A.FICARRA

Soluzioni Visual C++ (4.0) adottate nei programmi di gestione del personale: Anaass, Anaper, Adsenze e Dipcheck

IRA 239/97

INDICE

Introduzione

Recentemente sono stati realizzati alcuni programmi in linguaggio **Visual C++** (versione 4.0), che girano in ambiente **Windows 95**.

Questi programmi hanno la funzione di gestire le presenze e assenze giornaliere del personale (in attesa dell'installazione di un sistema completamente automatico). I loro nomi sono:

1) **Anaass**: gestisce l'anagrafe dei codici di assenza;
2) **Anaper**: gestisce l'anagrafe del personale;
3) **Adsenze**: gestisce le presenze e le assenze dei dipendenti, per ogni giornata lavorativa
4) **Dipcheck**: ricava informazioni individuali e statistiche sulle presenze e assenze dei dipendenti

Le informazioni risiedono tutte in un unico **database**, di tipo **MDB** (**M**icrosoft **D**ata**B**ase), gestito dal sistema di accesso **ODBC** (**O**pen **D**ata**B**ase **C**onnectivity).

La struttura dei dati e le caratteristiche operative dei programmi verranno dettagliatamente descritte in un altro Rapporto tecnico, di prossima preparazione. In questa sede, l'autore intende rivolgersi a specialisti del linguaggio **Visual C++**, per illustrare le principali tecniche di programmazione utilizzate e le soluzioni software, a volte originali, adottate per risolvere alcuni problemi che si sono posti nello sviluppo dei programmi. Pertanto, la terminologia usata in questo Rapporto e gli argomenti trattati sono strettamente specialistici.

Accesso al database

Il database si trova nel file **DipIRA.mdb** e contiene 6 tabelle:

1) **CodiciAssenze**, utilizzata in lettura e scrittura da **Anaass** e **Adsenze** e in sola lettura da **Dipcheck**;
2) **Dipendenti**, utilizzata in lettura e scrittura da **Anaper** e in sola lettura da **Adsenze** e **Dipcheck**;
3) **FerieInfo**, utilizzata in lettura e scrittura da **Anaper** e **Adsenze** e in sola lettura da **Dipcheck**;
4) **Festivi**, utilizzata in sola lettura da **Adsenze** (per le operazioni di gestione e aggiornamento bisogna utilizzare **MS Access**);
5) **Movimenti**, utilizzata in lettura e scrittura da **Adsenze** e in sola lettura da **Dipcheck**;
6) **Provvisori**, utilizzata in lettura e scrittura da **Adsenze** e in sola lettura da **Dipcheck**;

In tutti i programmi, a ogni tabella corrisponde un **recordset** associato a una classe derivata da **CRecordset**; una funzione di **CRecordset**, di nome **GetDefaultConnect**, restituisce una stringa che fra l'altro contiene il nome dell' **"ODBC data source"**, cioè il nome tramite il quale il *driver* **ODBC** riconosce il database: questo nome é **"Personale IRA"**; pertanto in ogni PC, prima di lanciare uno qualunque dei quattro programmi, é necessario eseguire *una tantum* la "registrazione" del database, associando il file **DipIRA.mdb** al **Data Source Name "Personale IRA"** (l'operazione si effettua tramite *pannello di controllo* e *icona* **ODBC 32 bit**).

Nei programmi **Anaass** e **Anaper** il database é dichiarato in **AppWizard**. Ciò comporta la creazione di una classe di *vista* derivata da **CRecordView**, con forti limitazioni alle possibilità operative (la classe di *vista* si appoggia a una *dialog box*, alla cui esistenza é legata quella del programma - d'altra parte non so se e come sia possibile creare una seconda *view* in una applicazione *single-document*). Tale struttura rende possibili soltanto operazioni di gestione dei record (lettura, aggiunta, modifica, cancellazione).

In **Anaass** la procedura é standard: viene creato un solo recordset (con **AppWizard**) e le classi derivate da **CDocument** e da **CRecordset** non hanno funzionalità aggiuntive; tutte le operazioni vengono svolte nella classe derivata da **CRecordView** e in particolare dalla sua funzione **OnMove**, che si comporta nel seguente modo (consideriamo per esempio **OnMove(ID_RECORD_NEXT)**):

1) verifica che i testi dei controlli della *dialog box* coincidano ancora con i corrispondenti membri dell'oggetto di **CRecordset** (l'associazione viene fatta dalla funzione **DDX_FieldText** chiamata dalla **DoDataExchange**);
2) se sono state fatte modifiche, esegue un **CRecordView::UpdateData(TRUE)** (trasferendo i dati dalla finestra a **CRecordset** tramite chiamata a **DoDataExchange**) e un **CRecordset ::Update()** (trasferendo i dati da **CRecordset** al database);

3) esegue la funzione **CRecordset::MoveNext()** che accede al record successivo del database (e automaticamente trasferisce il contenuto dei campi nei corrispondenti membri di **CRecordset**);
4) esegue un **CRecordView::UpdateData(FALSE)** per copiare i nuovi dati nella finestra;

Ne consegue che, se lo spostamento viene fatto direttamente con **MoveNext()**, occorre dopo chiamare la **UpdateData(FALSE)** per riallineare i dati ed evitare che, alla successiva **OnMove**, i vecchi dati siano copiati al posto dei nuovi.
Un'altra particolarità di **OnMove** consiste nel fatto che non riconosce le condizioni di EOF o BOF (un esempio di come risolvere il problema si trova nella funzione **OnFatto**: il dato di un *campo chiave* viene salvato prima dello spostamento e poi confrontato con il nuovo valore; se i due dati non sono diversi vuol dire che lo spostamento non é avvenuto e ciò é possibile solo se si é raggiunto il BOF o l'EOF).
Infine, sembra che non sia possibile svuotare del tutto un recordset (lo riconosce vuoto solo alla creazione, ma non dopo che sono stati aggiunti e poi cancellati i records).

L'associazione fra i controlli della *dialog box* e i campi del recordset (o meglio, i corrispondenti membri dell'oggetto di **CRecordset**) viene creata rapidamente posizionando il mouse sul controllo e facendo *doppio click* con il tasto CTRL premuto: questa operazione attiva **ClassWizard** che propone il nome e il tipo di una nuova variabile-membro (che può essere un *valore* oppure un *controllo* nel caso che si debba operare con funzioni-membro del controllo stesso): come risultato, **ClassWizard** sistema automaticamente, all'interno della funzione **DoDataExchange**, l'opportuna chiamata alla funzione **DDX_FieldText** oppure **DDX_Control**.

In **Anaass** sono state aggiunte le seguenti funzionalità alla classe derivata da **CRecordView**,:

- il membro **BOOL m_bModMode** (normalmente **FALSE**): serve ad identificare quando il recordset deve essere aggiornato (funzioni di modifica o aggiunta di un record, attivate da voci di menù);
- la funzione **void Visibility(BOOL vis)** : é chiamata (con **vis=TRUE**) dalle funzioni di aggiunta o modifica di un record (in questo caso toglie l'attributo di *read-only* ai campi e pone **m_bModMode=TRUE**); é chiamata (con **vis=FALSE**) alla fine della **DoDataExchange**, per ripristinare le condizioni di default dopo che é stata eseguita una modifica;
- la funzione **void OnAggiungi()** : é mappata da una voce di menù e serve per aggiungere un nuovo record (attivabile solo se **m_bModMode=FALSE**, per garantire che non si aggiunga un record mentre é in corso di modifica un altro); crea un record con campi vuoti, chiama la funzione **CRecordset::Requery()** (questa funzione rigenera l'accesso al recordset e muove la posizione al primo record), riposiziona al record appena creato chiamando la **OnMove(ID_RECORD_LAST)**, preceduta dalla necessaria **UpdateData(FALSE)** (vedi sopra), e infine chiama **Visibility(TRUE)**, per consentire la modifica del record;
- la funzione **void OnModifica()** : é mappata da una voce di menù e serve per modificare un record; si limita a chiamare **Visibility(TRUE)**;
- la funzione **void OnCancella()** : é mappata da una voce di menù e serve per cancellare un record (attivabile solo se **m_bModMode=FALSE**); eseguita la cancella-

zione, chiama la **CRecordset::MoveNext()** (o la **CRecordset::MoveLast()** nel caso sia stato raggiunto l'EOF) seguita dalla solita **UpdateData(FALSE)**;

- la funzione **void OnFatto()** : é mappata da un controllo a "bottone" e serve per registrare la modifica un record; chiama la **OnMove(ID_RECORD_PREV)** seguita da **OnMove(ID_RECORD_NEXT)**: con la prima operazione produce l'automatico aggiornamento sia dell'oggetto di **CRecordset** che dal database (vedi descrizione a pagg. 3-4), con la seconda riposiziona sul record modificato.

Anaper differisce da **Anaass** in quanto usa due recordset: il primo (tabella **Dipendenti**) é dichiarato in **AppWizard** come recordset principale del database (la classe assume automaticamente il nome **CAnaPerSet**) ; il secondo (tabella **FerieInfo**) é creato tramite **ClassWizard** nel seguente modo:

1) si aggiunge una nuova classe, di nome **CFerInfSet**, e si dichiara che deriva da **CRecordset**;
2) **ClassWizard** chiede di specificare il nome del database e della tabella;
3) chiuso **ClassWizard**, si dichiara, nella classe derivata da **CDocument**, un *membro-oggetto* di tipo **CFerInfSet** (oltre a quello di **CAnaPerSet** creato automaticamente) e, nella classe derivata da **CRecordView,** un *membro puntatore* a **CFerInfSet** (oltre all'altro, sempre creato automaticamente);
4) nella classe derivata da **CRecordView**, si corregge la funzione **OnInitialUpdate**, aggiungendo le assegnazioni degli indirizzi dei due membri di **CDocument** ai rispettivi puntatori e le istruzioni per l'apertura del recordset di **CFerInfSet** prima della chiamata della **CRecordView:: OnInitialUpdate** (questa funzione apre automaticamente il recordset principale); é inoltre opportuno che i membri **m_pDatabase** dei duc recordset coincidano: per ottenere questo, dopo l'apertura del recordset di **CFerInfSet** e prima dell'apertura automatica del recordset principale, bisogna assegnare il valore di **m_pDatabase** del primo al secondo.

Anaper utilizza anche i membri di **CRecordset**: **m_strFilter** e **m_strSort**:

- **m_strSort** é necessario per specificare l'ordine di successione dei records (non fidarsi di quello che appare usando **MS Access**!) e va definito nel *costruttore* della classe (per esempio, in **CFerInfSet()** é stata inserita l'istruzione: **m_strSort = "CodDip, Data"**, la quale stabilisce che i records devono essere ordinati secondo il *campo* **CodDip** e, a parità di **CodDip**, secondo il *campo* **Data**);
- **m_strFilter** permette di selezionare un sottoinsieme del recordset, mediante un "filtro", cioè una stringa che definisce una o più relazioni logiche sul valore dei *campi*; é utilizzato in modo diretto e non nel modo parametrizzato suggerito dai manuali (vedi "Visual C++ Tutorial", pag. 379), in quanto la parametrizzazione non funziona se il numero di parametri é variabile: ogni volta che serve, si ridetermina il valore del filtro da memorizzare in **m_strFilter** e poi si chiama la funzione **Requery()**, che applica il filtro e posiziona il record corrente all'inizio del sottoinsieme selezionato; queste operazioni vengono eseguite all'interno di *funzioni-membro* di **CFerInfSet**, come per esempio la funzione **GetAllwd**, che seleziona il record contenente il numero di giorni di ferie consentite a un dato dipendente in un dato anno.

Un'altra funzione-membro di **CFerInfSet**, utilizzata in **Anaper**, é la **SetAllwd**, che imposta il numero di giorni di ferie consentite, modificando il corrispondente record della tabella, se esiste, o aggiungendo un nuovo record, se non esiste. In realtà la classe

contiene parecchie altre funzioni, che illustreremo parlando della "gestione automatica delle ferie", ma tali funzioni non sono utilizzate da **Anaper**. Infatti, per evitare duplicazioni, i codici di *implementazione* delle 6 classi derivate da **CRecordset** (e di altre classi di tipo diverso, utilizzate da almeno due programmi) sono raggruppati in una particolare directory, di nome **Classi**, in cui ad ogni classe corrisponde una coppia di files (**.h** e **.cpp**); d'altra parte, ogni programma (escluso **Anaass**, che opera sul database in maniera rigorosamente standard) contiene un file, di nome **classi.cpp**, che é costituito esclusivamente da *direttive di inclusione* di files della directory **Classi**; in questo modo, la directory **Classi** funziona come una "libreria" e il file **classi.cpp** di ogni programma seleziona le classi che lo stesso programma deve utilizzare. In particolare il file **classi cpp** di **Anaper** contiene soltanto le *direttive di inclusione* del codice di *intestazione* (**.h**) e di *implementazione* (**.cpp**) della classe **CFerInfSet**, associata alla tabella **FerieInfo**: poiché tale classe é utilizzata anche da **Adsenze** e **Dipcheck**, essa é molto più ricca di funzionalità di quanto non sia strettamente necessario ad **Anaper**.

Per quello che riguarda la classe derivata da **CRecordView**, in **Anaper** sono presenti le stesse funzionalità aggiuntive già descritte in **Anaass**, con la differenza che, nella funzione **DoDataExchange**, sono inserite istruzioni che permettono di gestire non solo il recordset principale, ma anche quello associato alla classe **CFerInfSet**: tali istruzioni chiamano la **GetAllwd** per trasferire i dati dal database alla *dialog-box* e la **SetAllwd** per l'operazione contraria. Inoltre, tramite il membro **BOOL m_bNewDip**, il programma riconosce se sta esaminando un record esistente o se é stato appena aggiunto un nuovo record; in questo caso la **DoDataExchange** non usa la **GetAllwd** per presentare i dati ma imposta dei valori di default, che l'operatore, se vuole, può modificare.

Completamente diversa é la situazione dei programmi **Adsenze** e **Dipcheck**, nei quali il database non é dichiarato in **AppWizard** e quindi la classe di *vista* é fatta derivare non più da **CRecordView** ma da **CScrollView**: ciò da un lato comporta la rinuncia alle funzionalità e agli automatismi di **CRecordView** (non potendosi utilizzare la funzione **OnMove**, il database deve essere gestito direttamente con le funzioni **Edit** e **Update** di **CRecordset**), ma dall'altro offre la possibilità di utilizzare liberamente la finestra con le funzioni dell'interfaccia grafica.

