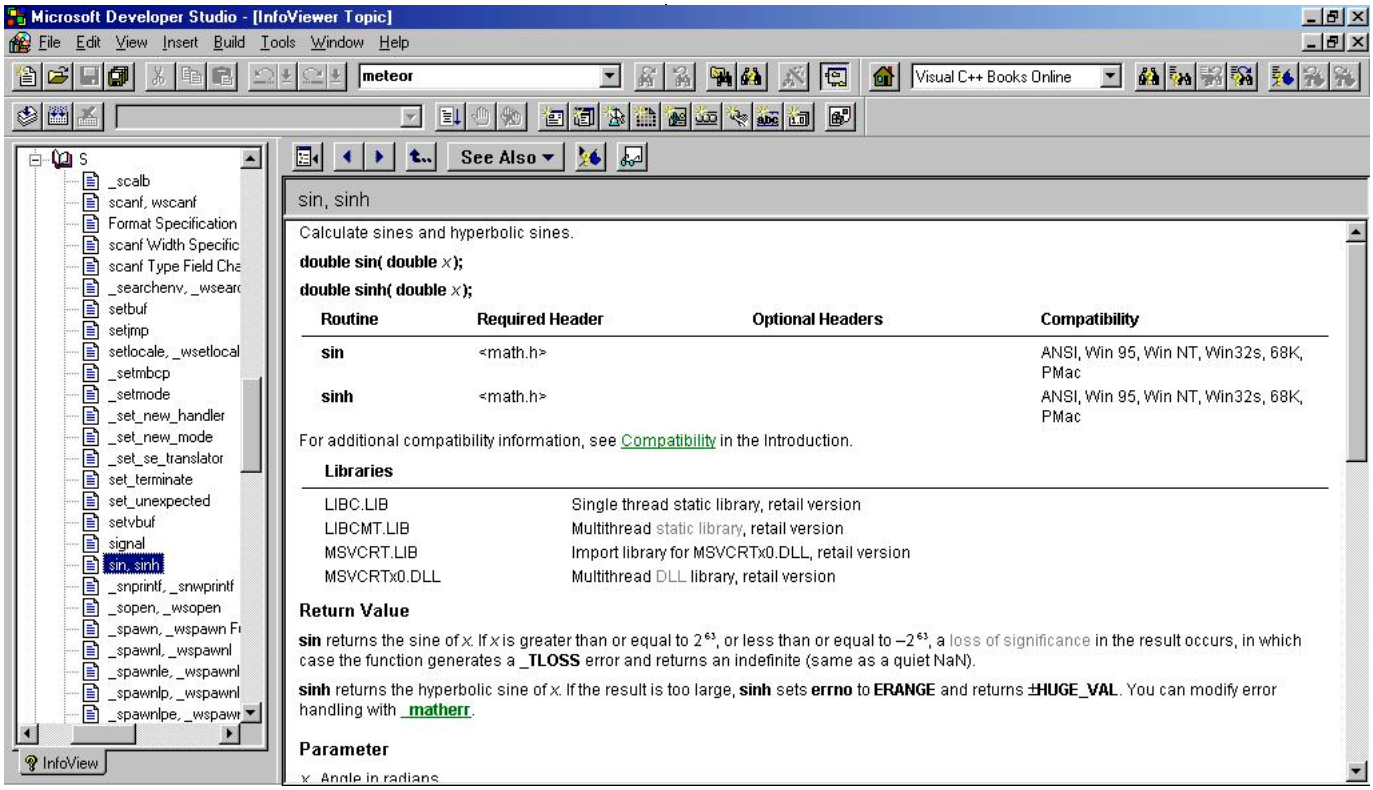
Il n'existe malheureusement pas de méthode infaillible qui permet de passer du problème posé au résultat. L'apprentissage des techniques de programmation est un travail de longue haleine. Le langage (ici *Visual C++*) n'est qu'un outil qui permet d'écrire ce qu'on a envie de faire. La structure du programme est indépendante de tout langage.



cadre 1 : Recherche d'une rubrique par mot clé.



cadre 2 : Documentation en ligne et barre d'outil InfoViewer.



cadre 3 : Appui sur F1.

B. LES BASES

Ce deuxième chapitre présente les bases du développement d'applications à l'aide du langage *Visual C++* (version 4), en particulier : types et variables, fonctions.

1. Les données du programme

Les données d'un programme se caractérisent habituellement suivant deux critères :

- leur type ;
- leur possibilité d'être modifiées ou non.

1.1. Notion de variable

Les données représentent toutes les informations manipulées par un programme. Pour pouvoir être manipulées et connues du programme,, il faut les identifier à l'aide d'un nom (ou encore identificateur). Suivant leur emploi, deux catégories de données peuvent être distinguées :

- les données constantes ;
- les données variables.

Les données constantes sont connues dès le début des traitements et représentent une valeur (ou information) qui n'est pas modifiée du début à la fin de l'exécution du programme.

Les données variables représentent des informations dont les valeurs sont élaborées par le programme. De plus, pour pouvoir avoir une existence effective, il faut qu'elles puissent contenir l'information, c-à-d qu'il faut leur associer une zone mémoire où elles seront stockées. Il faut garder à l'esprit qu'une variable est toujours au moins associée à un nom et à une place en mémoire.

Exemple : calcul de la moyenne de 4 nombres.

```
...
Note1=10;
Note2=8.5;
Note3=12.5;
Note4=14;
Res=(Note1+Note2+Note3+Note4)/4;
…
```

Nous disposons de 5 variables (identificateurs : *Note1*, *Note2*, *Note3, Note4*, *Res*). La valeur 4 qui est nécessaire au calcul de la moyenne est une donnée constante du problème.

Le symbole '=' est le symbole de l'affectation, '/' le symbole de la division. Voir *cadre 5*. J'insiste sur le terme affectation, et non pas égalité au sens mathématique. On pourra donc écrire :

```
...
Note1=20;
Note1=Note1-5;
//Note 1 est affecté avec 20-5=15
```

Remarque : Le langage *Visual C++* fait la différence entre les majuscules et les minuscules dans les identificateurs.

Exemple : *note1* et *Note1* sont deux identificateurs différents !

1.2. Notion de type

Cette notion permet d'expliciter le fait que les données n'ont pas toutes les mêmes propriétés : nombres réels, entiers, caractères, …

1.2.1. Types usuels

Voir *cadre 4*.

1.2.2. Déclaration

Les déclarations suivantes sont valides :

```
...
float Note1;
float Note2;
float Note3;
float Note4;
float Res;

…
```

ou encore :

```
...
float Note1, Note2, Note3, Note4, Res;
…
```

Toute variable doit être déclarée avant son utilisation :

```
...
float Note1;
Note1=10;
…
```

La variable *Note1* est déclarée de type *float*, puis on lui affecte la valeur 10. Dans ce cas, on peut écrire aussi :

```
...
float Note1=10;
…
```

Entiers	Taille (bits)	Intervalle	Application
unsigned char	8	0 à 255	Petits nombres, jeu de caractères complet du PC
char	8	- 128 à 127	Très petits nombres, carctères ASCII
unsigned short	16	0 à 65 535	Décomptes, petits nombres, boucles de contrôle
short	16	- 32 768 à 32 768	Décomptes, petits nombres, boucles de contrôle
unsigned int	32	0 à 4 294 967 295	Très grands nombres entiers
unsigned long int long	32	- 2 147 483 648 à 2 147 483 647	Très grands nombres entiers
Booléens			Sont traitées en 3.4.
BOOL	8	TRUE (vrai) ou FALSE (faux)	Données binaires
Chaînes de caractères			Sont traitées en 3.4.
char [] char * CString			
Réels			
float	32	$\pm$ 3.4 x $10^{\pm 38}$	Scientifique (précision sur 7 chiffres)
double	64	$\pm$ 1.7 x $10^{\pm 308}$	Scientifique (précision sur 15 chiffres)
Pointeurs			
*	32	Sans objet	Manipulations d'adresses mémoire Sont traités au chapitre C.

cadre 4 : Types de données standards.

1.3. Opérateurs

Type	Rôle	Priorité	Exemple
()	Appel de fonction	1	float y,M_PI=4*atan**(1)** ; y=sin**(**M_PI/6**)**;
[ ]	Indice du tableau	1	t**[i]**=4;
.	Opérateur de structure ou de classe Sélectionne un membre de la structure ou de la classe	1	CDialog d; d.DoModal() ;

→	Accès à un membre de structure ou de classe par pointeur	1	void CBkView::OnDraw(CDC* pDC)
			{
			CBkDoc* pDoc = GetDocument();
			pDC→TextOut(20,50,"Bonjour: "+ pDoc→m_sMon Nom);
			}

!	Non logique	2	BOOL OK ;
			if (**!**OK)
			{
			
			}

~	Complément à 1 bit à bit	2	unsigned char i=254;
			i=256+**~**i ;
			//Résultat: 1. Rq: 256: tient compte du bit signe

*	Adressage indirect (manipulation des pointeurs) Retourne la valeur à l'adresse *ptr_i*.	2	int i=12; int *ptr_i=**&**i; //Un pointeur sur un int, initialisé à l'adresse de la variable i.
&	Adresse Retourne l'adr. de i (et non pas la valeur i).	2	Idem ci-dessus

&	Passage de valeurs par référence
	La valeur de la var. r de l'ex. ci-contre peut être modifié par la fonc. grâce au symbole &.
	Rq: Cette écriture est spécifique C++.
	Un tableau n'est jamais passé par valeur, mais toujours par référence, implicitement.
	Ex: void val_abs_tab(float t[20]);// Pas de &
		2	//Soit la fonction:
			void val_absolue(float **&**r) //r est modifié
			{
			if (r<0) r=-r;;
			}
			//Utilisation:
			float r=-3.21458;
			val_absolue(r); //Résultat: 3.21458

++	Incrémentation Identique à i=i+1	2	int i=3; i**++**; //Résultat: 4.
--	Décrémentation Identique à i=i-1	2	int i=3; i**--**; //Résultat: 2.

