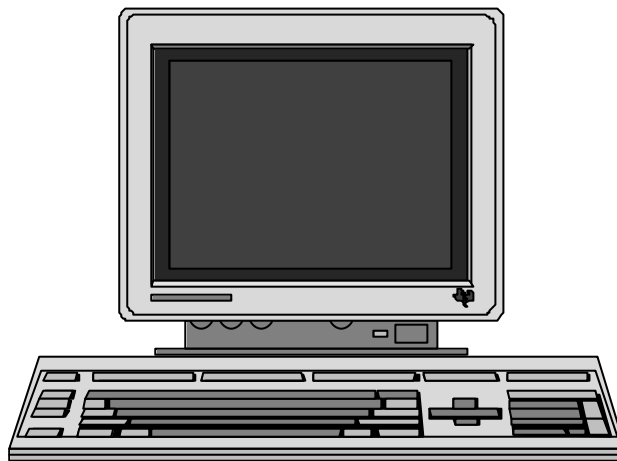


UNIVERSITE de CERGY

*Programmation Windows
En Borland C*



José GILLES - 1998 / 99

IUP GENIE ELECTRIQUE

Mes sincères remerciements à P.D. MATTHEY
et F. PALANGIE qui ont contribué à la préparation de ce polycopié.

Merci d'avance à tous ceux qui voudront bien
me faire part de leurs remarques et suggestions
visant à améliorer cette seconde édition.

§ I) Introduction

- I) Les environnements de développement sous Windows
 - 1) Programmation en C via l'API
 - 2) Programmation en C++ via OWL
 - 3) Comparatif
- II) Les principes de Windows
- III) Le but poursuivi
- IV) L'interface du Borland C 4.5 pour Windows

§ II) Création d'un programme Windows

- I) Création d'un programme Windows élémentaire
 - 1) Création du projet
 - 2) Code source
- II) Le fonctionnement de ce programme

§ III) Les Ressources

- I) Les ressources Windows
- II) Création d'un menu
 - 1) Création du sous projet de ressources
 - 2) Création de la ressource menu
 - 3) Intégration du menu au programme
- III) Création d'une boîte de dialogue
 - 1) Les boîtes de dialogue
 - 2) Création de la ressource boîte de dialogue
 - 3) Utilisation de la boîte de dialogue
- IV) Création d'une icône personnalisée
 - 1) Création d'une ressource icône
 - 2) Intégration de l'icône au programme
- V) Création d'une boîte de dialogue non modale
 - 1) Les boîtes non modales
 - 2) Création de la ressource boîte à outil
 - 3) Utilisation de la boîte à outils
- VI) Boîte de dialogue comme fenêtre principale
 - 1) Principe
 - 2) Création de la ressource
 - 3) Utilisation
- VII) Pilotage des contrôles d'une boîte de dialogue
 - 1) Rappels
 - 2) Activation des boutons poussoirs
 - 3) Boutons poussoirs personnalisés
 - 4) Barres de défilement
 - 5) Les menus
 - 6) Les contrôles 3D Borland

- VIII) Boites listes et combinées
 - 1) Présentation
 - 2) Création de la ressource
 - 3) Utilisation d'une boîte liste
 - 4) Utilisation d'une combo-box

§ IV) Texte et graphisme

- I) Le GDI Windows
 - 1) Définition
 - 2) Le Device Context
- II) Le texte
 - 1) Les polices
 - 2) Affichage de texte
- III) Le graphisme
 - 1) Crayons et brosses
 - 2) Les fonctions de tracé
 - 3) Les fonctions de remplissage
- IV) Application au programme démineur
 - 1) Principes
 - 2) Réalisation

§ V) Clavier et souris

- I) Introduction
- II) Le clavier
 - 1) Principes
 - 2) La table de caractères Windows
- III) La souris
 - 1) Principes
 - 2) Capture de souris
 - 3) Curseur souris
- IV) Application au démineur

§ VI) Manipulations de fichiers

- I) Manipulations de fichiers
 - 1) Fonctions de base
 - 2) Fonctions 16 bits
 - 3) Fonctions 32 bits
- II) Boites de dialogue prédéfinies Open / Save

§VII) Allocation dynamique et images Bitmap

- I) Allocation dynamique
- II) Images Bitmap

§VIII) L'impression des documents

- I) Les principes
 - 1) La gestion des documents
 - 2) Programmation
 - 3) Les problèmes d'échelle
- II) Les boîtes de dialogue Print et Print Setup

PROGRAMMATION EN C SOUS WINDOWS

IUP GE 2EME ANNEE

Matériel requis: Un PC fonctionnant sous Windows 3.1 ou Windows 95
Environnement: BORLAND C pour Windows 4.0 ou supérieur

D) Les environnements de développement pour Windows:

On suppose que le lecteur est familier du langage C et de la programmation sous DOS en Turbo C, afin de faciliter les explications et comparaisons ultérieures.

Il existe de nombreux environnements de développement sous Windows mettant en oeuvre divers langages de programmation; citons par exemple:

- Microsoft Visual Basic
- Microsoft Visual C/C++
- Borland C/ C++
- Windev et autres générateurs d'applications...

Nous avons retenu ici Borland C/C++ pour Windows — BCW en abrégé — version 4.0 ou supérieure, qui fonctionne sous Windows 3.1, Windows 95 et Windows NT et permet de générer des applications 16 ou 32 bits pour toutes ces plates-formes (et aussi pour MSDOS). Les applications 32 bits utilisant des instructions machine, des registres et un adressage sur 32 bits ne fonctionnent sous Windows 3.1 qu'à la condition d'installer l'extension Win32s.

BCW permet de développer sous Windows de deux façons assez différentes que nous allons comparer:

1) Programmation en C via l'API :

Dans cette utilisation la programmation a lieu en C uniquement et repose sur l'appel direct de l'API (Application Programming Interface) de Windows qui regroupe les fonctions mises par Windows à la disposition du programmeur pour réaliser les tâches élémentaires nécessaires.

2) Programmation en C++ via OWL:

Dans ce cas la programmation se fait en C++ et repose sur la bibliothèque de classes Object Windows Library (OWL) de Borland — équivalente aux Microsoft Foundation Classes (MFC) de Visual C++ — qui remplace l'API traditionnelle et offre des compléments intéressants.

3) Comparatif:

Si l'approche Object Windows offre de nombreuses classes et méthodes prédéfinies fort utiles, elle a aussi des inconvénients pour une première approche:

- Elle nécessite la connaissance du C++ orienté objet
- Elle masque le fonctionnement de Windows là où l'API le met en évidence
- Elle génère des exécutables titanesques ou nécessite des DLL externes

En somme, l'usage de l'API semble préférable dans un but didactique, l'aspect OWL pouvant être abordé ultérieurement pour la réalisation de programmes plus complexes ou plus professionnels.

La différence entre ces deux façons de procéder se manifeste par l'utilisation de fichiers sources d'extensions .C ou .CPP et par l'inclusion de headers différents permettant l'emploi des fonctions de l'API ou de OWL.



Attention: Dans tout ce qui suit nous utiliserons uniquement l'API Windows et le langage Borland C.

II) Les principes de Windows:

Le fonctionnement même de Windows rend la programmation très différente de ce qu'elle peut être sous MSDOS. En effet Windows est un environnement plus ou moins multitâche et multifenêtré dans lequel plusieurs processus peuvent coexister à un instant donné, et entre lesquels il est possible de commuter facilement.

Les entités de base sous Windows sont les applications et les fenêtres; en fait une même application pouvant être lancée plusieurs fois on parle plutôt d'instance d'une application pour identifier un processus s'exécutant sous Windows.

Chaque instance d'application possède une ou plusieurs fenêtres qui permettent l'interaction avec l'utilisateur via des drivers de clavier, de souris, de vidéo...

Notons au passage que Windows masque à travers ces drivers les spécificités du matériel afin qu'un même programme fonctionne sur des machines différentes.

L'affichage dans une fenêtre passe comme nous le verrons plus loin par un contexte de périphérique manipulé à l'aide de fonctions de l'API. Les saisies au clavier et les manipulations de la souris ne sont pas gérées directement par le programme. Windows génère lors de ces événements des messages qui sont placés dans une file d'attente associée à l'application en attendant d'en être

extraits par celle-ci puis traités par une fonction particulière associée à la fenêtre concernée, nommée procédure de fenêtre.

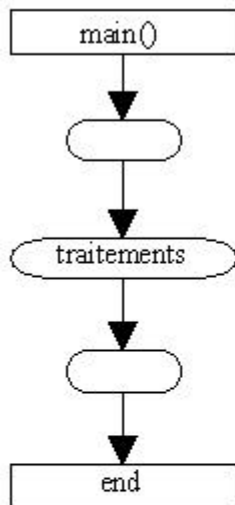
La structure générale d'un programme Windows est donc différente de celle d'un programme MSDOS:

Dans un programme MSDOS la fonction main() déroule linéairement le code du programme et appelle les fonctions nécessaires.

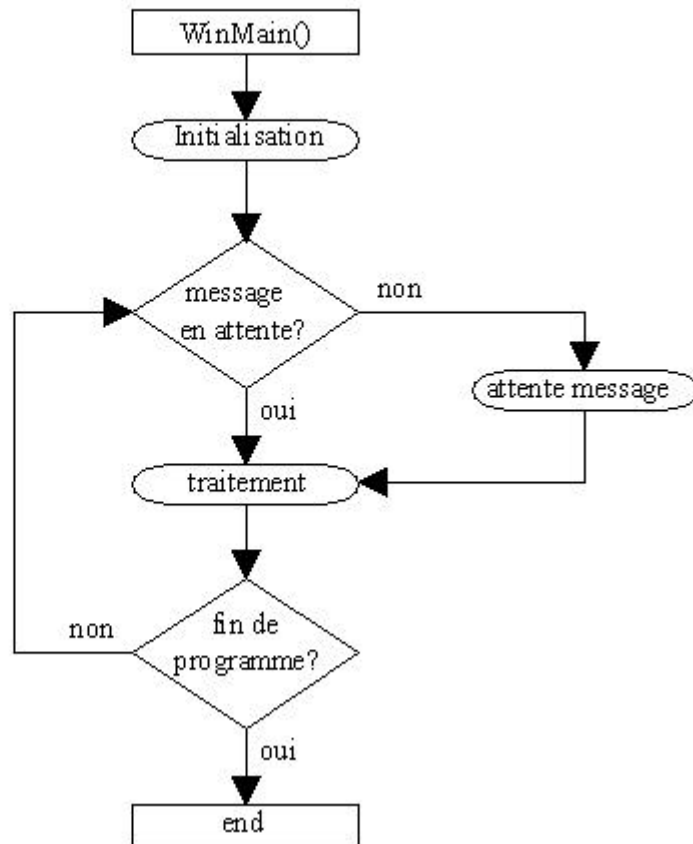
Dans un programme Windows la fonction WinMain() se contente d'initialiser l'instance d'application et les fenêtres nécessaires, puis se place dans une boucle d'attente de message qui ne se terminera qu'en quittant l'application. Les messages concernant les fenêtres de l'application sont extraits de la file d'attente dans cette boucle et sont transmis indirectement aux procédures de fenêtre; l'appel est indirect dans la mesure où ce n'est pas le programme qui appelle la procédure de fenêtre, mais Windows, qui en a mémorisé l'adresse lors de l'initialisation de l'application.

Cette approche peut paraître inutilement compliquée, mais elle répond en fait à des nécessités: d'une part le principe de file d'attente permet durant le traitement d'un message par la procédure de fenêtre de continuer à réceptionner d'autres messages, d'autre part Windows se réserve ainsi la possibilité, en cas d'urgence, d'envoyer certains messages directement à la procédure de fenêtre en court-circuitant la file d'attente.

Programme MSDOS



Programme Windows



Windows apporte encore d'autres innovations telles que les ressources parmi lesquelles:

- Les menus déroulants
- Les icônes de programme
- Les boîtes de dialogue avec boutons, cases, zones de saisie...

L'emploi de ressources allège la programmation en C en reportant la description de ces éléments dans des scripts de ressources annexes et en laissant à Windows le soin de gérer ces éléments à l'écran. Les scripts de ressources sont des fichiers ASCII d'extension .RC qui sont compilés à part en des fichiers de ressources binaires .RES qui peuvent ensuite être liés au programme .EXE. Ainsi les icônes associées à un programme sont elles puisées dans ses ressources.

Un éditeur de ressources intégré au BCW nommé Resource Workshop permet de créer facilement les menus , icônes, boîtes de dialogue, etc. dont l'usage est ensuite assez simple.

Signalons enfin parmi les spécificités de Windows la possibilité de créer et d'utiliser des fichiers DLL — Dynamic Link Library — permettant d'alléger le programme .EXE en plaçant certains groupes de fonctions dans des fichiers externes d'extension .DLL.

Une DLL est chargée à la demande lorsque l'on a besoin d'accéder aux fonctions qu'elle contient. Une même DLL peut être partagée par plusieurs processus, elle n'est alors chargée qu'une seule fois en mémoire par Windows.

Exemple: Voir § Réseau pour l'appel des fonctions d'une DLL.

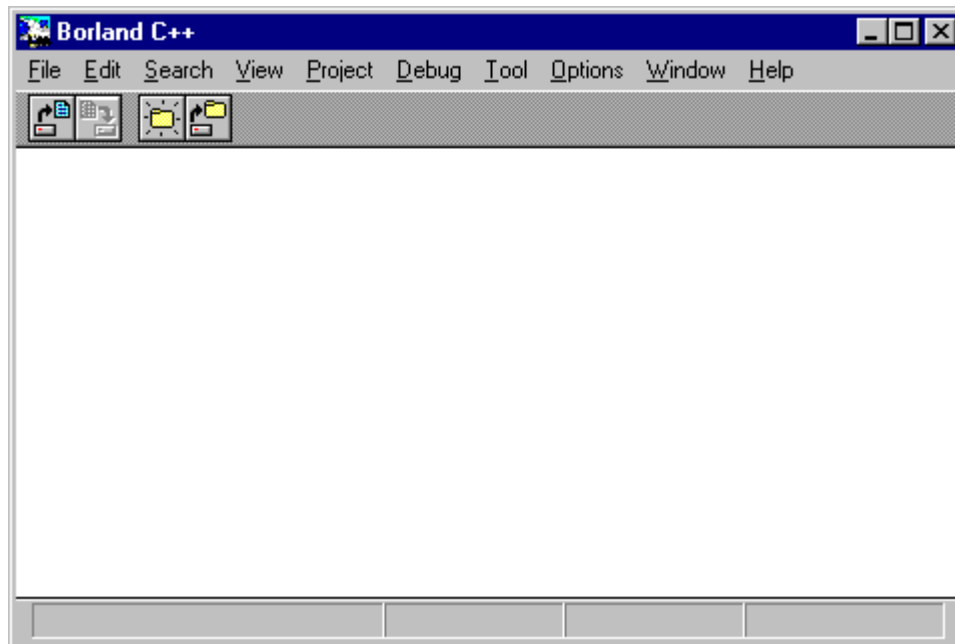
III) Le but poursuivi:

Le but de ce polycopié est non seulement d'exposer les principes de la programmation Windows mais aussi de réaliser concrètement des applications mettant en oeuvre les concepts les plus importants (Loin de nous l'idée d'être exhaustifs...)

Dans les chapitres qui suivent nous allons donc progressivement réaliser, dans un premier temps, un programme comparable au démineur de Windows (Winmine.exe); dans un deuxième temps nous réaliserons un programme plus complexe de visualisation d'images à partir de fichiers BMP.

IV) L'interface du Borland C 4.5 pour Windows:

L'interface utilisateur de Borland C 4.5 pour Windows se présente au démarrage sous la forme suivante dans la version US, sous Windows 95:



Les menus les plus utiles sont les suivants:

Menu File:	New:	Crée un nouveau fichier source.
	Open:	Ouvre un fichier source existant.
	Save:	Sauvegarde le fichier en cours d'édition.
Menu Edit:	Cut:	Coupe le texte enluminé.
	Copy:	Copie une portion de texte enluminée.
	Paste:	Colle ce qui a été coupé ou copié.
Menu Search:	Search:	Recherche un mot clé.
	Replace:	Remplace un mot par un autre.
Menu Project:	New project:	Crée un nouveau fichier de projet .EDI
	Open project:	Ouvre un fichier de projet existant.
	Close project:	Ferme le projet en cours.
	New target:	Définit un nouvel exécutable à générer (cible) à partir des fichiers sources voulus.
	Compile:	Compile le fichier source en cours d'édition.
	Make:	Recompile les fichiers modifiés et génère les exécutables cibles correspondants.
	Build all:	Recompile tous les fichiers et recrée tous les exécutables cibles.
Menu Debug:	Run:	Lance l'exécution du programme sélectionné.
Menu Tool:	Resource	

Workshop: Lance l'éditeur de ressources Borland.