Adsenze usa insieme tutte le 6 tabelle del database, in lettura e scrittura (salvo le tabelle **Dipendenti** e **Festivi**, che sono in sola lettura), **Dipcheck** ne usa 5 (é esclusa **Festivi**), tutte in sola lettura.

Per ogni tabella, la corrispondente classe, derivata da **CRecordset** , viene creata tramite **ClassWizard** (con la stessa procedura descritta per la creazione della classe **CFerInfSet** in **Anaper**, vedi pag.5). Ogni classe contiene varie *funzioni-membro* aggiunte, che, utilizzando il membro **m_strFilter**, permettono di effettuare selezioni e ricerche nel recordset in base ai parametri di chiamata. Inoltre, ogni classe associata a una tabella in lettura e scrittura possiede proprie funzioni di aggiornamento, aggiunta e cancellazione di records; in questo modo tutta la gestione del database é *incapsulata* nelle classi derivate da **CRecordset** e non in quelle di *documento-vista*, che si limitano a chiamare le funzioni, ma lasciano, secondo i canoni della *programmazione a oggetti*, che tutte le operazioni vengano eseguite all'interno degli oggetti stessi.

Per ogni classe derivata da **CRecordset** viene creato un *membro-oggetto* nella classe derivata da **CDocument**, con una eccezione: poiché le tabelle **Movimenti** e **Provvisori** hanno struttura identica, si é preferito definire una sola classe, di nome **CMovSet**, e crearne due istanze. Per distinguere fra le due tabelle, la *lista di inizializzazione* del *costruttore* della classe di **CDocument** trasmette ai due oggetti di **CMovSet** un parametro, che per il primo oggetto ha valore **TRUE** e per il secondo oggetto ha valore **FALSE**; a sua volta il *costruttore* di **CMovSet** trasferisce il parametro nella propria *variabile-membro* **BOOL m_flag**, testando la quale la funzione **GetDefaultSQL** (che é una *funzione-membro* di **CRecordset** opportunamente modificata) determina quale tabella associare all'oggetto (**Movimenti**, se **m_flag** é **TRUE**, **Provvisori** se **m_flag** é **FALSE**). Con questa sola modifica si é evitato di creare due classi, peraltro identiche.

In **Adsenze**, la classe derivata da **CDocument** fornisce la maggior parte delle funzionalità: oltre a contenere i *membri-oggetto* di tutti i recordset utilizzati, provvede anche alla loro "apertura", tramite la funzione **Open** di **CRecordset**, chiamata direttamente dalla **OnNewDocument**, cioè dal primo effettivo punto di entrata nel programma. Se qualche **Open** fallisce (per esempio nel caso che il database non esista o non sia stato registrato), la **OnNewDocument** ritorna **FALSE** (provocando l'immediata fuoriuscita dal programma), previa visualizzazione di un messaggio che informa quale tabella ha provocato l'errore. Se invece tutti i recordset vengono aperti correttamente, la **OnNewDocument** ritorna **TRUE**, per cui il programma crea il *frame* e le finestre di *vista* e resta in attesa che l'operatore selezioni una voce di menù; in queste condizioni tutte le tabelle del database restano accessibili per l'intera durata del programma, ad eccezione di **Festivi**, che viene chiusa dalla stessa **OnNewDocument**, in quanto le informazioni contenute in questa tabella (pochi records, ciascuno dei quali rappresenta una data, con *campi* corrispondenti al giorno, mese e anno) vengono trasferite in un unico array di oggetti della classe **CTime**; le altre tabelle verranno chiuse direttamente dal *distruttore* dell'oggetto di **CDocument**, quando l'operatore deciderà di far terminare il programma.

Un po' diverso é il comportamento di **Dipcheck**: questo programma praticamente non ha menù, in quanto l'operatore può eseguire una sola scelta, selezionando una stringa fra quelle di una *ListBox* di una finestra di dialogo. Anche **Dipcheck** apre il database tramite **OnNewDocument**, dal cui esito dipende la continuazione del programma, ma, terminata questa funzione, non resta in attesa: appena creata la finestra di *vista* (cioè la prima volta che é eseguita la funzione **OnDraw**) il programma visualizza la finestra di dialogo e, in base alla selezione dell'operatore, prepara un array di stringhe che contiene tutte le informazioni richieste; fatto ciò, chiude immediatamente tutti i recordset e visualizza le stringhe ponendosi finalmente in attesa; a questo punto l'operatore può esaminare le informazioni che appaiono sullo schermo per tutto il tempo che vuole, senza più impegnare il database (eseguendo lo "*scrolling*" se i dati occupano più di una pagina di video), in quanto dalla seconda esecuzione in poi della funzione **OnDraw** il programma accede direttamente all'array di stringhe e non ai dati originari.

Una classe per la visualizzazione interattiva

La principale funzione del programma **Adsenze** é quella di registrare e trasferire nel database le informazioni sulle presenze e assenze dei dipendenti. Nell'attesa di un sistema di rilevazione automatico, questa operazione deve essere eseguita manualmente da un operatore, che, ogni giorno, trascrive i dati copiandoli dal foglio di firme di presenza e attingendo le informazioni sulle assenze da quanto comunicatogli dagli stessi dipendenti. Poiché tale lavoro é alquanto ripetitivo e noioso, si é cercato, nel programma **Adsenze**, di sviluppare un'interfaccia grafica "amichevole", che consentisse di eseguire le suddette operazioni in un modo efficiente e rapido.

A questo scopo é stata creata una nuova classe, di nome **CProspetto**, derivata da **CObject**, con il compito di svolgere le seguenti funzioni:

1) acquisire le informazioni provenienti dal database (e riguardanti una giornata lavorativa) tramite proprie *funzioni-membro* (pubbliche) e memorizzarle in proprie *variabili-membro* (protette), costituite da oggetti della classe *template* **CArray** (con elementi, in questo caso, di tipo **int**) o della classe **CStringArray**;

2) visualizzare i dati presentandoli sotto forma di tabella, con i nomi dei dipendenti in testa a ogni riga e la descrizione del tipo di assenza (con relativo codice ufficiale CNR) in testa a ogni colonna (la prima colonna deve essere riservata alla "presenza"); per ogni riga, cioè per ogni dipendente, un cerchio scuro ("pallino") nella casella di intersezione con una colonna deve servire ad indicare la posizione del dipendente nella giornata in esame (ovviamente se non esistono ancora informazioni la riga deve essere vuota);

3) consentire all'operatore di inserire nuovi "pallini", oppure di cancellarli, oppure di spostarne la posizione; in ogni caso il programma deve assicurare che vi sia, al più, un "pallino" per ogni riga, cancellando quello esistente in caso di spostamento; in questo modo, la tabella può essere rappresentabile mediante un array monodimensionale **n(i)**, dove **i** é un indice che percorre i dipendenti e **n** é un numero che corrisponde al tipo di assenza (lo chiameremo "indice di assenza", da non confondere con il "codice di assenza", che può anche essere non numerico ed é definito dal CNR); convenzionalmente stabiliamo che **n(i)=0** vuol dire che il dipendente **i** é presente, **n(i)=1000** significa che non vi sono ancora informazioni sul dipendente **i**, e **n** negativo ha lo stesso significato di **-n**, con la differenza che l'informazione é ancora provvisoria (per esempio: previsione di ferie future); in quest'ultimo caso il "pallino" é di colore grigio;

4) restituire all'esterno, tramite proprie *funzioni-membro* (pubbliche), le informazioni ricavate dai valori, eventualmente modificati dall'operatore, delle proprie *variabili-membro*.

Come si può notare, la classe **CProspetto** non sa che i dati provengono da (e devono essere restituiti a) un database: il suo campo di applicazione é più generale e riguarda tutti i casi in cui si debbano porre in relazione due insiemi, con l'unico ma imprescindibile vincolo che ad ogni elemento del primo insieme corrisponda, al massimo, un elemento del secondo insieme (ovviamente non é richiesto il viceversa ... altrimenti ... avremmo un solo dipendente presente ogni giorno !!!).

Applicando le specifiche sopra descritte, la classe é stata implementata con le seguenti caratteristiche:

1) Inizializzazione e acquisizione dati

Le *variabili-membro* trasmesse al *costruttore* (e da questo inizializzate) sono:

CTime m_date; contiene la data della giornata in esame;

BOOL m_bdef; ha valore **TRUE** se il Prospetto é *definitivo* ("pallini" neri), oppure valore **FALSE** se il Prospetto é *temporaneo* ("pallini" grigi); vedremo più avanti in cosa consiste la differenza.

In realtà, sia la data che il tipo di Prospetto (che sono informazioni fondamentali per il resto del programma) interessano assai poco alla classe **CProspetto**, che utilizza tali informazioni esclusivamente per visualizzarle: la data e la frase "PROSPETTO DEFINITIVO", oppure "PROSPETTO TEPORANEO" (in funzione del valore di **m_bdef**) sono scritte in un riquadro in alto a sinistra dello schermo; invece, il colore di ogni "pallino" é determinato dal segno, positivo o negativo, del corrispondente elemento dell'array **n(i)** e non da **m_bdef**.

Le *variabili-membro* che rappresentano i dati sono:

CStringArray m_nameArray; contiene cognome e nome dei dipendenti;

CArray <int, int> m_absindArray; contiene gli "indici di assenza": é il già descritto array **n(i)**;

CStringArray m_absdescrArray; contiene le descrizioni dei tipi di assenza, secondo il dettato delle circolari del CNR;

CStringArray m_absdescrhyArray; simile alla precedente, il suo significato sarà chiarito più avanti;

CStringArray m_abscodArray; contiene i codici di assenza CNR;

Le *funzioni-membro* di caricamento dei dati sono:

void AddName(const CString& name, const int n=1000);

aggiunge il nuovo elemento **name** all'array **m_nameArray** e, corrispondentemente, il nuovo elemento **n** all'array **m_absindArray**; se il secondo argomento é omesso, vuol dire che l'informazione non é disponibile, da cui, di default: **n=1000** ;

void AddAbsCode(.....);

aggiunge un nuovo elemento a ciascuno dei tre array di stringhe che descrivono il tipo e il codice di assenza.

Queste funzioni sono chiamate, all'attivazione del Prospetto (deciso dall'operatore tramite selezione di una voce di menù), dalla classe derivata da **CDocument**, che attinge le informazione dal database tramite i propri *membri-oggetto*, *istanze* delle classi derivate da **CRecordset**; le chiamate sono inserite in due *loop* che percorro-

no, rispettivamente, la lista dei dipendenti in servizio nel giorno considerato (tabella **Dipendenti**) e la lista dei codici di assenza (tabella **CodiciAssenze**); inoltre, poiché i records delle due tabelle suddette sono univocamente identificati da due *campi-chiave* presenti anche nelle tabelle **Movimenti** e **Provvisori** (che contengono anche un *campo data*), per ogni dipendente **i** é possibile selezionare, nella tabella **Movimenti** (se il Prospetto é *definitivo*) o **Provvisori** (se il Prospetto é *temporaneo*), il record con dipendente **i** e data uguale a quella del giorno in esame; tale record può esistere o non esistere: se esiste é unico e contiene anche l'indice di assenza **n** da trasmettere nella chiamata della funzione **AddName**; se non esiste la funzione **AddName** trasmette solo il nome del dipendente.

2) Visualizzazione

In previsione che le dimensioni della tabella siano maggiori, sia come righe (dipendenti), che come colonne (tipi di assenza), di quelle dello schermo, la classe di *vista* é definita, in **AppWizard**, come classe derivata da **CScrollView**, anziché da **CView**; inoltre é implementata, sempre tramite **AppWizard**, la funzionalità di "**split window**": questa consente all'operatore di suddividere la finestra in quattro parti (dopo avere selezionato una voce di menù che visualizza le barre di suddivisione e avere trascinato tali barre con il mouse in modo da ottenere le dimensioni volute delle quattro finestre) e rende la consultazione della nostra tabella molto più "*friendly*" di come sarebbe se fosse gestita da **Excel** o da qualche altro prodotto standard. Infatti, posizionando opportunamente le barre di suddivisione, l'operatore può fare in modo che:

- il riquadro in alto a sinistra contenga la frase di intestazione (data e tipo di Prospetto - vedi pag. 10);
- il riquadro in basso a sinistra contenga le intestazioni delle righe (nomi dei dipendenti);
- il riquadro in alto a destra contenga le intestazioni delle colonne (descrizioni e codici delle assenze);
- il riquadro in basso a destra contenga tutte le caselle di incrocio fra righe e colonne;

in questo modo, manovrando lo "scrolling" esclusivamente con le barre annesse al quarto riquadro (quello delle caselle), si ottiene che:

- il riquadro con l'intestazione resta fisso;
- il riquadro con i nomi dei dipendenti é solidale con le caselle negli scorrimenti verticali, ma é fisso negli scorrimenti orizzontali;
- il riquadro con le descrizioni delle assenze é solidale con le caselle negli scorrimenti orizzontali, ma é fisso negli scorrimenti verticali;

in conclusione, qualunque sia la casella consultata, sia il nome del dipendente che la descrizione dell'assenza, corrispondenti a tale casella, sono sempre visibili sullo schermo.