(type)	Conversion de type	2	float r=2.3546;
			int i;
			i=**(int)**r+3; //Résultat: 5; conversion de float → int

*	Multiplication	3	i=4*****3; //Résultat: 12
/	Division	3	i=8/2; //Résultat: 4
%	Reste de la division (modulo)	3	int i=9, j=5,k; k=i**%**j; //Résultat: 4
+	Addition	4	i=4**+**3; //Résultat: 7

-	Soustraction	4	i=4**-**3; //Résultat: 1
			i=-i; //Changement de signe, résultat: -1

<<	Décalage gauche d'un certain nombre de bits	5	int i,j;
			i=2; //0 0 0 0 0 0 1 0 en binaire
			j=i**<<**1; //Résultat: 4; décal. de 1 bit vers gauche

>>	Décalage droite d'un certain nombre de bits	5	int i,j;
			i=4; //0 0 0 0 0 1 0 0 en binaire
			j=i**>>**1; //Résultat: 2; décal. de 1 bit vers droite

<	Strictement inférieur	6	if (i**<**j)
			{
			
			}

>	Strictement supérieur	6	if (i**>**j)
			{
			
			}

<=	Inférieur ou égal	6	if (i**<=**j)
			{
			
			}

>=	Supérieur ou égal	6	if (i**>=**j)
			{
			
			}

==	Egal
	Rq: l'égalité entre des nb. réels (float, ...) n'existera jamais (précision 7 chiffres. Ex: 1.000001 et 1.000000)
		6	if (i**==**j)
			{
			
			}

!=	Différent	6	if (i**!=**j)
			{
			
			}

&	ET bit à bit	7	Int i=10; //0 0 0 0 1 0 1 0 en binaire
			int j=12; //0 0 0 0 1 1 0 0 en binaire
			int k;
			k=i**&**j; //Résultat: 8 → 0 0 0 0 1 0 0 0

^	XOR, OU exclusif bit à bit	8	k=i**^**j; //Résultat: 6 → 0 0 0 0 0 1 1 0
|	OU inclusif bit à bit	9	k=i**|**j; //Résultat: 14 → 0 0 0 0 1 1 1 0

&&	ET logique	10	if (i>0 **&&** j>0)
			{
			
			}

||	OU logique	11	if (i>0 **||** j>0)
			{
			
			}

=
*=
/=
+=
-=
...
	Affectation	13	int x**=**4, y;
			y**=**3*x+2; //Résultat: y=14
			x***=**2; //Résultat: 8 - Identique à x=x*2;
			x**/=**4; //Résultat: 1 - Identique à x=x/4;
			x**+=**1; //Résultat: 5 - Identique à x=x+1;
			x**-=**4; //Résultat: 0 - Identique à x=x-4;

,	Enchaînement d'instructions Séparateur d'arguments d'une fonction	14	float x **,** y **,** z; z=pow(x **,** y); //Fonction puissance

cadre 5 : Opérateurs.

1.4. Variables booléennes

Elles sont utilisées pour mémoriser les données binaires. Elles ne peuvent prendre que la valeur TRUE (vrai) ou FALSE (faux).

Exemple :

```
BOOL OK;
OK=TRUE;
…
if (OK==TRUE)
{
 …
}
else
{
 …
}
//On peut écrire aussi:
if (OK) // ou éventuellement if (!OK)
{
 …
}
else
{
 …
}
```

1.5. Variables caractères

1.5.1. Particularité des variables de type *char* et *unsigned char*

D'après tableau *cadre 4*, les variables caractères contiennent des petits nombres entiers, mais aussi le jeu de caractères complet du PC.

On peut donc écrire :

```
char c;
c = 65;
c = 'A';
c = 0x41;
```

Les trois affectations ci-dessus sont équivalentes. La variable *c* contient la même donnée, c-à-d le caractère A, le nombre 65 en base 10 ou le nombre 41 en base 16 (symbole 0x).

1.5.2. Jeux de caractères

Jeux de caractères (entre 0 et 255)

Dec	Hex	Char	Code	Remarque
0	**00**	**€**	**NUL**	**Fin de chaîne**
1	01	€	SOH	
2	02	€	STX	
3	03	€	ETX	
4	04	€	EOT	
5	05	€	ENQ	
6	06	€	ACK	
7	07	€	BEL	
8	08	€	BS	
9	09	€	HT	
10	**0A**	**€**	**LF**	**Fin de ligne**
11	0B	€	VT	
12	0C	€	FF	
13	**0D**	**€**	**CR**	**Touche Entrée**
14	0E	€	SO	
15	0F	€	SI	
16	10	€	SLE	
17	11	€	CS1	
18	12	€	DC2	
19	13	€	DC3	
20	14	€	DC4	
21	15	€	NAK	
22	16	€	SYN	
23	17	€	ETB	
24	18	€	CAN	
25	19	€	EM	
26	1A	€	SIB	
27	1B	€	ESC	
28	1C	€	FS	
29	1D	€	GS	
30	1E		RS	
31	1F	€	US	
32	20	(space)		Barre espace
33	21	!		
34	22	"		
35	23	#		
36	24	$		
37	25	%		
38	26	&		
39	27	'		
40	28	(		
41	29	)		
42	2A	*		
43	2B	+		
44	2C	,		
45	2D	-		
46	2E	.		
47	2F	/		
48	30	0		
49	31	1		
50	32	2		
51	33	3		
52	34	4		
53	35	5		
54	36	6		
55	37	7		
56	38	8		
57	39	9		
58	3A	:		
59	3B	;		
60	3C	<		
61	3D	=		
62	3E	>		
63	3F	?		
64	40	@		
65	41	A		
66	42	B		
67	43	C		
68	44	D		

69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	72	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□
128	80	€
129	81	□
130*	82	‚
131*	83	ƒ
132*	84	„
133*	85	…
134*	86	†
135*	87	‡
136*	88	ˆ
137*	89	‰
138*	8A	Š
139*	8B	‹
140*	8C	Œ
141	8D	□
142	8E	Ž
143	8F	□
144	90	□
145	91	'
146	92	'
147*	93	"
148*	94	"
149*	95	•
150*	96	-
151*	97	—
152*	98	˜
153*	99	™
154*	9A	š
155*	9B	›
156*	9C	œ
157	9D	□
158	9E	ž
159*	9F	Ÿ
160	A0	
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	¬
173	AD	-
174	AE	®
175	AF	¯
176	B0	°
177	B1	±
178	B2	²
179	B3	³
180	B4	´

181	B5	µ
182	B6	¶
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

€ : Indique que ce caractère n'est pas affichable.
□ : Indique que ce caractère n'est pas supporté par Windows.
* : Indique que ce caractère n'est disponible que pour les polices True Type.
La colonne Code n'a de sens que pour les caractères 1 à 31.

1.5.3. Caractères spéciaux

Seront utilisés :

Écriture	Dec	Remarque
\0	**0**	**Fin de chaîne**
\n	**10**	**Fin de ligne**
\r	**13**	**Retour chariot**

1.6. Tableaux et chaînes de caractères

1.6.1. Déclaration

Dans la déclaration des types précédents (voir *1.2.2*), les éléments manipulés prenaient une seule valeur à un moment donné, il s'agissait de variables simples. Mais parfois, on peut vouloir associer à un même élément plusieurs valeurs. Il est intéressant de faire appel à la notion de tableau.

Le tableau est un type de variables qui permet d'associer sous un même nom un ensemble fini (et de taille fixe) de valeurs du même type. L'accès à une valeur particulière se fait à l'aide d'un numéro d'ordre appelé indice. Tous les types de base (voir *cadre 4*) et tous les types abstraits construits en C/C++ peuvent servir à définir un tableau. La syntaxe est la suivante :

Type Nom_Tableau[Taille_Tableau] ;

Taille_Tableau définit le nombre de valeurs stockées dans le tableau et *Type* le type de données qu'il contient.

Exemples :

- *float **t**[80] ;*
  Déclaration d'un tableau *t* à une dimension de 80 colonnes contenant des données de type *float*.

- *int **m_fT**[5][6] ;*
  Déclaration d'un tableau *m_fT* à deux dimension de 5 lignes et de 6 colonnes contenant des données de type *int*.

- *char **s**[40] ;*
  Déclaration d'un tableau *s* à une dimension de 40 colonnes contenant des données de type *char*. Il s'agit d'une chaîne de caractères.

- *unsigned char **T1**[5][3][8] ;*
  Déclaration d'un tableau *T1* à 3 dimensions contenant des données de type *unsigned char*.

1.6.2. Indice

L'indice est le numéro d'ordre permettant d'accéder à un élément particulier (une case du tableau).

La première case du tableau aura toujours l'indice 0, la dernière *Taille_Tableau*-1.

Exemples :

- *float **t**[80] ;*
  Voir *cadre 6*.

Tableau **t** indice **i**

0	1	2	…	77	78	79
2.35	**-2.95**	**1.25**	**…**	**8.65**	**16.3**	**65.9**

cadre 6 : Tableau à une dimension.

- *int **m_fT**[5][6] ;*
  Voir *cadre 8*.

1.6.3. Affectation

L'affectation d'une ou plusieurs données d'un tableau se fait en spécifiant le ou les indices de la case à affecter.

Exemples :

- *t[0] = 2.35 ; t[79] = 65.9 ;*
  Voir *cadre 6*.
  Ou :
- *int i=2 ;t[i] = 1.25 ;*

- *m_fT[0][0] = 4 ; m_fT[1][0] = 53 ;*
  Voir *cadre 8*.

Tableau **m_fT** **i**

j	0	1	2	3	4	5
0	**4**	**-9**	**-5**			
1	**53**					
2						
3						
4	**5**	**-8**	**9**	**0**		**1**

cadre 8 : Tableau à deux dimensions.

Tableau **s1** indice **i**

0	1	2	3	4	5	6
T	**S**	**1**	**P**	**H**	**\0**	

cadre 7 : Chaîne de caractères

1.6.4. Chaînes de caractères

Une chaîne de caractères est tout simplement un tableau de caractères à une dimension.

Exemple :

- *char **s1**[7] ;*
  Déclaration d'un tableau *s1* à une dimension de 7 colonnes contenant des données de type *char*.

Le tableau *s1* donné *cadre 7* contient la chaîne de caractères "TS1PH", chaque caractère de la chaîne occupant une case du tableau.

Etant donné que le chaîne pourra être plus courte que la taille du tableau qui est sensé la contenir, le dernier caractère de la chaîne doit toujours être le caractère 0 noté **\0**.

L'affectation de cette chaîne pourra donc s'écrire :

```
char s1[7];
s1[0] = 'T';
s1[1] = 'S';
s1[2] = '1';
s1[3] = 'P';
s1[4] = 'H';
s1[5] = '\0';
```

Cette écriture est évidemment fastidieuse, surtout si la chaîne est longue. Nous préférerons donc souvent la fonction *sprintf* du langage C/C++ ou *Format* du langage *Visual C++*.

1.6.5. Chaînes de caractères sous Visual C++

On peut sous *Visual C++*, en plus de la déclaration précédente, définir une chaîne de caractère en utilisant le type *CString*.