Menu Options:

Project: Permet de paramétrer finement la compilation.

Environment: Permet de paramétrer l'interface de Borland C.

Menu Help:

Windows

API: Documentation de l'API Windows, indispensable car non documentée sur papier.

OWL API Documentation de la bibliothèque Object Windows.

D) Création d'un programme Windows élémentaire:

Pour créer un premier programme Windows élémentaire, suivre le guide:

1) Création du projet:

- a) Lancer Borland C sous Windows, ouvrir le menu File → New et saisir le texte du source C nommé WSTEP1.C dont le listing est ci-après. Sauvegarder via File → Save ce fichier sous le nom WSTEP1.C
- b) Ouvrir le menu Project → New Project et compléter les champs comme suit:

Path & Name =	C:\. . . \WSTEP1.IDE Adapter le chemin d'accès au fichier
Target Name =	WSTEP1 Nom de l'exécutable
Target Type =	Application
Platform =	Windows 3.x ou Win32 Selon la version de Windows visée
Target Model =	Medium ou Large ou GDI
Standard Libraries =	Runtime + Static

Faire OK en fin de saisie.

- c) Dans la fenêtre Project qui vient d'apparaître, supprimer les fichiers proposés par défaut à savoir WSTEP1.CPP, WSTEP1.RC, et WSTEP1.DEF en pressant sur . Insérer à la place le fichier WSTEP1.C en pressant sur <INS>.
- d) Lancer dans le menu Project → Compile puis Project → Make all qui doivent se solder par un message «Success» malgré trois warnings sans grande importance.
- e) Lancer alors Debug → Run
Le programme doit apparaître, avec une fenêtre principale et une petite fenêtre affichant le message «Bonjour, bienvenue dans Windows».
- f) Refermer la fenêtre de bienvenue puis terminer l'application en fermant la fenêtre principale.

2) Code source:

Voir pages suivantes le code source de WSTEP1.C

```

/*
 * Programmation Windows en C via API -- by JG
 *
 * A compiler pour Win32 sous Borland C++ 4.5
 *
 */

#include <windows.h>

/* declaration des fonctions */
int WINAPI WinMain(HANDLE, HANDLE, LPSTR, int);
BOOL InitApplication(HANDLE);
BOOL InitInstance(HANDLE, int);
LRESULT CALLBACK MainWndProc(HWND, UINT, WPARAM, LPARAM);

HANDLE hInst;          /* handle d'instance d'application */
HWND hWnd;            /* handle de fenetre */

BOOL InitApplication(HANDLE hInstance)
/* initialise l'application */
{
    WNDCLASS wc;      /* structure de description de fenetre */

    wc.style = CS_DBLCLKS;
    wc.lpfWndProc = (WNDPROC) MainWndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;          /* nom de ressource ou NULL */
    wc.lpszClassName = "WSTEP1CLASS"; /* nom de classe unique */

    return(RegisterClass(&wc));
}

BOOL InitInstance(HANDLE hInstance, int nCmdShow)
/* initialise cette instance */
{
    hInst = hInstance;      /* memorise handle d'instance */

    /* Creation de la fenetre principale d'application */
    /* et emission d'un message WM_CREATE */
    hWnd = CreateWindow("WSTEP1CLASS", "Programmation Windows - Exemple",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

    if (hWnd == NULL)
        return (FALSE);
}

```

```

ShowWindow(hWnd, nCmdShow);      /* affichage fenetre */
UpdateWindow(hWnd);              /* genere un msg WM_PAINT pour zone client */
return (TRUE);
}

```

int WINAPI

WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)

/* fonction principale du programme */

```

{
MSG msg;                          /* variable de message */

/* teste si existe une autre instance de l'application */
if (hPrevInstance == NULL)
    /* si non initialise application */
    if (!InitApplication(hInstance))
        return(1);

/* initialise cette instance de l'application*/
if (!InitInstance(hInstance, nCmdShow))
    return(2);

/* message de bienvenue */
MessageBox(hWnd, "Bienvenue dans Windows", "Bonjour", MB_OK);

/* boucle de lecture des messages */
/* jusqu'a WM_QUIT */

while (GetMessage(&msg, NULL, 0, 0))
    {
    TranslateMessage(&msg);        /* conversion du message */
    DispatchMessage(&msg);       /* envoi du message */
    }

return(msg.wParam);              /* code de retour */
}

```

LRESULT CALLBACK

MainWndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)

/* procedure de fenetre */

```

{
HDC hdc;                          /* contexte de peripherique */
PAINTSTRUCT ps;

/* routine de gestion des messages entrants */
switch (message)
    {
    /* genere a la creation de la fenetre */
    case WM_CREATE:
        {
        }
        break;

/* Mise a jour de la fenetre client */

```

```

case WM_PAINT:
    {
        hdc = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
    }
    break;

/* fermeture de la fenetre principale */

case WM_CLOSE:
    {
        /* qui induit la fin de l'application */
        DestroyWindow(hWnd);
    }
    break;

/* genere par la destruction de la fenetre principale */
case WM_DESTROY:
    {
        /* emet le message WM_QUIT qui met fin a l'application */
        PostQuitMessage(0);
    }
    break;

default:
    {
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
}
return(0);
}

```

II) Le fonctionnement de ce programme:

Le programme WSTEP1.C commence très classiquement par l'inclusion du header <windows.h>, qui est le seul indispensable dans l'immédiat; il déclare une multitude de fonctions, de types, de structures, de constantes...

Ensuite viennent les déclarations de fonctions et de variables.

Notons au passage quelques types et constantes prédéfinis couramment utilisés:

BOOL	booléen TRUE ou FALSE
UINT	entier court non signé
LONG	entier long signé
WORD	entier court non signé
DWORD	entier long non signé
HANDLE	descripteur d'application
HWND	descripteur de fenêtre
LPSTR	pointeur long sur chaîne de caractères

style:	Style de la fenêtre, par exemple: CS_DBLCLKS fenêtre gérant les doubles clics de souris. CS_NOCLOSE fenêtre que l'on ne peut refermer manuellement. CS_ . . . Ces styles — Voir Aide de BCW pour la liste complète — sont combinables par l'opérateur
lpfnWndProc:	Pointeur sur la procédure de fenêtre responsable de la gestion des messages de cette classe de fenêtres; il suffit d'y mettre le nom de cette fonction.
cbClsExtra:	Option inutile ici donc égale à 0.
cbWndExtra:	Idem.
hInstance:	Handle de l'instance d'application en cours.
hIcon:	Handle de la ressource icône associée au programme; on choisit pour le moment une icône par défaut via LoadIcon().
hCursor:	Idem pour le Curseur de souris via LoadCursor().
hbrBackground:	Handle de la brosse de couleur utilisée pour remplir le fond de la fenêtre ; ici une brosse blanche fournie par GetStockObject().
lpszMenuName:	Nom de la ressource Menu associée à la fenêtre, pour le moment aucune donc champ NULL.
lpszClassName:	Nom attribué à cette classe de fenêtres, qui doit en principe être unique et généralement lié à l'application.

Ensuite on appelle InitInstance() dont le rôle consiste à créer la fenêtre principale de l'application. On utilise pour cela les fonctions suivantes:

```

syntaxe:        HWND CreateWindow( LPCSTR class, LPCSTR title, DWORD style,
                                  int x, y, nx, ny, HWND hParent, HMENU hMenu,
                                  HINSTANCE hInst, void far * ptr );

                ShowWindow( HWND hWnd, int nCmdShow );
                UpdateWindow( HWND hWnd );

```

On passe ici à CreateWindow() les paramètres suivants:

class:	Le nom de la classe de la fenêtre enregistrée plus haut.
title:	Le titre de la fenêtre.
style:	Un attribut de style de fenêtre obtenu par combinaison avec de constantes prédéfinies telles que: WS_OVERLAPPED fenêtre avec titre et bordure.

WS_OVERLAPPEDWINDOW

	idem plus menu système.
WS_HSCROLL	fenêtre avec barre de défilement horizontal.
WS_VSCROLL	fenêtre avec barre de défilement vertical.
WS_MAXIMIZE	fenêtre plein écran.
WS_ . . .	

x, y:	La position initiale de la fenêtre à l'écran.
nx, ny:	Les dimensions de la fenêtre à l'écran; ici des valeurs par défaut.
hParent:	Le handle de la fenêtre parent, NULL si aucune.
hMenu :	NULL si aucun ou si menu de la classe associée.

La fonction ShowWindow() se charge ensuite d'afficher la fenêtre ainsi créée. (Ce n'est pas indispensable si le style de fenêtre contient CS_VISIBLE).

Il y a plusieurs modes possibles, tels que:

SW_SHOW	fenêtre de la taille et à la position prévues.
SW_SHOWMAXIMIZED	fenêtre plein écran.
SW_SHOWMINIMIZED	fenêtre réduite en icône.
SW_ . . .	

On utilise en fait ici le mode demandé au lancement du programme.

Enfin UpdateWindow() génère un message WM_PAINT afin que la zone utile de la fenêtre, nommée zone client, soit actualisée par la procédure de fenêtre (Voir plus loin).

Passée cette initialisation le programme affiche un mot de bienvenue puis rentre dans sa boucle de messages. On y utilise les fonctions:

Syntaxe: GetMessage(MSG FAR * msg, HWND hWnd, UINT first, UINT last);
 TranslateMessage(MSG FAR * msg);
 DispatchMessage(MSG FAR * msg);

Le rôle de GetMessage() est d'extraire un message msg de la file d'attente de l'application qui soit destiné à la fenêtre référencée par le handle hWnd. (Les autres paramètres sont en général nuls).

GetMessage() renvoie 0 si le message extrait est WM_QUIT qui signifie la fin du programme, et une valeur non nulle dans tous les autres cas.

TranslateMessage() se charge de convertir certains messages claviers comme nous le verrons plus tard...

DispatchMessage() demande alors à Windows de transmettre le message définitif à la procédure de fenêtre concernée.

déclenche alors un appel à un appel à DestroyWindow() donc un message WM_DESTROY.

Lors du passage suivant dans la boucle de message, cela produira un appel à PostQuitMessage(), donc un message WM_QUIT dont on sait qu'il mettra un terme à la boucle de message et terminera le programme !!!

Il ne faut donc pas moins de trois messages pour terminer l'application...

Syntaxe: DestroyWindow(HWND hWnd);
 PostQuitMessage(int code);

DestroyWindow() se contente de détruire la fenêtre indiquée.

PostQuitMessage() demande la terminaison de l'application.

Le paramètre code de PostQuitMessage() est le code de retour du programme, en principe repris comme code de retour de la fonction WinMain().

I) Les ressources Windows:

Tout programme Windows peut intégrer des ressources telles que:

- Menu, raccourcis clavier
- Boite de dialogue
- Icône, curseur, bitmap
- Tableau de chaînes de caractères

Les ressources sont assez faciles à créer à l'aide de l'éditeur de ressources Borland Resource Workshop.

Elles sont faciles d'emploi également car prises en charge par Windows lui-même.

Ainsi un menu créé en quelques minutes est lié à l'application et sa prise en charge concernant les déroulements, clics de souris est du ressort de Windows, qui émet des messages (Encore !) pour informer le programme des actions réalisées dans le menu.

De même les boîtes de dialogue permettent un dialogue interactif entre l'application et l'utilisateur au moyen de boutons, cases à cocher, champs, listes...

Les icônes, curseurs, bitmaps servent quant à eux à associer une icône personnalisée au programme, un curseur de souris spécifique, ou à intégrer des graphiques bitmap.

Enfin les tableaux de chaînes de caractères servent par exemple à développer des logiciels multilingues dans lesquels les messages à traduire sont relégués au niveau des ressources pour ne pas avoir à modifier le code source de l'application !

Toutes ces ressources se présentent sous la forme de scripts sources ASCII d'extension .RC associés à des fichiers d'en-tête d'extension .RH

Les scripts peuvent être créés dans l'éditeur ou générés — de préférence — par l'outil Resource Workshop.

Lors de la compilation du projet, les scripts de ressources .RC sont compilés au format ressource binaire .RES puis linkés au programme .EXE

II) Création d'un menu:

Nous allons ajouter, pour commencer, un menu au programme de l'étape précédente. Pour cela, modifier le programme WSTEP1.C en suivant les indications suivantes.

1) Création du sous projet de ressources:

Lancer l'éditeur de ressources depuis le menu général Tools → Resource Workshop et agrandir la fenêtre au maximum.

Choisir dans le menu de Resource Workshop File → New project
et choisir le type de sous projet .RC

Dans la fenêtre Add to project qui apparaît choisir alors:

```
Le répertoire de travail C :\ . . en cliquant l'explorateur
File Name=  WSTEP1.RH
File Type=  RH,H (C header)
```

et accepter la création du header de ressource WSTEP2.RH destiné à contenir les identificateurs de ressources.

Faire ensuite File → Save project et lui attribuer le nom C:\ . . . \WSTEP1.RC

Le sous projet de ressources est prêt à l'emploi.

2) Création de la ressource menu:

Toujours dans Resource Workshop ouvrir Resource → New
et choisir MENU.

Modifier le menu de base proposé pour obtenir le menu suivant:

```
MAIN_MENU
POPUP "Jeu"
    MENUITEM "Nouveau"
    MENUITEM "Quitter"
_End Popup_
POPUP "Aide"
    MENUITEM "A propos"
    MENUITEM "Règle du Jeu"
_End Popup_
_End Menu_
```

Il convient de faire <INS> ou Menu → New Popup ou Menu → New MenuItem pour insérer un élément de menu et pour supprimer un élément existant.

Ensuite il faut définir les identificateurs des diverses options du menu; pour cela cliquer successivement sur chaque élément de menu pour faire apparaître à gauche la fenêtre de propriétés. Ne pas modifier les propriétés par défaut dans l'immédiat mais remplacer les identificateurs dans le champ Item Id par:

```
CM_JEU_NEW, CM_JEU_QUIT
CM_AIDE_ABOUT, CM_AIDE_REGLE
```

Ces identificateurs seront placés par Resource Workshop dans le header WSTEP1.RH puis utilisés dans le programme pour identifier les actions demandées.

Définir des raccourcis clavier en faisant précéder les lettres clés d'un & dans le champ Item Text de la fenêtre propriétés; par exemple l'intitulé &Jeu donnera dans le menu la commande Jeu, accessible par <ALT> J directement.

Le menu obtenu peut être testé en temps réel dans la partie supérieure de la fenêtre de droite où il apparaît !

Pour conclure, renommer ce menu via Resource → Rename et le nommer MAIN_MENU comme prévu plus haut.

☛ Attention: Ne pas créer d'identificateur numérique pour le menu MAIN_MENU si on veut pouvoir accéder à cette ressource par son nom dans le programme. Une ressource identifiée numériquement doit être appelée à l'aide de la macro-fonction MAKEINTRESOURCE(int IDENT).

Avant de quitter Resource Workshop, enregistrer le sous projet via File → Save project
Les fichiers WSTEP1.RC et WSTEP1.RH sont générés.

3) Intégration du menu au programme:

Commencer par ajouter au fichier de projet, au même niveau que WSTEP1.C, le script de ressource WSTEP1.RC (Qui appelle lui-même WSTEP1.RH)

Editer alors WSTEP1.C et apporter les modifications suivantes:

a) Insérer le header WSTEP1.RH en dessous de WINDOWS.H par une directive
`#include "wstep1.rh"`

afin de pouvoir utiliser les identificateurs du menu.

b) Dans la fonction InitApplication(), modifier la ligne qui définit le menu:
`wc.lpszMenuName= "MAIN_MENU";`

La ressource est appelée directement par son nom.

c) Ajouter dans la procédure de fenêtre MainWndProc() un cas WM_COMMAND correspondant au message WM_COMMAND émis par Windows lors d'une action dans le menu. Le paramètre Wparam contient lors de l'appel l'identificateur de la commande choisie dans le menu, d'où le code à insérer:

```
case WM_COMMAND:
{
switch(wParam)
{
case CM_JEU_NEW:
{
/* Jeu → Nouveau */
}
```

```

        break;
    case CM_JEU_QUIT:
        {
            /* Jeu → Quitter */
        }
        break;

    case CM_AIDE_ABOUT:
        {
            /* Aide → A propos */
        }
        break;
    case CM_AIDE_REGLE:
        {
            /* Aide → Règle du jeu */
        }
        break;
    }
}
break;

```

d) Pour vérifier la validité de ce code on pourra placer un appel à `MessageBox()` dans la rubrique Aide → A propos.

D'autre part, si on veut rendre opérationnelle l'option Jeu → Quitter, il faut y placer un appel à `SendMessage()`:

```
SendMessage(hWnd, WM_CLOSE, 0, 0);
```

qui demande la fermeture de la fenêtre et tout ce qui s'ensuit...

e) Lancer Project → Make All pour compiler la source .C et le script de ressources .RC
 Tester le fonctionnement du menu avec Debug → Run

III) Création d'une boîte de dialogue modale:

1) Les boîtes de dialogue:

Les boîtes de dialogue modales sont fréquemment utilisées sous Windows pour dialoguer avec l'utilisateur et collecter des informations en tous genres.