Le operazioni suddette sono eseguite automaticamente dalle classi di libreria **CScrollView** e **CSplitterWnd**, ma la funzionalità di visualizzazione vera e propria é trasferita dalla classe di *vista* a **CProspetto**: come é noto, é la funzione **CScrollView::OnDraw** che si occupa della scrittura dei dati nella finestra; questa funzione é chiamata implicitamente dal sistema ogni volta che la finestra cambia di

posizione o di dimensioni (per esempio durante lo *scrolling*), oppure esplicitamente dalla **UpdateAllViews** della classe *documento*; nella classe derivata da **CScrollView** la funzione **OnDraw** é *scavalcata*: il suo compito é soltanto quello di verificare se esiste un Prospetto attivo: se non esiste, non fa nulla, se esiste, trasferisce l'argomento **pDC**, puntatore al proprio *device-context*, alla classe **CProspetto**, chiamando la sua *funzione-membro* (pubblica) **Show(CDC* pDC)**.
La conoscenza del *device-context* della classe di *vista* é l'unica informazione "esterna" di cui **CProspetto** necessita: tramite **pDC**, la funzione **Show** può accedere alle funzioni della *Graphics Device Interface* (GDI) per eseguire la visualizzazione dei dati, che ricava dai propri array; inoltre la classe incapsula alcune *variabili-membro* (protette, inizializzate dal *costruttore*), che sono "oggetti GDI", di tipo **CPen** e **CBrush**: gli oggetti **CPen** servono per definire spessore e colore delle linee che delimitano righe e colonne (alternativamente rosse e blu), mentre gli oggetti **CBrush** definiscono il colore dello sfondo (ciano) e il colore di riempimento dei "pallini" (nero e grigio), che vengono disegnati, mediante la funzione **pDC->Ellipse**, nelle posizioni determinate in base ai corrispondenti valori di **i** e **n(i)**; gli stessi numeri che indicano i vari colori utilizzati, insieme ad altre costanti (come per esempio le dimensioni delle caselle) sono associati a simboli tramite direttive **#define** (e quindi sono facilmente modificabili, secondo il proprio gusto). Infine, per quello che riguarda la scrittura dei testi, la funzione **Show** utilizza due *fonts*, che crea indirettamente tramite la classe **LOGFONT** (oggetto **logFont**): la prima, per i nomi dei dipendenti e le descrizioni delle assenze, con **logFont.lfHeight=15** (altezza del carattere in unità logiche, con *mappatura* standard MM_TEXT) e **logFont.lfWeight=400** (spessore *normale*), la seconda, per l'intestazione, con **logFont.lfHeight=20** e **logFont.lfWeight=700** (*bold*) (gli altri valori sono quelli di default).
Un'attenzione particolare merita la parte della funzione **Show** che provvede alla scrittura delle frasi di descrizione dei vari tipi di assenza nelle caselle di intestazione delle colonne: per maggiore comodità dell'operatore si é preferito che in ogni casella non comparisse soltanto il codice CNR, ma l'intera descrizione, suddivisa su più righe, e si é scelto, per motivi estetici, che la suddivisione rispettasse le regole di sillabazione delle parole. Per questo motivo gli array delle stringhe di descrizione sono in realtà due: il primo, **m_absdescrArray**, contiene le frasi "normali", il secondo, **m_absdescrhyArray**, contiene le stesse frasi, ma con la differenza che, all'interno di ogni parola, é inserito il carattere "*/*" in tutte le posizioni in cui la stessa parola può essere suddivisa in sillabe (ricordiamo che i due array sono caricati, insieme a quello dei codici CNR, dalla funzione **CProspetto::AddAbsCode**, chiamata dalla classe *documento*; questa a sua volta fa uso di una classe derivata da **CRecordset** per accedere alla tabella **CodiciAssenze** del database, in cui trova, in tre *campi* separati, i valori da trasferire nei tre array). Con questo accorgimento, alternativo a quello di utilizzare una procedura automatica di sillabazione, che sarebbe stata troppo onerosa (considerato anche che i caratteri non sono a spaziatura fissa), si può fare in modo che ad ogni stringa di **m_absdescrhyArray** corrisponda un'array di stringhe, e che ciascuna stringa di tale array rappresenti una riga di testo; questa operazione viene eseguita dalla *funzione-membro* (protetta) **CProspetto::Sillabario** (chiamata dalla **Show**), che riceve come argomento la stringa da spezzare, separa la stringa in parole e le parole in sillabe (utilizzando il carattere "*/*" che, ovviamente, rimuove dalle stringhe di output) e costruisce l'array in modo che

ogni stringa riempia il massimo possibile dello spazio a disposizione (conoscendo la larghezza della casella, in unità logiche, e facendo uso della funzione **pDC->GetTextExtent**, che restituisce le dimensioni di una stringa in unità logiche); infine l'array di stringhe così costruito (che é una variabile *locale* della **Show**, trasmessa alla **Sillabario** *by reference*) viene ripreso dalla **Show** che lo trasferisce nella casella che gli compete, una stringa per ogni riga, allineando le stringhe dal basso a partire dalla penultima riga, in quanto l'ultima é riservata al codice CNR (che proviene dall'array **m_abscodArray**).

Un'altra funzione *scavalcata* nella classe di *vista* é la **OnEraseBkgnd** (chiamata automaticamente dal sistema ogni volta che é necessario ridisegnare lo sfondo): se il Prospetto non é attivo, lo sfondo é di colore standard (cioè, normalmente, bianco) e la **OnEraseBkgnd** si limita a chiamare la sua omonima della classe base; se invece il Prospetto é attivo, la **OnEraseBkgnd** chiama la funzione **EraseBk**, *membro* (pubblico) di **CProspetto** (esattamente come la **OnDraw**, che chiama la **Show**), trasmettendole, oltre al solito **pDC**, anche un oggetto di tipo **CRect** che definisce l'area *client* della finestra; a sua volta, nella **CProspetto**, la **EraseBk** colora l'area trasmessale, mediante il proprio apposito oggetto di tipo **CBrush**.

3) Interattività

L'operatore interagisce con il Prospetto posizionando il mouse e *cliccando*:
- con il tasto sinistro, ottiene di inserire un nuovo "pallino" in una casella;
- con il tasto destro, cancella l'eventuale "pallino" esistente.

Nel primo caso il programma risponde a un *messaggio* di **Windows**, *mappato* dalla funzione **OnLButtonDown**; questa funzione riceve, come é noto, un argomento di tipo **CPoint**, che contiene le coordinate del punto in cui in cui é posizionato il mouse. Nella classe di *vista*, la funzione é *scavalcata*: per prima cosa verifica se esiste un Prospetto attivo e, se non esiste, non fa nulla; se il Prospetto esiste, crea un oggetto della classe **CDC** (di nome **dc**), mediante la funzione **CClientDC**, lo "prepara" con la funzione **OnPrepareDC**, e lo usa per convertire le coordinate del punto (di nome **point**) da *device* a logiche, chiamando la funzione **dc.DPtoLP**; infine chiama una *funzione-membro* (pubblica) di **CProspetto**, il cui prototipo é:
int Change(CDC* pDC, const CPoint& point), trasmettendole come argomenti **&dc** e **point**.
Come si può notare, anche in questo caso la classe di *vista* serve soltanto come "tramite", ma la parte rilevante della funzionalità é *incapsulata* nella classe **CProspetto**: la funzione **Change** si occupa di identificare la casella entro cui si trova **point** (se il punto é esterno a tutte le caselle, la funzione non fa nulla e *ritorna* il valore **0**), disegna un "pallino" al suo interno, cancella quello eventualmente esistente sulla stessa riga (riempiendo la casella con il colore di fondo) e aggiorna l'array **n(i)**; alla fine, se tutto é andato bene, ritorna il valore **1**, dopo avere memorizzato (vedremo perché), nella *variabile-membro* **CRect m_rectBounding**, il rettangolo costituito dall'insieme delle caselle della riga.
Nella classe di *vista*, la **OnLButtonDown** esamina il valore di ritorno della funzione **Change**: se questo é **0**, non fa nulla, se é **1** chiama (per aggiornare le altre finestre

visibili) la **UpdateAllViews** (tramite il suo puntatore alla classe *documento*), trasmettendole, fra gli argomenti, l'indirizzo dell'oggetto della classe **CProspetto**. A sua volta la **UpdateAllViews** chiama la funzione *virtuale* **OnUpdate** della classe di *vista*, la quale, come é noto, prevede fra i suoi argomenti un puntatore a **CObject** (lo chiamiamo **pHint**): la **OnUpdate** (*scavalcata*) riconosce che il realtà **pHint** punta a **CProspetto** (con **IsKindOf(RUNTIME_CLASS(CProspetto))** - é questo il motivo per cui la classe **CProspetto** é fatta derivare da **CObject**), e quindi ricava il valore di **m_rectBounding** e lo trasmette come argomento alla funzione **InvalidateRect**; quest'ultima chiama la **OnDraw** (e dunque la **CProspetto::Show**) producendo il ridisegno, non dell'intera tabella, ma solo della riga modificata (con notevole risparmio nel numero di operazioni). Se invece **pHint** non punta a **CProspetto** (come succede, per esempio, quando la **OnUpdate** é eseguita automaticamente dopo uno *scrolling*), é chiamata la funzione **Invalidate** (senza argomenti) e l'intera finestra viene riscritta.

Un comportamento particolare della funzione **Change** si ha quando l'operatore *clicca* (sempre con il tasto sinistro del mouse) in una qualunque casella dell'ultima colonna a destra, con intestazione "ALTRO": questo significa che il dipendente a cui corrisponde la casella selezionata é assente per motivi ancora sconosciuti e che tali "motivi" vanno comunicati al programma; in altre parole, i tre array delle assenze possono crescere di dimensioni durante la gestione di un Prospetto, e ciò succede, appunto, quando é selezionata la colonna con descrizione "ALTRO". Infatti in questo caso la **Change**, prima di eseguire le altre operazioni, chiama la *funzione-membro* (protetta) **BOOL AddNewCod**, la quale a sua volta attiva una *dialog-box* di tipo *modale*, che si sovrappone al Prospetto. I *controlli* di tale finestra sono costituiti, oltre che dai soliti "bottoni" **OK** e **Cancel**, da tre *caselle di testo*, corrispondenti ai tre nuovi elementi degli array da aggiornare (esattamente come succede quando, eseguendo il programma **Anaass**, si prepara un nuovo record da aggiungere alla tabella **CodiciAssenze** del database). La finestra di dialogo é, come sempre, associata a una classe (derivata da **CDialog**), che abbiamo chiamato **CNewCod**: in questa sono definite tre *variabili-membro* (pubbliche), di tipo **CString**, che la funzione **DoDataExchange** associa ai tre *controlli* della finestra, permettendo così di trasferire all'esterno i valori digitati dall'operatore nelle *caselle di testo*; in aggiunta, per rendere più "*friendly*" l'operazione, i due *controlli* della descrizione sono associati a *variabili-membro* di tipo **CEdit**, utilizzati dalla *funzione-membro* (protetta) **OnChangeDescr** (che *mappa* un messaggio di **Windows** emesso ogni volta che viene aggiornata la prima *casella*) per trasferire il contenuto della prima *casella* nella seconda (tramite le funzioni di **CEdit**: **GetWindowText** e **SetWindowText**); pertanto l'operatore deve prima riempire la *casella* con descrizione "normale" e poi, nella seconda *casella* (in cui é comparsa automaticamente la stessa frase), inserire solamente caratteri "*/*" per la separazione in sillabe. Quando l'operatore, introdotti i dati nelle *caselle*, disattiva la finestra di dialogo *cliccando* sul "bottone" **OK** (se *clicca* su **Cancel** annulla tutto, in quanto la **AddNewCod** *ritorna* **FALSE** alla **Change** e questa a sua volta termina immediatamente, *ritornando* **0** alla **OnLButtonDown**), la **CProspetto::AddNewCod** trasferisce i dati delle *caselle di testo* negli array, inserendo un nuovo elemento in ciascuno di essi, in penultima posizione (cioè lasciando sempre in ultima l'elemento con descrizione "ALTRO"); fatto questo la **AddNewCod** ritorna **TRUE** alla **Change**, che aggiorna

l'array **n(i)** e termina, ritornando il particolare valore **-1** alla **OnLButtonDown**, per informarla che in questo caso tutta la finestra va ridisegnata; finalmente, attraverso la catena di chiamate: **OnLButtonDown - UpdateAllViews - OnUpdate - Invalidate - OnDraw - CProspetto::Show**, il Prospetto riappare presentando una colonna in più (con relativo "pallino" nella casella selezionata), subito prima della colonna con intestazione "ALTRO".

Nel caso che l'operatore *clicchi* con il tasto destro del mouse, il programma risponde al *messaggio* di **Windows** *mappato* dalla funzione **OnRButtonDown**; questa, *scavalcata* nella classe di *vista*, si comporta analogamente alla **OnLButtonDown**, con la differenza che, se il Prospetto é attivo, chiama la *funzione-membro* (pubblica) **BOOL CProspetto::Erase**; a sua volta la **Erase** é analoga alla **Change**: verifica che il mouse sia posizionato all'interno di una casella (in più verifica che nella casella sia presente un "pallino") e, in caso positivo (altrimenti esce *ritornando* **FALSE** e la **OnRButtonDown** non fa nulla), procede con le operazioni: cancella il "pallino" e aggiorna l'array **n(i)**, ponendo **n(i)=1000** (assenza di informazioni sul dipendente **i**); fatto questo, crea in **m_rectBounding** il rettangolo "*invalidato*" e *ritorna* **TRUE** alla **OnRButtonDown**, che, da questo punto in poi, esegue esattamente le stesse operazioni che fa la **OnLButtonDown** quando la **Change** le *ritorna* il valore **1**.

4) Restituzione dei risultati

La disattivazione di un Prospetto avviene quando l'operatore seleziona l'apposita voce di menù, che é *mappata* da una *funzione-membro* della classe *documento*: questa, prima di cancellare l'oggetto creato come istanza della classe **CProspetto**, ne interroga il contenuto, utilizzando le sue *funzioni-membro* (pubbliche) adibite a tale scopo; queste funzioni sono:

- **int GetAbsInd(const int& ind)** : restituisce **m_absindArray[ind]** (più volte citato come **n(i)**);
- **CString GetDescr(const int& ind)** : restituisce **m_absdescrArray[ind]** ;
- **CString GetDescrHy(const int& ind)** : restituisce **m_absdescrhyArray[ind]** ;
- **CString GetCode(const int& ind)** : restituisce **m_abscodArray[ind]** ;
- **int GetAbsSize()** : restituisce la dimensione dei tre array di descrizione delle assenze.

Queste informazioni servono alla classe *documento* per aggiornare il database: in particolare la prima é utilizzata per aggiornare la tabella **Movimenti** (o **Provvisori**), le altre quattro per ampliare la tabella **CodiciAssenze**, nel caso che il valore restituito dalla **GetAbsSize** sia maggiore di quello che era al momento dell'attivazione del Prospetto (i nuovi records vengono inseriti prima di quello con descrizione "ALTRO", esattamente come avviene per gli array di **CProspetto**). Grazie a questa procedura, non é necessario che la tabella **CodiciAssenze** sia completa sin dall'inizio (come deve essere, invece, la tabella **Dipendenti**): ogni nuovo codice può essere inserito solo quando se ne presenta l'esigenza (i codici stabiliti dal CNR sono più di cento e sarebbe veramente oneroso inserirli tutti in anticipo, considerato anche che molti di essi resteranno sempre inutilizzati); perciò, é il programma **Adsenze** e non **Anaass** che deve occuparsi di far crescere la tabella: **Anaass** va usato

solamente per la correzione degli errori (la classe **CProspetto** può aggiungere nuovi codici ma non modificare quelli esistenti) e, all'inizio, per inserire i codici di cui il programma richiede la presenza obbligatoria, che sono:

1) PRESENZA (codice **00**);
2) FERIE (codice **FE**);
3) FERIE ANNO PRECEDENTE FRUITE NEL SECONDO SEMESTRE DELL'ANNO SUCCESSIVO (codice **37**);
4) ALTRO (codice *blank*).