De nombreux opérateurs et fonctions membres de la classe *CString* existent, tels que :

- + : Concaténation de chaînes,
- *GetLength* : Renvoie la longueur de la chaîne,
- *GetAt* : Renvoie un car. à une position,
- *IsEmpty* : Teste si la chaîne est vide,
- *Mid, Left, Right* : Extraction de sous chaînes,
- …

1.6.6. Fonction *CString::Format*

Prototype	void Format(const char **format* [, *argument*, …]);
Action	Affecte des données formatées dans une chaîne de caractères de type *CString*.
Description	La fonction *Format*, membre de la classe *CString*, accepte une séquence d'arguments et affecte les données formatées dans la chaîne de caractères.
Spécifica-teurs de format	*%[indicateurs][taille][.prec][F|N|h|l|L]type* • *%* : Début de spécification de format • *indicateurs* : Justification de la sortie, signe pour les nombres, zéros à droite des nombres, … • *taille* : Nombre mini de caractères à afficher, blancs ou zéros pour compléter. • *Prec* : Nb. maxi de caractères à afficher. Pour les entiers, nb. mini de chiffres à afficher. • *F|N|h|l|L* : *F* : pointeur long (far pointer), *N* : pointeur court (near pointer), *h* : short int, *l* : long ou double, *L* : long double • *type* : d ou i :entier décimal signé u : entier décimal non signé f : flottant signé de la forme [-]ddd.dd e : flottant signé de la forme [-][d.dde[+/-]ddd c : caractère simple s : chaîne de caractères x : entier hexadécimal non signé avec a, b, c, d, e, f).
Valeur ren-voyée	Aucune.

Exemple	CString s;
	int i=20;
	s.Format("Valeur de i : %u",i);

Remarque :

La fonction *Format* n'affiche pas le chaîne de caractères dans la fenêtre. Pour cela, on utilisera, par exemple, la fonction *TextOut* (membre de CDC).

Exemple :

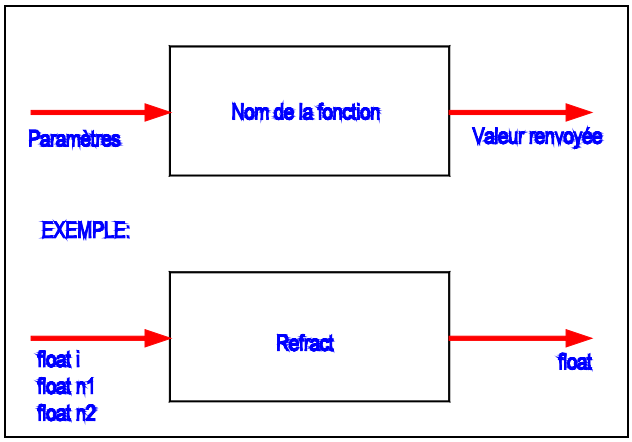
```
CString s;
int i=20;
s.Format(("Valeur de i : %u",i);
pDC→TextOut(100,50,s);
```

2. Fonctions

2.1. Définition

Comme expliqué paragraphe *2*, l'analyse d'un problème fait apparaître un ensemble d'activités qui échangent des informations. Ces activités peuvent être assimilées à des fonctions.

Une fonction apparaît alors comme un nom donné à un ensemble d'opérations, de calculs et de traitements. Voir *cadre 9*.



cadre 9 : Réprésentation symbolique d'une fonction.

Des données sont communiquées à la fonction afin qu'elle puisse effectuer les traitements. Les données passées à la fonction sont appelées les paramètres ou les arguments.

En outre, la fonction peut produire des informations qu'elle communique ensuite en retour, c'est la valeur renvoyée.

2.2. Prototype d'une fonction

La manipulation de la fonction dans des programmes demande de respecter cet ensemble de caractéristiques, qui définissent son interface externe, encore appelé en C/C++ le prototype de la fonction.

Globalement, le prototype d'une fonction précise :

- le nom de la fonction ;
- le nombre des paramètres passés à la fonction ;
- le type de chacun de ces paramètres ;
- le type de valeur renvoyée par la fonction.

Exemple :
Pour la fonction *Refract*, le prototype s'écrit :

```
float Refract(float i, float n1, float n2);
```

Remarques :

- Si la fonction appartient à une classe, le nom de celle-ci est définie avant le nom de la fonction, suivi de ::
- Si la fonction n'admet pas de paramètre(s), on remplace la liste des paramètres par le mot *void*.
- Si la fonction ne retourne pas de valeur, on remplace aussi par *void*.

Exemples :

```
void CBkView::OnDraw(CDC* pDC);
```

- Nom de la fonction : *OnDraw* ;
- Appartient à la classe *CBkView* ;
- Paramètre : un pointeur sur un objet de type *CDC* ;
- Pas de valeur en retour, donc *void*.

```
BOOL CGdiObject::DeleteObject(void);
```

- Nom de la fonction : *DeleteObject* ;
- Appartient à la classe *CGdiObject* ;
- Aucun paramètre, donc *void*.
- Valeur en retour : *BOOL* (un booléen, une variable qui prend deux valeurs : *TRUE* ou *FALSE*, c-à-d vrai ou faux).

Le prototype d'une fonction est généralement défini dans un fichier **.h**. Exemples : *bkdoc**.h***, *bkview**.h***.

2.3. Corps d'une fonction

Dans le corps de la fonction il s'agit de spécifier les traitements et calculs à effectuer en partant des paramètres pour arriver à la valeur renvoyée.

Exemple :
Pour la fonction *Refract*, le corps s'écrit :

```
float Refract(float i, float n1, float n2)
{
 float r;
 if (n2<1||fabs(n1*sin(i)/n2)>1)
 {
  r=1e10;
 }
 else
 {
  r=asin(n1*sin(i)/n2);
 }
 return r;
}
```

Remarques :

- On récrit le prototype sans ; à la fin de la ligne.
- Les traitements à effectuer sont écrits entre le 1er niveau d'accolades : { } ;
- Si la fonction renvoie une valeur, le mot *return* doit figurer à l'intérieur du corps.
- La variable *float r* déclarée à l'intérieur du corps de la fonction n'est pas connue à l'extérieur. Sa portée est limitée à la fonction.
- A l'intérieur d'une fonction, on peut appeler d'autres fonctions (*fabs*, *sin*, *asin*).

Le corps d'une fonction est généralement défini dans un fichier **.cpp**. Exemples : *bkdoc**.cpp***, *bkview**.cpp***.

3. Instructions de sélection et d'itération

Type	Syntaxe	Rôle	Exemple

if	if (*expression*) *instruction*	Instruction de sélection (ou test)
		Si *expression* est:
		⇒ vrai: *instruction* est exécuté
		⇒ faux: *instruction* est ignoré
			... ;
			if (note<10)
			{
			…
			}
			... ;

if, else	if (*expression*) *instruction1*
	else *instruction2*
		Instruction de sélection
		Si *expression* est:
		⇒ vrai: *instruction1* est exécuté
		instruction2 n'est pas exécuté
		⇒ faux: *instruction2* est exécuté
			... ;
			if (y==0)
			{
			z=1e10;
			}
			else
			{
			z=x/y;
			}
			... ;

if,
else if,
else
	if (*expression*) *instruction1*
	else if (*expression2*) *instruction2*
	else if ...
	else *instructionN*
		Instruction de sélection
		Une et une seule des N instructions est exécutée, celle qui correspond à la première expression vraie.
		Dans le cas où il n'y a aucune expression vraie et si l'instruction else est présente, c'est *instructionN* qui est exécuté. L'instruction else est facultative.
			... ;
			if (note<5)
			{
			…
			}
			else if (note>15)
			{
			…
			}
			else
			{
			…
			}
			... ;

while	while (*expression*)
	instruction
		Instruction d'itération
		Si *expression* est:
		⇒ vrai : *instruction* est exécuté tant que *expression* est vrai.
		⇒ faux : *instruction* est ignoré. On passe à l'instruction suivant la boucle while.
			... ;
			x=0;
			while (x<=6.28)
			{
			y=sin(x);
			…
			x=x+0.02;
			}
			... ;

do
while
	do
	instruction
	while (*expression*)
		Instruction d'itération
		Si *expression* est :
		⇒ vrai: *instruction* est exécuté à nouveau.
		⇒ faux: *instruction* est ignoré. On passe à l'instruction suivant la boucle while.
		Quelle que soit la valeur de *expression*, *instruction* est exécuté au moins une fois.
			... ;
			x=0;
			do
			{
			y=sin(x);
			…
			x=x+0.02;
			}
			while (x<=6.28);
			... ;

for	for (*expression1*; *expression2; expression3*)
	instruction
		Instruction d'itération
		expression1: initialise la boucle.
		expression2: test avant chaque itération.
		Si *expression2* est :
		⇒ vrai : on exécute *expression3*.
		Itération jusqu'à ce qu'*expression2* soit faux.
		⇒ faux : on sort de la boucle.
		expression3: est évalué après chaque itération.
			... ;
			for (x=0;x<=6.28;x=x+0.02)
			{
			y=sin(x);
			…
			}
			... ;

cadre 10 : Instructions de sélection et d'itération.

C. VARIABLES ET POINTEURS

La notion de variable a été abordée en *A.1.1.* Nous aborderons dans ce chapitre le concept de variable de type pointeur.

1. Concept

L'adresse ou la valeur à cette adresse ?

Un *pointeur* représente l'endroit de la mémoire où est stockée une information. C'est une variable qui contient l'adresse mémoire d'une donnée particulière.

En fait, il faut prendre conscience ici que, pour la machine, une information en mémoire est caractérisée par trois éléments :

- L'endroit précis où est mémorisée l'information dans la machine, c'est-à-dire son *adresse mémoire*.
- La nature de l'information, définissant les opérations machine à utiliser lors de sa manipulation, c'est-à-dire son *type* (int, float, ...).
- La *valeur* de l'information elle-même.

L'endroit où sont stockées les informations est défini par un nombre entier, donnant le nombre d'octet entre sa position actuelle et le début de la zone mémoire.

Adresse	Mémoire	Contenu
	...	
....4092	3.142	Valeur de *r*
	...	
....4080	4092	*ptr_r* : valeur de l'adresse de *r*
	...	

Un pointeur est une variable qui peut stocker la valeur d'une adresse mémoire, c'est-à-dire une valeur entière représentant un nombre d'octet.

Pour que cette adresse soit utilisable, il faut connaître la nature de l'information mémorisée à cette adresse, c'est-à-dire que son type. *Un pointeur sera donc toujours associé à un type de données particulier* (le pointeur sur r est associé à un flottant).

On peut avoir en C/C++ des pointeurs sur tous les types de données existants ou définis par le programmeur.

On aura ainsi des pointeurs sur des entiers, des pointeurs sur des flottants, des pointeurs sur des tableaux de caractères et des pointeurs sur des fonctions !

