CardsWorkShop V1.7c Help Contents

Dedicated to the memory of Sophie D'Amboise
à la mémoire d'une complicité parfaite

Introduction to CardsWorkShop V1.7c

CardsWorkShop is a integrated editor/compiler/player allowing the quick design and play of solitary card games. The language used ressembled PASCAL and is object oriented. A good number of examples are included.

Your feedbacks will be appreciated. And of course don't forget to register.

Quick How-to-use

Playing :

Double click on the icon representing the game you want to play.

Compiling :

Load a *.cdl file from the file|open menu or by double-clicking on its icon in a Games Icons List Box. Compile it by pressing the appropriate button in the Game Window.

Menu Commands

File Menu
- **Play Menu**
- **Full Menu**
- **New**
- **Open...**
- **Save**
- **Save As...**
- **Exit**

Edit Menu
- **Undo**
- **Cut**
- **Copy**
- **Paste**
- **Clear**
- **Find...**
- **Replace...**
- **Next**

Game Menu
- **Compile**
- **Run**
- **Open/Run...**
- **Directories...**
- **Make All...**
- **Include .cdl**
- **Include .cvc**
- **Choose New Game**
- **Choose Deck...**
- **Make Icon**

Play Menu
- **Rules**
- **ReStart**
- **Enter Seed...**
- **Shadows**
- **Redraw**

Start
Undo

File|Full Menu

This option lets you use the full menu with more options, suitable for development.

File|Play Menu

This option let you use the play menu with options useful for playing only.

File|New

New opens a new Edit window with the default name <ANONYMOUS> and automatically makes the new Edit window active.

These anonymous files are used as a temporary edit buffer.

CWS prompts you to name an anonymous file when you save it.

File|Open...

The File|Open dialog box appears. It is where you open a file by typing the file name in the input box or using the list boxes to find and open the file.

File Name input box

The File Name input box is where you enter the name of the file to load, or the file-name mask to use as a filter for the Files list box.

Files list box

The Files list box lists the names of files in the current directory that match the file-name mask in the File Name input box, plus the parent directory and all subdirectories.

Directories list box

You view the contents of different directories by selecting a directory name in the Directories list box.

File|Save

The Save command saves the file in the active Edit window to disk.

If the file has a default name (<ANONYMOUS>), CWS opens the File|Save As dialog box so you can rename the file and save it in a different directory or on a different drive.

File|Save As...

Save As opens up the File Save As dialog box, where you can save the file in the active Edit window under a different name, in a different directory, or on a different drive.

The File Save As dialog box is where you type in the new name in the File Name input box (you can include a drive and directory path) or use the Directories list to select a new path.

If you choose an existing file name, CWS asks if you want to overwrite the existing file.

The window containing this file is updated with the new name.

File|Exit

The Exit command exits CWS and removes it from memory.

If you've modified a source file without saving it, CWS prompts you to do so before exiting.

Also, you can press Alt+F4 to exit.

Edit|Undo or Undo

In an Edit Window :

The Undo command "undoes" the most recent edit or cursor movement.

Undo inserts any characters you deleted, deletes any characters you inserted, replaces any characters you overwrote, and moves your cursor back to a prior position.

If you undo a block operation (Cut, Copy, Paste or Clear, your file will appear as it was before you executed the block operation.

In a Game Window :

The Undo command "undoes" your most recent transaction on the playfield.

If you keep on pressing Undo, it continues to undo the changes you made during the current game.

Edit|Cut

The Cut command removes the selected text from your document and places the text in the Clipboard.

You can then choose Edit|Paste to paste the cut text into any other document (or somewhere else in the same document).

The text remains selected in the Clipboard so you can paste it as many times as you want.

Edit|Copy

The Copy command leaves the selected text intact but places an exact copy of it in the Clipboard.

To paste the copied text into any other document, choose Edit|Paste.

Edit|Paste

The Paste command inserts the selected text from the Clipboard into the current window at the cursor position.

Edit|Clear

The Clear command removes the selected text but does not put it into the Clipboard.

This means you can't paste the text as you could if you had chosen Cut or Copy.

Although you can't paste the cleared text, you can undo the Clear command with Undo.

Edit|Find...

You use the Find Text dialog box to specify the text you want to search for.