Une boîte de dialogue contient des **contrôles** de diverses natures, telles que:

- Boutons poussoirs ou radio, cases à cocher
- Textes, champs d'édition
- Boîtes listes à choix multiples
- Barres de défilement

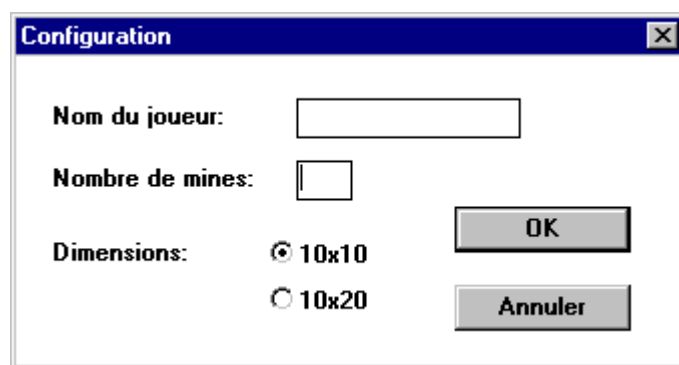
En plus des contrôles désirés une boîte de dialogue contient des boutons standards du genre Ok, Cancel, Yes ou No...

2) Création de la ressource boîte de dialogue:

La création d'une boîte de dialogue se fait encore sous Resource Workshop; il suffit de l'ajouter au sous projet existant en cliquant sur WSTEP1.RC dans la fenêtre Project.

Nous allons y créer une boîte de dialogue simple comprenant:

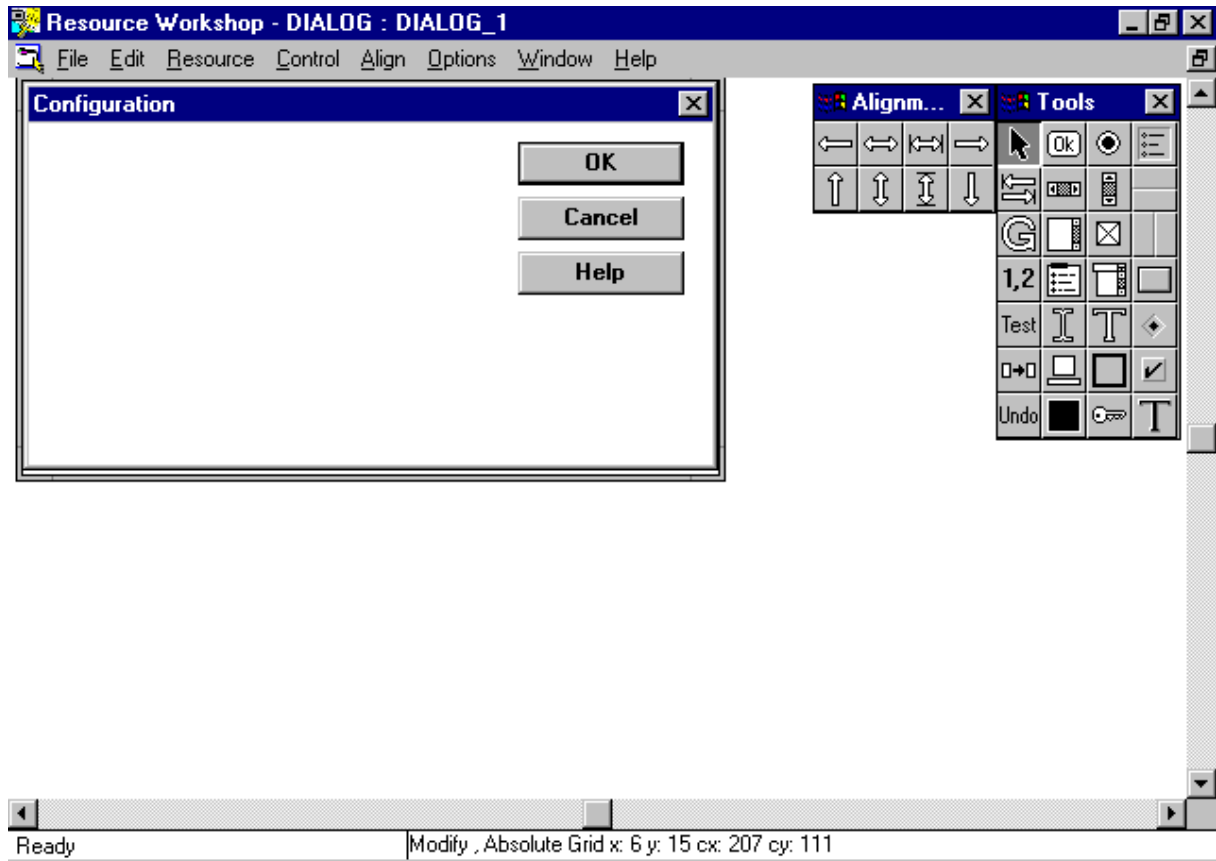
- Un champ d'édition texte pour le nom du joueur.
- Un champ d'édition numérique pour le nombre de mines.
- Une paire de boutons radio pour choisir la taille de la grille de jeu parmi 10x10 et 10x20.
- Les boutons Ok et Annuler pour terminer le dialogue.



Pour cela ouvrir le menu **Resource** → **New** et opter pour **DIALOG**; choisir alors le style de boutons, par exemple style Windows et situés à droite. Une boîte de dialogue de base apparaît.

Cliquer deux fois sur le titre de la boîte et la renommer en "Configuration".
Supprimer le bouton Help à l'aide de .

Les boîtes à outils à droite permettent de créer et placer les contrôles dans la boîte de dialogue:



Insérer les textes statiques "Nom du joueur", "Nombre de mines" et "Dimensions" en choisissant le 'T' blanc dans la boîte à outils et en positionnant les champs dans la boîte; éditer les champs pour y insérer les textes voulus.

Insérer les champs d'édition correspondants en choisissant le 'T' blanc dans la boîte à outils; cliquer deux fois sur chaque champ pour faire apparaître la fenêtre propriétés et modifier les identificateurs proposés par défaut, qui ne sont pas du tout significatifs.

On nommera IDC_NAME l'identificateur du champ Nom du joueur, IDC_MINES celui du champ Nombre de Mines, les deux étant déclarés Text.

Les champs statiques ont un identificateur égal à -1.

Choisir dans la boîte à outils deux boutons radio à intituler 10x10 et 10x20, ayant pour identificateurs IDC_10 et IDC_20. Ces boutons sont par défaut associés dans un même groupe et ne peuvent être cochés qu'en alternance.

Placer convenablement les boutons Ok et Cancel que l'on renommera en Annuler sans en changer toutefois les identificateurs.

Dimensionner la boîte de dialogue et la placer par rapport à la fenêtre mère car cela a une importance à l'exécution.

Tester alors la boîte de dialogue obtenue en cliquant sur Test dans la boîte à outils.

Vérifier en particulier l'ordre des contrôles lors du déplacement dans la boîte avec <TAB>;

si l'ordre ne convient pas, cliquer sur (1,2) dans la boîte à outils et redéfinir à la souris l'ordre de parcours des contrôles. (Choisir la Flèche de la boîte à outils pour terminer)

Enregistrer la boîte de dialogue achevée via Resource → Rename sous le nom CONFIG par exemple, sans créer encore une fois d'identificateur numérique pour cette ressource, puis enregistrer le sous projet dans File → Save project

3) Utilisation de la boîte de dialogue:

L'utilisation de la boîte de dialogue CONFIG ainsi créée nécessite plusieurs modifications dans WSTEP1.C:

- L'appel de la boîte de dialogue dans la rubrique Jeu → Nouveau de la procédure de fenêtre gérant le menu.
- L'initialisation du contenu de la boîte de dialogue, qui est par défaut vide.
- La récupération en fin de dialogue des modifications effectuées.
- La destruction de la boîte de dialogue.

Une boîte de dialogue étant une fenêtre on va lui associer une procédure de dialogue chargée de gérer les messages de la boîte.

Remarquons que dans le cas d'une boîte de dialogue, les messages ne transitent pas par la file de message du programme mais sont adressés directement par Windows à la boîte de dialogue.

Une procédure de dialogue sera toujours déclarée sous la forme:

Syntaxe: `BOOL CALLBACK ConfigDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);`

Les paramètres sont de même nature que pour une procédure de fenêtre.

Le message WM_INITDIALOG est émis lors de la création de la boîte afin d'en préparer l'initialisation.

Un message WM_COMMAND est ensuite émis à chaque action sur un contrôle de la boîte, avec pour paramètre wParam= Identificateur du contrôle.

Ainsi toute action dans la boîte génère un message et l'on peut piloter la boîte de dialogue à distance à l'aide de fonctions qui permettent de récupérer ou modifier le contenu ou le statut des différents contrôles.

La sortie de la boîte est caractérisée par l'utilisation d'un des boutons standards ayant pour identificateurs IDOK, IDCANCEL, IDYES, IDNO...

Pour rendre opérationnelle la boîte de dialogue, suivre les étapes suivantes:

- a) Créer la procédure de dialogue ConfigDlgProc() sur le modèle indiqué plus haut et y placer le code suivant:

BOOL CALLBACK

ConfigDlgProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)

/* procedure de dialogue CONFIG */

```
{
    BOOL pb ;

    switch (message)
    {
        case WM_INITDIALOG: /* emis avant l'affichage de la boite */
        {
            SetDlgItemText(hDlg, IDC_NAME, name);
            SetDlgItemInt(hDlg, IDC_MINES, nb_mines, 0);
            if (size == 10)
                SendDlgItemMessage(hDlg, IDC_10, BM_SETCHECK, TRUE, 0);
            else
                SendDlgItemMessage(hDlg, IDC_20, BM_SETCHECK, TRUE, 0);

            return(TRUE);
        }
        // break;

        case WM_COMMAND: /* emis lors d'une action dans la boite */
        {
            switch(wParam)
            {
                case IDOK: /* sortie OK */
                {
                    GetDlgItemText(hDlg, IDC_NAME, name, sizeof(name));
                    nb_mines= GetDlgItemInt(hDlg, IDC_MINES, &pb, 0) ;

                    if (IsDlgButtonChecked(hDlg, IDC_10)) size= 10;
                    else size= 20;
                    EndDialog(hDlg, IDOK);
                }
                break;

                case IDCANCEL: /* sortie ANNULER */
                {
                    EndDialog(hDlg, IDCANCEL);
                }
                break;
            }
            return(TRUE);
        }
        // break;
    }

    return(FALSE);
}
```

Le message WM_INITDIALOG est utilisé pour initialiser la boîte. Les différents types de contrôles disposent de fonctions spécifiques pour agir sur leur contenu ou leur statut.

Ici on utilise SetDlgItemText(), SetDlgItemInt(), GetDlgItemText() et GetDlgItemInt() pour accéder au contenu des champs d'édition:

Syntaxe: SetDlgItemText(HWND hDlg, int IDENT, LPCSTR string);
 GetDlgItemText(HWND, hDlg, int IDENT, LPSTR string, int len);
 SetDlgItemInt(HWND hDlg, int IDENT, UINT value, BOOL sign);
 GetDlgItemInt(HWND, hDlg, int IDENT, BOOL * error, BOOL sign);

Par exemple SetDlgItemText(hDlg, IDC_NAME, name) affecte au champ IDC_NAME contenant le nom du joueur le contenu de la chaîne de caractères name.

Inversement GetDlgItemText(hDlg, IDC_NAME, name, sizeof(name)) récupère dans la variable chaîne de caractères name le contenu du champ IDC_NAME.

Les fonctions SetDlgItemInt() et GetDlgItemInt() font de même pour le champ IDC_MINES et la variable nb_mines; le paramètre signe est TRUE si la valeur est signée et la variable error contient 0 en cas d'erreur de conversion du texte en nombre !

On sollicite également SendDlgItemMessage() et IsDlgButtonChecked() pour positionner les boutons radio ou en consulter l'état:

Syntaxe: SendDlgItemMessage(HWND hDlg, int IDENT, UINT msg,
 WPARAM wParam, LPARAM lParam);
 UINT IsDlgButtonChecked(HWND, hDlg, int IDENT);

Par exemple SendDlgItemMessage(hDlg, IDC_10, BM_SETCHECK, TRUE, 0) a pour but d'activer le bouton de dimension 10x10 identifié par IDC_10.

L'appel IsDlgButtonChecked(hDlg, IDC_10) permet donc de récupérer une valeur nulle ou non selon que le bouton IDC_10 est activé ou pas.

La fonction EndDialog() ferme la boîte de dialogue, donc met fin à DialogBox() qui est en fait en cours d'exécution !

Syntaxe: EndDialog(HWND hDlg, int IDENT);

EndDialog() détermine le code de retour de DialogBox() à travers le paramètre IDENT.

☛ Attention: Une procédure de dialogue doit impérativement retourner TRUE lorsqu'elle a traité le message et FALSE dans le cas contraire.

b) Appeler la boîte de dialogue CONFIG à l'aide la fonction DialogBox():

Syntaxe: int DialogBox(HANDLE hInst, lpszDlgName Name, HWND hParent,
 DLGPROC DlgProc);

Les paramètres transmis sont le handle de l'application, le nom de la ressource dialogue sous forme de chaîne de caractères, le handle de la fenêtre mère de la boîte de dialogue et enfin le nom de la procédure de dialogue.

Elle retourne un entier égal à -1 en cas d'erreur (Ressource inexistante...) et le code de fin de dialogue IDOK, IDCANCEL... dans le cas contraire.

On doit appeler cette fonction dans MainWndProc() rubrique Jeu → Nouveau sous la forme:

```
res= DialogBox( hInst, "CONFIG", hWnd, ConfigDlgProc );  
if (res != IDOK) break;
```

c) Ajouter quelques déclarations et variables au programme pour faire fonctionner l'ensemble.
Ajouter en début de programme:

```
/* prototype de la procédure de dialogue */  
BOOL CALLBACK ConfigDlgProc(HWND, UINT, WPARAM, LPARAM);  
char name[20]= "Untel"; /* nom du joueur */  
UINT nb_mines= 10; /* nombre de mines */  
UINT size= 10; /* taille de la grille */
```

Ajouter aussi une variable à la procédure MainWindowProc():

```
int res; /* code de retour de dialogue */
```

Recompiler alors le projet et tester la boîte de dialogue.

IV) Création d'une icône personnalisée

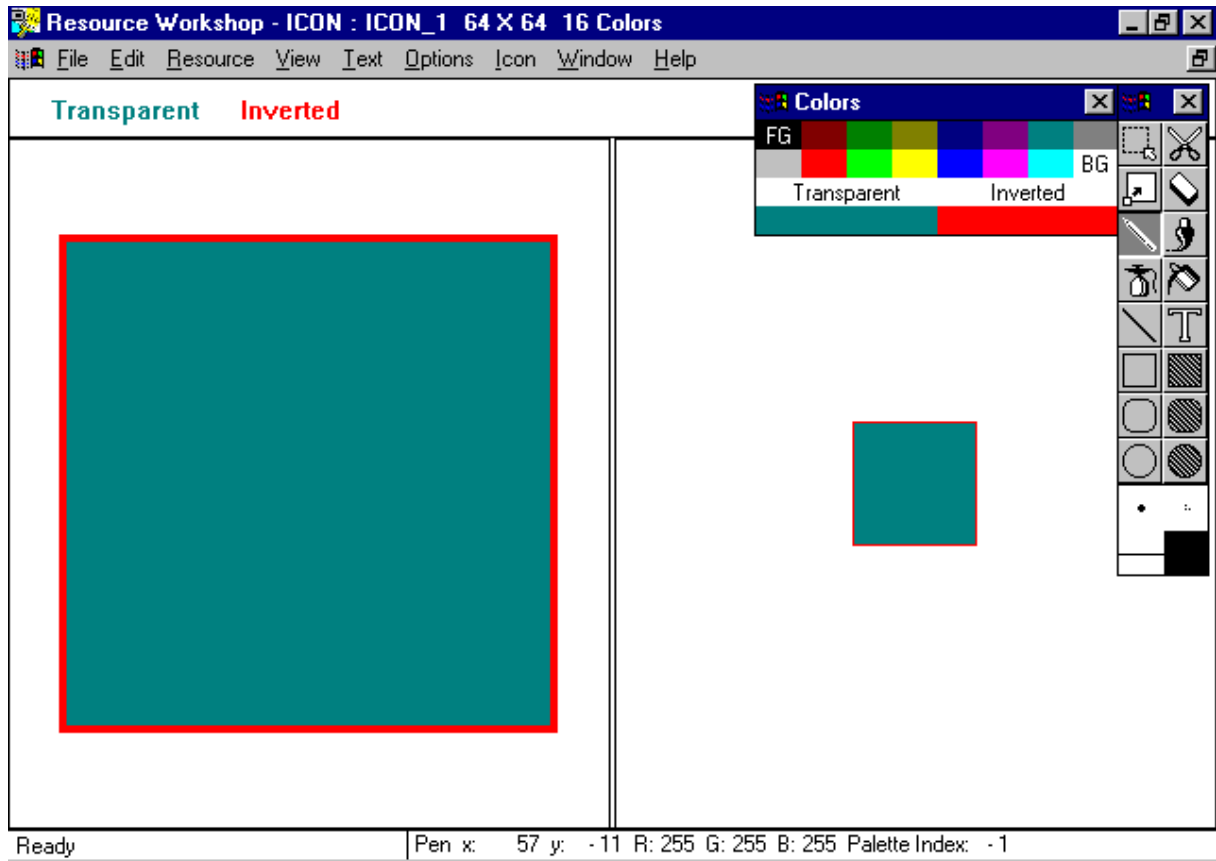
Plutôt que d'utiliser une icône standard fournie par le gestionnaire de programme, il est préférable d'attribuer une icône personnalisée au programme, voire même plusieurs.