A proposito della voce FERIE, notare che il codice attribuito **FE** non coincide con quelli ufficiali del CNR; la ragione di questo sta nel fatto che, per ogni dipendente dichiarato in ferie, il programma controlla, attingendo alla tabella **FerieInfo**, di quale tipo di ferie si tratta (e controlla anche che il dipendente abbia diritto a fruirne), e determina automaticamente il codice (che può essere **31**, **32** o **94**), oppure notifica che il dipendente ha esaurito le sue ferie (nel qual caso il Prospetto resta *temporaneo*, finché il problema non é risolto). La totale automatizzazione della gestione delle ferie é uno dei punti qualificanti del programma (se ne occupa la classe *documento*), in quanto libera l'operatore dall'onere di controllare continuamente la situazione dei dipendenti mentre inserisce i dati dei Prospetti giornalieri.

Abbiamo parlato più volte dell'attivazione e della disattivazione di un Prospetto e delle circostanze in cui sia la classe *documento* che quella di *vista* controllano l'esistenza o meno di un Prospetto attivo. Chiariamo ora come ciò avviene:

- nella classe *documento* é dichiarata una *variabile-membro*, di nome **m_pProspetto**, e di tipo *puntatore* a **CProspetto**;
- nel *costruttore* della classe *documento* **m_pProspetto** é inizializzata con il valore **NULL**;
- quando l'operatore seleziona l'apposita voce di menù, la *funzione-membro* della classe documento *mappata* da questa voce controlla il valore di **m_pProspetto**: se non é **NULL** (Prospetto già attivo), non fa nulla, se é **NULL**, attiva una finestra di dialogo (associata alla classe **CDataPros**), che permette all'operatore di inserire la data del giorno in esame; con tale informazione, la funzione accede alla tabella **Movimenti**, o, in mancanza di dati sufficienti, alla tabella **Provvisori**;
- superati i controlli preliminari, la funzione crea un'*istanza* di **CProspetto** nell'area *heap* e copia l'indirizzo di tale area in **m_pProspetto**; poi trasferisce le informazioni selezionate dal database nell'oggetto appena creato, utilizzando le sue *funzioni-membro* di ingresso, *puntate* da **m_pProspetto**;
- durante un Prospetto attivo, nessun altra operazione di menù può essere eseguita: ciò significa che tutte le funzioni *mappate* da voci di menù controllano, per prima cosa, il valore di **m_pProspetto** e, se non lo trovano **NULL**, terminano immediatamente; non é neppure permesso uscire dal programma se prima non si é disattivato il Prospetto: infatti, la funzione di attivazione annulla l'attributo WS_SYSMENU dello *stile* della finestra *frame*, mediante la chiamata: **AfxGetMainWnd()->ModifyStyle(WS_SYSMENU,0)**; questa operazione, eliminando l'intera barra del menù di sistema, fa sparire anche le due barrette incrociate a "per", che costituiscono l'unico mezzo a disposizione dell'operatore per uscire dal programma; l'istruzione **ModifyStyle** ottiene il suo effetto, ma non cancella l'apparenza della barra di menù (che non esiste più ma é ancora visibile): per

farla sparire non ho trovato di meglio che far sparire l'intera finestra (mediante la chiamata: **AfxGetMainWnd()->ShowWindow(SW_HIDE)**), e, immediatamente dopo, ripristinarla (usando la stessa funzione con parametro **SW_SHOW**);

- anche il comportamento delle funzioni della classe di *vista* (**OnDraw**, **OnEraseBkgnd**, **OnLButtonDown**, **OnRButtonDown**) dipende dall'esistenza o meno di un Prospetto attivo: tutte queste funzioni che, per decidere cosa fare, hanno bisogno di testare il valore di **m_pProspetto**, possono accedervi tramite la *funzione-membro* (pubblica) **CProspetto* GetProspect()** della classe *documento* (a sua volta raggiungibile dalla classe di *vista*, come é noto, tramite la funzione **GetDocument**);
- come l'attivazione, anche la disattivazione di un Prospetto é decisa dall'operatore, che seleziona l'apposita voce di menù, *mappata* da un'altra *funzione-membro* della classe *documento*; tale funzione esegue anzitutto, come le altre, il test sul valore di **m_pProspetto**, ma si comporta in maniera opposta, cioè non fa nulla se il suo valore é **NULL** e va avanti in caso contrario; la funzione accede all'oggetto *puntato* da **m_pProspetto** ed estrae i valori contenuti nei suoi array, utilizzando le sue *funzioni-membro* di uscita; poi decide (vedremo più avanti in base a quali criteri) se i dati vanno inseriti nella tabella **Movimenti** o in quella **Provvisori** del database, procede, se necessario, all'ampliamento della tabella **CodiciAssenze**, e infine cancella l'oggetto di **CProspetto** (rimuovendolo, con **delete**, dall'area *heap*) e riassegna il valore **NULL** a **m_pProspetto** (da questo momento le altre voci di menù sono di nuovo utilizzabili e le funzioni della classe di *vista* ritornano al comportamento di *default*); prima di terminare, la funzione ripristina la barra del menù di sistema e quindi la possibilità di uscire dal programma.

Un'altra classe del programma **Adsenze**, utilizzata dalla classe *documento* e simile, sotto molti aspetti, alla **CProspetto**, si chiama **CProspView** e serve per visualizzare le informazioni aggregate sulle presenze e assenze dei dipendenti, in un dato periodo di tempo. Le sue caratteristiche principali sono:

- é identica alla **CProspetto** per quello che riguarda il meccanismo di attivazione e disattivazione e i controlli che le classi *documento* e *vista* effettuano per evitare che qualunque altra operazione sia eseguita mentre é attivo un oggetto di **CProspView**; infatti le sue *istanze* sono *puntate* da un'altra *variabile-membro* della classe *documento*, di nome **m_pProspView**, il cui valore é fissato a **NULL** quando non esistono *istanze* della classe; sia le funzioni che *mappano* voci di menù, sia quelle della classe di *vista* testano in realtà entrambe le variabili **m_pProspetto** e **m_pProspView** per decidere cosa fare;
- é una classe di visualizzazione pura, senza interattività; pertanto, possiede funzioni di ingresso, tramite le quali riceve dalla classe *documento* le informazioni estratte dal database (tabelle **Dipendenti**, **CodiciAssenze** e **Movimenti**, cioè solo Prospetti *definitivi*), ma non funzioni di uscita, in quanto non ha dati da restituire;
- produce un output grafico molto simile a quello della classe **CProspetto**, per ciò che riguarda il significato delle righe (lista dei dipendenti) e delle colonne (descrizioni delle assenze); possiede perciò, come la **CProspetto**, *variabili-membro* di tipo **CPen** e **CBrush** per il colore delle linee e del fondo, la funzione **Show** per la visualizzazione dei dati, la funzione **EraseBk** per il ridisegno del fondo, e la funzio-

ne **Sillabario** per la suddivisione in sillabe delle stringhe di descrizione (le ultime due funzioni sono identiche alle loro omonime della classe **CProspetto**); sono ovviamente assenti le funzioni di **CProspetto**, quali la **Change**, la **Erase** e la **AddNewCod**, che attengono all'aspetto interattivo; infine la classe sfrutta le funzionalità di "**split window**" per mantenere sempre visibili sullo schermo, come la **CProspetto**, le intestazioni dei dipendenti e delle assenze;

- a differenza della **CProspetto**, presenta informazioni che riguardano, non una sola giornata, ma un intero periodo di giornate lavorative (il *costruttore* inizializza due *variabili-membro* di tipo **CTime**, che rappresentano rispettivamente la data iniziale e finale del periodo); le caselle di incrocio fra righe e colonne non contengono "pallini", ma numeri, che indicano per quanti giorni del periodo in esame il dipendente (riga) é stato presente, o assente per un dato tipo di assenza (colonna); in particolare nella colonna delle "PRESENZE" é riportata un'espressione del tipo **m/t**, dove **m** é il numero di giorni di presenza e **t** é il numero totale dei giorni lavorativi del periodo (che in generale sarà uguale per tutti i dipendenti, ma non sempre, in quanto in sedi diverse vi possono essere festività locali differenti).

Da quanto detto si evince che la classe **CProspView** si distingue dalla **CProspetto** soprattutto per quello che riguarda la corrispondenza fra i due insiemi (dipendenti e assenze), che non può più essere rappresentata da una relazione del tipo **n(i)**, ma da un array bidimensionale **m(i,n)**, dove **i** corrisponde al dipendente, **n** all'*indice di assenza* e **m** al numero di giorni. Dal punto di vista del programma, ciò ha comportato una leggera difficoltà, in quanto gli array bidimensionali, pur previsti dal linguaggio **C**, non sono supportati da classi di libreria e quindi il loro uso diretto come *variabili-membro* di una classe, senza conoscerne a priori le dimensioni, avrebbe caricato a il programmatore dell'onere aggiuntivo di gestire la loro allocazione in memoria (oppure di allocare staticamente una quantità di memoria molto più grande, per sicurezza, di quella effettivamente necessaria); d'altra parte non sarebbe stato efficiente trasformare l'array in monodimensionale, in quanto é più semplice ed elegante che le *funzioni-membro* di ingresso trasferiscano i dati mantenendo separati i due insiemi (come in **CProspetto**, queste funzioni sono chiamate, dalla classe *documento*, in due *loop* distinti). Il problema é stato risolto trasformando l'array bidimensionale in una *lista concatenata* di array monodimensionali, con la specifica che ogni elemento della *lista* riguardi il dipendente **i** e contenga l'array **m(n)** (giorni **m** in funzione dell'*indice di assenza* **n**).

A tale scopo é stata creata una nuova classe, di nome **CListIntArray**, costituita da sole *variabili-membro* (private) e così strutturata:

```
class CListIntArray : public CObject   (*)
{  CString m_name;              cognome e nome del dipendente
   int m_iPres;                 indice di assenza corrispondente alla "PRESENZA"
   int m_nTot;                  numero totale di giorni lavorativi
   CArray<int, int> m_nm;       array m(n)
   CListIntArray* m_pNext;      punta al prossimo elemento della lista
   friend class CProspView; };  (**)
```

(*) la derivazione di **CListIntArray** da **CObject** permette di usare l'operatore **new** per creare le *istanze* di **CListIntArray** nell'area *heap*;

(**) la classe **CProspView** é definita *friend* della **CListIntArray** e perciò ogni *funzione-membro* della prima può accedere alle *variabili-membro* della seconda, anche se queste sono (di *default*) private.

Nella classe **CProspView** sono dichiarate due *variabili-membro*, entrambe di tipo **CListIntArray*** : la prima (inizializzata con **NULL**) punta al primo elemento della *lista*, la seconda punta all'ultimo; utilizzando queste due variabili, la *funzione-membro* **AddName** (completamente diversa dalla sua omonima nella classe **CProspetto**) é in grado di aggiungere ogni volta un nuovo elemento alla *lista*, copiandone l'indirizzo nella variabile **m_pNext** dell'elemento precedente e creando così la *concatenazione*; in questo modo, la *lista concatenata* percorre i dipendenti, mentre, in ogni elemento della *lista*, l'array **m_nm** percorre gli *indici di assenza*.
Le altre *variabili-membro* di **CListIntArray** hanno significato ovvio: **m_name** serve per l'intestazione della riga, mentre **m_iPres** e **m_nTot** servono per scrivere l'espressione: **m_nm[m_iPres] / m_nTot** nella colonna delle "PRESENZE" (la barra non vuol dire "operazione di divisione", ma va proprio scritta come "barra"!).

Restano invece nella classe **CProspView** gli array di descrizione delle assenze: ne bastano due, cioè quelli che servono per scrivere le intestazioni delle colonne; il terzo, di descrizione senza separazione in sillabe, non serve ai fini della visualizzazione (**CProspetto** aveva bisogno di tutti e tre gli array in previsione di trasmetterli al database, nel caso di aggiunta interattiva di nuovi records nella tabella **CodiciAssenze**). La *funzione-membro* **AddAbsCode**, che aggiorna tali array, é sostanzialmente identica alla sua omonima della **CProspetto**, salvo il fatto che trasporta due argomenti anziché tre.

Infine il *distruttore*: in **CProspetto** non aveva nulla da fare; qui ha il compito di rimuovere la *lista* dalla memoria *heap*; prima di fare questo chiede all'operatore se vuole una stampa dei dati (ne parleremo nel capitolo dedicato alle stampe).

Le funzioni delle classi di "documento-vista"

Come é noto, lo "scheletro" delle classi di *documento-vista*, creato automaticamente da **AppWizard**, costituisce una base che é di grande aiuto al lavoro del programmatore, in quanto gli offre la possibilità di implementare le funzionalità caratteristiche della sua applicazione, espandendo ed eventualmente modificando strutture già preparate e disponibili.

Abbiamo visto che nei programmi **Anaper** e **Anaass** il ruolo della classe *documento* é marginale: quasi tutto il lavoro é svolto dalla classe di *vista*, derivata da **CRecordView**.

Invece, in **Adsenze**, la situazione é ribaltata: la classe di *vista*, derivata da **CScrollView**, offre soltanto le funzionalità di "manovra" della finestra (compresa la "**split window**") e funge da "ponte" di collegamento fra i *messaggi* automatici di **Window** e le classi **CProspetto** e **CProspView**, mentre la parte rilevante del programma é concentrata nella classe *documento* (e in quelle derivate da **CRecordset**).