Search for input box

This input box is where you enter the search string. Choose OK to begin the search, or choose Cancel to forget it.

Case Sensitive

When the Case Sensitive option is on, CWS differentiates uppercase from lowercase when performing a search.

Case Sensitive Off is the default.

Edit|Replace...

The Replace Text dialog box is where you specify the text to search for and what to replace it with.

Search for input box

Enter the search string in the Text to Find input box and choose OK to begin the search, or choose Cancel to forget it.

Replace with input box

Enter the replacement string in the New Text input box.

Case Sensitive

When the Case Sensitive option is On, CWS differentiates uppercase from lowercase when performing a search.

Case Sensitive Off is the default.

All Occurences

Set All Occurences On if you want CWS to replace all occurrences of the search string found.

Prompt On Replace

When the Prompt On Replace option is On, CWS prompts you before replacing each time it finds the search string.

When Prompt On Replace is Off, CWS automatically replaces the search string.

Edit|Next

The Next command repeats the last Find or Replace command.

The last settings made in the Find Text or Replace Text dialog box remain in effect when you choose Next.

Game|Compile

The Compile command compiles the file in the active edit window.

If an error occurs, the status bar displays the error and a token near the error is highlighted, in the file that caused the error.

Game|Run

The Run command executes the last compiled program.

Game|Choose New Game

This command opens a new Games Icons List Box if none is openned.

Game|Open/Run

This dialog works like the File|Open dialog, except that the file choosen must be an executable file (*.cvc) and is executed after selection.

Game|Directories...

The Game|Directories dialog box appears. It is where you set path for three type of files :

.CDH
include files, included by #include at compile time
.CVC
executable files, created by compiling and icon files created by making an icon
.BMP
graphical files for bitmaps in the FACE instruction

You can set the path by typing it in the input box or using the browse... button to move thru directories.

Game|Make All...

After you specify a directory, all sources files in it are compiled. If an error happens, the compiling stops and an error report is given.

Game|Choose Deck...

This opens a dialog in which you can choose the default deck of card for all games. Only one deck at a time is active.

Use the scroll bar to choose the deck and then press the Ok button.

Game|Shadows

Turns moving shadow accompanying game transactions On or Off.

Game|Include *.cdl

Indicate if source files (*.cdl) get icons in Games Icons List Box

Game|Include *.cvc

Indicate if executable files (*.cvc) get icons in Games Icons List Box

Game|Make icon

When you select this, a snapshot of the current playing window is made and will serve as icon in Games Icons List Box.

a *.bmp file is created with the same name as the executable.

Play|Rules

Puts you in inspecting mode. In this mode when you click on a stack with a mouse button, information (rules of the game) will be displayed (if the game programmer created some).

Technically, the Help procedure of the selected stack is called.

Play|ReStart

Lets you start the current game over again.

Play|Enter Seed...

Lets you see the current game random seed and lets you change it.

Entering the same seed twice will permit you to play the same game twice.

Play|ReDraw

Redraw the current window.

In Games Icons List Box this command lets you update the list of icons if you just compiled a new game for the first time, or just created an icon.

In Game Window this is useful when the fireworks go awry.

Start

Starts a new game in the current Game Window.

Playing compiled games

Glossary of solitaire terms
Games Icons List Box
Game Window

Glossary

Values	As in most card games, Ace, Two, Three and so on , including the picture cards, Jack, Queen, King.
Suits	Consisting of Clubs, Diamonds, Hearts, Spades.
Colors	Of which there are two, Red and Black.
The Tableau	Consists of single cards, groups or piles which have their own purposes and limitations, as described in each game.
Foundations	Are cards upon which others are built to form complete sequences, thus terminating the game. The Foundations may be part of the original Tableau, or they may be established during play, according to the individual game.
Sequences	
Ascending	Run from a low card, usually an Ace, on up to the high card, as A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K.
Descending	Run from a high card , usually a King, on down to the low card, as K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, A.
Auxiliary Cards	Belong to The Tableau, which may be built upon Foundations, or may be used for forming temporary sequences, according to the rules.
Rows	Are cards dealt horizontally in The Tableau, either singly or overlapping, as specified.
Columns	Are cards dealt vertically in The Tableau, either singly or overlapping, as specified.
The Stock	Is a term applied to the remainder of the pack after The Tableau has been arranged.
The Reserve	Is a packet or group of cards that is laid aside or specially retained for building on Foundations.
Available Cards	Are any that are free for building on Foundations, or for transfer to auxiliary cards or columns.
Blocked Cards	Are those which must in some way be released to become available.
Waste Pile	Consists of cards that can not be used when dealt and therefore must be laid aside. Some games are lost when all the stock has gone into the Waste Pile. Others allow the Waste Pile to be used as a new Stock, as specified in the rules of individual games.