1) Création d'une ressource icône:

Relancer Resource Workshop en cliquant sur WSTEP1.RC dans la fenêtre projet.

Choisir Resource → New puis ICON et préciser le type d'icône désiré, par exemple au format 32x32 ou 64x64 pixels et 256 couleurs.

Cliquer deux fois sur la ligne qui apparaît pour ouvrir l'éditeur d'icônes:



Créer l'icône voulue à l'aide de la boîte à outils.

Lorsque l'icône est achevée, renommer cette nouvelle ressource comme d'habitude via Resource → Rename, sans créer d'identificateur, sous le nom BOMBS par exemple.

Enregistrer dans File → Save project et quitter Resource Workshop.

2) Intégration de l'icône au programme:

L'intégration de cette icône au programme ne nécessite qu'une modification au niveau de la fonction InitApplication(); modifier la ligne suivante:

```
wc.hIcon= LoadIcon(hInstance, "BOMBS");
```

La fonction LoadIcon() est alors invoquée en référençant par son nom l'icône créée:

Syntaxe: HICON LoadIcon(HANDLE hInst, LPCSTR IconName);

Recompiler le projet WSTEP1 et vérifier à l'exécution l'apparition de l'icône dans la barre de titre, au moins sous Windows 95.

Dans tous les cas tout raccourci créé vers WSTEP1.EXE utilisera cette nouvelle et unique icône pour matérialiser le programme.

V) Création d'une boîte de dialogue non modale:

1) Les boîtes non modales:

Une boîte de dialogue modale comme celle décrite dans le §III apparaît lors d'une action de l'utilisateur et doit être refermée avant que le programme puisse se poursuivre.

Au contraire, une boîte non modale (modeless) peut rester présente à l'écran sans empêcher le déroulement du programme, et constituer par exemple une boîte à outils comme nous allons le voir ci-après. Cette boîte à outils pouvant être montrée ou cachée à volonté.

2) Création de la ressource boîte à outils:

On va créer ici une boîte non modale destinée à remplacer un menu. Elle contiendra les contrôles suivants:

- Un bouton pour quitter l'application
- Un bouton pour fermer la boîte à outils



La boîte est créée comme d'habitude sous Resource Workshop. Toutefois afin de la rendre non modale on **décochera** dans la feuille de propriétés la case **Modal Frame**.

3) Utilisation de la boîte à outils:

L'utilisation nécessite la prise en compte de plusieurs choses:

- Appel de la boîte par `CreateDialog()`
- Gestion de la fermeture de la boîte par `DestroyWindow()`
- Gestion des actions dans la boîte

En fait une telle boîte n'est vraiment intéressante que si on peut placer des icônes dans les boutons, mais nous verrons cela plus loin.

On doit associer une procédure de dialogue à la boîte à outils:

Syntaxe: `BOOL CALLBACK ToolProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam);`

La procédure gère alors les actions dans la boîte à travers le message `WM_COMMAND` et

les identificateurs de contrôles dans wParam. Par exemple les boutons ci-dessus auront pour identificateurs IDC_HIDE et IDC_END.

Contrairement à une boîte modale qui est appelée par DialogBox() et fermée par EndDialog(), une boîte modale est créée par CreateDialog() et fermée par un DestroyWindow().

Syntaxe: HWND CreateDialog(HANDLE hInst, lpszDlgName Name, HWND hParent, DLGPROC ToolProc);

Les paramètres transmis étant le handle de l'application, le nom de la ressource boîte à outils, le handle de la fenêtre mère et le nom de la procédure de dialogue.

Le handle hToolWnd récupéré au retour du CreateDialog() doit être conservé jusqu'à la destruction de la boîte par DestroyWindow(hToolWnd).

Remarque: Il est judicieux de créer la boîte à outils dans le WM_CREATE de la fenêtre mère et de la supprimer dans le WM_DESTROY de celle-ci.

Pour masquer temporairement la boîte à outils, on peut utiliser ShowWindow() avec le paramètre SW_HIDE, puis avec SW_SHOW pour la faire réapparaître.

On fera un double clic dans la fenêtre principale pour la faire réapparaître.

On peut savoir si la fenêtre existe déjà ou si elle est visible en faisant appel aux fonctions IsWindow() et IsWindowVisible().

On peut aussi positionner et dimensionner la boîte à outils à volonté via MoveWindow():

Syntaxe: MoveWindow(HWND hWnd, int x0, y0, width, height, BOOL fRepaint);

VI) Boîte de dialogue comme fenêtre principale:

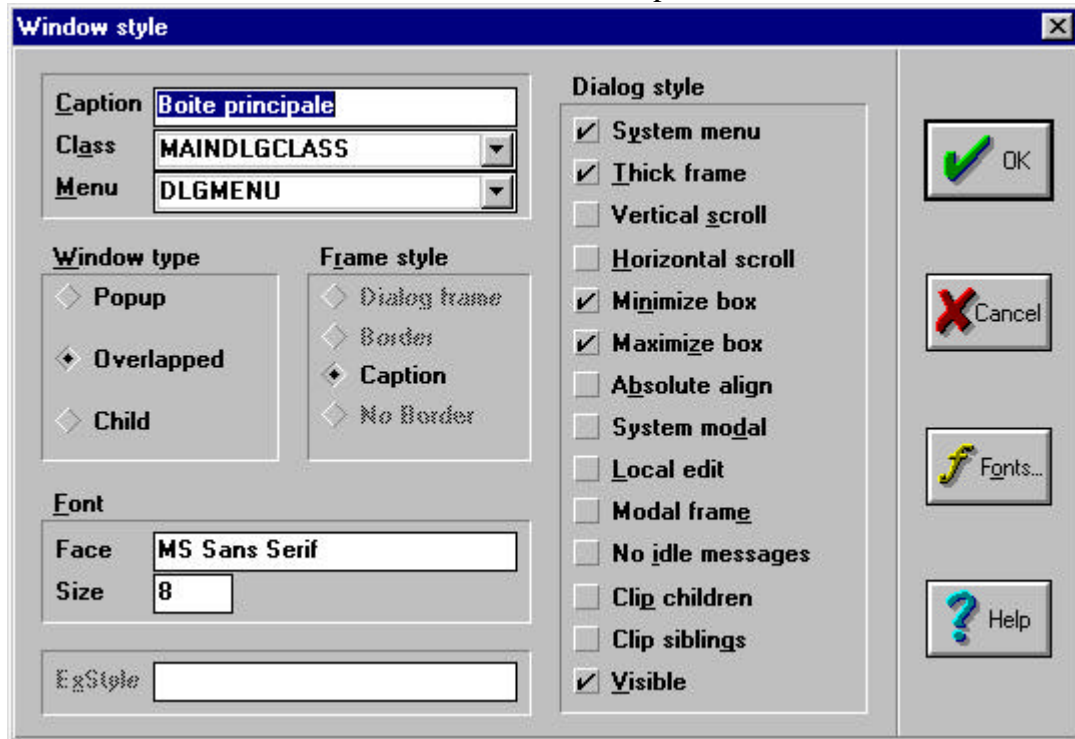
1) Principe:

Certaines applications pas trop compliquées peuvent se contenter d'une boîte de dialogue et de contrôles en guise d'interface utilisateur, c'est à dire de fenêtre principale. Dans ce cas la boîte de dialogue aura une classe propre et un menu.

2) Création de la ressource boîte principale:

Sous Resource Workshop on crée une boîte de dialogue contenant les contrôles souhaités, et on ajoute, dans la feuille de propriétés une classe de fenêtre MAINDLGCLASS et un nom de ressource menu DLGMENU, et on coche les cases indiquées ci-dessous.

Le menu associé sera créé comme d'habitude sous le nom précédent.



3) Utilisation de la boîte principale:

On définit dans InitApplication() la classe de fenêtre MAINDLGCLASS comportant en plus des champs usuels le champ:

```
wc.cbWndExtra= DLGWINDOWEXTRA;          /* Constante prédéfinie */
```

La fenêtre est ensuite créée par CreateDialog() et non CreateWindow() dans InitInstance() !

Le paramètre hParent désignant la fenêtre mère est naturellement NULL.

On associe à la fenêtre une procédure de fenêtre comme d'habitude.

VII) Pilotage des contrôles d'une boîte de dialogue ou d'une fenêtre:

1) Rappels:

On a déjà mentionné dans la partie §III les fonctions SetDlgItemText(),SetDlgItemInt(), GetDlgItemText(), GetDlgItemInt() qui permettent d'accéder au contenu texte ou nombre d'un champ de saisie.

On a aussi utilisé SendDlgItemMessage() pour cocher par exemple une case ou activer un bouton radio, ainsi que la fonction IsDlgButtonChecked() pour tester l'état d'un bouton radio.

2) Activation des boutons poussoirs:

Un contrôle, il faut bien le dire un jour, n'est pour Windows qu'une (petite) fenêtre parmi tant d'autres. Un contrôle a donc des propriétés semblables aux autres fenêtres: on peut l'activer, le masquer, lui donner le focus...

Pour cela il faut récupérer le handle du contrôle par GetDlgItem():

Syntaxe: HWND GetDlgItem(HWND hDlg, int IDENT);

Exemple: hButton= GetDlgItem(hDlg, IDC_ADD);

On peut alors réaliser les opérations voulues avec les appels suivant:

```
ShowWindow( hButton, SW_HIDE ); /* Cacher le bouton */
ShowWindow( hButton, SW_SHOW ); /* Montrer le bouton */
EnableWindow( hButton, FALSE); /* Griser le bouton */
EnableWindow( hButton, TRUE ); /* Réactiver le bouton */
SetFocus( hButton, SW_SHOW ); /* Donner le focus à ce bouton */
```

3) Boutons poussoirs personnalisés:

On peut créer des boutons poussoirs personnalisés, pour une boîte à outils par exemple, en combinant un bouton et une ressource icône ou bitmap..

Il faut pour cela créer le bouton (sans titre) dans la boîte à outils, en **cochant** dans la feuille de propriétés du bouton la case **OwnerDraw** afin que le contenu du bouton puisse être redessiné. Il faut créer dans la foulée le dessin sous forme de bitmap ou d'icône.

Ensuite, la procédure de dialogue doit gérer le message WM_DRAWITEM qui est émis à la création du bouton afin d'en dessiner le contenu; lParam contient pour cela l'adresse d'une structure DRAWITEMSTRUCT contenant l'identificateur du bouton du bouton. et autres informations:

```
struct DRAWITEMSTRUCT
{
    UINT CtlType;
    UINT CtlID;
    UINT itemID;
    UINT itemAction;
    UINT itemState;
    HWND hwndItem;
    HDC hDC;
    RECT rcItem;
    DWORD itemData;
};
```

On y voit en particulier un champ hDC qui doit servir au dessin dans le bouton.

Pour cela on charge le bitmap par LoadBitmap() ou l'icône par LoadIcon() et on récupère un handle de type HBITMAP qui permet de sélectionner celui-ci au niveau du fond de bouton de la façon suivante:

```
hBitmap= LoadBitmap( BMPNAME);  
SelectObject( DrawItemStruct.hDC, hBitmap );
```

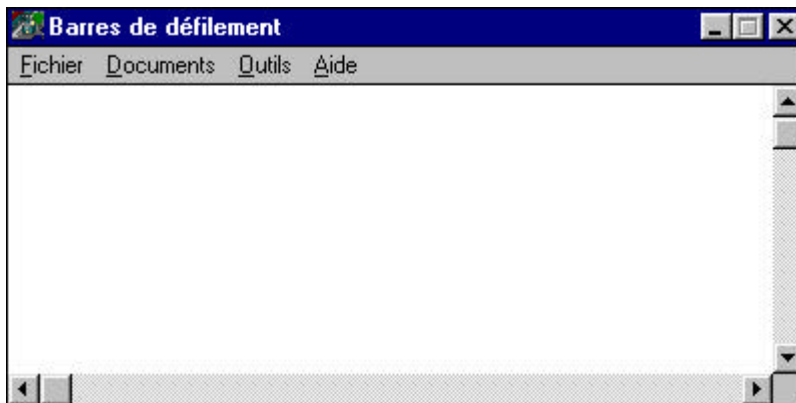
Il ne reste plus qu'à modifier l'aspect du bouton lorsque l'on clique dessus par FrameRect() ou InvertRect()...

4) Barres de défilement:

Les barres de défilement, alias scroll bars sont très utiles et très faciles à gérer sous Windows. Elles permettent de scroller du texte ou des images, horizontalement ou verticalement, à l'aide de la souris ou du clavier.

La souris agit sur les flèches ou le curseur de la scroll bar, tandis que la gestion clavier consiste à simuler les actions équivalentes avec PostMessage() — qui respecte la file d'attente contrairement à SendMessage().

Pour attribuer une barre de défilement à une fenêtre il faut inclure le style WS_VSCROLL ou WS_HSCROLL dans l'appel à CreateWindow(). Les barres viennent s'ajouter à la zone client mais n'en font pas partie.



Les dimensions (épaisseur) des barres sont fixées par Windows et accessibles par GetSystemMetrics().

Au niveau programmation, on doit tout d'abord définir l'intervalle associé aux positions extrêmes de la scroll bar, par deux valeurs min et max passées à SetScrollRange():

Syntaxe: SetScrollRange(HWND hWnd, int BARID, int min, max, BOOL fRedraw);

BARID: SB_VERT ou SB_HORZ selon la barre concernée.

min: 0 par exemple

max: nombre de positions possibles de la scroll bar.

fRedraw: TRUE si la barre doit être redessinée (la taille du curseur ou ascenseur est fonction de l'intervalle spécifié).

A chaque clic sur la barre de défilement Windows génère un message circonstancié WM_VSCROLL ou WM_HSCROLL avec dans LOWORD(wParam) l'action: SB_LINEUP, SB_LINEDOWN, SB_PAGEUP, SB_PAGEDOWN, ou SB_THUMBPOSITION, SB_THUMBTRACK si on a touché à l'ascenseur; dans ces deux derniers cas HIWORD(wParam) contient la position actuelle de l'ascenseur, représentée par une valeur comprise entre min et max selon une règle de trois donc proportionnelle au défilement.

Attention: Ne pas omettre de déclarer wParam en WPARAM (= LONG) et non en WORD sans quoi HIWORD(wParam) = 0 dans tous les cas !!!

Cela dit, le déplacement du curseur dans la barre à l'écran est du ressort du programmeur qui utilise pour cela la fonction SetScrollPos():

Syntaxe: SetScrollPos(HWND hWnd, int BARID, int ScrollPos, BOOL fRedraw);

Ici le paramètre essentiel est

ScrollPos: Nouvelle position de défilement, telle qu'obtenue ci-dessus en général.

Remarque: De plus, contrairement à ce que l'on pourrait croire, Windows ne scrolle absolument pas le texte ou l'image dans la fenêtre; cela incombe au programmeur qui doit redessiner la zone client !!!

5) Les menus:

On peut griser ou activer les options d'un menu via EnableMenuItem():

Syntaxe: EnableMenuItem(HMENU hMenu, UINT ITEMID, UINT MODE);

hMenu : Le handle du menu, obtenu si nécessaire par GetMenu().

ITEMID: L'identificateur de l'item tel qu'il est défini dans la ressource.

MODE: MF_GRAYED ou MF_ENABLED.

Signalons aussi la possibilité de cocher ou décocher une option de menu par √ en utilisant MF_CHECKED ou MF_UNCHECKED.

6) Les contrôles 3D Borland:

En plus des contrôles standard Windows et des boîtes de dialogue de style Windows (3.1), Resource Workshop offre des boîtes et contrôles 3D (gris avec effets de bord, style Win 95). Ces contrôles sont reconnaissables à leur nom (Borld) dans la feuille de propriété, et les boîtes à leur classe spécifique Borld_GRAY.