Il s'agit de variables comme les autres, et par conséquent elles se déclarent, s'initialisent et se manipulent par des opérations particulières. Elles respectent les mêmes règles que les autres variables (portée, statut, ...).

De même, on pourra avoir des tableaux de pointeurs, des fonctions renvoyant des valeurs de pointeurs, des pointeurs passés en arguments et des pointeurs de pointeurs !

2. Définition

La syntaxe est la suivante :

```
type * nom_pointeur;
```

Exemples de déclarations de pointeurs :

- float *ptr; //Un pointeur de nom *ptr* sur une zone mémoire stockant un objet de type float.
- int *truc; //Un pointeur de nom *truc* sur une zone mémoire stockant des int.
- CString *s; //Un pointeur *s* sur une classe de type CString donc sur une chaîne de caractères).
- float (*t)[2] ; //Un pointeur de nom *t* sur un tableau de *float* à 2 colonnes.
- class CBkDoc * m_pDoc ;//Un pointeur de nom *m_pDoc* pointant sur un objet (une classe) de type *CBkDoc*. Voir C.

3. Initialisation

3.1. Pointeur sur objet existant

L'initialisation d'un pointeur peut s'effectuer en lui affectant la valeur de l'adresse d'un objet déjà existant. Pour cela, on dispose de *l'opérateur d'adressage* ***&***.

```
float r=3.142;
float *ptr_r=&r;
```

Le pointeur *ptr_r* est initialisé à l'adresse de la variable *r* (On affecte ...4092 à *ptr_r*).

La seconde opération importante est la manipulation de la valeur stockée à l'adresse contenue par le pointeur. Pour cela, on dispose de *l'opérateur de déférencement noté* *******.

```
float x;
float r=3.1415;
float *ptr_r=&r;
x=cos(r); //x=-1;
x=cos(*ptr_r); //x = -1 !!, idem ci-dessus
```

**ptr_r* représente la valeur stockée à l'adresse contenue dans *ptr_r*.

L'expression **ptr_r* peut maintenant remplacer la variable *r* partout où l'on souhaite.

3.2. Pointeur sur nouvel object. Opérateur *new* et *delete*.

L'initialisation d'un pointeur peut aussi s'effectuer

en demandant directement de réserver la zone mémoire de l'objet qu'il va pointer.

On parlera d'allocation dynamique de la mémoire, et de désallocation dynamique. Ceci présente le grand avantage de pouvoir allouer et libérer la mémoire occupée par les objets à tout moment.

Exemple d'utilisation d'un pointeur dynamique :

- float *ptr; //Déclaration d'un pointeur *ptr* sur une zone mémoire pouvant stocker un *float*.
- ptr = **new** float ; //Réservation de l'espace mémoire.
- *ptr = 3.14 ; //Ecriture de 3.14 dans la zone mémoire réservée.
- …
- **delete** ptr ; //Libération de la mémoire

Autres exemples d'allocation et de désallocation :

```
int *p_i;
int *p_i;
CString *p_s;
float (*p_fT)[2];

p_i=new int;
*p_i=3;
p_s=new CString[3];
p_s[0].Format("%u",*p_i);
AfxMessageBox(p_s[0]); // Voir cadre 11.

(*p_i)++;
p_s[1].Format("%u",*p_i);
AfxMessageBox(p_s[1]);

p_fT=new float[1000][2];
int i;
for (i=0;i<1000;i++)
{
 p_fT[i][0]=0;p_fT[i][1]=0;
}
p_s[2].Format("%8.4f,%8.4f",p_fT[500][0],p
_fT[500][1]);
AfxMessageBox(p_s[2]);

delete p_i;
delete []p_s;
delete []p_fT;
```

4. Pointeur sur les variables membres et les fonctions membres d'une classe

cadre 11.

On utilisera l'opérateur → pour pointer les membres de la classe. Voir les opérateurs en *B.1.3*.

Exemple :

```
void CBkView::OnDraw(CDC* pDC)
{
 CBkDoc* pDoc = GetDocument();
 pDC→TextOut(20,50,"Bonjour: "+ pDoc→m_sMo
nNom);
}
```

Explications :

a) La fonction *OnDraw* est membre de la classe *CBKView*, ne retourne pas de valeur, et admet comme paramètre un pointeur de nom *pDC* sur un objet de type *CDC.*
b) *pDoc* est un pointeur de type *CBkDoc* et est affecté avec l'adresse du document.
c) La fonction *TextOut* (membre de *CDC*) est appelée par le pointeur *pDC*. On notera ici la présence de la flèche →.
d) La variable *m_sMonNom* est membre de la classe *CBkDoc*. Elle est appelée ici en utilisant le pointeur *pDoc* pointant sur cette classe. On notera encore la présence de la flèche →.

Pour plus d'informations, voir le chapitre suivant.

D. LES CLASSES C++ ET MFC

Le langage *Visual C++* prend toute son ampleur lorsqu'on l'utilise avec la librairie de classe : « la Microsoft Fondation Class » appelée *MFC*. Cette librairie contient de nombreuses fonctions qui facilitent grandement l'écriture d'applications destinées à être utilisées dans un environnement Windows.

1. Défintion d'une classe

La notion de classe peut être assimilée à une structure qui contient des fonctions et des variables. Ces fonctions et ces variables seront donc regroupées sous un nom générique (le nom de la classe) et seront donc membres de cette classe.

1.1. Exemple

On souhaite créer une classe *Ccercle* contenant les variables membres *xc, yc, ray* de type *float* et la fonction membre *surface* retournant un *float*.

Nous allons donc écrire :

```
class Ccercle
{
 float xc, yc;
 float ray;
 float surface();
};
```

La corps de la fonction *surface* est le suivant :

```
float Ccercle::surface()
{
 return ray*ray*3.1415;
}
```

Cette fonction *surface* calcule la surface d'un cercle de rayon *ray*.

La variable *ray* n'est pas à redéclarer. Elle est déjà membre de la classe.

1.2. Création d'une instance d'une classe

Pour pouvoir utiliser la classe, il faut créer un objet du type de la classe. Cet objet peut être une variable ou un pointeur.

1.2.1. Variable

```
Ccercle c;
c.ray=2.0;
surf=c.surface();
```

- Définit un objet *c* (une variable) de type *Ccercle*. Cet objet existe maintenant en mémoire et ne pourra plus être supprimé.
- Affecte la variable membre *ray* avec la valeur de 2.
- Calcule la surface du cercle de rayon 2.

1.2.2. Pointeur

```
Ccercle *p_c;
p_c=new Ccercle();
p_c→ray=2.0;
surf=p_c→surface();
delete p_c;
```

- Définit un pointeur *p_c* de type *Ccercle*.
- Fait pointer le pointeur *p_c* sur un nouvel objet de type *Ccercle*. Cet objet existe maintenant en mémoire et pourra être supprimé par *delete*.
- Affecte la variable membre *ray* avec la valeur de 2.
- Calcule la surface du cercle de rayon 2.
- Supprime l'objet pointé par *p_c*. La mémoire occupée par l'objet *Ccercle* est maintenant libérée.

2. Spécificateurs d'accès

La classe *Ccercle* ainsi définie n'est pas accessible de l'extérieur. En effet, toutes les variables et les fonctions d'une classe sont par défaut de type *private* (privé). Les variables ne peuvent donc pas être initialisées par des instructions extérieures à la classe. Pour résoudre ce problème, il suffit d'ajouter un spécificateur d'accès public comme suit :

```
class Ccercle
{
 private:
 float xc, yc;
 public:
 float ray;
  float surface();
};
```

Dans l'exemple ci-dessus, les variables membres *xc* et *yc* sont privées, et la variable membre *ray* et la fonction membre *surface* sont publiques.

```
Ccercle c;
c.ray=2.0;          //Ecriture correcte
surf=c.surface();   //Ecriture correcte

c.xc=12.0;          //Ecriture incorrecte,
car var. privée…
```

3. Fonction membre constructeur

Dans la version actuelle du programme, la variable rayon est définie par l'intermédiaire de *c.ray=2.0* ou de *p_c→ray=2.0*. De plus, les variables *xc* et *yc* ne sont pas initialisées.

Lors de la création d'un objet, une fonction membre particulière est exécutée si elle existe. Cette fonction est appelée le constructeur. Il porte obligatoirement le même nom que la classe.

```
class Ccercle
{
 private:
 float xc, yc;
 public:
 float ray;
 Ccercle(float x, float y, float r);
 //Constructeur
  float surface();
};
```

Le constructeur de la classe *Ccercle* admet 3 pa-

ramètres.

```
Ccercle::Ccercle(float x, float y, float r)
{
 xc=x;
 yc=y;
 ray=r;
}
```

Dans le constructeur, on effectue l'affectation des variables membres.

L'appel du constructeur lors de la création de la classe simplifie l'écriture :

```
Ccercle c(10.0, 10.0, 2.0);
```

Les variables membres *xc, yc* et *ray* sont initialisées avec les valeurs 10, 10 et 2. L'écriture *c.ray=2* n'est donc plus nécessaire.

On peut écrire aussi :

```
Ccercle *p_c;
p_c=new Ccercle(10.0, 10.0, 2.0);
```

4. Fonction membre destructeur

On sait maintenant que la fonction membre constructeur est automatiquement exécutée lors de la définition de l'objet. De la même manière, une fonction membre destructeur est automatiquement exécutée lorsqu'un objet est détruit. Le nom de la fonction destructeur est le même que celui de la classe, mais un tilde (~) est placé devant.

```
class Ccercle
{
 private:
 float xc, yc;
 public:
 float ray;
 Ccercle(float x, float y, float r);
 //Constructeur
 ~Ccercle();
 //Destructeur
  float surface();
};
```

```
Ccercle:: ~Ccercle()
{
 //On peut placer ici les instructions à
faire lors de la destruction de la classe
}
```

5. Classe dérivée

La dérivation d'une classe est un processus élégant qui est utilisé lorsqu'on désire ajouter des fonctionnalités à une classe existante sans avoir à réécrire le code qui la compose. En effet, lorsqu'on dérive une classe, la nouvelle classe hérite de toutes les données membres et de toutes les fonctions membres de la classe dont elle est issue.

```
class Ccylindre:public Ccercle
{
 private:
 float haut;
 public:
 Ccylindre(float r,float h);
 ~Ccylindre();
 float volume();
};
```

- Déclaration d'une classe *Ccylindre* dérivée de la classe *Ccercle*. La classe *Ccylindre* hérite de toutes les propriétés de la classe *Ccercle*, c-à-d des variables membres *xc, yc, ray* et de la fonction membre *surface*. Voir *1.1.*
- La classe *Ccylindre* comporte une variable membre privée *haut* de type *float.*
- Elle comporte un constructeur et un destructeur.
- Elle comporte une fonction membre *volume* retournant un *float*.

```
Ccylindre::Ccylindre(float r,float h)
{
 ray=r;
 haut=h;
}
//_____________________________________________
Ccylindre::~Ccylindre()
{
 //…
}
//_____________________________________________
float Ccylindre::volume()
{
 return surface()*haut;
}
```

- Corps du constructeur : on assigne les variables membres *ray* (membre de *Ccercle*, donc aussi de *Ccylindre*) et *haut* (membre de *Ccylindre)*.
- Corps du destructeur (vide pour l'instant).
- Corps de la fonction *volume*. On calcule le volume d'un cylindre en appelant la fonction *surface* membre de la classe *Ccercle*.

```
 float surf, vol;
 char s[80];

 Ccylindre *p_cy;
 p_cy=new Ccylindre(1.0,10);
 surf=p_cy→surface();
 sprintf(s,"Surface du cercle de base:
%8.3f",surf);
 pDC→TextOut(10,200,s);
 vol=p_cy→volume();
 sprintf(s,"Volume du cylindre:
%8.3f",vol);
 pDC→TextOut(10,220,s);
 delete p_cy;
```

Valeurs affichées :

- Surface du cercle de base : 3.141
- Volume du cylindre : 31.415

E. CONTEXTES DE PÉRIPHÉRIQUES

D'après Visual C++ 6, LE TOUT EN POCHE.