Games Icons List Box

The purpose of the Game Icons List Box is to ease the work when playing with files, a little like the File Manager. The surface is divided in three parts described below.

Path Box

In the upper left corner is a Path Box. It indicates the current path for all the icons in the Icons Box. When you quit CWS this informations is saved in CWS.INI.

Directory Box

In the upper right corner is a drop down list diplaying all the directories and drives you can choose to change the path in the Path Box.

Icon Box

The lower, biggest part of the window displays icons connected to source files and executable files. You can set what kind of icons are displayed here by using the Game|Include .cdl and Game|Include .cvc menu.

To run a executable file or to open a source file, double-click on its icon.

Game Window

The Game Window is the actual place where you play solitaire. It is one big playfield covered with different stacks depending on which game you are playing.

Transactions on the playfield are made with the mouse by clicking on stacks and dragging cards. Some stacks will respond automatically when you click a button on them while others will let you drag cards elsewhere.

Usually by choosing the Play|Rules item in the menu you can inspect the different stack and get informations about how they work.

You can Undo transactions at any time and up to the beginning of the game.

Creating new Solitaire Games

Introduction to Creation

CWS is a form of object-oriented language. You can define and modify object and you can customize instance of that object. The customization can be complemented by inheritance from other instances. Some call this prototyping and some call this actors. I would go with the prototyping because you can easily program and test your ideas within as little as one hour.

Here is a list of important concept in CWS :

Objects
Instances
Stacks

Transaction
System Predicates
Log
Playfield

Object definition

An object is a collection of constants, variables, predicates, procedures and functions. An object do not really exists, only instance of that object can exists. Every procedure of an object, when executed, can access the variables and constants of the instance.

See also Objects for a complete description.

Instance definition

An instance is a separate entity created from an object definition. Each instance created can have its own values for its constants and its variables. Also its behavior (methods) can be customized. For example, A stack is an object while a foundation is an instance of a stack.

See also Instance for a complete description.

Stack definition

A stack is an object. It is a container for an ordered set of cards. A stack can contain 0, 1 or more cards (up to 204) and a bottom of pile drawing indicating the state of the pile. This drawing is usually a red cross, a green circle or a shaded card.

Each stack can respond to different messages sent to it by the playing environment : start of game, selection with the mouse, destination of a drop with the mouse or request for information.

The mouse itself is a stack with restriction (Cursor).

See also Stacks for a complete description.

Transaction

A transaction is the transport of cards from one stack to another between the time the user press the mouse button and the time he releases it.

For the programmer this means :

First case :

A) A stack answer the mouse button selection message and put some cards on the mouse stack (Cursor)

B) The user moves the mouse without releasing the mouse button to another stack.

C) The user releases the button and the destination stack is informed of the drop. If this stack refuses the cards on the mouse stack (if he doesn't removed them all) then the transaction is cancelled, otherwise the transaction is accepted and completed.

Second case :

A) A stack answer the mouse button selection message and send some cards to another stack(s) on the playfield but none to the mouse stack (Cursor). The transaction is completed when the mouse button is released.

If the user, when dragging, releases the cards someplace that doesn't answer to the drop message, the transaction is cancelled.

When a transaction is cancelled, the state of the game goes back to just before the transaction.

The last transaction can be cancelled with Undo.

System Predicates

There are three global system predicates that can be defined by the programmer : integrity? to check system Integrity, win? to check if player has won and loose? to check if he has lost.

integrity? predicate

Is executed after each transaction to check the system integrity. If False is returned, the last transaction is undone.

If absent, the system doesn't check for integrity. It is equivalent to always returning true.

This can also be used for special operations like turning some new cards side up.

```
predicate integrity? is
    begin
    with it do
        if IsSideDown?(it[it!]) then Turn it[it!] side up
    for A1, A2, A3, A4;
    return True;
    end;
```

win? predicate

Is executed after each transaction to check if the game is won. If True is returned, the game is over and the player is told of his success.