Resource Workshop mélange les contrôles des deux types, mais il convient de se méfier:

Pour pouvoir utiliser les objets 3D, en particulier en dehors de BCW, dans un exécutable, il faut respecter l'approche suivante:

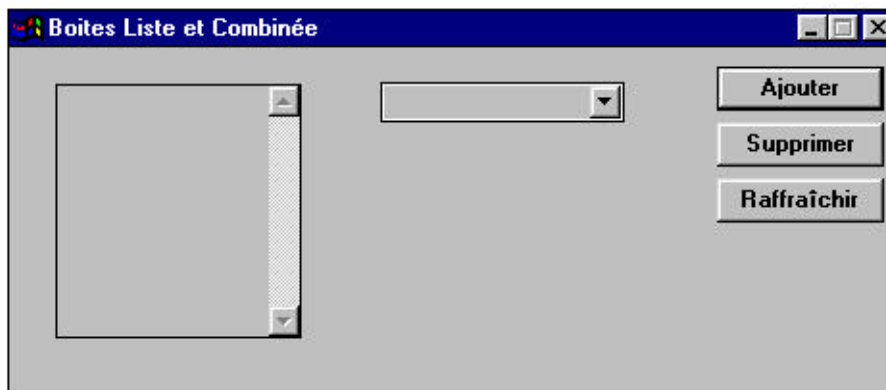
- Cocher BWCC (Borland Windows Custom Controls) dans les librairies du projet.
- Inclure bwcc.h dans le programme C.
- Appeler depuis WinMain() la fonction BWCCRegister(HINSTANCE hInst)
- Si les librairies sont dynamiques, prévoir avec l'exécutable les librairies BWCC.DLL ou BWCC32.DLL.

VIII) Boites listes et combinées:

1) Présentation:

Une boîte liste permet d'effectuer un choix parmi une liste de propositions imposées;
Une boîte combinée ou combo-box y ajoute un champ de saisie, au cas où les choix proposés dans la liste ne conviendraient pas.

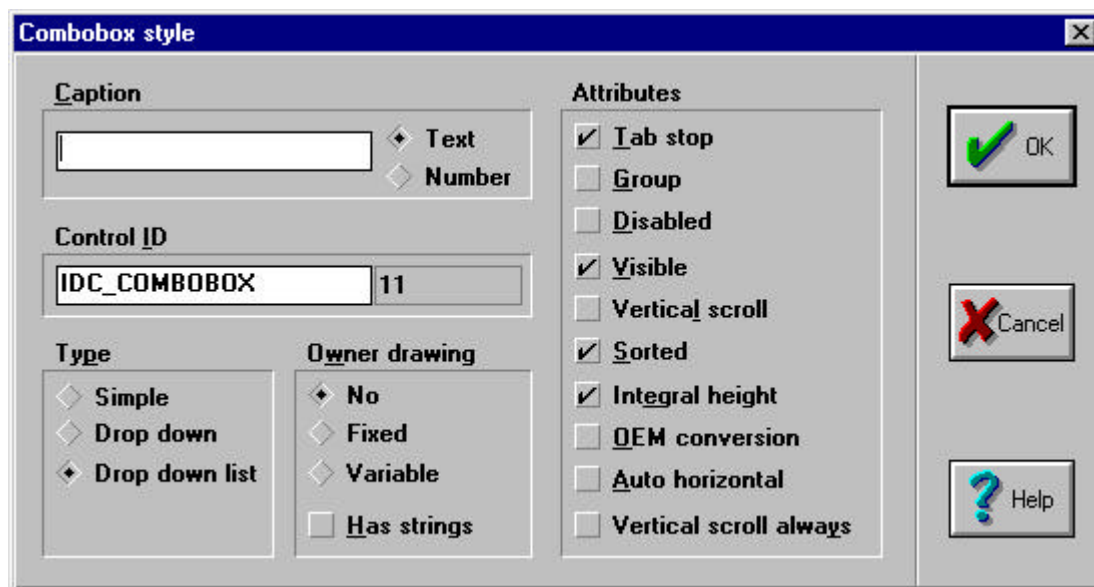
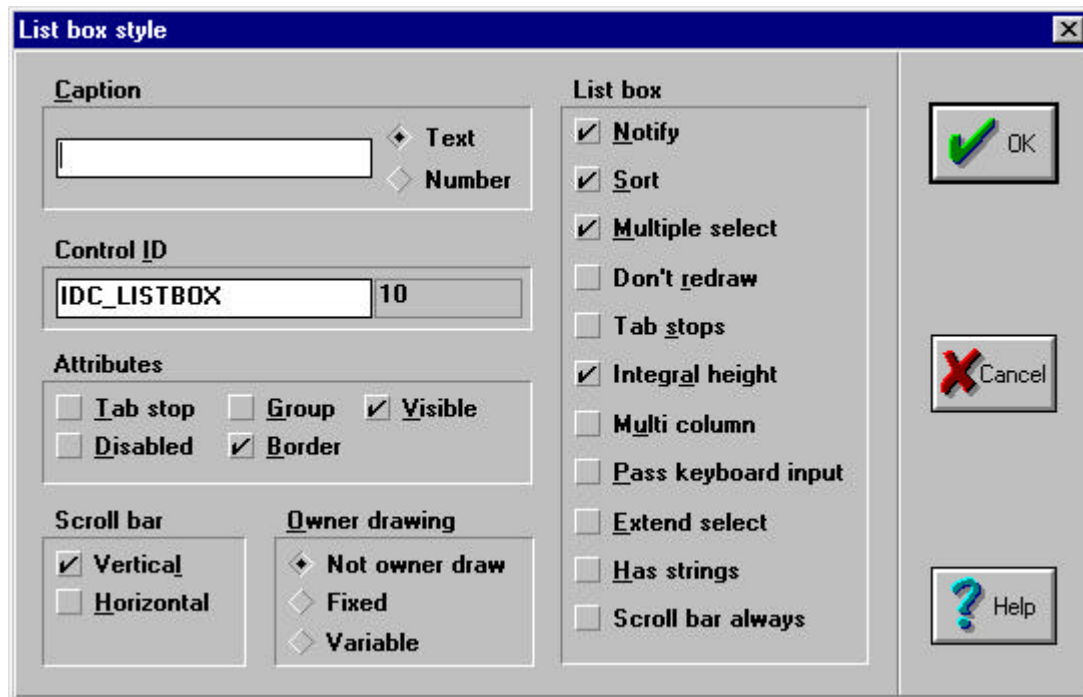
On peut dans une liste sélectionner un ou plusieurs items:



Une liste peut être triées de façon automatique par ordre alphanumérique si on le souhaite.
La liste peut même s'étaler sur plusieurs colonnes !

2) Création de la ressource:

Dans Resource Workshop on choisit et positionne la boîte liste ou combo dans la boîte de dialogue, en cochant dans la feuille de propriétés les options voulues:



3) Utilisation d'une boîte liste:

L'utilisation d'une boîte liste passe par des messages:

- de la boîte liste vers la procédure de dialogue via WM_COMMAND lors d'actions dans la boîte (sélection, désélection...)
- de la procédure de dialogue vers la boîte liste (pour ajouter, supprimer, sélectionner des items dans la liste....) par SendMessage().

Lors d'un WM_COMMAND, lParam contient le handle hListe de la boîte liste et LOWORD(wParam) est le code de notification, par exemple:

LBN_SELCHANGE	Sélection simple
LBN_DBLCLK	Double clic
LBN_SELCANCEL	Désélection complète
...	

Dans l'autre sens l'appel à SendMessage() aura la forme suivante:

```
result= SendMessage( hListe, LB_***, p1, p2)
```

Les paramètres p1 et p2 dépendent du message LB_*** que l'on envoie:

Action	LB_***	p1	p2	result
Ajouter à la liste	LB_ADDSTRING	0	(LPARAM) stg	
Ajouter à telle position	LB_INSERTSTRING	pos	(LPARAM) stg	
Ajouter une liste de fichiers	LB_DIR	attributs	(LPARAM) format	
Supprimer l'élément à telle position	LB_DELETESTRING	pos	0	
Effacer toute la liste	LB_RESETCONTENT	0	0	
Compter les éléments de la liste	LB_GETCOUNT	0	0	nb
Sélectionner tel élément	LB_SETCURSEL	pos	0	
Sélectionner le 1er commençant par...	LB_SELECTSTRING	0	(LPARAM) stg	pos ou LB_ERR
Rechercher une sous chaîne	LB_FINDSTRING	0	(LPARAM) stg	pos ou LB_ERR
Longueur d'un élément de la liste	LB_GETTEXTLEN	pos	0	nb octets
Copier un élément	LB_GETTEXT	pos	(LPARAM) stg	nb octets
Modifier l'état de sélection d'un élément (choix mult)	LB_SETSEL	(WPARAM) 0 select, 1 deselect	pos	
Lire l'état de sélection	LB_GETSEL	pos	0	0 ou 1
Nombre de lignes sélectionnées	LB_GETSELCOUNT	0	0	nb lignes
Première ligne visible de la liste	LB_GETTOPINDEX	0	0	pos
Modifier zone visible de la liste	LB_SETTOPINDEX	pos	0	

4) Utilisation d'une combo-box:

Pour le pilotage d'une combo-box on retrouve les mêmes principes que pour une boîte liste, avec quelques actions supplémentaires.

Les messages de combo-box sont CB_*** et les notification CBN_***.

1) Le GDI Windows:

1) Définition:

Le GDI (Graphic Device Interface) de Windows est responsable de l'interface entre les programmes graphiques et l'affichage correspondant, au travers des drivers de périphériques. C'est lui qui assure l'indépendance de Windows par rapport au matériel sous-jacent, ici la carte vidéo et l'écran.

Le GDI travaille par défaut en pixels pour déterminer les dimensions de l'écran, d'une fenêtre, la taille d'une police, l'épaisseur d'un trait...et c'est sous cette forme que nous l'utiliserons.

Toute manipulation de texte ou de graphique à l'écran passe par l'obtention d'un DC (Device Context ou Contexte de Périphérique). Un tel DC est géré par un handle obtenu auprès de Windows et il permet d'appeler les fonctions de la bibliothèque GDI pour réaliser les tracés dans une fenêtre. Le DC contient une structure interne qui mémorise certains paramètres courants qu'il ne sera pas nécessaire de spécifier aux fonctions GDI (par exemple la position courante du crayon); ces paramètres peuvent eux-mêmes être modifiés à l'aide de fonctions appropriées.

2) Le Device Context:

Dans le premier programme réalisé au chapitre 2 on a utilisé furtivement un contexte de périphérique pour gérer le message WM_PAINT:

ce message est généré à chaque fois qu'une partie de la zone client de la fenêtre est altérée, afin que le programme puisse régénérer son contenu. Pour cela on appelle BeginPaint() pour obtenir le fameux handle de contexte de périphérique de type HDC; ce handle permet alors l'appel des fonctions GDI, jusqu'à la libération du DC par appel à EndPaint().

Syntaxe: HDC BeginPaint(HWND hWnd, struct PAINTSTRUCT * ps);
 EndPaint(HWND hWnd, struct PAINTSTRUCT * ps);

Ces deux fonctions utilisent également une structure pré-définie PAINTSTRUCT servant à mémoriser quelques paramètres, parmi lesquels le champ rcPaint, de type RECT, composé lui-même de quatre champs LONG nommés left, top, right, bottom. Une telle structure RECT sert à mémoriser les coordonnées d'un rectangle dans une fenêtre.

Ici rcPaint contient en fait après l'appel à BeginPaint() les coordonnées du **rectangle invalide** de la fenêtre à repeindre: il s'agit de la partie de la fenêtre ayant effectivement été altérée.

En principe les fonctions GDI utilisant le handle HDC reçu de BeginPaint() ne peuvent pas effectuer de tracés en dehors de ce rectangle invalide; pour y remédier on peut appeler la fonction InvalidateRect():

Syntaxe: InvalidateRect(HWND hWnd, RECT * rc, BOOL erase);

Le second paramètre rc représente alors la zone à invalider; s'il est NULL toute la zone client est invalidée. On peut aussi faire appel à GetClientRect() pour connaître les coordonnées de la zone client.

Le dernier paramètre erase est TRUE si le fond de la fenêtre doit être effacé et FALSE dans le cas contraire.

Pour le moment il n'y a dans le programme rien à restaurer dans le traitement de WM_PAINT en dehors de la MessageBox et c'est pourquoi on fait appel immédiatement à EndPaint().

Windows ne pouvant gérer qu'un nombre limité de handles de périphériques simultanément il est indispensable de les libérer aussitôt après usage !!!

En dehors de WM_PAINT on utilise un DC pour créer du texte ou du graphisme dans une fenêtre en passant par GetDC() et ReleaseDC():

Syntaxe: HDC GetDC(HWND hWnd);
 ReleaseDC(HWND hWnd, HDC hdc);

La fonction GetDC() retourne un handle associé à un Device Context permettant de réaliser des tracés dans la fenêtre référencée par hWnd. Ce handle est libéré en fin de traitement par ReleaseDC().

Remarque: GetDeviceCaps() permet d'obtenir des informations sur le périphérique associé à un DC telles que la taille en pixel, le nombre de couleurs...

En incluant dans le style de la fenêtre lors de sa création le flag CS_OWNDC on peut obtenir qu'une fenêtre mémorise les paramètres du DC qu'elle utilise au fil des appels à GetDC() ... ReleaseDC(). Autrement les modifications apportées (crayon, brosse...) sont perdues et doivent être mémorisées au sein du programme dans des variables globales ou statiques et restaurées à chaque utilisation du DC. Nous opterons pour cette seconde approche dans le programme proposé.

II) Le texte:

Le texte est géré sous Windows comme du graphisme et il passe donc par un DC.

Le texte repose sur la notion de police ou fonte de caractères.

1) Les polices:

Une police représente un jeu de caractères affichables d'un style particulier. Il existe des polices à espacement fixe dans lesquelles tous les caractères occupent la même largeur, et des polices à espacement proportionnel.

Ces polices peuvent être de type bitmap, vectoriel ou true-type;

les polices bitmap sont figées dans leur taille, les vectorielles qui sont décrites par des vecteurs sont paramétrables dans une plage de tailles raisonnables et les true-type qui sont décrites par des courbes sont paramétrables à volonté.

En pratique Windows s'adapte à toutes les situations, mais gare au rendu de certaines fontes dans certaines tailles...

Les polices fournies en standard sous Windows sont:

Courier New (Bold, Italic)	Polices à espacement fixe.
Times New Roman (Bold, Italic)	Polices proportionnelles.
Arial (Bold, Italic)	Polices proportionnelles.
System	Police système utilisée par Windows.
Symbol	Police de symboles spéciaux.

Sous Windows 95 il s'agit de polices true-type.

Pour utiliser une police il faut appeler CreateFont() qui crée une image mémoire de la fonte externe dans la taille voulue, afin d'en optimiser la vitesse d'affichage.

On peut aussi appeler GetStockObject() pour utiliser une fonte interne pré-définie.

La fonte choisie doit être sélectionnée au niveau du DC via SelectObject().

Les fontes créées par CreateFont() doivent être détruites par DeleteObject() en fin de programme.

Syntaxe: HFONT GetStockObject(int FONT_ID);
 SelectObject(HDC hdc, HFONT font);

Le paramètre FONT_ID de GetStockObject() peut prendre ici les valeurs pré-définies SYSTEM_FONT, SYSTEM_FIXED_FONT...

Exemple: SelectObject(hdc, GetStockObject(SYSTEM_FONT));

Remarque: On peut obtenir des informations sur une police à l'aide de GetTextMetrics() qui utilise une structure TEXTMETRIC dans laquelle les champs tmHeight et tmExternalLeading donnent entre autres la hauteur totale des caractères de la fonte et l'interligne conseillé.

2) Affichage de texte:

Une fois associée la police voulue au DC on peut afficher du texte dans la fenêtre avec les fonctions GDI TextOut(), DrawText() et modifier la couleur du texte et de son fond avec SetTextColor(), SetBkColor() et SetBkMode():

Syntaxe: TextOut(HDC hdc, int x, y, char * text, int len);
 DrawText(HDC hdc, char * text, int len, RECT * rc, int format);
 SetTextColor(HDC hdc, COLORREF color);
 SetBkColor(HDC hdc, COLORREF color);
 SetBkMode(HDC hdc, int mode);

TextOut() affiche la chaîne text à partir des coordonnées (x,y) dans la fenêtre; len est la longueur du texte, calculée par strlen(text) par exemple.

DrawText() affiche le texte dans un rectangle rc selon un format qui peut être précisé, et en particulier sur plusieurs lignes...

D'autre part, SetTextColor() définit la couleur du texte au niveau du DC à l'aide d'un codage RGB des couleurs exposé ci-après.

De même SetBkColor() définit la couleur du fond entre les caractères.