Infine, in **Dipcheck**, i ruoli delle due classi sono abbastanza "atipici", in quanto é la classe di *vista* a lanciare le operazioni: la prima volta che va in esecuzione (automaticamente) la sua funzione **OnDraw**, questa trasferisce il controllo alla classe *documento*, che si occupa della selezione e preparazione dei dati da visualizzare, e alla fine, se tutto é andato bene, pone il valore **TRUE** nella propria *variabile-membro* (pubblica) **BOOL m_bdone** (inizializzata dal *costruttore* con **FALSE**); con ciò la classe *documento* ha esaurito la sua funzione e il controllo torna alla **OnDraw**, che esamina il valore di **pDoc->m_bdone** (**pDoc** é il *puntatore* all'oggetto della classe *documento*, restituito dalla funzione **GetDocument**) e, se lo trova **TRUE**, visualizza i dati; le volte successive, invece, la **OnDraw** (chiamata per esempio quando l'operatore effettua lo "*scrolling*" della finestra) trova i dati già pronti e si limita a ripresentarli sullo schermo (vedi anche pag.7).

Nel seguito di questo capitolo tratteremo in dettaglio della classe *documento* nel programma **Adsenze**, che presenta le caratteristiche più interessanti. Rispetto allo "scheletro" creato da **AppWizard**, la classe derivata da **CDocument** risulta profondamente modificata: sono state soppresse praticamente tutte le *funzioni-membro* originarie, salvo la **OnNewDocument**, che peraltro il programma utilizza, non per aprire un nuovo "documento", ma per sfruttare la sua caratteristica di *entry-point* effettivo dell'applicazione (come già detto a pag. 7, la **OnNewDocument** si occupa di eseguire la **Open** delle tabelle del database). Le altre *funzioni-membro* della classe base, e in particolare quelle riguardanti la gestione dei files (apertura, salvataggio, *serializzazione*, chiusura, e relative voci di menù), non servono, in quanto l'unico "file-documento" presente é il database, che é gestito dalle classi di **CRecordset**.
Al loro posto la classe *documento* contiene un gran numero di nuove *variabili* e *funzioni-membro* (per esempio i *membri-oggetto* delle classe derivate da **CRecordset**, di cui abbiamo già parlato) e si collega a un menù completamente diverso da quello pre-

parato da **AppWizard**; restano solo alcune voci, riposizionate e raggruppate sotto il comando **Frame**, che servono per: a) visualizzare le *barre di divisione* della **split window**, b) visualizzare o meno la *barra di stato*, c) visualizzare l'*icona* del programma nella *dialog-box* associata alla classe **CAboutDlg**; dal menù é anche sparita l'opzione di uscita dal programma (come già detto in fondo a pag. 16, l'unico modo per terminare l'esecuzione di **Adsenze** é quello di *cliccare* sulla barra del *system menù*). Tutte le nuove voci di menù (*mappate* da *funzioni-membro* della classe *documento*) si riferiscono a operazioni specifiche dell'applicazione, e sono:

- attivazione di un Prospetto giornaliero: *mappata* dalla funzione **OnProspettoAttiva**, che seleziona i dati dal database e li trasmette alla classe **CProspetto**;
- disattivazione di un Prospetto giornaliero: *mappata* dalla funzione **OnProspettoDisattiva**, che esamina ed elabora i dati ricevuti dalla classe **CProspetto** e trasferisce i risultati nel database, cancellando dalla memoria *heap* l'oggetto *istanza* della classe **CProspetto**;
- attivazione di un Prospetto riassuntivo: *mappata* dalla funzione **OnGenerico**, che seleziona i dati dal database e li trasmette alla classe **CProspView**;
- disattivazione di un Prospetto riassuntivo: *mappata* dalla funzione **OnTermina**, che elimina dalla memoria *heap* l'oggetto *istanza* della classe **CProspView**;
- inserimento dei dati di previsione di assenza di un dipendente nella tabella **Provvisori** del database: *mappata* dalla funzione **OnPrevis**, che riceve l'informazione dall'operatore, tramite *dialog-box*, e aggiorna conseguentemente il database;
- stampa del rapporto mensile da trasmettere al CNR: *mappata* dalla funzione **OnMensile**, che chiede all'operatore di scegliere il mese in una *ListBox* di una finestra di dialogo, ed estrae dalla tabella **Movimenti** i dati relativi al mese selezionato, elaborandoli ed organizzandoli in un file di stampa (vedere capitolo dedicato alle stampe).

A loro volta le funzioni sopramenzionate chiamano altre *funzioni-membro* della classe per operazioni specifiche: per esempio, sia la **OnGenerico** che la **OnMensile** chiamano la funzione **DoProspView**, in quanto molte operazioni di preparazione dei dati sono comuni a entrambe (quando é necessario, la funzione distingue fra le due vie da seguire, interrogando la *variabile-membro* **BOOL m_bMensile**, che é posta **FALSE** dalla **OnGenerico** e **TRUE** dalla **OnMensile**).

Particolarmente importanti sono le funzioni che si occupano della preparazione e dell'interpretazione dei Prospetti, distinguendoli fra *definitivi* e *temporanei*, e le funzioni che gestiscono le ferie dei dipendenti.

Nella terminologia del programma **Adsenze**, le parole *definitivo* e *temporaneo*, associate ai Prospetti giornalieri, assumono il seguente significato:

- un Prospetto é *definitivo* se le informazioni sono complete e valide, cioè se tutti i dipendenti in servizio nella data in esame sono registrati (come presenti o assenti) e tutte le assenze sono consentite (il programma controlla il numero di giorni di ferie che ogni dipendente ha ancora diritto di fare); comunque, nonostante la parola, un Prospetto *definitivo* può essere sempre modificato (per esempio per correggere informazioni erroneamente inserite); i dati dei Prospetti *definitivi* risiedono nella tabella **Movimenti** del database (ogni record contiene tre *campi*, che identificano, rispettivamente, il dipendente, il tipo di assenza e la data);

- un Prospetto é *temporaneo* se non vale almeno una delle due condizioni sopramenzionate; inoltre un Prospetto é sempre *temporaneo* se si riferisce a una data "futura", cioè posteriore alla data di sistema presente nel PC dell'operatore (la data di sistema deve essere impostata e controllata, in ogni PC che usa **Adsenze**, in modo che coincida sempre con la data reale !); i dati dei Prospetti *temporanei* risiedono nella tabella **Provvisori** del database (la cui struttura é identica a quella della tabella **Movimenti**).

Il programma provvede a che i dati delle due tabelle siano sempre autoconsistenti, cioè che nella tabella **Movimenti** siano presenti, per ogni giornata, le informazioni relative a tutti i dipendenti (questo non é richiesto per la tabella **Provvisori**, in cui si possono trovare dati relativi anche a un solo dipendente, ma per un lungo periodo di tempo, come per esempio nel caso di previsioni di assenza per maternità) e che, nelle trasformazioni dei Prospetti da *temporanei* a *definitivi* (o viceversa) i records copiati in una tabella vengano cancellati dall'altra (per modo che non esistano contemporaneamente dati di una stessa giornata in entrambe le tabelle).

La funzione **OnProspettoAttiva**, dopo aver chiesto all'operatore di inserire la data del giorno da esaminare, esegue, per prima cosa, una ricerca nella tabella **Movimenti**: se trova dei records con la stessa data, controlla che "ci siano tutti" e, in questo caso, crea un Prospetto *definitivo* (in caso contrario, possibile soltanto se il database é stato manomesso, il programma "abortisce" segnalando il tipo di errore); se non trova informazioni nella tabella **Movimenti**, crea un Prospetto *temporaneo*, trasferendogli i dati che, eventualmente e anche parzialmente, trova nella tabella **Provvisori**. La funzione usa la variabile-membro **BOOL m_bDayFlag** per ricordare se il Prospetto é stato attivato come *definitivo* (**TRUE**) o *temporaneo* (**FALSE**), e la *variabile-membro* **BOOL m_bFuture** per indicare se il Prospetto contiene informazioni su una giornata già trascorsa (**FALSE**) o si riferisce a previsioni future (**TRUE**).
Più arduo é il compito della funzione **OnProspettoDisattiva**, che deve controllare i dati restituiti dalla classe **CProspetto** e stabilire, fra l'altro, in quale tabella del database trasferirli. Per fare questo, utilizza due *funzioni-membro* (protette), che si chiamano **UpdateDay** e **ChangeProsp**:

- la funzione **BOOL UpdateDay** controlla che ogni dipendente sia registrato e confronta i dati originari nel database con quelli restituiti dal Prospetto: dove trova una variazione, corregge il database e, nel caso particolare che la variazione riguardi un'assenza con codice "**FE**" (ferie) o codice "**37**" (ferie con permesso speciale), aggiorna la tabella **FerieInfo** tramite due funzioni membro (pubbliche) di **CFerinInfSet** (di cui parleremo fra poco), dalle quali ricava anche l'informazione sulla ammissibilità dell'assenza per ferie del dipendente in esame; controllati tutti i dipendenti, la **UpdateDay** restituisce **TRUE** se il Prospetto deve restare (o diventare) *definitivo*, **FALSE** in caso contrario (cioè se esiste almeno un caso di dipendente non registrato o dichiarato in ferie senza averne diritto)
- la funzione **void ChangeProsp** esegue materialmente il trasferimento dei dati dalla tabella **Movimenti** alla tabella **Provvisori** (o viceversa), cancellandoli nella tabella originaria; viene chiamata quando é necessario cambiare la connotazione di un Prospetto e, precisamente, in uno dei seguenti casi:
  a) se **m_bDayFlag** é **FALSE**, **m_bFuture** é **FALSE** e **UpdateDay** restituisce **TRUE** (trasformazione di un Prospetto da *temporaneo* a *definitivo*);

b) se **m_bDayFlag** é **TRUE** e **UpdateDay** restituisce **FALSE** (trasformazione di un Prospetto da *definitivo* a *temporaneo*); questo secondo caso é più raro, ma non lo si può escludere a priori: potrebbe succedere che l'operatore, nel correggere un Prospetto già *definitivo*, abbia cancellato, anche involontariamente, un "pallino" senza ricrearlo in un'altra casella della stessa riga, oppure abbia dichiarato in ferie un dipendente senza più diritto a usufruirne.

La gestione delle ferie dei dipendenti, automatizzata, come s'é detto, per semplificare il lavoro dell'operatore, svolge un'altra funzione, oltre quella di controllare che ogni dipendente dichiarato in ferie ne abbia diritto: si occupa di separare, per le visualizzazioni dei Prospetti riassuntivi e per i Rapporti mensili da inviare al CNR, la voce FERIE nelle tre voci "burocratiche" in cui vengono classificate le ferie di un dipendente (a parte quelle con codice **37**, che costituiscono un caso a sé e sono già distinte a livello di tabella **CodiciAssenze**):

1) Codice **31**: ferie dell'anno precedente fruibili nel primo semestre dell'anno successivo;
2) Codice **94**: ferie sostitutive delle festività soppresse (in generale 4 giorni all'anno);
3) Codice **32**: ferie dell'anno corrente (in generale 28 giorni all'anno).

E' chiaro che ogni dipendente tende prima ad esaurire le ferie dell'anno precedente, poi le festività soppresse (fruibili entro l'anno) e, per ultime, quelle dell'anno corrente, che può riportare all'anno successivo. Sulla base di questo assunto, l'assenza per FERIE, che, nelle tabelle **CodiciAssenze**, **Movimenti** e **Provvisori**, é identificata da un unico codice, viene separata nei tre diversi codici secondo la successione cronologica degli eventi e l'informazione viene registrata nella tabella **FerieInfo**. Questa tabella contiene vari *campi*, fra cui assume particolare importanza il *campo* **CodDip** (codice dipendente), il cui valore permette di distinguere fra due diversi tipi di record:

- se **CodDip** é negativo, il record identifica il numero di giorni di ferie con codice **94** (chiamiamolo **tot94**) e con codice **32** (**tot32**), che il dipendente (identificato dal codice **-CodDip**) ha diritto di fare in un dato anno; i valori di **tot94** e **tot32** sono impostati dalla funzione **SetAllwd** e letti dalla funzione **GetAllwd**, entrambe *membri* della classe **CFerInfSet** (vedere pag. 6); questo tipo di record é gestito dal programma **Anaper**, che, come abbiamo visto, permette di definire ed eventualmente correggere i valori di **tot94** e **tot32** relativi all'anno corrente: pertanto, ad ogni nuovo anno, l'operatore deve provvedere, utilizzando **Anaper**, a impostare i nuovi valori, ma non é necessario che lo faccia se questi non si discostano dagli standard (**tot94=4** e **tot32=28**), che vengono assunti di *default*; ne consegue che l'impegno di inserire ogni anno i valori di **tot94** e **tot32** riguarda pochi casi eccezionali (nuovi assunti, dipendenti che vanno in pensione entro l'anno, non laureati in servizio da meno di tre anni ecc...), a parte durante la fase di preparazione iniziale del database (se non capita esattamente all'inizio di un anno), in cui, per ottenere risultati corretti, bisogna detrarre dai valori reali di **tot94** e **tot32** il numero di giorni di ferie fruite in precedenza (per esempio, se un dipendente, prima del giorno di operatività delle procedure automatiche, ha già fruito di **5** giorni di ferie codice **32**, bisogna simulare che lo stesso dipendente abbia diritto a **tot32=28-5=23** giorni per compensare la mancata registrazione di detti **5** giorni di ferie nel database);
- se **CodDip** é positivo, il record identifica una giornata di ferie del dipendente con codice **CodDip**; in questo caso un altro *campo* della tabella (di nome **Totali**), a sua volta frazionato in *campi di bit*, contiene vari numeri (uno per ogni codice di assen-

za), ciascuno dei quali rappresenta il totale di giorni ferie fruite dall'inizio dell'anno; identifichiamo tali numeri con le seguenti sigle:

- **np** : giorni di ferie dell'anno precedente;
- **n94** : giorni di ferie dell'anno corrente, codice **94**;
- **n32** : giorni di ferie dell'anno corrente, codice **32**;
- **nq** : giorni di ferie in esubero rispetto al massimo consentito dell'anno corrente;
- **ne** : giorni di ferie in esubero rispetto al massimo consentito dell'anno precedente;

un altro *campo* della tabella, di tipo "stringa", riceve il codice della giornata di ferie (chiamiamolo **cod**); il suo valore viene determinato nel seguente modo, partendo dal contenuto del campo **Totali** dell'ultimo record precedente (con lo stesso valore di **CodDip**):

- se il codice dell'assenza é **FE**, il valore di **np** viene confrontato con **tot32** (dell'anno precedente): se é minore, **np** viene incrementato di **1** e la procedura termina con **cod="31"**; altrimenti il valore di **n94** viene confrontato con **tot94** (dell'anno corrente): se é minore, **n94** viene incrementato di **1** e la procedura termina con **cod="94"**; altrimenti il valore di **n32** viene confrontato con **tot32** (dell'anno corrente): se é minore, **n32** viene incrementato di **1** e la procedura termina con **cod="32"**; altrimenti (il dipendente ha esaurito le ferie dell'anno corrente), il valore di **nq** viene incrementato di **1** e la procedura termina con **cod="??"**;
- se il codice dell'assenza é **37**, il valore di **np** viene confrontato con **tot32** (dell'anno precedente): se é minore, **np** viene incrementato di **1** e la procedura termina con **cod="37"**; altrimenti (il dipendente ha esaurito le ferie dell'anno precedente), il valore di **ne** viene incrementato di **1** e la procedura termina con **cod="!!"**;

come si può notare, in ogni record (con **CodDip** positivo) della tabella **FerieInfo** (che contiene anche un *campo* con la data) vengono memorizzate tutte le informazioni relative a un giorno di ferie di un dipendente: tali informazioni riguardano sia il totale dei giorni di ferie accumulate fino a quel momento (*campo* **Totali** suddiviso nei 5 *campi di bit* sopramenzionati), sia il codice "burocratico" dell'assenza per ferie (*campo* **cod**); infine, se risulta **cod="??"** oppure **cod="!!"**, il record ci informa che il dipendente ha, in quella data, esaurito le ferie e pertanto il Prospetto con la stessa data non può diventare *definitivo*.