If absent, the game will never end with success. It is equivalent to always returning false.

The win? predicate is always checked before the loose? predicate, so if they both return true the player wins.

```
predicate win? is
    return (A1!=13) and (A2!=13) and (A3!=13) and (A4!=13);
```

loose? predicate

Is executed after each transaction to check if the game is lost. If True is returned, the game is over and the player is told of his failure.

If absent (sometimes it is easier for the player to see he has lost than to program it), the game will never end with failure. It is equivalent to always returning false.

The win? predicate is always checked before the loose? predicate, so if they both return true the player wins.

```
predicate loose? is
    return (D1!=0) and not MovePossibleOnTableau?;
```

Playfield

The playfield is divided in a serie of row and columns giving a big matrix. The size of this matrix is defined independently for each game in its header.

Each cell of the matrix is one square unit but will not necessarily be square physically on the screen. For example if you define the matrix to be 10 by 10, and the game is played on a 640 by 480 window, then each square of the matrix will be 64 by 48 pixels.

The space taken by each stack is described by defining a rectangular sub-matrix inside the screen matrix. See X, Y, W and H constants in Stacks.

Stacks can overlap.

One limitation is that the width and height of the matrix must not exceed 320 by 200.

When the Game Window is resized the physical size of the matrix changes and the new size for each object on the playfield is computed and the window is redrawn.

With the normal card handler VCARDS and HCARDS, the size of a card is computed like this : every stack is checked and we keep the minimum height and minimum width found. We take the physical size of the resulting minimum rectangle and try to fit the biggest possible card frame in it. The card frame is always at a ration of 2 horizontally for 3 vertically.

If a the **grid** variable is defined with:
#define grid
Then the playfield matrix is visible when playing.

See also visual aspect

Visual Aspect

Every modified stack in the execution of a Select Method are redrawn at the end of the Method. This lets you do many operations on a stack (like turning cards, reversing some subsequence order, etc.) without worrying about the visual aspect.

Modified stacks are also redrawn after the execution of the integrity? predicate or explicitly with the execution of the DRAW instruction.

Log

When a transaction is accepted it is added to a transactions log. At this point, the user can undo the last transaction by choosing the undo commands in the menu.

The user can undo every transaction up to the start of the game. The system starts logging transaction after the execution of every start method.

The programmer has nothing to do for all this, it is done automatically. It should be noted that only global variables are logged. So if you use working variables which don't need to be global, better make them local so to not overload the log.

Editor Window

Edit windows are where you type in and edit your CWS code. You can also do the following in an edit window:

compile your programs
run your programs
read them from disk files
save them to disk files

You can open as many edit windows as you want but each one is limited to around 32K of text.

To open an edit window, choose File|Open. You can open the same file in more than one window.

The buttons at the top of the window are shortcuts for menu items of that name.

CardsWorkShop language description

Program layout

program ::= **header ';' (object_def | instance_def | const_def | var_def | pred_def | proc_def | func_def)* order_def [statement] '.'**

the statement after the order_def is executed only once when the game is loaded for execution. It is useful for defining the title variable and any global options variables.

PreProcessor

CWS includes a preprocessor much like the usual C preprocessor. the following commands are available :

#include 'file'
include the text of 'file' at the current point. The file is taken from the include directory

#define n
defines the symbol n for the preprocessor only. A symbol can be defined or undefined.

#ifdef n
will compile the following text up to a #else or a #endif if n has been previously defined by #define.

#ifndef n
will compile the following text up to a #else or a #endif if n has not been previously defined by #define.

#else
change the state of compilation if used between a #ifdef or and #ifndef and a #endif.

#endif
stop the restrictions put by the previous #ifdef or #ifndef.

#define grid

If a the **grid** variable is defined, then the playfield matrix is visible when playing.

Header

header ::= **GAME *id* IS *integer* *BY* *integer***

The *id* is used to assign a name to the executable file that is generated by compiling. The file is named *id*.cvc .

The first *integer* describes the width of the playfield and the second *integer*, the height.

ORDER instruction

order_def ::= ORDER *id* (',' *id*)*

id is of type STACK.