Enfin SetBkMode() précise si les espaces intercaractères doivent être remplis avec la couleur de fond ou laissés dans la couleur existante, selon la valeur OPAQUE ou TRANSPARENT du paramètre mode.

Remarque: Les fonctions GetBkMode() et GetBkColor() permettent de récupérer les valeurs courantes au niveau du DC.

Les couleurs RGB sont codées sur 32 bits avec dans les trois octets de poids faible les composantes Red, Green et Blue de la couleur. La macro-fonction RGB() permet de créer des couleurs à partir de leurs composantes fondamentales:

Syntaxe: COLORREF RGB(BYTE r, BYTE g, BYTE b);

Exemples:	Rouge pur	RGB(255, 0, 0)
	Vert pur	RGB(0, 255, 0)
	Bleu pur	RGB(0, 0, 255)
	Blanc	RGB(255,255,255)
	Noir	RGB(0, 0, 0)
	Magenta	RGB(255, 0, 255)
	Jaune	RGB(255, 255,0)
	Cyan	RGB(0, 255, 255)
	...	

D'autres combinaisons intermédiaires sont acceptables selon le nombre de couleurs supportées par la vidéo; Windows utilisera de toute façon la couleur la plus proche que puisse supporter le périphérique !

III) Le graphisme:

Toujours à l'aide d'un DC on peut effectuer de nombreux tracés et remplissages rien qu'avec les fonctions GDI existantes. Les tracés sont réalisés à l'aide d'un crayon et les remplissages à l'aide d'une brosse.

1) Crayons et brosses:

Il existe trois crayons pré-définis, WHITE_PEN, BLACK_PEN et NULL_PEN, auxquels on accède à l'aide de GetStockObject() qui retourne dans ce cas un handle de crayon de type HPEN. Pour sélectionner ce crayon au niveau du DC, il faut appeler SelectObject() comme pour les polices vues plus haut...

Exemple: HDC hdc;
 HPEN pen;

 hdc= GetDC(hWnd);
 pen= GetStockObject(BLACK_PEN);
 SelectObject(hdc, pen);
 ...
 ReleaseDC(hWnd, hdc);

Remarque: Par défaut BLACK_PEN est sélectionné dans le DC.
 NULL_PEN servira à faire des remplissages sans bordure plus tard.

Les crayons standards font un trait plein de 1 pixel de large, mais on peut créer des crayons personnalisés via CreatePen(). Ils doivent être détruits par DeleteObject().

Syntaxe: HPEN CreatePen(int penstyle, int width, COLORREF color);

Le paramètre penstyle détermine le style de tracé parmi:

PS_SOLID	Trait plein
PS_DASH	Tirets
PS_DOT	Trait pointillé
PS_DASHDOT	Trait mixte
PS_DASHDOTDOT	Trait mixte

Bien entendu width est la largeur du trait, mais seuls les traits pleins supportent une largeur autre que 1 !

Enfin color est la couleur RGB du trait.

Il existe aussi quelques brosses pré-définies, nommées WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH, DKGRAY_BRUSH, BLACK_BRUSH, NULL_BRUSH.

On y accède toujours avec GetStockObject() qui retourne pour la circonstance un handle de brosse de type HBRUSH , que l'on sélectionne dans le DC avec SelectObject().

On peut aussi créer des brosse avec CreateSolidBrush() et CreateHatchBrush() qui devront être détruites par DeleteObject().

Syntaxe: HBRUSH CreateSolidBrush(COLORREF color);
 HBRUSH CreateHatchBrush(int style, COLORREF color);

Les brosses solides peignent entièrement dans la couleur RGB indiquée tandis que les autres font des hachures de divers styles:

HS_HORIZONTAL	Hachures horizontales
---------------	-----------------------

HS_VERTICAL	Hachures verticales
HS_FDIAGONAL	Hachures obliques
HS_BDIAGONAL	Hachures obliques inverses
HS_CROSS	Hachures croisées
HS_DIAGCROSS	Hachures croisées obliques

Le traitement du fond entre les hachures dépend du mode de couleur de fond déterminé par SetBkMode().

Remarque: On pourra mémoriser les crayons, brosses, fontes utiles au programme dans des variables **statiques** initialisées dans le traitement du message WM_CREATE de la fenêtre associée au DC et les détruire lors du traitement du message WM_DESTROY.

2) Les fonctions de tracé:

Les fonctions de tracé les plus importantes sont SetPixel(), GetPixel(), MoveTo(), LineTo(), et GetCurrentPosition().

Elles utilisent le crayon sélectionné dans le DC. Pour tracer des cercles, ellipses, rectangles on utilisera les fonctions de remplissage avec la brosse NULL_BRUSH !

Syntaxe: SetPixel(HDC hdc, int x, y, COLORREF color);
 COLORREF GetPixel(HDC hdc, int x, y);
 MoveTo(HDC hdc, int x, y);
 LineTo(HDC hdc, int x, y);
 DWORD GetCurrentPosition(HDC hdc);

La fonction SetPixel() permet de colorier le pixel de coordonnées (x,y) dans la couleur RGB spécifiée, tandis que GetPixel() permet de récupérer la couleur d'un pixel.

La fonction MoveTo() déplace le crayon, dont la position est mémorisée au niveau du Device Context. LineTo() trace une ligne allant de la position courante du crayon jusqu'au point de coordonnées (x,y) sans toutefois colorier ce dernier point, qui devient la position courante du crayon.

Quant à GetCurrentPosition(), elle permet de récupérer cette position courante du crayon dans un entier dont les mots de poids faible et fort contiennent respectivement les coordonnées x et y. On peut extraire x et y avec les macro-fonctions LOWORD() et HIWORD():

Syntaxe: int LOWORD(DWORD u);
 int HIWORD(DWORD u);

3) Les fonctions de remplissage:

Les fonctions de base dans ce domaine sont Rectangle(), RoundRect(), Ellipse(), Arc(), Chord() et Pie().

Elles utilisent le crayon courant pour le contour et la brosse courante pour le remplissage.

Syntaxe: Rectangle(HDC hdc, int x1, y1, x2, y2);
 RoundRect(HDC hdc, int x1, y1, x2, y2, xe, ye);
 Ellipse(HDC hdc, int x1, y1, x2, y2);
 Arc(HDC hdc, int x1, y1, x2, y2, xs, ys, xe, ye);
 Chord(HDC hdc, int x1, y1, x2, y2, xs, ys, xe, ye);
 Pie(HDC hdc, int x1, y1, x2, y2, xs, ys, xe, ye);

La fonction Rectangle trace et remplit le rectangle indiqué. RoundRect() réalise un rectangle aux angles arrondis en ellipses d'axes de longueurs xe et ye.

Ellipse() réalise une ellipse contenue dans le rectangle circonscrit indiqué.

La fonction Arc() réalise un arc d'ellipse où (xs,ys) et (xe,ye) sont les coordonnées de points qui ne sont pas nécessairement sur l'ellipse mais sur des demi-droites partant du centre de celle-ci et qui délimiteront l'arc...

Chord() et Pie() font de même tout en joignant les extrémités de l'arc en une corde ou en les reliant au centre de l'ellipse.

IV) Application au programme démineur:

1) Principes:

Il faut maintenant mettre en oeuvre toutes ces notions afin de réaliser la grille du démineur et de prévoir son évolution au cours du jeu...

On va compléter pour cela WSTEP1.C afin de gérer une grille de jeu de taille 10x10 ou 10x20 apparaissant lors de l'accès à Jeu → Nouveau dans le menu.

On créera une fonction affichage() chargée d'afficher la grille en début de partie, mais qui sera aussi appelée pour le traitement des messages WM_PAINT; à cette fin on doit mémoriser en permanence l'état de la grille de jeu, par exemple dans un tableau d'entiers bidimensionnel:

```
int grille[20][10]
```

L'état de chaque case est reflété par une valeur parmi les suivantes:

```
#define VIDE            0        case vide, sans mine voisine  
                          1 à 8    case vide avec nombre de mines voisines  
#define MINE           255     case minée  
#define SHOW          1024    case découverte
```

La constante SHOW sera ajoutée à la valeur de chaque case qui a été découverte au cours de la partie.

Au départ il faudra placer aléatoirement les mines dans la grille, puis il faudra calculer la valeur de toutes les autres cases en fonction du nombre de mines voisines.

Exemple: 0 1 1 1

```

1 2 1
1 2 1
1 1 1 0

```

Les cases non encore découvertes seront grisées.

2) Réalisation:

Réaliser les organigrammes puis écrire les fonctions:

Fonction:

Rôle:

`void init_grille()`

Initialisation aléatoire de la grille de jeu.

`void affichage(HDC)` Affichage de l'état actuel de la grille, comprenant la grille elle-même, les cases grisées non encore découvertes et les cases découvertes contenant éventuellement un chiffre.
Affichage des textes adjacents: nom du joueur, nombre total de mines, nombre de cases résiduelles.

Ces deux fonctions seront appelées en début de partie, et `affichage()` sera aussi appelée dans le traitement de `WM_PAINT`, après un `InvalidateRect()` permettant de redessiner toute la zone client de la fenêtre.

On pourra commencer par la fonction `affichage()` et réaliser une grille entièrement masquée !

I) Introduction:

La plupart — sinon toutes — les applications Windows nécessitent une interaction avec l'utilisateur à travers le clavier et la souris. ces interactions sont prises en charge par Windows lorsqu'il s'agit de ressources telles que les menus, raccourcis clavier ou boîtes de dialogue. Lorsqu'il s'agit de gérer le **clavier ou la souris** au niveau d'une fenêtre de l'application, il est du ressort du programmeur de traiter les **messages** émis par ces deux périphériques. Les messages clavier et souris passent par la file d'attente de messages de la fenêtre concernée.

Remarque: Seule la fenêtre active peut recevoir les messages clavier. Signalons au passage qu'une fenêtre reçoit le message WM_SETFOCUS lorsqu'elle devient active et le message WM_KILLFOCUS lorsqu'elle devient inactive.

II) Le clavier:

1) Principes:

Windows distingue deux types de messages clavier:

- les messages d'événement clavier
- les messages de caractère

Les premiers sont:

WM_KEYDOWN, WM_KEYUP,
WM_SYSKEYDOWN, WM_SYSKEYUP

L'appui sur une touche ordinaire du clavier génère un WM_KEYDOWN et son relâchement génère un WM_KEYUP. En cas de répétition automatique, plusieurs WM_KEYDOWN peuvent être émis suivis d'un seul WM_KEYUP.

Le paramètre lParam d'un WM_KEYDOWN contient quelques renseignements parfois utiles mais parlons surtout de wParam qui contient le code de touche virtuelle qui identifie la touche pressée au clavier par un code VK_... , tel que

VK_TAB, VK_RETURN, VK_SHIFT, VK_UP, ...
VK_F1, VK_A ...

Les messages WM_SYSKEYDOWN et WM_SYSKEYUP sont émis pour des combinaisons de touches prédéfinies (Alt + Tab, Alt + F4 ...). Ces messages sont en général laissés à Windows, c'est à dire transmis à DefWindowProc() !

L'utilisation directe des messages d'événements clavier est problématique dans la mesure où une même touche peut correspondre à '&' sur un clavier Azerty et à '1' sur un clavier Qwerty...

Heureusement Windows propose l'approche des messages de caractère, qui permet d'obtenir des caractères conformes à ce qui est inscrit sur le clavier !

Pour profiter de cette possibilité il suffit de transmettre tous les messages à TranslateMessage() avant d'appeler DispatchMessage(), ce que nous avons prévu dès le début !

Les messages de caractère à traiter sont alors:

WM_CHAR, WM_DEADCHAR
WM_SYSCHAR, WM_SYSDEADCHAR

WM_CHAR est le seul réellement indispensable; il correspond aux caractères usuels du clavier.

WM_DEADCHAR est émis pour les préfixes de caractères comme ^ et `` qui doivent être suivis d'un caractère normal; toutefois il n'est pas nécessaire de s'en préoccuper car Windows en tiendra compte dans le prochain WM_CHAR !

Le paramètre lParam de WM_CHAR contient la même chose que pour un WM_KEYDOWN tandis que wParam contient le code ASCII du caractère.

Remarque: Les messages d'événements sont transmis quand même mais on ne s'en préoccupe plus.

2) La table de caractères Windows:

Les caractères de codes ASCII 0 à 127 (7 bits) sont toujours les mêmes quel que soit l'environnement, mais il existe des extensions diverses pour les codes 128 à 255. Windows utilise la table ANSI étendue disponible en annexe.

Elle contient surtout des lettres accentuées et certains codes ne sont pas utilisés. MS-DOS utilise des pages de codes différentes. Il existe des fonctions de l'API Windows pour la conversion, telles que OemToAnsi() et AnsiToOem(). Ces fonctions peuvent être utiles pour convertir un fichier ASCII du format MS-DOS au format Windows.

Remarque: Pour convertir des caractères (accentués) en majuscules il convient d'utiliser AnsiUpper() et non la fonction C toupper().

III) La souris:

1) Principes:

La présence d'une souris peut être détectée par GetSystemMetrics(SM_MOUSEPRESENT) mais rien ne permet de déterminer le nombre de boutons de la souris !

Les messages souris sont essentiellement:

WM_LBUTTONDOWN	Clic sur le bouton gauche
WM_MBUTTONDOWN	Clic sur le bouton central

WM_RBUTTONDOWN	Clic sur le bouton droit
WM_LBUTTONUP	Relâchement du bouton gauche
WM_MBUTTONUP	Relâchement du bouton central
WM_RBUTTONUP	Relâchement du bouton droit
WM_LBUTTONDOWNBLCLK	Double clic sur le bouton gauche
WM_MBUTTONDOWNBLCLK	Double clic sur le bouton central
WM_RBUTTONDOWNBLCLK	Double clic sur le bouton droit
WM_MOUSEMOVE	Déplacement de la souris

Pour tous ces messages le champ IParam contient les coordonnées du curseur de la souris au moment du message:

x= LOWORD(IParam)
y= HIWORD(IParam)

Quant à wParam il contient l'état des touches SHIFT et CTRL qui peuvent être utilisées conjointement à la souris:

(wParam & MK_SHIFT) == MK_SHIFT si SHIFT appuyé,
(wParam & MK_CONTROL) == MK_CONTROL si CTRL appuyé.

Les messages WM_MOUSEMOVE sont émis périodiquement au cours du déplacement de la souris, mais de façon un peu aléatoire...

Remarque: On peut à tout moment récupérer la position du curseur de souris ou la modifier avec GetCursorPos() et SetCursorPos().

2) Capture de souris:

Les messages de la souris sont adressés par défaut à la fenêtre où se trouve le curseur. cela peut poser des problèmes lorsqu'une application (de dessin par exemple) attend la fin d'une manipulation marquée par un relâchement d'un bouton de la souris: si ce relâchement a lieu dans une autre fenêtre, l'application ne le verra pas et risque de rester bloquée...

Pour parer à cela l'application peut capturer la souris afin de recevoir tous les messages souris. Pour cela on appelle SetCapture(). La capture doit être limitée dans le temps afin de ne pas paralyser Windows; on doit donc appeler ReleaseCapture() pour libérer la souris:

Syntaxe: SetCapture(HWND hWnd);
 ReleaseCapture();

Les coordonnées souris sont toujours exprimées par rapport au coin supérieur gauche de la zone client de la fenêtre; elles peuvent donc devenir négatives lors d'une capture !

3) Curseur souris:

La forme du curseur souris est initialement définie dans la classe de fenêtre, mais on peut en changer avec LoadCursor(), SetCursor() et ShowCursor():

Syntaxe: HCURSOR LoadCursor(HINSTANCE hInst, LPCSTR CURSOR_NAME);
 HCURSOR SetCursor(HCURSOR Curs);
 ShowCursor(BOOL mode);

LoadCursor() reçoit comme paramètres NULL pour une ressource système et un nom de curseur prédéfini tel que IDC_ARROW (flèche), IDC_WAIT (sablier), IDC_CROSS (croix) et retourne un handle de type HCURSOR.

SetCursor() active ce curseur souris et retourne le handle du précédent curseur.

ShowCursor() reçoit comme paramètre TRUE ou FALSE selon que l'on veut montrer ou cacher le curseur souris.

Attention: Le curseur souris est partagé par toutes les fenêtres et toutes les applications et doit donc être restauré lorsque la fenêtre perd le focus.

ShowCursor() comptabilise les appels et par conséquent un certain nombre d'appels avec le paramètre FALSE doivent être compensés par le même nombre d'appels avec le paramètre TRUE si on veut que le curseur revienne !

IV) Applications au démineur:

Il s'agit dans le programme démineur de gérer les messages WM_LBUTTONDOWN et de récupérer les coordonnées graphiques de la souris, de les traduire en coordonnées virtuelles dans le tableau grille[][].

On pourra alors réaliser l'action nécessaire si la case n'est pas déjà découverte.

Si la case est minée on mettra fin à la partie, qui est perdue.