1. Définition

Un contexte de périphérique (*device context* ou *DC* en anglais) est un structure gérée par Windows qui stocke les informations requises lorsqu'un programme affiche une sortie sur un périphérique (écran, imprimante, …). Le contexte de périphérique stocke des informations concernant la zone de sortie et ses fonctionnalités. Avant de pouvoir utiliser une fonction de sortie graphique GDI (Graphic Device Interface), il faut créer un contexte de périphérique.
Exemple : la fonction *OnDraw* admet comme argument un pointeur sur la classe CDC :
*void CBkView::OnDraw(CDC *pDC);*

Les contextes de périphériques sont en réalité des structures utilisées par l'interface GDI pour suivre l'état de sortie courante d'une fenêtre. Cependant, vous n'accédez jamais directement aux variables membres d'un contexte de périphérique ; l'accès se fait toujours par l'intermédiaire d'appel de fonctions.
Exemple : *pDC→TextOut(10,10,"BONJOUR") ;*

La raison pour laquelle les contextes de périphériques sont utilisés est que leur emploi est obligatoire ; il n'existe pas d'autre moyen de créer une sortie dans un programme Windows. Cependant, l'utilisation de contextes de périphériques est également la première étape de l'utilisation des nombreuses fonctions GDI (plumes, fontes, …) disponibles sous Windows. La compréhension du fonctionnement des contextes de périphériques vous aidera donc à optimiser vos programmes Windows.

1.1. Objets GDI et contextes de périphériques

Lorsqu'on associe un objet GDI à un contexte de périphérique, on dit souvent qu'on *sélectionne l'objet dans le contexte de périphérique*.

Un objet GDI peut être associé à un contexte de périphérique pour lui fournir des fonctionnalités de dessin spécifique.
Exemple : Changer la couleur d'une ligne, …

Les différents objets GDI permettent d'obtenir différents types de sorties. Les objets GDI les plus couramment utilisés avec des contextes de périphériques sont donnés dans le *tableau 1*.

1.2. Indépendance par rapport aux périphériques

L'intérêt des contextes de périphériques est de rendre les programmes Windows indépendants du matériel utilisé. En prenant certaines précautions, votre programme pourra être exécuté avec n'importe quel type d'affichage ou d'imprimante pris en charge par un pilote Windows. La plupart des nouveaux périphériques fournissent des pilotes si Windows ne les prend pas en charge, ce qui signifie que les programmes que vous écrivez aujourd'hui fonctionneront avec les périphériques de demain.

Les précautions à prendre pour obtenir une véritable indépendance vis-à-vis du matériel sont les suivantes :

- N'indiquez pas de dimensions absolues dans votre programme. Des écrans ou des imprimantes plus grands ou plus petits déformeront l'affichage de votre programme.
- Ne calculez pas en fonction d'une résolution d'affichage donnée. Différents utilisateurs emploient différentes résolutions d'affichage.
- Ne partez pas du principe qu'un certain ensemble de couleurs sera toujours disponible ou adapté à toutes les situations. Ainsi, ne supposez pas que le fond de l'espace de travail est toujours blanc. De nombreux utilisateurs changent leur configuration de couleurs.

2. Utilisation des contextes de périphériques

Lorsqu'on utilise Visual C++, on fait presque toujours appel à une classe MFC pour accéder à un contexte de périphérique. La bibliothèque MFC propose quatre classes de contextes de périphériques pour vous faciliter les choses :

- CDC. La classes de base de toutes les classes de contextes de périphériques.
- CPaintDC. Permet de faire le « ménage » lorsqu'une fenêtre répond à WM_PAINT.
- CClientDC. Utilisé lorsqu'un contexte de périphérique ne sera utilisé pour une sortie vers le zone client d'une fenêtre.
- CWindowDC. Utilisé lorsque toute la fenêtre peut être utilisée pour l'affichage.
- CPrinterDC…

Objet	Fonction
Crayon (Pen)	Dessin de lignes
Pinceau (Brush)	Remplissage de régions
Bitmap (Bitmap)	Affichage d'images
Police (Font)	Caractéristiques des polices de caractères

tableau 1 : Objets GDI couramment utilisés dans les programmes Windows.

2.1. MFC, ClassWizard et les contextes de périphériques

Lorsque vous créez une classe avec ClassWizard ou lorsque vous créez une nouvelle application MFC, le code utilisant ou créant un contexte de périphérique est crée de manière automatique.
Exemple : La fonction *OnDraw* typique ci-dessous.

```
void CBkView::OnDraw(CDC* pDC)
{
 CDocument* pDoc = GetDocument();
 //TODO: add draw code here

}
```

Le contexte de périphérique utilisé pour la fonction *OnDraw* est crée par l'infrastructure MFC avant l'appel de la fonction *OnDraw*. Dans la mesure où toutes les fonctions *OnDraw* doivent afficher quelque chose, le contexte de périphérique est généré automatiquement sans que vous ayez besoin d'écrire de code.

La plupart des fonctions ayant besoin d'un contexte de périphérique, y font appel sous la forme d'un paramètre, comme c'est le cas pour *OnDraw*.

2.2. Sélection d'un objet GDI

L'une des erreurs les plus courantes, lors de l'utilisation de contextes de périphériques, se produit lors de l'association d'un objet GDI (exemple, une plume) avec un contexte de périphérique. Lorsque le contexte de périphérique est crée, il contient un ensemble d'objets GDI par défaut (voir *tableau 2*).

Device context
Font
Brush
Pen
Bitmap

tableau 2 : Un contexte de périphérique crée avec un ensemble d'objets GDI par défaut.

Lorsqu'un nouvel objet GDI, une plume, par exemple, est associée à un contexte de périphérique, la plume par défaut du GDI est passée comme valeur de renvoi à la fonction appelante. Cette valeur de renvoi doit être stockée afin de pouvoir être renvoyée ensuite au contexte de périphérique.

Exemples :

```
CPen plumeA, *m_pOldPen;
plumeA.CreatePen(PS_SOLID,2,RGB(0,0,0));
m_pOldPen=pDC→SelectObject(&plumeA);
…
pDC→SelectObject(m_pOldPen);
plumeA.DeleteObject();
```

```
CFont fntA, *m_pOldFont;
fntA.CreateFont(15,0,0,0,FW_MEDIUM,FALSE,FA
LSE,FALSE,ANSI_CHARSET,OUT_DEFAULT_PRECIS,C
LIP_DEFAULT_PRECIS,DEFAULT_QUALITY,DEFAULT_
PITCH,"Arial");
m_pOldFont=pDC→SelectObject(&fntA);
…
pDC→SelectObject(m_pOldFont);
fntA.DeleteObject();
```

Dans les listings ci-dessus, les pointeurs *m_pOldPen* et *m_pOldFont* contiennent l'adresse de la plume originale et de la fonte originale du contexte de périphérique.

Vous devez rétablir l'état initial du contexte de périphérique une fois que vous en avez fini avec lui. A défaut, les ressources que vous avez associées au contexte de périphérique ne seront pas libérées et votre programme consommera de plus en plus de ressources GDI. Ce problème est nommé fuite de ressources; il faut donc s'habituer à toujours rétablir l'état original dans le contexte de périphérique.

2.3. Choix d'un système de coordonnées pour le contexte de périphérique

Sous Windows, différents systèmes de coordonnées (*map mode* en anglais) permettent de définir la taille et la direction d'unités utilisées avec les fonctions de dessin. Selon les systèmes, il est possible d'utiliser des dimensions physiques et logiques, et l'origine des coordonnées peut se situer en haut, en bas ou à un endroit quelconque de l'écran.

Huit systèmes de coordonnées sont disponibles sous Windows. Pour déterminer le système de coordonnées en cours d'un contexte de périphérique, utilisez la fonction *GetMapMode* ; pour le modifier, utilisez *SetMapmode*. Voici les différents systèmes de coordonnées :

- MM_ANISOTROPIC. Convertit l'unité logique en unité arbitraire, les échelles des axes étant elles-mêmes arbitraires. Les fonctions membres *SetWindowExt* et *SetViewportExt* sont utilisées pour modifier les unités, l'orientation et l'échelle.
- MM_HIENGLISH. L'unité logique est convertie en une valeur physique de 0,001 pouce. X positif vers la droite, Y positif vers le haut.
- MM_HIMETRIC. L'unité logique est convertie en une valeur physique de 0,01 mm. X positif vers la droite, Y positif vers le haut.
- **MM_ISOTROPIC**. Semblable à MM_ANISOTROPIC, par contre l'échelle des deux axes étant

identique. Utiliser *SetWindowExt* et *SetViewportExt* pour définir unité et orientation des axes.

- MM_LOENGLISH. Idem que MM_HIENGLISH avec une conversion de 0,01 pouce.
- MM_LOMETRIC. Idem que MM_HIMETRIC avec une conversion de 0,1 mm.
- MM_TEXT. L'unité logique est convertie en un pixel de périphérique. X positif vers la droite, Y positif vers le bas.
- **MM_TWIPS**. L'unité logique est convertie en 1/20e de point, 1 twip égale 1/1440e de pouce (soit environ 1/57e de mm). Ce mode est utilisé lors de l'envoi de données à l'imprimante. X positif vers la droite, Y positif vers le haut.