The body of a program end by an ORDER statement. It a list ordering the stack instances. This order applies to initialisation (Start method) and general redrawing of stacks. So the first stack is under all others if overlap occurs.

The order of the stacks is followed in reverse order to determine stacks selection with the mouse.

Stacks not registered in the ORDER instruction must not be accessed in any other way than by inheritance.

Generaly, the first stack is where you add the deck(s) or cards and shuffle them :

```
Add Ace+Spade .. King+Diamond;
Shuffle;
```

Then in the Start method of the other stacks you pull cards from that first stack (here C1) :

```
Pull 4 from C1;
```

CONST

gives an immutable value to an identifier.

const_def ::= **CONST const_elm (',' const_elm)* ';'**
const_elm ::= ***id*** **':=' const_exp**
const_exp ::= **expression_without_var**

There is many predefined constants in CWS, see the file system.cdh .

VAR

var_def ::= **VAR (var_elm ',')***
var_elm ::= ***id*** **(',' *id*)* ':' types**

A variable (var) declaration associates an identifier and a type with a location in memory where values of that type can be stored.

TITLE string variable

var TITLE : string;

If defined it will be the title string used in the window header of the Game Window and in the Games Icons List Box.

SCORE integer variable

var SCORE : integer;

One day it will be used to keep high-core. One day...

Types

CWS has predefined simple types that are used in variables and functions definitions.

If new objects are created, a new type with the same name as the object is created, like STACK and CARDS. The SELF variable has the type of the object to which the code is associated to.

The following is a list of these types.

types ::= **STACK | INDEX | CARD | INTEGER | BOOLEAN | STRING | CARDS | GROUP | object_name**
object_name ::= ***id***

STACK type

Variable of this type can be associated with any defined Stack. Any operation to a Stack can be applied to a Variable of type STACK.

The iteration variable in a with instruction is of type STACK.

INDEX type

Indexes are used as indices to access cards on a stack. They are used inside the [...] in a stack.

Integers can be used too.

CARD type

A card is what is contained in the array of a stack. It acts like an integer.

In the normals card handlers VCARDS and HCARDS the card type is defined like this :

0..12	is the range from Ace to King of Spade
13..25	Ace to King of Heart
16..38	Ace to King of Club
39..51	Ace to King of Diamond
52..103	is the range from Ace of Spade to King of Diamond but side down
104..155	is the range from Ace of Spade to King of Diamond side up but shaded
156	is the green circle
157	is the red cross
158	is an empty card. It is not seen nor can it be selected. It is useful in horizontal and vertical stacks in reduction games (like pyramid).

So simple integer arithmetic can be applied on a value of CARD type.

```
{****are c1 & c2 of different color}
predicate AlternateColor?(c1, c2 : Card) is
    return (((c1 / 13) + (c2 / 13)) mod 2) = 1;
```

INTEGER type

Integers are the whole numbers you learned to count with (1, 5, -21, and 752, for example).

The allowed range is -32768 to 32767.

BOOLEAN type

Indicates variables which can have the value TRUE or FALSE.

STRING type

Strings are a combination on characters inside a couple of quote.

For example, 'This is a string'.

You can use \ before a special character inside a string :

\t	tab
\n	newline
\'	quote
\\	back-slash

CARDS type

Variables of this type can be associated with any defined card handler. Any operation to a card handler can be applied to a Variable of type CARDS.

GROUP type

group_constant ::= '|' [stack (',' stack)*] '|'

A variable or constant of this type is a list of stack instance. The list variable in a with instruction can be of type GROUP.

A group can be empty.

OBJECT

An object (class) is an abstract (not tangible) structure, from which tangible instance can be created.

Many instances can be constructed from the same object definition. In CWS each instance can be customized by modifying its behavior on message activation (method call). This is different from the usuals object-oriented languages. The default behavior can be specified in the object definition.

object_def ::= OBJECT *id* IS object_body END *id* ';'
object_body ::= (const_Init | meth_Init | multi_Init)*
const_init ::= CONST *id* ':' types [':=' const_exp] ';'
var_init ::= VAR *id* ':' types [':=' const_exp] ';'
meth_init ::= (PREDICATE | PROCEDURE | FUNCTION) *id* [parms] [IS statement] ';'
multi_Init ::= *id* IS *id* (',' *id*)* ';'

Note that in object_def, OBJECT *id* and END *id* must be the same *id*.