Le operazioni di aggiornamento della tabella **FerieInfo** vengono eseguite dalla funzione **Add**, *membro* della classe **CFerInfSet**; tale funzione, chiamata dalle funzioni della classe *documento* **UpdateDay** (che controlla i Prospetti giornalieri) e **OnPrevis** (che inserisce nella tabella **Provvisori** le previsioni di giorni di assenza futuri) opera non solo sulla giornata in esame, ma anche su tutti i record della tabella, eventualmente presenti, che si riferiscono a giornate successive (relativi allo stesso dipendente, cioè con lo stesso valore di **CodDip**); questa precauzione é indispensabile, in quanto la successione cronologica é una componente fondamentale per determinare il codice appropriato (infatti potrebbe succedere che, dichiarando in ferie un dipendente nella correzione di un Prospetto, il tipo di ferie del dipendente cambi valore in Prospetto successivo, passando per esempio da codice **"94"** a codice **"32"**, o addirittura da codice **"32"** a codice **"??"**, con il risultato che l'introduzione di un giorno di ferie farebbe degradare da *definitivo* a *temporaneo* non il Prospetto in esame, ma quello di una data successiva!!!). Fra le altre operazioni, la funzione **Add** aggiorna un array di stringhe,

trasmessole come argomento *by reference*, dove memorizza le date dei Prospetti in cui in codice é passato da "buono" a "cattivo" (cioè é diventato **"??"** o **"!!"**); la funzione chiamante (**UpdateDay** o **OnPrevis**) controlla l'array, mantenendolo in ordine di data ed eliminando i "doppioni" (l'array é unico per tutti i dipendenti), mediante la *funzione-membro* **UpdateArr**; alla fine, se l'array non é vuoto (normalmente lo é, per fortuna!), chiama la *funzione-membro* **FerieFinite**, che comunica all'operatore quali dipendenti hanno esaurito le ferie e in quali giorni; infine (se la funzione chiamante é la **UpdateDay**), la **OnProspettoDisattiva** (qualunque sia l'esito della **UpdateDay**, che riguarda solo la giornata in esame) cerca se nell'array ci sono date di giornate successive: se le trova e ad esse corrispondono Prospetti *definitivi*, provvede a ritrasformare tali Prospetti in *temporanei* (chiamando la **ChangeProsp**).
Un'altra *funzione-membro* della classe **CFerInfSet**, di nome **Remove**, é chiamata (sempre dalla **UpdateDay** o dalla **OnPrevis**) ogni volta che un record indicante una giornata di ferie deve essere cancellato o sostituito con un altro con diverso tipo di assenza; la **Remove** svolge l'operazione inversa della **Add**: cancella il record dalla tabella **FerieInfo** e aggiorna gli eventuali record in data successiva, relativi allo stesso dipendente, procedendo nell'esame dei codici in ordine inverso e, quando é necessario, sottraendo **1** ai valori contenuti nel campo **Totali**; come la **Add**, anche la **Remove** aggiorna un array, memorizzando in esso le eventuali date in cui i codici sono passati da "cattivi" a "buoni"; analogamente, la funzione chiamante esamina tali date e, per ciascuna di quelle che corrispondono a Prospetti *temporanei*, chiama la *funzione-membro* **CheckProvv**, che controlla se il Prospetto ha titolo per essere trasformato in *definitivo* (se sì, chiama la **ChangeProsp**).

La funzione **DoProspView** (che, ricordiamo, prepara sia i Prospetti riassuntivi che i Rapporti mensili), organizza e presenta i dati che trova nella tabella **Movimenti**, salvo quando incontra il codice corrispondente alla voce generica di FERIE, nel qual caso attinge l'informazione dalla tabella **FerieInfo**: per questo motivo, mentre i Prospetti giornalieri interattivi presentano una sola colonna, con codice **FE** (con grande sollievo dell'operatore!!!), nei Prospetti riassuntivi di pura visualizzazione compaiono le tre colonne "burocratiche", con codici **"31"**, **"94"** e **"32"**.

Il programma **Dipcheck** ha comportamento analogo alla funzione **DoProspView** di **Adsenze**, con la differenza che **Dipcheck** visualizza anche i dati dei Prospetti *temporanei*, accodando, per ogni dipendente, le informazioni ricavate dalla tabella **Provvisori** a quelle estratte dalla tabella **Movimenti**; in entrambi i casi il programma ricava i dati delle ferie dalla tabella **FerieInfo**, con il risultato di visualizzare qualunque tipo di codice, anche eventualmente, nella parte relativa alle "previsioni" di ferie, i codici che abbiamo definito "cattivi" (**"??"** o **"!!"**).

Per concludere, segnaliamo altre due soluzioni software adottate nella classe *documento* di **Adsenze**:

1) E' stato detto che la *funzione-membro* **OnNewDocument** *ritorna* **FALSE** (e quindi fa terminare immediatamente il programma) se l'apertura di una tabella del database fallisce; prima però chiama un'altra *funzione-membro*, di nome **OpenError**, che segnala all'operatore, con un messaggio, il nome della tabella che ha provocato l'errore; d'altra parte é noto che, mentre é in esecuzione la **OnNewDocument**, il *frame* della finestra non é ancora visibile e in particolare non é ancora stata eseguita automaticamente dal sistema la funzione **OnInitialUpdate**, che invece é indispensabile, quando la classe di *vista* é derivata da **CScrollView**, per definire il *size* delle aree di *scrolling*; in altre parole, qualunque tentativo di visualizzare messaggi o altro dalla **OnNewDocument** produce normalmente un "*errore fatale*" che fa *abortire* il programma; per evitare questo, la stessa funzione **OnNewDocument** forza l'inizializzazione delle finestre di *vista*, chiamando la **GetFirstViewPosition** e, in *loop*, la **GetNextView** (che fornisce l'indirizzo **pView** di ogni finestra) e la **pView->OnInitialUpdate.**

2) La fase di preparazione di un Prospetto, sia giornaliero (con la **OnProspettoAttiva**), che riassuntivo (con la **DoProspView**), impegna l'elaboratore per alcuni secondi; la stessa coda accade quando il programma elabora i dati di un Prospetto giornaliero (in risposta alla **OnProspettoDisattiva**) oppure memorizza i dati di "previsione" (con la **OnPrevis**); durante questi periodi il cursore assume la forma caratteristica della "clessidra" e ciò si ottiene chiamando, all'inizio, la funzione di sistema: **::SetCursor(AfxGetApp()->LoadStandardCursor(IDC_WAIT))**; alla fine, per ripristinare il cursore normale (una "freccia"), viene chiamata la stessa funzione, con argomento **IDC_ARROW**.

Utilizzo avanzato delle finestre di dialogo

Come é noto, le *finestre di dialogo* servono essenzialmente per la "comunicazione" fra programma e utente: l'utente se ne serve per inserire i dati richiesti dal programma, mentre il programma le usa per notificare i messaggi all'utente.

I programmi **Adsenze** e **Dipcheck** fanno largo uso di finestre di dialogo, sempre associate a classi derivate da **CDialog** (abbiamo già citato, oltre alla consueta **CAboutDlg**, creata da **AppWizard**, le classi **CNewCod** e **CDataPros** di **Adsenze**); qui parleremo solo delle *dialog-box* utilizzate da **Dipcheck**, che presentano le caratteristiche più interessanti.

Dipcheck é un programma di visualizzazione pura e "*one-purpose*": accede al database in sola lettura ed é utilizzabile da chiunque (**Adsenze** é invece consentito ai soli autorizzati, tramite controllo della *password*); esegue una sola operazione, selezionabile, non da menù, ma da *dialog-box* (infatti il programma non ha praticamente menù: le uniche voci presenti sono, oltre alla solita che attiva la **CAboutDlg**, l'opzione di uscita dal programma (ottenibile anche con il tasto *Escape*) e un'altra, normalmente disattivata, di cui parleremo più avanti). Come già detto altre volte, le operazioni partono dalla funzione **OnDraw** della classe di *vista*, la quale, riconoscendo di essere eseguita per la prima volta, chiama un'altra *funzione-membro*, **BOOL PreDraw**, che a sua volta chiama la *funzione-membro* **BOOL InfSelect** della classe *documento*.
La classe derivata da **CDocument** ha il compito, in **Dipcheck**, di estrarre dal database le informazioni selezionate sulla base delle scelte dell'operatore e di preparare i dati, mettendoli a disposizione della classe di *vista*. A questo scopo la **InfSelect** crea un'*istanza* della classe **CSelDlg**, associata a una finestra di dialogo *modale*, trasmettendo al suo *costruttore* l'indirizzo **this** dell'oggetto *documento* e l'indirizzo di una propria variabile *locale* di tipo **CStringArray**: il primo argomento deve servire alla classe **CSelDlg** per accedere ai *membri* pubblici della classe *documento* (e in particolare ai *membri-oggetto* di **CRecordset** associati alle tabelle del database), il secondo argomento trasmette alla stessa **CSelDlg** la lista dei codici e nomi dei dipendenti, ricavata dalla tabella **Dipendenti**. Fatto questo la **InfSelect** trasferisce il controllo all'oggetto di **CSelDlg**, chiamando la sua *funzione-membro* **DoModal**; se questa funzione *ritorna* **FALSE** (l'operatore ha *cliccato* sul "bottone" **Cancel** della finestra di dialogo), la **InfSelect** esce immediatamente *ritornando* **FALSE** alla **PreDraw** della classe di *vista* e la **PreDraw** a sua volta esce *ritornando* **FALSE** alla **OnDraw**, che, in questo caso, fa terminare l'esecuzione del programma. Se invece l'operatore clicca sul "bottone" **OK**, la **DoModal** *ritorna* **TRUE** e l'elaborazione procede. Non descriveremo in questo contesto le successive operazioni della **InfSelect** e delle altre *funzioni-membro* della classe *documento*, chiamate dalla **InfSelect** per la preparazione dei dati (queste operazioni sono analoghe a quelle della **DoProspView** del programma **Adsenze**, che abbiamo descritto nel capitolo precedente); concentreremo la nostra attenzione, invece, su quello che succede nella classe **CSelDlg**.

La finestra di dialogo associata alla classe **CSelDlg** presenta, oltre ai soliti "bottoni" **OK** e **Cancel**, un *controllo* di tipo *ListBox a selezione singola*, tre *controlli a casella di testo* (*Edit*) e alcuni *controlli di testo statico* (*Static*); nella *ListBox* la prima stringa contiene la frase "SITUAZIONE GENERALE FERIE" , mentre le stringhe successive contengono la lista dei nomi dei dipendenti, trasferita dall'array trasmesso dalla **InfSelect** al *costruttore* della classe. Sia la *ListBox* che i *controlli Edit* sono associati a *variabili-membro* di **CSelDlg**, rispettivamente di tipo **CListBox** e **CEdit** (l'associazione é stata creata da **ClassWizard**, che ha introdotto, nel codice della funzione **DoDataExchange**, le opportune chiamate alle funzioni **DDX_Control**).

La caratteristica più rilevante della classe **CSelDlg** consiste nel fatto che gestisce l'apparenza della *dialog-box* in maniera "dinamica", in funzione delle scelte dell'operatore. Infatti, all'inizio, nella intestazione della *dialog-box* appare la scritta: "Seleziona la situazione generale delle ferie oppure il nome di un singolo dipendente", e inoltre l'unico *controllo* visibile é la *ListBox*; se l'operatore seleziona la prima riga ("SITUAZIONE GENERALE FERIE"), la scritta di intestazione cambia e diventa: "Seleziona l'anno di ferie da esaminare", e inoltre appare un *controllo Static* con la scritta: "Anno:", seguito accanto da una *casella Edit* contenente (come scelta di *default*) il valore dell'anno corrente; se invece l'operatore seleziona una qualunque riga successiva (nome di un dipendente), la scritta di intestazione diventa: "Seleziona le date iniziale e finale del periodo che interessa", e appaiono due *controlli Static* con le scritte: "Data da:" e "a:", seguite ciascuna da una *casella Edit* contenente il valore di una data (le date iniziale e finale, proposte come scelte di *default*, rappresentano i limiti estremi dell'intervallo di tempo entro cui esistono informazioni sul dipendente selezionato; tali informazioni provengono sia dalla tabella **Movimenti** che dalla tabella **Provvisori**); se l'operatore "ci ripensa" e seleziona un altro dipendente, il valore delle date iniziale e finale cambia; se il ripensamento é "totale" cioé l'operatore decide di selezionare la prima di riga della *ListBox* dopo aver selezionato un singolo nome (o viceversa), il *controlli* che erano apparsi precedentemente spariscono, per fare posto a quelli appropriati con la seconda scelta. In questo modo, con una sola *dialog-box* vengono gestiti percorsi diversi, definiti dalla selezione nel *controllo* principale (la *ListBox*), che genera dinamicamente la configurazione e il contenuto degli altri *controlli*.