Si le nombre de cases résiduelles est égal au nombre de mines de la grille on mettra fin à la partie, qui est alors gagnée.

D) Manipulations de fichiers:

1) Fonctions de base:

Il est possible, programmant en C, d'utiliser pour les accès fichiers les fonctions standard du langage C:

- Fonctions de bas niveau de la famille `open()`, `read()`, `write()`, `close()`... qui utilisent un accès direct à l'aide d'un handle de type `int`.
- Fonctions de haut niveau de la famille `fopen()`, `fget...()`, `fput...()`, `fclose()`... autorisant des accès tamponnés à l'aide de pointeurs de type `FILE*`.

Consulter à ce sujet le polycopié de C. Le principal inconvénient de ces fonctions est qu'elles ne peuvent manipuler que des blocs de données de taille inférieure à 64 KO, ce qui oblige à faire des boucles pour manipuler de gros fichiers.

2) Fonctions 16 bits (Windows 3.X):

Windows offre des fonctions spécifiques qui étaient initialement à usage interne et ont par conséquent des noms underscoreés. Elles utilisent un handle de type `HFILE` et peuvent manipuler des blocs de données supérieurs à 64 KO.

On leur préférera toutefois d'autres fonctions sous Windows 95 ou 98, en particulier pour les noms longs.

Syntaxe: `HFILE _lopen(char * filename, int amode);`
 `HFILE _lcreat(char * filename, UINT attrib);`

On donne le nom du fichier suivi du mode d'ouverture parmi les constantes `OF_READ`, `OF_WRITE` ou `OF_READ_WRITE` pour `_lopen()`, et suivi du type de fichier à créer parmi 0= Normal, 1= ReadOnly, 2= Hidden, 3= System pour `_lcreat()`.

Elles retournent le handle associé au fichier, ou `HFILE_ERROR` en cas de problème.

Les fonctions de lecture / écriture associées varient selon la taille des données:

`_lread()` et `_lwrite()` pour des blocs de moins de 64 KO et `_hread()` et `_hwrite()` pour des blocs jusqu'à 2 GO, qui doivent alors être alloués dynamiquement !

Syntaxe: `UINT _lread(HFILE hd, LPSTR data, UINT taille);`
 `LONG _hread(HFILE hd, LPVOID data, LONG taille);`
 `UINT _lwrite(HFILE hd, LPSTR data, UINT taille)`
 `LONG _hwrite(HFILE hd, LPVOID data, LONG taille)`

Ces fonctions retournent le nombre d'octets effectivement lus ou écrits dans le fichier.

Pour ce déplacer dans un fichier (en lecture) on dispose de la fonction `_llseek()`:

Syntaxe: `LONG _llseek(HFILE hd, LONG offset, int origine);`

On y indique le nombre d'octets de déplacement par rapport à l'origine choisie parmi

Début du fichier= 0
Position courante= 1
Fin du fichier= 2

La fonction retourne la position absolue par rapport au début du fichier.

Pour fermer le fichier un simple appel à `_lclose()` suffit en lui passant le handle du fichier:

Syntaxe: `_lclose(HFILE hd);`

3) Les fonctions 32 bits:

Sous Windows 95, 98 ou NT, on préférera utiliser les fonctions 32 bits suivantes qui supportent des blocs de 4 GO de données et les noms longs (qui ont une taille maximale de 256 caractères).

L'inconvénient est le nombre de paramètres à préciser lors de l'appel à `CreateFile()`, qui sert à créer comme à ouvrir les fichiers ou les répertoires... Cette fonction retourne un handle de type HANDLE en cas de succès et `INVALID_HANDLE_VALUE` en cas d'erreur.

Syntaxe: `CreateFile(char * filename, DWORD amode, DWORD shmode, void * secmode, DWORD ofmode, DWORD flags, HANDLE hf);`

Dans cette litanie de paramètres, retenons que

- filename= nom du fichier,
- amode= mode d'accès `GENERIC_READ`, `GENERIC_WRITE` ou les deux,
- shmode= mode de partage `FILE_SHARE_READ` ou `FILE_SHARE_WRITE`,
- ofmode= mode d'ouverture parmi
 - `CREATE_NEW` (échec si existe déjà)
 - `CREATE_ALWAYS` (efface si existe déjà)
 - `OPEN_EXISTING` (échec si n'existe pas)
 - `TRUNCATE_EXISTING` (échec si n'existe pas et efface sinon)
- flags= combinaison ! de constantes de type attribut telle que
 - `FILE_ATTRIBUTE_NORMAL` et éventuellement de flags
 - `FILE_FLAG_*`
- secmode= options de sécurité (pour NT), NULL par défaut,
- hf= handle d'un fichier modèle, NULL par défaut.

Exemple: Ouverture d'un fichier standard existant en lecture seule partagée...

```
hd= CreateFile("toto.txt", GENERIC_READ, FILE_SHARE_READ, NULL,  
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

Le plus dur étant fait, on peut alors lire ou écrire dans le fichier avec ReadFile() et WriteFile():

```
Syntaxe:   ReadFile( HANDLE hd, LPVOID data, DWORD taille, DWORD * nb,  
              LPVOID over );  
           WriteFile( HANDLE hd, LPVOID data, DWORD taille, DWORD * nb,  
              LPVOID over );
```

Ici les paramètres sont:

- data= adresse des données à lire ou à écrire,
- taille= nombre d'octets à lire ou à écrire,
- *nb= nombre d'octets effectivement lus ou écrits,
- over= pointeur NULL en général.

On peut se déplacer dans un fichier avec SetFilePointer() et on doit le fermer après usage par un appel à CloseHandle():

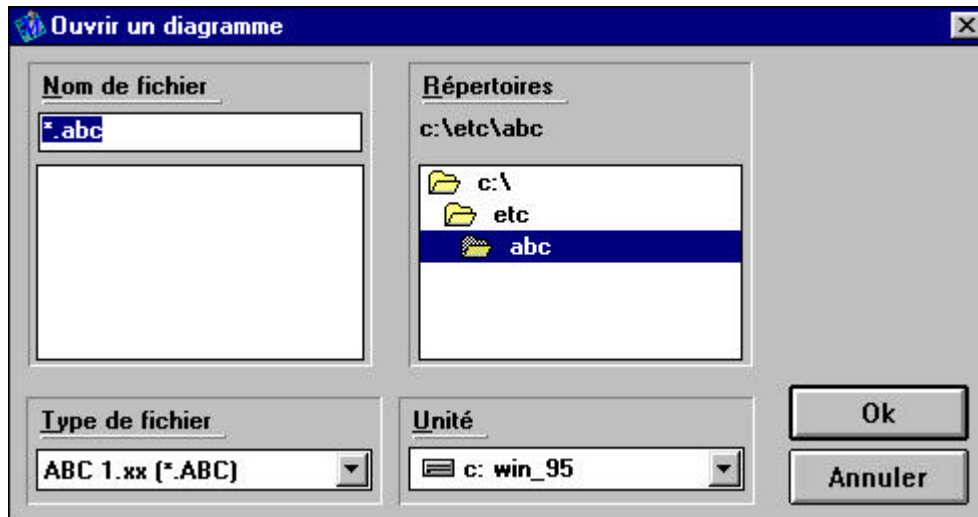
```
Syntaxe:   CloseHandle( HANDLE hd );
```

Remarque: Toutes ces fonctions 32 bits permettent aussi d'ouvrir des pipes et des périphériques de communication...

II) Les boîtes de dialogue prédéfinies Open / Save:

Depuis Windows 3.1 existent divers modèles de boîtes de dialogues prédéfinies (appelées Common dialogs) et en particulier des boîtes pour l'ouverture ou l'enregistrement des fichiers offrant le choix du lecteur, du répertoire, du nom (etc.) ce qui simplifie grandement la programmation.

Leur utilisation nécessite l'inclusion du header <commdlg.h> et l'accès aux bibliothèques d'importation COMDLG.LIB ou COMDLG32.LIB associées aux mêmes DLL.



Ces boîtes sont dans la langue utilisée par Windows.

Le principe consiste à préparer la boîte avant son apparition et à récupérer le contenu lors de la fermeture; tout le reste est du ressort du programme. Elles utilisent une structure prédéfinie OPENFILENAME et les fonctions respectives GetOpenFileName() et GetSaveFileName(), qui se chargent d'appeler les boîtes de dialogue.

```

struct OPENFILENAME
{
    DWORD           lStructSize;
    HWND           hwndOwner;
    HINSTANCE      hInstance;
    LPCTSTR        lpstrFilter;
    LPTSTR         lpstrCustomFilter;
    DWORD          nMaxCustFilter;
    DWORD          nFilterIndex;
    LPTSTR         lpstrFile;
    DWORD          nMaxFile;
    LPTSTR         lpstrFileTitle;
    DWORD          nMaxFileTitle;
    LPCTSTR        lpstrInitialDir;
    LPCTSTR        lpstrTitle;
    DWORD          Flags;
    WORD           nFileOffset;
    WORD           nFileExtension;
    LPCTSTR        lpstrDefExt;
    DWORD          lCustData;
    LPOFNHOOKPROC  lpfnHook;
    LPCTSTR        lpTemplateName;
};

```

Les champs utiles sont les suivants:

- lStructSize= sizeof(OPENFILENAME),
- hwndOwner= handle de la fenêtre mère,
- lpstrFilter= filtre d'affichage des noms de fichiers; on peut cumuler plusieurs filtres en les séparant par des \0 et des ;
Par exemple:
"Fichiers Textes\0*.txt;*.doc\0Tous fichiers\0*.*\0"
Cette chaîne se termine de fait par un double NUL final !
- lpstrFile= nom et chemin d'accès au fichier,
- nMaxFile= longueur maximale de la chaîne lpstrFile,
- lpstrInitialDir= Lecteur et répertoire à lister, NULL si répertoire courant,
- lpstrTitle= titre de la boîte de dialogue,
- lpstrFileTitle= nom du fichier sans chemin,
- Flags= 0 en général.

Les autres champs sont à mettre à la valeur 0 ou NULL selon leur type.

Avant l'appel de GetOpenFileName() ou GetSaveFileName() on doit initialiser la structure, et en particulier préciser le lecteur, le répertoire, les filtres, un nom de fichier par défaut...

Syntaxe: BOOL GetOpenFileName(struct OPENFILENAME * ofn);
 BOOL GetSaveFileName(struct OPENFILENAME * ofn);

On passe alors l'adresse de la structure préparée et on reçoit TRUE ou FALSE selon que l'utilisateur est sorti de la boîte par OK ou Annuler. On consulte alors les modifications apportées à la structure pour réaliser les opérations d'ouverture ou de sauvegarde.

D) Allocation dynamique:

S'il est possible, pour allouer dynamiquement une zone mémoire — dans le tas ou heap — d'utiliser les fonctions du C standard, il est néanmoins préférable sous Windows d'utiliser les fonctions système dans la mesure où la gestion de la mémoire sous Windows est assez complexe; en effet, l'aspect multitâche de Windows fait que plusieurs applications se partagent les ressources machine à un instant donné, qu'un programme peut être lancé ou arrêté à tout moment ce qui a tendance à fragmenter la mémoire ...

Aussi, pour réduire cette fragmentation, Windows doit-il pouvoir déplacer des blocs mémoire (code, données ...) ou même les swapper sur disque temporairement pour faire de la place, sans que cela n'affecte les processus concernés, qui ne doivent se rendre compte de rien ...

Il est donc délicat d'allouer une zone de mémoire dynamique, sur laquelle on possède alors un pointeur qui est utilisé dans le programme, et qui ne peut donc être déplacée.

Pour remédier à cela, Windows manipule la mémoire allouée dynamiquement par GlobalAlloc() à l'aide de handles mémoire de type HGLOBAL. Un pointeur n'est attribué que lorsqu'on accède effectivement à la zone allouée, qui est alors verrouillée par GlobalLock().

Dès que possible on déverrouille la zone par GlobalUnlock(), sans pour autant perdre les données qu'elle contient: la zone mémoire redevient simplement mobile.

Quand la zone n'est plus utile on la libère par GlobalFree().

Syntaxe: HGLOBAL GlobalAlloc(unsigned memflag, DWORD Size);
 void * GlobalLock(HGLOBAL hmem);
 GlobalUnlock(HGLOBAL hmem);
 GlobalFree(HGLOBAL hmem);

Pour allouer un bloc mémoire dans le tas global on indique à GlobalAlloc() la taille du bloc en octets dans Size, jusqu'à 4 GO en théorie. On doit aussi indiquer un flag de mobilité de la zone, parmi les constantes:

memflag=	GMEM_MOVEABLE	Zone amovible
	GHND= GMEM_MOVEABLE GMEM_ZEROINIT	Zone amovible initialisée à 0
	GMEM_FIXED	Zone inamovible (déconseillé)

Le handle mémoire obtenu en retour est utilisé pour tous les accès ultérieurs. En cas d'erreur ou d'impossibilité, ce qui doit être impérativement testé, la fonction GlobalAlloc() retourne NULL. Il en est de même pour GlobalLock() si le verrouillage de la zone s'avère impossible.

Remarque: Chaque verrouillage d'un bloc incrémente en fait un compteur, et chaque déverrouillage le décrémente; Windows ne peut déplacer un bloc mémoire que si ce compteur est à 0. Il est souhaitable de procéder au verrouillage puis au déverrouillage d'un bloc de façon aussi

rapide que possible, et si possible dans le traitement du même message, afin d'éviter des soucis à Windows qui entre deux messages peut passer la main à une autre application !

II) Images Bitmap:

Les images Bitmap sont stockées dans des fichiers DIB (Device Independent Bitmap) d'extension .BMP

La taille souvent imposante de ces fichiers fait que leur manipulation nécessite en général l'allocation dynamique, d'où le lien avec la partie précédente ...

On va écrire une fonction de chargement en mémoire d'un fichier Bitmap et voir comment on peut alors afficher l'image dans une fenêtre.

Il faut savoir qu'un fichier BMP comporte plusieurs sections:

- structure BITMAPFILEHEADER informations sur le fichier BMP
- structure BITMAPINFOHEADER informations sur l'image bitmap
- palette de couleurs (facultative)
- bits de donnée de l'image.

```
La structure      struct BITMAPFILEHEADER /* bmfh */
                  {
                    UINT      bfType;
                    DWORD      bfSize;
                    UINT      bfReserved1;
                    UINT      bfReserved2;
                    DWORD      bfOffBits;
                  } ;
```

Les champs intéressants y sont les suivants:

- bfType= les lettres BM sur deux octets pour identifier le fichier,
- bfSize= taille totale du fichier BMP en octets,
- bfOffBits= offset des données de l'image par rapport au début du fichier.