An object can be extended in multiple parts except that every instances defined before an extention doesn't possess the new fields.

A constant in an object can be defined to a specific static value in an instance.
A variable in an object can be given dynamicaly a value and is local to procedures of the object or of the instance.
A procedure in an object can be totally abstract (without a statement body) or have a default behavior (a statement body).
A multiple interface is a way to a init many slots in an instance by assigning statically a value to the multiple interface name. For exemple, in stacks selectleft is a multiple interface for selectleftfrom and selectleftto.

Contextual Object

When you call a procedure, function or predicate defined by specifying the type of object, this procedure is in the object instance context. That means that you can directly access the variables of the calling instance.

The SELF variable has the type of the contextual object.

For example :

```
stack procedure DoShade(Spos : index; c1 : Card) is
  begin
  [Spos]:=c1;
  Turn [Spos] side shaded;
  Turn [!] Side down;
  end;

stack D1 is
  Select(Spos : index) is
    DoShade(Spos,King+Spade);
end D1;
```

In the DoShade procedure, [Spos] becomes implicitly D1[Spos] and [!] becomes D1[D1!].

SELF variable

var SELF : object_name

SELF is a variable that can be accessed inside any method or procedure, function or predicate in the context of an object. It is a variable of type object_name and referenced the current Instance.

Instance

An instance is a tangible (not abstract) structure, coming from an object definition. Many instances can be constructed from the same object definition. In CWS each instance can be customized by modifying its behavior on message activation (method call). This is different from usual object-oriented languages.

instance_def ::= object_name *id* [FROM *id2*] IS stack_body END *id* ';'
object_name ::= *id*
stack_body ::= (const_Init | meth_Init)*
const_init ::= *id* ':=' const_exp ';'
meth_init ::= *id* [parms] (FROM *id2* | IS statement) ';'

Note that in instance_def, object_name *id* and END *id* must be the same *id*.

An instance can inherit all or any fields from another one.
An instance definition can be split up in multiple parts.
An instance must be defined before it is referenced. You can use forward declaration to help you :

stack W1;

You can even defined multiple instance in one declaration:

stack A1, A2, A3, ..., An;

Instances defined before object extend do not possess the newer slots.

Instance inheritance

If FROM *id2* is present in instance_def then all fields defined to this point are duplicated in the new instance.

If FROM *id2* is present in meth_init then this methods is duplicated from the *id2* instance. If the method being defined can be broke up in multiple parts, then all lower methods are duplicated. See Select.

STACK object

A stack object instance is an aggregate of 10 fields : 6 constants, 2 special variables and 3 methods including one that can be broke up in 2 or 4 parts.

See the description of stacks in the introduction section.

The five constants are :
- X : Integer
- Y : Integer
- W : Integer
- H : Integer
- Direction : Integer
- Handler : Cards

The two special variables are :
- ! : Index
- [...] : card

The three methods are :
- Start
- Select
- Which can be sub-divised :
  - SelectFrom
  - SelectTo
  - SelectLeft
  - SelectLeftFrom
  - SelectLeftTo
  - SelectRight
  - SelectRightFrom
  - SelectRightTo
- Help

Many instructions work with stacks :
- Add
- Draw
- Flash
- Inverse
- Move
- Pull
- Remove
- Shuffle
- Turn
- With

Example of a Stack

CURSOR stack instance

CURSOR is a Stack but with restriction. For one the Direction is always DOWN and cards can be on the CURSOR stack only while in a Transaction.

You don't have to and shouldn't declare it or write methods for it.

Also, it cannot be extended

X and Y stack constant

X and Y give the upper left position of the Stack in the virtual matrix of the playfield defined in the game header.

see also W and H

W and H stack constant

W and H give the width and height of the Stack in the virtual matrix of the playfield defined in the game header.

see also X and Y

DIRECTION stack constant

DIRECTION describes the way the card are stacked on one another. The possible values are : UP, DOWN, LEFT, RIGHT, OVER, HORIZONTAL, LHORIZONTAL, VERTICAL or UVERTICAL.

UP, DOWN, LEFT, RIGHT and OVER stacks have their cards stacked consecutively on their surfaces and the range of selection with the mouse can go one beyond the length of the stack.