Tutto ciò é reso possibile grazie a un uso molto particolare della funzione **DoDataExchange**, di cui, a beneficio dei meno esperti, ricordiamo brevemente le caratteristiche:

> La **virtual void CWnd::DoDateExchange(CDataExchange* pDX)** é una funzione *virtuale*, utilizzabile da qualunque classe derivata da **CWnd** (e quindi in particolare da **CDialog**), e serve soprattutto per lo scambio dei dati fra i *controlli* di una *dialog-box* e le *variabili-membro* della classe (cioè, in sostanza, fra operatore e programma). Non deve mai essere chiamata dal programma, in quanto é chiamata automaticamente dalla *funzione-membro* **CWnd::UpdateData(b)**, dove, se **b=FALSE**, significa che i dati devono essere trasferiti dalle *variabili-membro* ai *controlli* della finestra (all'inizio, il sistema esegue automaticamente una **UpdateData(FALSE)**), mentre, se **b=TRUE**, i dati devono essere trasferiti dai *controlli* della finestra alle *variabili-membro* (il sistema esegue automaticamente una **UpdateData(TRUE)** in risposta all'**OK** dell'operatore),

La **DoDataExchange** riceve un argomento, **pDX**, che punta a un'*istanza* di una classe speciale, denominata **CDataExchange**; di questa classe sono importanti i seguenti *membri*:

- **BOOL m_bSaveAndValidate** (d'ora in poi abbreviato in **m_b**): assume lo stesso valore dell'argomento di chiamata della **UpdateData**; per cui, istruzioni del tipo: **if (pDX-> m_b)** servono per capire se i dati sono in ingresso nella *dialog-box* (**FALSE**) o in uscita (**TRUE**);
- **void Fail()** : questa funzione (di solito preceduta da un messaggio) viene chiamata quando si é verificato un errore (per esempio l'operatore ha inserito in un *controllo* un valore non consentito) e inibisce l'uscita del programma dalla *dialog-box*, anche se l'operatore ha premuto **OK**.

La **DoDataExchange**, di norma, non usa direttamente i *membri* **m_b** e **Fail()** della classe **CDataExchange**, ma stabilisce le associazioni fra *controlli* e variabili chiamando particolari funzioni *globali* (cioé non appartenenti a nessuna classe), identificate dal prefisso **DDX_** (per lo scambio dei dati) o **DDV_** (per il controllo di validità degli stessi) e trasmettendo a tali funzioni l'argomento **pDX**, che permette di accedere alla classe **CDataExchange**; per esempio, la funzione **DDX_Text(pDX,ID,var)** gestisce lo scambio di dati fra la variabile **var** e un *controllo a casella di testo*, identificato dal simbolo **ID** (a sua volta definito in fase di generazione grafica della *dialog-box*, mediante un "*resource editor*"), mentre la funzione **DDV_MinMaxInt(pDX,n,1,20)** stabilisce che la variabile intera **n** deve essere compresa fra **1** e **20**. La libreria del sistema mette a disposizione un gran numero di funzioni **DDX_** e **DDV_** , che si differenziano per il tipo delle variabili e dei *controlli*; tali funzioni sono selezionate automaticamente da **ClassWizard** che prepara, sulla base della lista dei *controlli* della *dialog-box* e delle *variabili-membro* da associare ad essi, il codice della funzione **DoDataExchange** con le chiamate alle funzioni **DDX_** e **DDV_** appropriate. In condizioni normali, tutto ciò é sufficiente, e il programmatore non ha necessità di modificare il testo della **DoDataExchange** preparato da **ClassWizard**; se però vuole ottenere risultati particolari (come nel nostro caso), ha piena libertà di intervento, per esempio creando blocchi di codice separati per i dati in entrata e quelli in uscita (discriminati dal test della variabile **pDx->m_b**), oppure sviluppando proprie funzioni **DDX_** e **DDV_** ecc...

Nella classe **CSelDlg** la funzione **DoDataExchange** é stata completamente riscritta: nella stesura definitiva é costituita da circa 100 righe di codice, di cui solo 4 provengono dalla versione originaria creata da **ClassWizard** (chiamate di funzioni **DDX_Control** che associano la *ListBox* e i 3 *controlli Edit* a *variabili-membro* di tipo **CListBox** e **CEdit**). Inoltre si é fatto in modo che la stessa **DoDataExchange** non sia eseguita soltanto al momento della generazione della *dialog-box* (**UpdateData(FALSE)** automatico) e in quello della sua cancellazione con **OK** (**UpdateData(TRUE)** automatico), ma anche ogni volta che nella *ListBox* cambia la stringa selezionata: ciò é stato realizzato molto semplicemente, *mappando* il *messaggio* di **Windows**: **ON_LBN_SELCHANGE**, associato alla *ListBox*, con una *funzione-membro* di **CSelDlg** (che abbiamo chiamato **OnSelchangeNames**) contenente l'istruzione **UpdateData(FALSE)**; questo accorgimento fa sì che la **DoDataExchange** venga rieseguita (con **pDX->m_b =FALSE**) ogni volta che l'operatore seleziona una stringa della *ListBox* oppure modifica la selezione precedente.

L'aspetto più interessante della funzione **CSelDlg::DoDataExchange** consiste nella parte che utilizza una nuova classe (che abbiamo chiamato **CExchDate**), creata per scambiare le informazioni di tipo "data" fra *dialog-box* e programma; i codici di *intestazione* e *implementazione* di questa classe si trovano in files della directory **Classi** (insieme alle classi derivate da **CRecordset**), in quanto la **CExchDate** é utilizzata anche dai programmi **Anaper** e **Adsenze**. Per capire da dove é nata l'esigenza di creare questa nuova classe, richiamiamo l'attenzione sulla classe di libreria **CTime**, che serve per gestire le date e i tempi ed é estremamente utile e ricca di funzionalità: premesso che un *oggetto* **CTime** é in sostanza una variabile intera che rappresenta il numero assoluto di secondi trascorsi da un certo istante di riferimento (le ore **0:0:0** dell'**1/1/1970**), la classe é costituita da varie *funzioni-membro* che permettono di estrarre dall'*oggetto* una sua qualunque componente (anno, mese, giorno, giorno della settimana, ore, minuti, secondi), oppure di convertire l'*oggetto* in una stringa, secondo il formato voluto (con la *funzione-membro* **Format**), oppure ancora di eseguire assegnazioni (=) e confronti (==, <, > ecc...) fra due *oggetti* (la classe gestisce perfino l'ora legale!); inoltre, vari *costruttori* in *overload* permettono di creare un nuovo *oggetto* date le sue componenti, o dato il numero assoluto di secondi, o per *copia* di un *oggetto* esistente. Tutte le volte che i nostri programmi gestiscono delle date, utilizzano *oggetti* **CTime** (abbiamo stabilito convenzionalmente che le ore, i minuti e i secondi siano **12:00:00**, fissi e uguali per tutti, per rendere possibili confronti fra le date senza errori); in particolare sono *oggetti* **CTime** i valori di due *controlli Edit* della *dialog-box* associata alla classe **CSelDlg** (le date iniziale e finale del periodo da cui estrarre le informazioni sul dipendente selezionato). Sfortunatamente la libreria non dispone di funzioni **DDX_** e **DDV_** per lo scambio e il controllo di *oggetti* **CTime** e quindi, in casi come il nostro, deve essere lo stesso programma applicativo a provvedere; ed é esattamente quello che abbiamo fatto, con la differenza che, al posto di due funzioni *globali* "cani sciolti", abbiamo preferito, per motivi di comodità, creare un'unica funzione, di nome **DDX_Date**, che svolgesse insieme le operazioni di scambio e di controllo; ma, per fare questo, la funzione necessitava di varie informazioni e quindi abbiamo trovato più funzionale, compatto ed elegante *incapsulare* la funzione e le altre informazioni in una classe: così é la nata la **CExchDate**, di cui la **DDX_Date** é una *funzione-membro* (pubblica), chiamata dalla **DoDataExchange**. Altri *membri* (privati) della **CExchDate** sono:

- due variabili di tipo **CTime**, inizializzate dal *costruttore* con i valori che gli sono passati come argomento; queste variabili, che si chiamano **m_tmin** e **m_tmax**, rappresentano i limiti di accettabilità inferiore e superiore della data fornita dall'operatore;
- la funzione **void Set**, chiamata dalla **DDX_Date** se **pDX->m_b** é **FALSE**; ha il compito di scrivere il valore della variabile **CTime** nella sua *casella di testo* associata: per fare questo, prima converte la variabile in stringa, con la funzione **CTime::Format**, e poi scrive la stringa nella *casella*, con la funzione **SetWindowText**;
- la funzione **BOOL Get**, chiamata dalla **DDX_Date** se **pDX->m_b** é **TRUE**; questa funzione svolge la massima parte del lavoro, perché deve, non solo trasferire il contenuto della *casella di testo* nella variabile **CTime**, ma anche controllare che la data sia stata inserita correttamente nella casella, in base alle seguenti regole:

- la data deve essere costituita da tre numeri, che rappresentano, nell'ordine, il giorno, il mese e l'anno;
- il giorno e il mese devono essere di 1 o 2 cifre, l'anno di 4 cifre (siamo vicini al 2000, non possiamo permetterci di indicare l'anno con solo due cifre!);
- i tre numeri possono essere separati da *blank*, *trattino* (**-**) o *slash* (*/*);
- la data deve essere compresa fra **m_tmin** e **m_tmax**;

la funzione **Get**, per prima cosa, trasferisce il contenuto della casella in una stringa, con la funzione **GetWindowText**; poi analizza la stringa, estraendo i tre numeri, che utilizza per costruire l'*oggetto* **CTime**; infine confronta l'*oggetto* con **m_tmin** e **m_tmax**; se tutte le regole sono state rispettate, lo copia nella variabile **CTime**, trasmessale *by reference* dalla **DDX_Date**, e termina restituendo **TRUE** alla **DDX_Date**, che a sua volta termina trasferendo la variabile (passata sempre *by reference*) alla **DoDataExchange**; se invece la data non é valida, la **Get** notifica il tipo di errore con un messaggio e termina restituendo **FALSE** alla **DDX_Date**, che chiama la **pDX->Fail** per inibire l'uscita del programma dalla *dialog-box* (finché i dati non sono corretti ... non si esce! ... oppure si preme il tasto "**Cancel**"!).

Torniamo ora alla funzione **DoDataExchange** ed illustriamone in dettaglio il contenuto. La funzione svolge, in successione, 4 gruppi di operazioni:

1) associazione fra i *controlli* della *dialog-box* e le *variabili-membro* di tipo **CListBox** e **CEdit**: questa parte, creata da **ClassWizard**, non é stata modificata; il nome delle variabili é: **m_ctlDip** (associata alla *ListBox*), **m_ctlvdatain**, **m_ctlvdatafi** (date iniziale e finale del periodo da cui estrarre le informazioni, nel caso di selezione di un dipendente), **m_ctlanno** (anno da esaminare nel caso di selezione della "SITUAZIONE GENERALE DELLE FERIE");
2) operazioni eseguite se **pDX->m_b** é **FALSE** (dati in ingresso): la funzione controlla se é stata selezionata una stringa della *ListBox* (con la chiamata **m_ctlDip.GetCurSel()**) e imposta di conseguenza la *variabile-membro* **m_CodDip**, a cui assegna il valore **-1** se nessuna stringa é selezionata, **0** se é selezionata la prima riga, un valore **> 0** se é selezionata una riga con il nome di un dipendente (in questo caso **m_CodDip** coincide con il codice del dipendente); quindi, in funzione di **m_CodDip**, cambia la scritta di intestazione della *dialog-box* (con la funzione **SetWindowText**), annulla o crea le stringhe appropriate dei *controlli Static* (che associa ai *controlli* stessi tramite funzioni **DDX_Text**) e, soprattutto, definisce quali *controlli Edit* devono essere visibili e quali nascosti; per esempio, se é stato selezionato un dipendente devono essere visibili i *controlli* con le date e nascosto quello con l'anno, cosa che si ottiene con le seguenti chiamate:

```
m_ctlvdatain.ShowWindow(SW_SHOW);
m_ctlvdatafi.ShowWindow(SW_SHOW);
m_ctlanno.ShowWindow(SW_HIDE);
```

inoltre, se **m_CodDip** é **> 0**, viene chiamata la *funzione-membro* **GetDipInterval** che, accedendo al database tramite il puntatore **this** alla classe *documento*, a sua volta trasmesso al *costruttore* della classe **CSelDlg** e da questo memorizzato in una propria *variabile-membro*, restituisce le date della prima e dell'ultima registrazione relative al dipendente selezionato (ricavate dalle due tabelle, **Movimenti** e **Provvisori**, unite insieme); tali date, che cambiano in funzione del dipendente, sono memorizzate in due *variabili-membro* **CTime**, di nome **m_tmin** e **m_tmax**;

3) operazioni eseguite qualunque sia il valore di **pDX->m_b** (dati in ingresso o in uscita): la funzione esegue il test di **m_CodDip** e quindi:
   a) se **m_CodDip** é **> 0** (selezionato un dipendente), crea un'*istanza* della classe **CExchDate** trasmettendole come argomenti di chiamata le *variabili-membro* **m_tmin** e **m_tmax** (che il *costruttore* di **CExchDate** memorizza nelle proprie omonime *variabili-membro*); usa poi (due volte) la funzione **DDX_Date** (come *membro* dell'*oggetto* **CExchDate** appena creato) per lo scambio bidirezionale dei dati fra i due *controlli Edit* associati a **m_ctlvdatain** e a **m_ctlvdatafi** e due *variabili-membro* **CTime**, di nome **m_vdatain** e **m_vdatafi**;
   b) se **m_CodDip** é uguale a **0** (selezionata la "SITUAZIONE GENERALE DELLE FERIE"), chiama la funzione di libreria **DDX_Text** per lo scambio bidirezionale dei dati fra il *controllo Edit* associato a **m_ctlanno** e una *variabile-membro* intera, di nome **m_anno**; inoltre, chiama la funzione di libreria **DDV_MinMaxInt** per controllare che il valore introdotto di **m_anno** sia compreso entro i limiti prefissati;
4) operazioni eseguite se **pDX->m_b** é **TRUE** (dati in uscita, quando l'operatore preme il bottone "**OK**"): la funzione esegue il test di **m_CodDip** e quindi:
   a) se **m_CodDip** é **< 0** (nessuna stringa selezionata nella *ListBox*), emette un messaggio di errore e poi chiama **pDX->Fail**;
   b) se **m_CodDip** é uguale a **0**, pone **m_vdatain** uguale all'1 gennaio e **m_vdatafi** uguale al 31 dicembre dell'anno selezionato; poi controlla che vi siano registrazioni nella tabella **Movimenti** o nella tabella **Provvisori** e, in caso contrario, emette un messaggio di errore e chiama **pDX->Fail**;
   c) se **m_CodDip** é **> 0**, esegue tre ulteriori controlli: che **m_vdatafi** non sia minore di **m_vdatain**, che l'anno delle due date sia lo stesso e che ci sia almeno una registrazione, relativa al dipendente selezionato, nel periodo di tempo selezionato; se almeno uno dei tre controlli dà esito negativo, la funzione emette un messaggio di errore e chiama **pDX->Fail**;

   infine, se tutto va bene, la **DoDataExchange** termina; il controllo ritorna alla **DoModal**, che cancella la *dialog-box* (ma non l'*oggetto* di **CSelDlg**) e restituisce **TRUE** alla **InfSelect** della classe *documento*: questa utilizzerà, per selezionare i dati, le *variabili-membro* (pubbliche) **m_CodDip**, **m_vdatain** e **m_vdatafi** di **CSelDlg.**