```
La structure      struct BITMAPINFOHEADER /* bmih */
                  {
                    DWORD      biSize;
                    LONG       biWidth;
                    LONG       biHeight;
                    WORD       biPlanes;
                    WORD       biBitCount;
                    DWORD      biCompression;
                    DWORD      biSizeImage;
                    LONG       biXPelsPerMeter;
                  }
```

```

LONG      biYPelsPerMeter;
DWORD    biClrUsed;
DWORD    biClrImportant;
};

```

Les champs utiles sont dans cette structure:

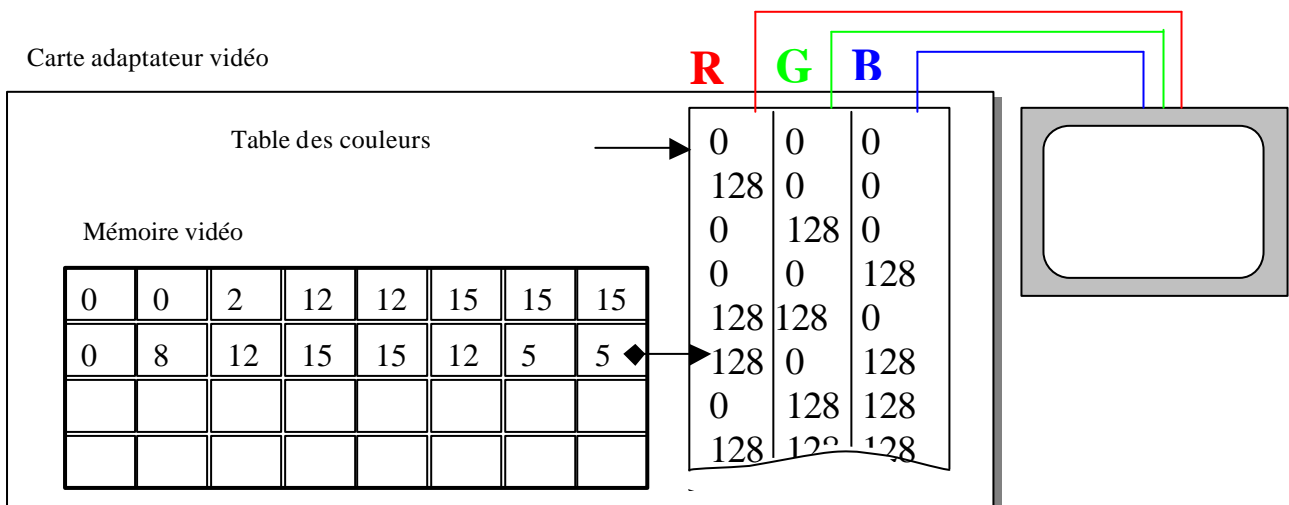
- biSize= taille de la structure,
- biWidth= largeur de l'image en pixels
- biHeight= hauteur de l'image en pixels
- biBitCount= 1, 4, 8 ou 24 càd le nombre de bits de couleur par pixel
 - 1 = bitmap noir et blanc
 - 4 = bitmap en 16 couleurs
 - 8 = bitmap en 256 couleurs
 - 24 = bitmap en 16 millions de couleurs
- biClrUsed= nombre de couleurs utilisées
 - 0 = comme ci dessus, valeur par défaut
- bmiColors= Palette de couleurs, excepté pour les bitmaps 24 bits qui sont codés directement en RGB

La palette de couleur ci-dessus est nécessaire pour les bitmaps en 2, 16 ou 256 couleurs; elle définit ces couleurs au standard RGB de la façon suivante:
 c'est un tableau de double-mots, appelés RGBQUADs contenant chacun la définition d'une couleur, selon son numéro dans la palette, de 0 à 255 par exemple pour un Bitmap 16 bits:

```

RGBQUAD:      Octet      0      1      2      3
               Composante Blue Green Red 0

```



Ensuite commence la description de l'image elle-même — qui peut être compressée — que les fonctions Windows se chargeront d'analyser dans tous les cas ...

Pour l'affichage correct de l'image, ces mêmes fonctions ont besoin de la structure INFO et de tout ce qui suit; aussi va-t-on charger tout le fichier BMP en mémoire (en supposant que sa taille le permette) et ensuite on pourra l'afficher à l'aide des fonctions SetDIBitsToDevice() ou StretchDIBits(); la seconde est capable de dilater ou compresser l'image pour l'adapter à la taille de la fenêtre cible.

Syntaxe: int SetDIBitsToDevice(HDC hdc, int x0, y0, w1, h1, x1,y1, UINT
 startline, nblines, BYTE * Image, BITMAPINFO * bmih, UINT mode);
 int StretchDIBits(HDC hdc, int x0, y0, w0, h0, x1, y1, w1, h1, BYTE * Image,
 BITMAPINFO * bmih, UINT mode, DWORD comb);

Les nombreux paramètres de la fonction SetDIBitsToDevice() sont:

- x0, y0 coordonnées d'affichage du Bitmap dans la fenêtre associée au hdc,
- w1, h1 dimensions de la portion de l'image à afficher,
- x1, y1 coordonnées du coin supérieur de la partie d'image à afficher
- startline numéro de la première ligne de l'image pointée par Image
 (=0 pour la première)
- nblines nombre de lignes de l'image à afficher
- Image adresse de début des données de l'image
- bmih adresse de la structure INFO associée
- mode constante DIB_RGB_COLORS

Pour la fonction StretchDIBits() il y a quelque nuances:

- w0, h0 dimensions de la zone de la fenêtre devant recevoir l'image
- comb mode de combinaison des bits avec le fond de la fenêtre.

Ces fonctions retournent le nombre de lignes effectivement affichées. En cas d'erreur elles retournent la valeur 0.

En pratique, on récupère les champs biWidth et biHeight de la structure INFO pour avoir les dimensions de l'image, on transmet l'adresse de cette même structure INFO calculée par un décalage de sizeof(BITMAPFILEHEADER) par rapport au début de l'ensemble.

Quant à l'adresse des données de l'image, on l'obtient par un décalage de bfOffbits.

Remarque: On utilisera StretchDIBits() si on doit redimensionner l'image, en cas de zoom par exemple; les calculs d'interpolation de l'image sont alors pris en charge par Windows.

Exemple:

```
#include <windows.h>
#include <stdio.h>

void ReadBitmapFile(char * bmpname, HWND hwnd);

void ReadBitmapFile(char * bmpname, HWND hwnd)
{
    HANDLE handFile;
    BITMAPFILEHEADER bmfh;
    HGLOBAL hmem;
    BITMAPINFOHEADER * bmi;
    BYTE * bmp;
    HDC hdc;
    DWORD Size, Offset;
    long l;
    int res;

    handFile=CreateFile(bmpname, GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL, NULL);

    if (handFile == INVALID_HANDLE_VALUE)
        MessageBox(hwnd, "Pb_Open()", "", 0);
    ReadFile(handFile, &bmfh, sizeof(BITMAPFILEHEADER), &l, NULL);
    if (bmfh.bfType != * (WORD *) "BM") // ce n'est pas un fichier .BMP
        MessageBox(hwnd, "Pb_BMP", "", 0);

    // taille du fichier BMP = taille des donnees + taille des entete
    Size = bmfh.bfSize + bmfh.bfOffBits;

    // On lit BITMAPINFO structure + donnees.
    hmem = GlobalAlloc(GHND, Size);
    if (hmem == NULL) MessageBox(hwnd, "Pb alloc()", "", 0);
    bmi = GlobalLock(hmem);
    if (bmi == NULL) MessageBox(hwnd, "Pb lock()", "", 0);

    ReadFile(handFile, bmi, Size, &l, NULL);
    // On calcule l'offset des donnees par rapport au debut de BITMAPINFO
    Offset= bmfh.bfOffBits - sizeof(BITMAPFILEHEADER);
    bmp= (BYTE *) bmi + Offset;

    // Affichage du bitmap
    hdc= GetDC(hwnd);
    res = SetDIBitsToDevice(hdc, 0,0, bmi->biWidth, bmi->biHeight,
                           0, 0, 0, bmi->biHeight, bmp, (BITMAPINFO *) bmi, DIB_RGB_COLORS);
    if (res == 0)
        MessageBox(hwnd, "Pb_BMP", "", 0);
    // On peut aussi utiliser StretchDIBits()
    ReleaseDC(hwnd,hdc);

    // liberation memoire maintenant que tout est dans le hdc
    GlobalUnlock(hmem);
    GlobalFree(hmem);
    // fermeture fichier .BMP
    CloseHandle(handFile);
}
```


D) Les principes:

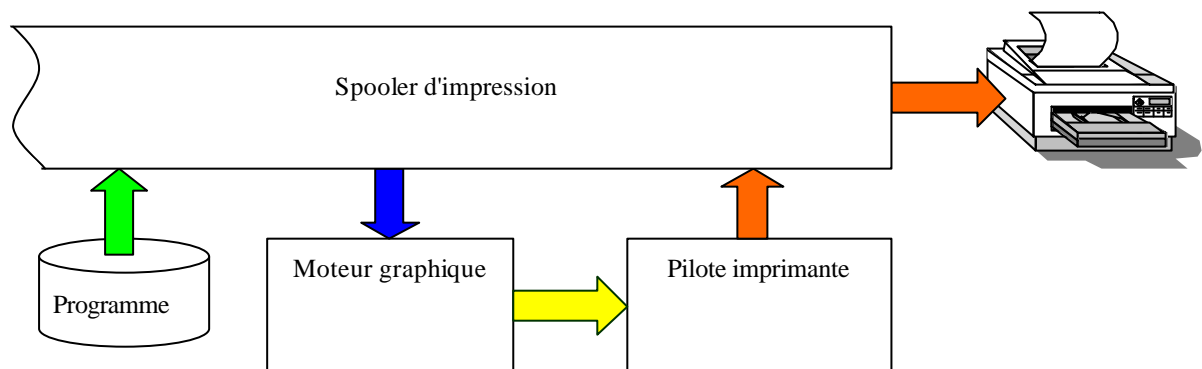
1) La gestion des impressions:

La gestion des imprimantes connectées au système est du ressort de Windows, à travers des drivers de périphériques fournis avec Windows ou avec les imprimantes.

Nous ne parlerons que des imprimantes en mode point (matricielles, jet d'encre ou laser).

Windows assure aux applications une certaine indépendance vis à vis du matériel, comme pour les affichages à l'écran. On va d'ailleurs utiliser, pour imprimer un document un Device Context (DC) d'imprimante et les mêmes fonctions GDI ...

On peut, en simplifiant un peu, résumer la procédure d'impression d'un document passant par le gestionnaire d'impression ou spooler:



Le rôle du driver d'imprimante est de convertir les appels GDI (Graphic Device Interface) en codes de contrôle d'imprimante, ou en langage d'imprimante tel que PCL ou POSTSCRIPT.

Le rôle du spooler (SPOOLER.EXE) est de mettre en attente les impressions et de les soumettre à l'imprimante au fur et à mesure.

2) Programmation

L'impression d'un document se fait en plusieurs étapes:

- Obtention d'un DC imprimante,
- Création du document, page par page, avec les fonctions GDI habituelles pour les textes et graphiques,
- traduction et Impression de chaque page,
- Libération du DC.

On peut obtenir un DC imprimante avec CreateDC() et EnumPrinters(), mais il est plus simple d'avoir recours à la fonction PrintDlg() qui utilise en plus la boîte d'impression prédéfinie printdlg que nous détaillerons au prochain paragraphe.

Le DC imprimante doit être libéré après usage par DeleteDC().

Syntaxe: BOOL PrintDlg (PRINTDLG * pdlg);
 DeleteDC (HDC hdc);

Cette fonction PrintDlg() renvoie dans le champ hdc de la structure pointée par pdlg ---- voir plus loin ---- le DC imprimante voulu (ou NULL en cas d'erreur !) après que l'utilisateur a fermé la boîte de dialogue d'impression.

La structure générale du programme d'impression est conforme à l'exemple suivant:

Exemple:

```
PRINTDLG pdlg;  
HDC hdcprn;  
  
PrintDlg(&pdlg);  
hdcprn= pdlg.hDC;  
  
if (StartDoc(hdcprn, &diprn) > 0)  
  {  
    StartPage(hdcprn);  
    ...  
    /* appels fts GDI pour écrire et dessiner sur la page */  
    ...  
    EndPage(hdcprn);  
  }  
EndDoc(hdcprn);  
  
DeleteDC(hdcprn);
```

Les fonctions utiles sont les suivantes

Syntaxe: int StartDoc (HDC hdcprn, DOCINFO * diprn);
 int StartPage (HDC hdcprn);
 int EndPage (HDC hdcprn);
 int EndDoc (HDC hdcprn);

Ces fonctions utilisent le HDC obtenu par PrintDlg() et renvoient une valeur > 0 en cas de succès. En cas d'échec il ne faut pas poursuivre car Windows intervient avec un message d'erreur et ferme le DC.

En plus du DC, StartDoc() reçoit l'adresse d'une structure DOCINFO qui doit être complétée avant l'appel de la fonction:

```
struct DOCINFO  
{  
  int            diSize;  
  LPSTR         diTitle;  
  LPTSTR        ptr;  
};
```

Dans cette structure les champs sont:

- diSize= sizeof(DOCINFO),
- diTitle= titre du document dans le gestionnaire d'impression,
- ptr= NULL en général.

Chaque page est encadrée par StartPage() et EndPage(), ce qui transmet la page au driver pour interprétation.

L'appel a EndDoc() provoque l'envoi du document au spooler d'impression

3) Les problèmes d'échelle:

Pour une première approche de l'impression on peut se contenter des considérations précédentes, mais on constatera que le résultat n'est pas franchement WYSIWYG en raison des problèmes d'échelle dus aux résolutions variées des imprimantes (180 dpi pour une matricielle, 300 dpi pour une jet d'encre et 600 ou 1200 dpi pour une laser par exemple).

Que va donc devenir un texte écrit dans une police 12 points ?

Il fera 12 points à l'écran en taille normale, et 12 points à l'impression mais autant dire que cela ne se verra pas sur une imprimante 1200 dpi !

L'impression d'un Bitmap pose exactement le même genre de problèmes.

La solution, car il y en a forcément une, consiste à analyser les résolutions verticale et horizontale de l'imprimante et à adapter l'impression en conséquence. Il faut aussi songer que l'imprimante possède des marges à l'impression, et qu'il faut en tenir compte pour le rendu de chaque page.

Le champ hDevMode de la structure PRINTDLG contient un handle sur une zone mémoire allouée par PrintDlg() qui contient des informations sur le format du papier, son orientation, les marges etc.

Les autres renseignements concernant directement l'imprimante sont accessibles via GetDeviceCaps():

Syntaxe: `int GetDeviceCaps (HDC hdcprn, UINT CTE_CAP);`

Ainsi CTE_CAP= LOGPIXELSX ou LOGPIXELSY renvoie un résultat qui est la résolution horizontale ou verticale en pixels / pouce c'est à dire en dpi. Ces valeurs doivent être converties en pixel par point en les divisant par 72 car les tailles de fontes sont exprimées en points et

$$1 \text{ point} = 1/72 \text{e de pouce}$$

Donc une fonte 12 points à l'écran aura pour hauteur à l'impression

$$h = (\text{float}) \text{GetDeviceCaps}(\text{hdcprn}, \text{LOGPIXELSY}) / 72 * 12$$

Valeur h qui sera arrondie à l'entier le plus proche et utilisée por créer la fonte voulue

par CreateFont() ou CreateFontIndirect() ...

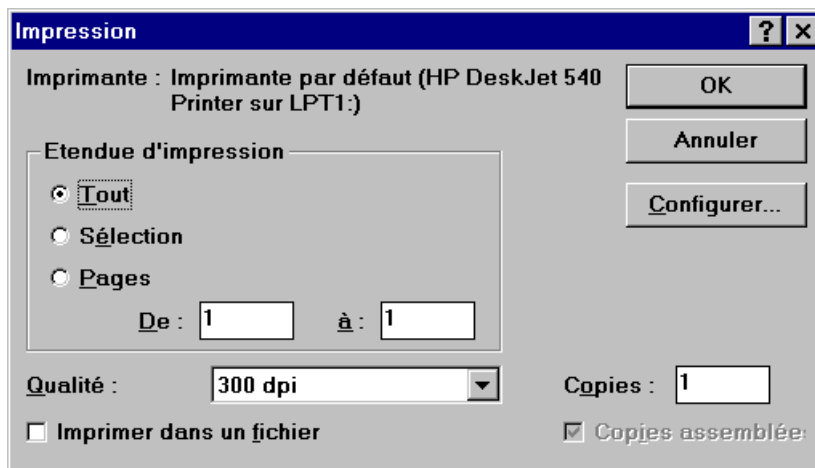
On ne peut donc pas utiliser les mêmes fontes pour l'écran et pour l'imprimante !

Il convient de sélectionner ladite fonte au niveau du DC imprimante hdcprn via SelectObject() avant d'écrire avec TextOut() ou DrawText() par exemple.

Remarque: En ce qui concerne les Bitmaps, on utilisera pour les imprimer la fonction StretchDIBits() comme pour un affichage à l'écran, mais en faisant un changement d'échelle basé sur les calculs précédents, qui donneront les dimensions du Bitmap à l'impression.

II) Les boîtes de dialogue Print et Print Setup:

Nous avons déjà mentionné la boîte Common Dialog Print qui est appelée par PrintDlg():



Dans cette première boîte, l'option Configurer en appelle une autre nommée Print Setup:



La structure PRINTDLG à utiliser contient les champs suivants:

```

struct PRINTDLG
{
    DWORD           lStructSize;
    HWND           hwndOwner;
    HANDLE         hDevMode;
    HANDLE         hDevNames;
    HDC            hDC;
    DWORD         Flags;
    WORD           nFromPage;
    WORD           nToPage;
    WORD           nMinPage;
    WORD           nMaxPage;
    WORD           nCopies;
    HINSTANCE      hInstance;
    DWORD         lCustData;
    LPPRINTHOOKPROC lpfnPrintHook;
    LPSETUPHOOKPROC lpfnSetupHook;
    LPCTSTR        lpPrintTemplateName;
    LPCTSTR        lpSetupTemplateName;
    HANDLE         hPrintTemplate;
    HANDLE         hSetupTemplate;
};

```

Ceux à compléter avant l'appel sont:

- lStructSize= sizeof(PRINTDLG),
- hwndOwner= handle de la fenêtre mère,
- Flags= PD_RETURNDC pour que la fonction demande un hdc,
- nFromPage= première page à imprimer,

- nToPage= dernière page à imprimer,
- nCopies= nombre d'exemplaires,

Au retour la fonction PrintDlg() renvoie TRUE si on est sorti par OK et FALSE si on est sorti de la boîte de dialogue par ANNULER.

Dans le premier cas le champ hDC contient le DC imprimante (vérifier qu'il est non NULL) et les autres champs les valeurs modifiées par l'utilisateur que le programme devra prendre en compte..

Table de caractères Windows															
	!	"	#	\$	%	&	'	[]	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{		}	~	
	'	'													
	ı	€	£	¤	¥		§	"	©	®	«	»	-	®	-
°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