2.4. Définition du système de coordonnées pour dessiner dans un repère orthonormé.

2.4.1. Listing

```
void CBkView::OnDraw(CDC* pDC)
{
 CBkDoc* pDoc=GetDocument();
 float
  xmi,ymi,xma,yma,
 ech;

 xmi=-70.0;
 xma=220.0;
 ymi=-100.0;
 yma=110.0;

 CRect rcClient;
 GetClientRect(rcClient);
 pDC→DPtoLP(rcClient);
 if (pDC→IsPrinting())
 {
  pDC→SetMapMode(MM_TWIPS) ;
  ech=1440.0/25.4;//1 twip = 1/1440e de
pouce!
  //Décalage axes
  pDC→SetWindowOrg(ech*xmi,ech*yma);
 }
 else
 {
  pDC→SetMapMode(MM_ISOTROPIC) ;
  ech=100;
  //1 mm est converti vers 100 unités
logiques!
  pDC→SetWindowExt(ech*(xma-xmi),-ech*(yma-
ymi));
  //Décalage axes
  pDC→SetWindowOrg(ech*xmi,ech*yma);

 pDC→SetViewportExt(rcClient.right,rcClient
.bottom);
 }
…
}
```

cadre 12.

Le listing donné *cadre 12* permet de spécifier un repère au format A4 paysage (297 X 210 mm) avec l'origine à 70 mm du bord gauche et à 100 mm du bas.
Le système de coordonnées sélectionné est MM_TWIPS pour l'imprimante et MM_ANISOTROPIC pour l'écran.

2.4.2. Fonctions

- **Fonction CDC::SetMapMode**

Prototype	int SetMapMode(int Mode);
Action	Spécifier le système de coordonnées.
Description	Permet de définir le système de coordonnées à utiliser. *Mode* peut être l'une des constantes : MM_HIENGLISH, MM_HIGHMETRIC, MM_LOENGLISH, MM_LOMETRIC, MM_TEXT, MM_TWIPS, MM_ANISOTROPIC et MM_ISOTROPIC. Dans le cas de l'utilisation de MM_ANISOTROPIC ou MM_ISOTROPIC, il faut spécifier les unités et les échelles des axes avec les fonctions *SetWindowExt* et *SetViewPortExt*.
Val. retournée	L'ancien système de coordonnées.
Exemple	SetMapMode(MM_TEXT);

- **Fonction CDC::SetWindowExt**

Prototype	CSize SetWindowExt(int cx, int cy);
Action	Spécifier la taille de la fenêtre.
Description	Permet de définir la taille de la fenêtre en unités logiques. *cx* : taille suivant x ; *cy* : taille suivant y. A utiliser avec MM_ANISOTRPIC ou MM_ISOTROPIC.
Val. retournée	L'ancienne taille.
Exemple	//1 mm est converti vers 100 unités logiques! int ech=100, xmi=0, ymi=0, xma=297, yma=210 ; pDC→SetWindowExt(ech*(xma-xmi),-ech*(yma-ymi));

- **Fonction CDC::SetWindowOrg**

Prototype	CPoint SetWindowOrg(int x, int y);
Action	Spécifier la position de l'origine dans la fenêtre.
Description	Permet de spécifier le décalage d'origine en unités logiques. A utiliser avec MM_ANISOTRPIC ou MM_ISOTROPIC.
Val. retournée	L'ancien décalage.
Exemple	//1 mm est converti vers 100 unités logiques! int ech=100, xmi=0, ymi=0, xma=297, yma=210 ; pDC→SetWindowExt(ech*(xma-xmi),-ech*(yma-ymi)); pDC→SetWindowOrg(ech*xmi,ech*yma);

2.5. Dessin dans la fenêtre

Pour écrire, dessiner,… dans la fenêtre nous utiliserons les multiples fonctions membres de la classe CDC de la bibliothèque MFC.

Exemple : Pour tracer une ligne de le repère précédent du point (10, 10) au point (50, 50), nous écrirons :

```
pDC→MoveTo(10*ech, 10*ech);
pDC→LineTo(50*ech, 50*ech);
```

La valeur de ech = 1440/25.4 (imprimante) ou ech = 100 (écran), voir fonction *IsPrinting* du *cadre 12*.

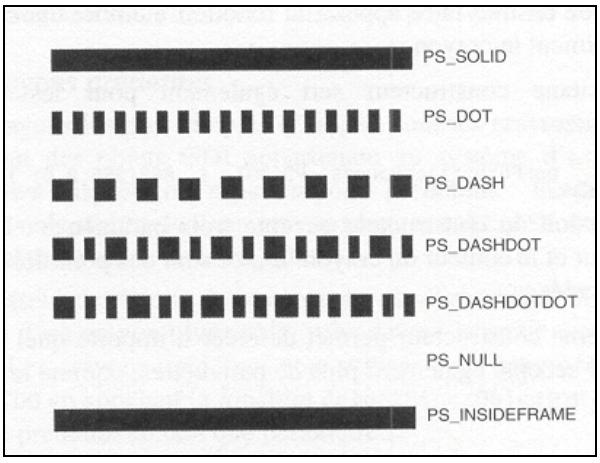
3. Objets GDI

3.1. Crayons

3.1.1. Définition

Un crayon ou plume (pen en anglais) est un objet GDI Windows servant à dessiner des lignes et des formes.
Comme pour les autres objets GDI, pour utiliser un crayon, il faut créer un objet MFC. **C'est la classe *CPen* qui permet de créer et de gérer les crayons**.
Lors de la création d'un crayon, on spécifie généralement trois attributs :

- **L'épaisseur**. Généralement d'un pixel, mais vous pouvez choisir n'importe quelle valeur pour un trait continu (PS_SOLID, PS_INSIDEFRAME). Pour les autres styles, l'épaisseur est de 1 pixel.
- **Le style**. L'un des styles du *cadre 13*. PS_NULL crée un crayon nul, qui ne fera appraître aucune ligne. PS_INSIDEFRAME crée un crayon dessinant des lignes à l'intérieur de formes fermées produites par des fonctions GDI, par exemple *Ellipse* ou *Rectangle*.
- **La couleur**. L'une des couleurs Windows par la fonction *RGB*.



cadre 13 : Exemples de styles de crayon.

3.1.2. Classe CPen

Pour définir, sélectionner et utiliser un crayon, on écrira :

```
CPen
 plumeVerte,
 plumeRouge,
 *m_pOldPen;

plumeVerte.CreatePen(PS_SOLID,5,RGB(0,255,0
));
plumeRouge.CreatePen(PS_DASHDOT,1,RGB(255,0
,0));

m_pOldPen=pDC→SelectObject(&plumeRouge);
//On dessine en rouge…
…
pDC→SelectObject(&plumeVerte);
//On dessine en vert…
…

//Fin de tracé
pDC→SelectObject(m_pOldPen);
plumeRouge.DeleteObject();
plumeVerte.DeleteObject();
```

On n'oubliera pas de mémoriser l'adresse du crayon original (ici *m_pOldPen*) qui sera restitué à la fin du tracé afin de rétablir l'état initial du contexte de périphérique comme expliqué en *2.2*.

3.2. Pinceaux

3.2.1. Définition

Un pinceau (brush en anglais) est un objet GDI permettant de remplir un contrôle, une fenêtre ou tout autre zone de l'écran.
Les pinceaux se différencient des crayons par le fait qu'ils servent à remplir une zone et non à dessiner un trait.
Les pinceaux ont plusieurs attributs :

- **La couleur**. L'une des couleurs Windows par la fonction *RGB*.
- **Le motif** (*Pattern*). Définit le motif utilisé par le pinceau.
- **Les hachures** (*Hatching*). Définit le type de hachures utilisé.

3.2.2. Classe CBrush

Pour définir, sélectionner et utiliser un pinceau, on écrira :

```
 CBrush
 *m_pOldBrush,
 pinceauA,
 pinceauB,
 pinceauC;

pinceauA.CreateSolidBrush(RGB(0,255,255));
pinceauB.CreateHatchBrush(HS_HORIZONTAL,RGB
(0,0,255));
m_pOldBrush=pDC→SelectObject(&pinceauA);

CRgn Region;
Region.CreateRectRgn(10, 10, 100, 100);
pDC→PaintRgn(&Region);
```

```
Region.DeleteObject();

pDC→SelectObject(&pinceauB);
…

CBitmap *pBip;
pBip=new CBitmap();
pBip→LoadBitmap(IDB_BITMAP1);
pinceauC.CreatePatternBrush(pBip);
pDC→SelectObject(&pinceauC);
…

pDC→SelectObject(m_pOldBrush);
pinceauA.DeleteObject();
pinceauB.DeleteObject();
pinceauC.DeleteObject();
delete pBip;
```

Contrairement aux crayons, où le style est passé comme paramètre lors de la création de l'objet, on utilise pour les pinceaux des fonctions distinctes pour différents styles. Après avoir construit l'objet *CBrush*, on peut appeler :

- *CreateSolidBrush* : La région est remplie avec une couleur sans motif ;
- *CreateHatchBrush* : On peut utiliser les motifs de hachures suivants : HS_BDIAGONAL, HS_CROSS, HS_DIAGCROSS, HS_FDIAGONAL, HS_HORIZONTAL, HS_VERTICAL ;
- *CreatePatternBrush* : La région est remplie avec le motif personnalisé défini sous la forme d'un bitmap, voir exemple *cadre 14*.

3.3. Les polices de caractères

3.3.1. Définition

Les polices sont des objets GDI tout comme les crayons et les plumes. Elles permettent de définir les caractères utilisés pour les sorties dans les programmes Windows. Une série de caractères et autres symboles ayant les mêmes attributs est une **police**.
Les polices sont gérées par Windows. Les informations concernant l'ensemble des polices installées sont stockées dans la *table des polices*.

cadre 14 : Bitmap IDB_BITMAP1.

3.3.2. Attributs d'une police

```
CFont fnt ;
fnt.CreateFont( 100,                        //Hauteur
                0,                          //Largeur
                0,                          //Angle
                0,                          //Orientation
                FW_BOLD,                    //Graisse
                TRUE,                       //Italique
                FALSE,                      //Souligné
                FALSE,                      //Barré
                ANSI_CHARSET                //Jeu de car.
                OUT_DEFAULT_PRECIS,         //Préc. sortie
                CLIP_DEFAULT_PRECIS,        //Clipping
                DEFAULT_QUALITY,            //Qualité
                DEFAULT_PITCH,              Chasse
                "Arial");                   //Nom police
```

Voir l'aide intégrée au logiciel pour les informations complémentaires.