Un'altra classe interessante di **Dipcheck**, associata a una finestra di dialogo *modale*, é la **CLegenda**. Per chiarire a cosa serve questa classe, richiamiamo l'attenzione su quando, citando il (poverissimo) menù di **Dipcheck**, abbiamo accennato a una voce normalmente disattivata (vedi pag.29): questa voce, con *caption* "Legenda", non é visibile mentre é attiva la *dialog-box* associata a **CSelDlg**. Quando l'operatore ha effettuato la selezione e premuto il bottone "**OK**", la funzione **InfSelect** della classe *documento* riprende il controllo e, sulla base delle scelte dell'operatore, provvede alla preparazione dei dati da visualizzare, impostando, fra l'altro, il valore della *variabile-membro* (pubblica) **BOOL m_bGenFerie** con il valore **TRUE** se é stata selezionata la prima riga della *ListBox* ("SITUAZIONE GENERALE DELLE FERIE"), oppure con il valore **FALSE** se é stato selezionato un singolo dipendente. Nel primo caso le informazioni da visualizzare, che riguardano tutti i dipendenti e il periodo di tempo di un anno, devono essere preparate in una forma molto compatta e non sufficientemente

autoesplicativa: é quindi necessario associare alla presentazione dei dati un dispositivo di tipo "*help*", attivabile su chiamata, che illustri il significato dei simboli utilizzati.
A tale scopo, quando la **InfSelect** termina restituendo il controllo alla funzione **Pre-Draw** della classe di *vista*, quest'ultima interroga il valore di **m_bGenFerie** e, se lo trova **TRUE**, abilita la voce di menù "Legenda" (ricavando l'indirizzo della finestra *frame* con **AfxGetMainWnd** e, tramite questo, l'indirizzo del menù con **GetMenu**, e infine chiamando la funzione **EnableMenuItem).** Quando all'operatore si presenterà sullo schermo il prospetto annuale completo delle ferie (già fatte o in previsione di fare) di tutti i dipendenti, gli sarà offerta anche la possibilità di attivare, selezionando la voce di menù "Legenda", una *dialog-box* in sola lettura contenente tutte le informazioni esplicative necessarie.
Torniamo ora alla classe **CLegenda**: questa é associata a una *dialog-box* semplicissima, costituita dal bottone "**OK**" e da un *controllo Static*; il testo del *controllo* é a sua volta associato, tramite una **DDX_Text** di libreria chiamata dalla **DoDataExchange**, a una *variabile-membro* (pubblica), di tipo **CString** e di nome **m_txtLegenda**; dall'altra parte la **InfSelect** prepara un array di stringhe con le frasi esplicative, che memorizza in una *variabile-membro* (pubblica) di nome **LegendaArray** (il contenuto di questo array non é fisso, ma dipende dai parametri selezionati, e quindi deve essere preparato ogni volta - inoltre **LegendaArray** serve anche per la stampa e perciò non può essere generato direttamente nella *dialog-box*). I due percorsi confluiscono nella *funzione-membro* **OnLegenda** della classe di *vista*, che *mappa* la voce di menù "Legenda": questa funzione crea un'*istanza* di **CLegenda**, nella cui *variabile-membro* **m_txtLegenda** copia l'intero **LegendaArray** (concatenando insieme le sue stringhe e inserendo un carattere *new-line* fra una stringa e l'altra); poi chiama la **CLegenda::DoModal** per attivare la *dialog-box*, che appare con un unica stringa di testo suddivisa in più righe, e infine, quando l'operatore, con suo comodo, ne ha letto il contenuto e ha premuto "**OK**", termina, cancellando sia la *dialog-box* che l'*oggetto* di **CLegenda** (senza interrogare il valore di ritorno della **DoModal**, che non interessa, in quanto la *dialog-box* é di sola lettura).

Infine, riteniamo utile citare un'altra *dialog-box* di **Dipcheck**, ancora più semplice della precedente, ma con una caratteristica particolare: non é di tipo *modale*, cioé non é attivata e disattivata dall'operatore, ma dal programma. In altre parole, la classe associata non usa la *funzione-membro* **DoModal**, che ha l'effetto di trasferire il controllo dal programma alla *dialog-box* e di restituirlo al programma solo quando l'operatore preme il bottone "**OK**" o "**Cancel**", ma le *funzioni-membro* **SendMessage(WM_INITIDIALOG)** per l'attivazione e **DestroyWindow()** per la disattivazione (fra le due chiamate la *dialog-box* resta visibile, ma non "cattura" il flusso del programma); inoltre, per generare una *dialog-box* non usando **DoModal**, occorre che il *costruttore* della classe associata chiami esplicitamente la *funzione-membro* **Create.**
Questo tipo di *dialog-box* é usato in **Dipcheck** quando il programma é impegnato in elaborazioni piuttosto lunghe (per esempio le operazioni eseguite dalla **InfSelect** per la preparazione dei dati, che possono occupare il computer per una decina di secondi) e quindi, per non dare all'operatore l'impressione che il programma stesso si sia "bloccato", compare nel frammentre una *box* con un *controllo Static* contenente la scritta "ATTENDERE, PREGO", che scompare non appena il programma é di nuovo in grado di ricevere comandi dall'operatore. Tutto ciò si ottiene con la classe

CWaitDlg, che supporta una *dialog-box non modale*, costituita esclusivamente da un *controllo Static*. Questa classe é usata anche in **Adsenze** e pertanto i suoi files di codice si trovano nel directory **Classi**. Un'*istanza* della **CWaitDlg** viene creata come variabile locale della funzione **InfSelect**, dopo il ritorno dalla **DoModal** della **CSelDlg**, e cancellata alla fine della stessa **InfSelect**, appena i dati sono pronti per essere visualizzati.

Stampe

Tutti e quattro i programmi di gestione del personale permettono di organizzare i dati in formato di stampa, su richiesta dell'operatore. A volte le stampe vengono prodotte in risposta a una selezione da menù, a volte é lo stesso programma che chiede, mediante una *dialog-box* interrogativa (con risposte SI/NO), di trasformare in stampa i dati presentati in visualizzazione sullo schermo.

- In **Anaass** la stampa può essere prodotta selezionando una voce di menù; viene creata una lista che elenca il contenuto della tabella **CodiciAssenze** e presenta, in ciascuna riga, il codice numerico, la descrizione e il codice CNR di un'assenza. Poiché, come abbiamo visto, la tabella non é precostituita (a parte quattro voci obbligatorie, vedi pag. 16), ma cresce dinamicamente con l'aggiunta di nuove voci solo quando se ne presenta la necessità durante la registrazione delle assenze giornaliere, é importante che nel titolo della lista compaia anche la data di creazione della stampa, come indicatore dello stato di aggiornamento della tabella.
- Anche in **Anaper** la stampa può essere prodotta selezionando una voce di menù; in questo caso la lista elenca il contenuto della tabella **Dipendenti**, una riga per ogni dipendente (compresi quelli non più in servizio), che presenta il codice numerico, il nome e cognome, e le date di assunzione e cessazione dal servizio (tali date sono inessenziali se il dipendente era già in servizio al momento dell'introduzione delle procedure automatiche e non cesserà dal servizio entro l'anno corrente; in questi casi le date formali di assunzione e cessazione sono fissate, nella tabella, rispettivamente all'1/1/1996 e al 31/12/2020 e non vengono riportate nella lista, cioé vengono sostituite, al loro posto, da spazi bianchi). In aggiunta, la lista riporta alcune informazioni estratte dalla tabella **FerieInfo**, e precisamente quelle (con **CodDip** negativo, vedi pag. 24) riguardanti il numero di giorni di ferie fruibili dell'anno in corso (suddivisi fra codice **94** e codice **32**) e dell'anno precedente (anche in questo caso é importante la data di produzione della stampa, che infatti appare insieme al titolo della lista).
- In **Adsenze** si possono produrre stampe in due circostanze:
  1) quando l'operatore seleziona da menù l'opzione di uscita da un Prospetto riassuntivo: in questo caso compare una *dialog-box* (creata dallo stesso *distruttore* dell'*oggetto* di **CProspView**) con la richiesta di riportare in stampa le informazioni visualizzate sullo schermo; se l'operatore risponde SI, il contenuto informativo della stampa prodotta é identico a quello del Prospetto, ma l'apparenza formale se ne discosta notevolmente: per far stare più dati possibile sulla stessa riga (relativa a un dipendente, come nel Prospetto), i valori numerici dei giorni di assenza per ogni codice non sono incolonnati ma sono scritti l'uno di seguito all'altro (solo se diversi da zero) e accompagnati, ciascuno, dal codice dell'assenza a cui si riferiscono;
  2) quando l'operatore seleziona da menù l'opzione di stampa di un Rapporto mensile (i Rapporti mensili non vengono visualizzati, ma solo stampati): in questo caso la stampa prodotta contiene le informazioni richieste dal CNR, in quanto il Rapporto dovrà essere trascritto su apposito modulo e inviato alla Sede Cen-

trale; tali informazioni consistono (sempre una riga per ogni dipendente) nell'elenco individuale delle date di assenza con, accanto a ogni data, il codice CNR dell'assenza (se le date sono consecutive vengono scritte le sole date iniziale e finale separate da un trattino).

- Infine in **Dipcheck** non vengono prodotte stampe durante la sua esecuzione, ma solo alla fine, quando l'operatore seleziona la voce di menù che comanda l'uscita dal programma; questa voce é *mappata* dalla funzione **OnAppExit**, che, nella classe base, si limita a inviare il messaggio WM_CLOSE alla finestra *frame*. La **OnAppExit** della classe *documento* di **Dipcheck**, che *scavalca* la funzione della classe base, esegue alla fine la stessa operazione, ma prima chiede all'operatore se vuole la stampa: se la risposta é SI viene riprodotta una copia identica (sia nel contenuto che nella forma) di quello che appare sullo schermo, con l'aggiunta che, se **m_bGenFerie** é **TRUE** (vedi pagg. 34 e 35), vengono trascritte anche le stringhe di **LegendaArray**, per accompagnare le note esplicative all'elenco dei dati.

Come si può notare, i programmi generano complessivamente 6 diverse stampe, ma il procedimento adottato é lo stesso in tutti casi: si é preferito rinunciare alle funzionalità della classe di *vista* messe a disposizione dal sistema (*funzioni-membro* **OnPreparePrinting**, **OnBeginPrinting** ecc...), che, secondo il nostro parere, richiedono complessità di strutture e organizzazioni in questo caso non necessarie, e si é deciso di non produrre mai direttamente delle stampe, ma di creare, ogni volta che é richiesta una "stampa", un semplice file di testo, da visualizzare e stampare successivamente con qualche utility di **Windows** (per esempio con il comando *Print* di **notepad**). Per evitare ambiguità o ripetizioni e garantire che ogni volta venga creato un file con nome diverso, si utilizza l'ora corrente (ottenuta con la funzione **GetCurrentTime**, *membro* di **CTime**), da cui vengono estratti il giorno (**dd**), l'ora (**hh**), i minuti (**mm**) e i secondi (**ss**), generando la stringa **fn="ddhhmmss.txt"**; successivamente si crea un'*istanza* della classe di libreria **CStdioFile**, passando al *costruttore* la stringa **fn** come nome del file da creare e la specifica della *modalità di accesso* in sola scrittura. Questa operazione produce il triplice risultato di creare un *oggetto* di **CStdioFile**, di aprire un file di nome **fn** e di associare il file all'*oggetto*, e permette, in seguito, di utilizzare la funzione **WriteString**, *membro* di **CStdioFile**, per scrivere stringhe di testo nel file. Alla fine si usa *funzione-membro* **Close** per chiudere il file (l'*oggetto* invece resta in vita, ma verrà automaticamente *distrutto* al termine della funzione che l'ha creato, essendo una sua *variabile locale*).
A volte (per esempio nelle stampe di **Dipcheck**), é sufficiente trasferire i dati nel file così come sono, una riga dopo l'altra: in questo caso basta chiamare una sola volta la **WriteString** per la scrittura di ogni riga, aggiungendo, in fondo alla stringa da scrivere, il carattere *new-line*; altre volte (come nella stampa di **Anaper**) i dati vanno prima preparati e incolonnati: per ottenere questo risultato nel modo più semplice, é stata fatta la funzione **MyCopy**, che copia una stringa all'interno di un'altra, ma, a differenza della funzione di libreria disponibile per questa operazione, non inserisce il *terminatore* **NULL** alla fine della stringa di destinazione e quindi permette di effettuare l'inserimento in qualunque posizione (colonna), senza distruggere la parte successiva (alla fine, la stringa di destinazione, composta delle varie parti e con un *new-line* in fondo, viene scritta su file, sempre tramite la **WriteString**).

Per concludere, vogliamo precisare che questa gestione (forse un po' "casalinga") dei files, con l'uso della classe **CStdioFile** al posto delle più note **CFile** e **CArchive**, potrà forse sembrare poco "elegante", ma é, a nostro parere, pienamente indicata allo scopo, perché permette un controllo più diretto dei dati, che compensa la rinuncia a certi automatismi, peraltro non necessari quando i files da produrre sono di puro testo.