3.3.3. Listing

Pour déclarer, définir et utiliser une fonte, on écrira :

```
CFont
 fntA, *m_pOldFont;

COLORREF clrOld;

fntA.CreateFont(100,0,0,0,FW_MEDIUM,FALSE,F
ALSE,FALSE,ANSI_CHARSET,OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,DEFAULT
_PITCH,"Comic Sans Ms");
m_pOldFont=pDC→SelectObject(&fntA);
pDC→SetTextAlign(TA_LEFT);
clrOld=pDC→SetTextColor(RGB(0,0,255));
pDC→TextOut(20,20,"Lycée J. Mermoz - 68300
ST.-LOUIS");

pDC→SelectObject(m_pOldFont);
fntA.DeleteObject();
pDC→SetTextColor(clrOld);
```

F. BOÎTES DE DIALOGUE SOUS VISUAL C++

1. Définition

Les boîtes de dialogue représentent l'une des fonctionnalités les plus importantes des applications Windows. Quel que soit le type de programme Windows que vous ayez décidé de créer, il est très probable que vous ayez à développer et à gérer des

cadre 15 : Exemple de boîte de dialogue (modale).

boîtes de dialogue.

Les boîtes de dialogue sont divisées en deux catégories principales :

- Les boîtes modales : Une boîte modale empêche l'utilisateur d'exécuter des applications tant qu'elle n'est pas refermée. Exemple : voir *cadre 15*.
- Les boîtes non modales : L'utilisateur peut agir pendant l'ouverture de la boîte de dialogue. Exemple : voir *cadre 16*, la boîte Rechercher/Remplacer de *Word*.
- Une troisième catégorie, les boîtes de message, est elle-même modale. Voir *cadre 17*. Elle est affichée à l'aide de la fonction *AfxMessageBox*.
  Exemple : AfxMessageBox("C'est le travail à faire…") ;

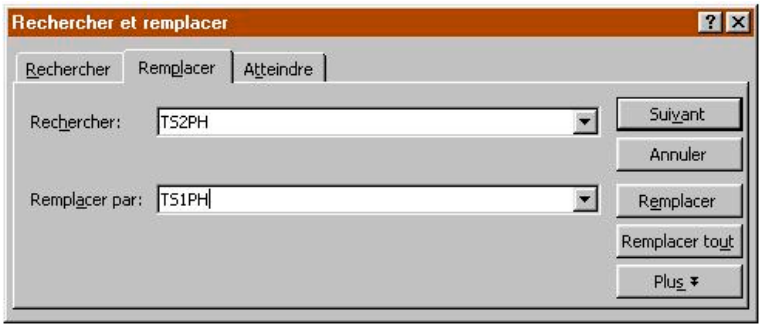
Dans la suite du cours, nous nous intéresserons aux boîtes modales.

2. Composants d'une boîte de dialogue

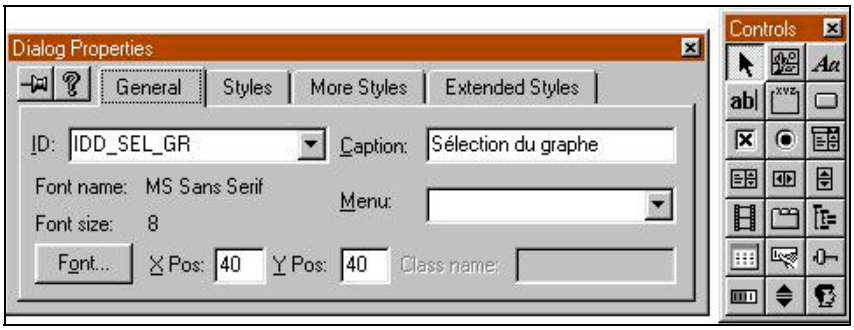
cadre 17.

Dans les programmes créés avec une architecture MFC, toute boîte de dialogue comporte deux composants :

- Une ressource qui identifie la boîte de dialogue et spécifie ses contrôles et leur emplacement dans la fenêtre de la boîte de dialogue. Voir *cadre 15* et *cadre 18*. Cette boîte de dialogue comporte : 2 boutons (Button), deux boutons radio (Radio Button), une case à cocher (Check Box), une zone de texte (Edit Box), une zone de texte statique (Static Text), deux boîtes de groupe (Group Box).
- Une classe C++ dérivée de la classe *CDialog* de la librairie MFC. Cette classe fournit l'interface permettant de gérer la boîte de dialogue : entre autres les fonctions membres et les variables membres.
  Exemple : La classe associée à la boîte de dialogue *cadre 15* s'appelle *CXDlg*.(fichiers *xdlg.cpp* et *xdlg.h*) Les variables membres et les fonctions membres sont données *cadre 19* et *cadre 20*. C'est l'utilitaire ClassWizard qui a été utilisé.



cadre 16 : Exemple de boîte de dialogue (non modale).



cadre 18 : Propriétés de la boîte de dialogue et contrôles sous Visual C++ V4.0.

3. Ouverture d'une boîte de dialogue

L'appel de la fonction *int CDialog::DoModal()* permet d'ouvrir la boîte de dialogue. On écrira :

```
#include "xdlg.h"
...
CXDlg d;
d.DoModal();
…
```

4. Affectation des variables membres

4.1. Avant appel de la fonction *DoModal*

D'après *cadre 19*,on peut écrire :

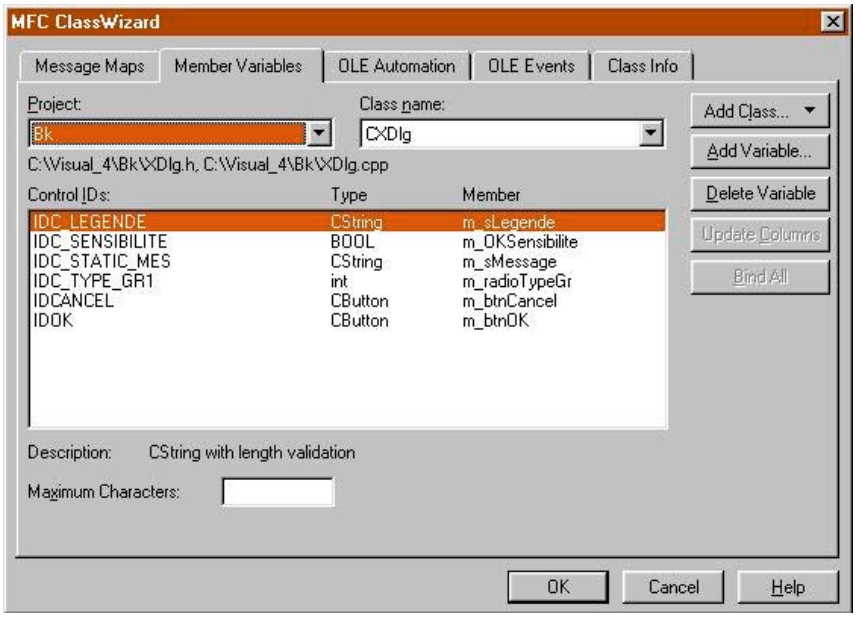
```
...
CXDlg d;
d.m_sLegende="Acquisition monochromateur
CHROMEX";
d. m_sMessage="Sélectionner le graphe à
tracer…";
d. m_OKSensibilite=TRUE;
d. m_radioTypeGr=0;
d.DoModal();
…
```

4.2. Fonction membre *OnInitDialog*

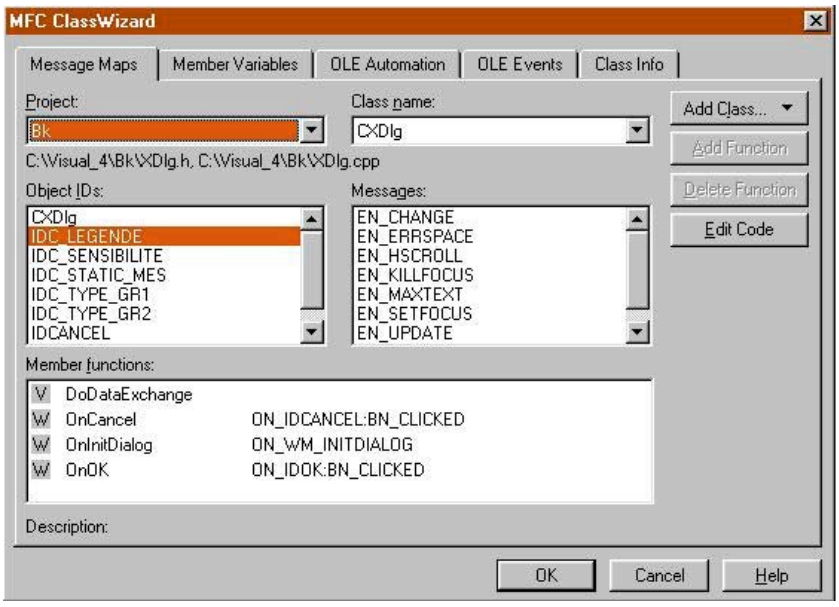
L'autre possibilité consiste à créer avec ClassWizard une fonction membre *OnInitDialog* qui sera appelée avant l'ouverture de la boîte de dialogue voir *cadre 20*.

On peut la compléter comme suit :

```
BOOL CXDlg::OnInitDialog()
{
 CDialog::OnInitDialog();
 m_sLegende="Acquisition monochromateur
CHROMEX";
 m_sMessage="Sélectionner le graphe à
tracer";
 m_OKSensibilite=TRUE;
 m_radioTypeGr=0;
 UpdateData(FALSE);
 return TRUE;
}
```



cadre 19 : Variables membres de la classe *CXDlg*.



cadre 20 : Fonctions membres de la classe *CXDlg*.

5. Prise en compte du clic sur le bouton OK

5.1. Par la valeur retournée par *DoModal*

Les valeurs retournées par la fonction *int CDialog::DoModal()* sont notamment : *IDOK*, *IDCANCEL*, *IDABORT*, …

On pourra donc écrire :

```
...
CXDlg d;
int i;
d.m_sLegende="Acquisition monochromateur
CHROMEX";
d. m_sMessage="Sélectionner le graphe à
tracer…";
d. m_OKSensibilite=TRUE;
d. m_radioTypeGr=0;
i=d.DoModal();
if (i==IDOK)
{
 //L'utilisateur à cliqué sur OK
 …
}
…
```

5.2. Par la fonction membre *OnOK*

```
void CXDlg::OnOK()
{
 UpdateData(TRUE);
 //L'utilisateur à cliqué sur OK
 ...

 CDialog::OnOK();
}
```

6. Mécanisme DDX/DDV

C'est le mécanisme qui gère l'échange des données entre les variables membres et les contrôles de la boîte de dialogue. Ce mécanisme est géré par la fonction *UpdateData*.

*UpdateData(**TRUE**) ;* → Mise à jour des variables membres de la boîte de dialogue avec les données des contrôles.

*UpdateData(**FALSE**) ;* → Mise à jour des contrôles avec le contenu des variables membres de la boîte de dialogue .

Exemple : L'exemple ci-dessous vérifie la valeur de la longueur d'onde *m_fLambda* (en nm) introduite par l'utilisateur dans une zone de texte et lui affecte la valeur de 632.8 nm en cas d'erreur.

```
void CXDlg::OnOK()
{
 UpdateData(TRUE);
 if (m_fLambda<100 || m_fLambda>1000)
 {
  m_fLambda=632.8
 }
 UpdateData(FALSE);
 ...

 CDialog::OnOK();
}
```

7. Pointeur sur l'application, le document et la vue

Les lignes ci-dessous permettent d'obtenir le pointeur sur l'application, le document ou la vue de n'importe quel endroit du programme, notamment dans les fonctions membres d'une boîte de dialogue.

L'application est gérée par les fichiers *Bk.cpp* et *Bk.h*.

Le(s) document(s) est(sont) géré(s) par les fichiers *BkDoc.cpp* et *BkDoc.h*.

La(les) vue(s) est(sont) gérée(s) par les fichiers *BkView.cpp* et *BkView.h*.

```
BOOL CXDlg::OnInitDialog()
{
 //Pointeur sur l'application
 class CBkApp *pApp;
 pApp=((CBkApp*)AfxGetApp());

 //Pointeur sur le document
 //En SDI:
 CFrameWnd *pFrame=(CFrameWnd
*)(AfxGetApp()->m_pMainWnd);
 pDoc=(CBkDoc*)pFrame->
GetActiveDocument();

 //En MDI:
 CMDIChildWnd * pChild= ((CMDIFrameWnd*)
(AfxGetApp()-> m_pMainWnd))->
MDIGetActive();
 if (pChild) pDoc=(CBkDoc*)pChild->
GetActiveDocument();

 //Pointeur sur la vue_____________________
 //En SDI:
 //CFrameWnd *pFrame=(CFrameWnd *)
(AfxGetApp()->m_pMainWnd);
 CView *pView=pFrame->GetActiveView();
 CBkView *pView=(CBkView*)pFrame->
GetActiveView();
 //pDoc=pView->GetDocument();
 //En MDI:
 CMDIChildWnd * pChild=
((CMDIFrameWnd*)(AfxGetApp()-> m_pMainWnd))
-> MDIGetActive();
 CView *pView=pChild->GetActiveView();
 //…
}
```

Remarque :

Une application SDI (Single Document Interface) comporte une seule fenêtre. Une application MDI (Multiple …) comporte plusieurs fenêtres.

Il peut y avoir plusieurs documents et vues, mais seulement une seule application.

G. EXÉCUTION EN TÂCHE DE FOND (TIMER)

Dans ce chapitre, vous allez faire connaissance avec une technique qui permet d'exécuter du code en « tâche de fond ». Cette technique repose sur l'implémentation d'un timer (ou temporisateur). Depuis la mise à jour régulière d'un contrôle, jusqu'à la réalisations d'acquisitions des mesures, les timers peuvent être utilisés dans de nombreux domaines où un traitement régulier et non bloquant doit être effectué.

1. Un peu de théorie

La mise en place d'un timer est très simple. Il suffit d'appeler la fonction *SetTimer*, dont voici le prototype :

unsigned int SetTimer(int Ident, unsigned int Intervalle, void Traitement()) ;

avec :

- *Ident* : entier sans signe qui identifie le timer.
- *Intervalle* : intervalle de temps en ms entre deux appels du timer *WM_TIMER*. Raisonnablement, cette valeur ne peut pas être inférieure à 50 ms.
- *Traitement* est le nom de la fonction chargée du traitement des messages *WM_TIMER*. Généralement, ce paramètre vaut NULL, et c'est la fonction *OnTimer* de l'application qui se charge du traitement des messages *WM_TIMER*.

La valeur renvoyée par la fonction *SetTimer* est différente de zéro si le timer a pu être déclaré. Elle est égale à zéro dans le cas contraire.

Exemple : Déclaration du timer n°1 avec appel de la fonction *OnTimer* toutes les 100 ms.

SetTimer(1,100,NULL) ;

Lorsqu'un timer n'est plus utilisé, ou lorsqu'il doit être suspendu, utilisez la fonction *KillTimer*, dont voici le prototype :

BOOL KillTimer(int Ident) ;

avec :

- *Ident* : entier sans signe qui identifie le timer.

La valeur renvoyée par la fonction *KillTimer* est vraie (TRUE) ou fausse (FALSE).

Exemple : Suppression du timer n°1 précédent.

KillTimer(1) ;

2. Application

Faire 1000 acquisitions de tensions toutes les 50 ms sur une carte analogique/numérique *Candibus*, avec possibilité de les interrompre à tout moment.

On clique sur le bouton Démarrer de la boîte de dialogue *AcqDlg* pour lancer les acquisitions :

```
void CAcqDlg::OnDemarrer()
{
 m_iNb=0;//Initialiser le nombre d'acq.
 SetTimer(1,50,NULL);//Toutes les 50 ms
}
```

Toutes les 50 ms, la fonction *OnTimer* est appelée, et une tension *u* est lue sur la carte *Candibus*.

```
void CAcqDlg::OnTimer(UINT nIDEvent)
{
 float u;
 class Ccandibus *m_pCandi;
 m_pCandi=new CCandibus();
 u=m_pCandi->LitTension(0,0);
 //Mettre les tensions dans un tableau…
 m_iNb++;//Incrémenter le nb. de mesures
 if (m_iNb==1000) KillTimer(1);
 delete m_pCandi;
}
```

On peut arrêter à tout moment par un clic sur le bouton Arrêter.

```
void CAcqDlg::OnArreter()
{
 Killtimer(1);
//…
}
```

SOMMAIRE

Chapitre	Page
A. Les fondements de la programmation	2
1. Documentation en ligne du logiciel Visual C++	2
2. Avant de programmer : Faire l'analyse du problème	2
B. Les bases	3
1. Les données du programme	3
1.1. Notion de variable	3
1.2. Notion de type	4
1.2.1. Types usuels	4
1.2.2. Déclaration	4
1.3. Opérateurs	5
1.4. Variables booléennes	7
1.5. Variables caractères	7
1.5.1. Particularité des variables de type *char* et *unsigned char*	7
1.5.2. Jeux de caractères	7
1.5.3. Caractères spéciaux	9
1.6. Tableaux et chaînes de caractères	9
1.6.1. Déclaration	9
1.6.2. Indice	10
1.6.3. Affectation	10
1.6.4. Chaînes de caractères	10
1.6.5. Chaînes de caractères sous Visual C++	10
1.6.6. Fonction *CString::Format*	11
2. Fonctions	12
2.1. Définition	12
2.2. Prototype d'une fonction	12
2.3. Corps d'une fonction	12
3. Instructions de sélection et d'itération	13
C. Variables et pointeurs	14
1. Concept	14
2. Définition	14
3.Initialisation	14
3.1. Pointeur su objet existant	14
3.2. Pointeur sur nouvel objet. Opérateur *new* et *delete*	15
4. Pointeurs sur les variables membres et les fonctions membres d'une classe	15
D. les classes C++ et MFC	16
1. Définition d'une classe	16
1.1. Exemple	16
1.2. Création d'une instance d'une classe	16
1.2.1. Variable	16
1.2.2. Pointeur	16
2. Spécificateurs d'accès	16
3. Fonction membre constructeur	16
4. Fonction membre destructeur	17
5. Classe dérivée	17
E. Contextes de périphériques	18
1. Définition	18
1.1. Objets GDI et contextes de périphériques (Pen ,Brush, Bitmap, Font)	18
1.2. Indépendance par rapport aux périphériques	18
2. Utilisation des contextes de périphériques	18
2.1. MFC, ClassWizard et les contextes de périphériques	19
2.2. Sélection d'un objet GDI	19
2.3. Choix d'un système de coordonnées pour le contexte de périphérique	19
2.4. Définition du système de coordonnées pour dessiner dans un repère orthonormé	20
2.4.1. Listing	20
2.4.2. Fonctions *SetMapMode, SetWindowExt, SetWindowOrg*	20
2.5. Dessin dans la fenêtre	21
3. Objets GDI	21
3.1. Crayons	21
3.1.1. Définition	21
3.1.2. Classe *CPen*	21

Chapitre	Page
3.2. Pinceaux	21
3.2.1. Définition	21
3.2.2. Classe CBrush	21
3.3. Les polices de caractères	22
3.3.1. Définition	22
3.3.2. Attributs d'une police	22
3.3.3. Listing	22
F. Boîtes de dialogue sous Visual C++	23
1. Définition	23
2. Composants d'une boîte de dialogue	23
3. Ouverture d'une boîte de dialogue	24
4. Affectation des variables membres	24
4.1. Avant appel de la fonction *DoModal*	24
4.2. Fonction membre *OnInitDialog*	24
5. Prise en compte du clic sur le bouton OK	25
5.1. Par la valeur retournée par *DoModal*	25
5.2. Par le fonction membre *OnOK*	25
6. Mécanisme DDX/DDV	25
7. Pointeur sur l'application, le document et la vue	25
G. Exécution en tâche de fond (Timer)	26
1. Un peu de théorie	26
2. Application	26

INDEX ALPHABÉTIQUE

Mot clé	Page
Adresse mémoire	14
Affectation (symbole =)	3
Aide en ligne de Visual C++	2
Analyse du problème	2
Arguments d'une fonction	12
Boîte de dialogue	23
Booléen	7
Caractères	7
Caractères spéciaux	9
CFont	22
Chaînes de caractères	4, 10
char	7
Classes C++ et MFC	16
Constantes	3
Constructeur	16
Contextes de périphériques	18
Corps d'une fonction	12
CFont	21
CPen	21
CString	22
delete	15
Destructeur	17
Dérivation d'une classe	17
do	13
Données du programme	3
else	13
Entiers	4
Fonctions	12
Fontes	19
for	13
Format (formater une chaîne de caractères)	11
if	13
Instructions de sélection et d'itération (*if, then, else, while, do, …*)	13
int	4
Jeu de caractères ASCII	7, 8, 9
long	4
MFC	16
new	15
Objets GDI	21
Opérateurs	5, 6
Ouverture d'une boîte de dialogue	24
Paramètres d'une fonction	12
Pen (crayon)	18
Pointeurs	14
Pointeur sur document, vue et application	26
Privé ou public	16
Prototype d'une fonction	12
Réels	4
short	4
Systèmes de coordonnées	19
Tableaux	9
Timer	16
Types de données standards	4
unsigned	4
Variables	3, 4
Variables membres d'une boîte de dialogue	24
void	12
while	13