HORIZONTAL and VERTICAL stacks have their cards spreaded equally on their surfaces and the range of selection with the mouse cannot go beyond the length of the stack.

LHORIZONTAL and UVERTICAL stacks have like their HORIZONTAL and VERTICAL counterpart but draw their cards in the reverse direction.

HANDLER stack constant

This constant must refer to a defined card handler (CARDS). This handler will assign a graphical face to each value possible for a card on this stack. If no handler is specified, a default handler is associated to the stack (VCARDS or HCARDS) in fonction of the Direction constant.

! stack variable

! is represent the length (number of cards) on the Stack. It can be preceded by the stack it qualifies or else it qualifies the contextual object.

! is of type INDEX.

[...] stack variable

[...] is the array of cards in the Stack. It can be preceded by the stack it qualifies or else it qualifies the contextual object.

It is indexed by variable of type INDEX or INTEGER.

The card at position 0 is the empty pile drawing. It is better to limit it to EmptyCard, CrossCard or a shaded Card.

[0]:=EmptyCard

By default the empty pile drawing is the EmptyCard (green circle) with the default handler VCARDS and HCARDS.

START stack procedure

At the beginning of a game (after the user press Start) this method is called by the system. It is called for each stack in the order defined in the Order part of a program.

Example of a Stack

SELECT method

Select(Spos : Index)

This method is called to answer a message from the mouse. It means that a mouse button was pressed or released over the card at the Spos position in this Stack. In UP, DOWN, LEFT, RIGHT and OVER stacks Spos can be one greater than the length of the stack.

The Spos parameter must be present but its name can be customised to your taste.

This method can be subdivised in two disjoint set :

SelectRight(SPos : Index)
SelectLeft(SPos : Index)

Works like Select but differentiate between the left mouse button and the right mouse button.

SelectRightFrom(SPos : Index)
SelectRightTo(SPos : Index)
SelectLeftFrom(SPos : Index)
SelectLeftTo(SPos : Index)

Works like the SelectRight or SelectLeft but differentiate if a button is pressed (from) or released (to).

Select...From are used to pickup and Select...To are used to drop.

For example if you define the select behavior for a Stack, all messages (button right or left, pressed or released) are treated by the same code.

HELP stack procedure

When the Game Window is in inspecting mode (see Game|Rules menu) and a mouse button is pressed over a Stack, the Help method of that stack, if any, is executed.

Usually this method will open a Text Box (using Clear), write text (using Write) and/or add button (using Wait).

STACK Example

Here's an example of a Stack object :

```
stack W1 is
  X := 2;
  Y := 2;
  Direction := down;
  W := 2;
  H := 12;
  //****************************
  Start is
    begin
    Pull 6 From D1;
    Turn [1..3] Side Up;
    end;
  //****************************
  Select(Spos : Index) is
    Pull 1 To D2;
end W1;
```

A stack definition can be split up in multiple parts.

```
stack D1 is
  Select(Spos : index) is
    DoShade(Spos,King+Spade);
end D1;

stack C1 is ...

stack D1 is
  Start is
    Pull 1 from C1;
end D1;
```

D1 is the same instance, methods and constant definitions are accumulated in D1. If methods or constant are redefined, the lowest definition prevails.

CARDS object

A cards object instance is an aggregate of 7 fields : 6 constants and 1 method.

A card handler will be used in a stack Handler variable to assign a graphical face to each value possible for a card. This is done in the init method of the handler. One handler can be used in many stacks.

The five constants are :

- W : Integer
- H : Integer
- RatioW : Integer
- RatioH : Integer
- Default : Card
- Sync : Cards

The method is :

- Init

Example of a Card handler

VCARDS cards instance

This constant handler can be used in the handler variable of a stack.

It shows a standard deck of card with the value and suit drawn horizontally at the top of the card so it can be read easily in vertical stacks.

HCARDS cards instance

This constant handler can be used in the handler variable of a stack.

It shows a standard deck of card with the value and suit drawn vertically on the top-left of the card so it can be read easily in horizontal stacks.

W and H cards constant

W and H gives the width and height of a card in the virtual matrix of the playfield defined in the game header.

If any of these variable is 0, a largest fit algorithm will be applied : scan all stack which uses this handler card and keep the largest fit as possible.

The default handler VCARDS and HCARDS use the largest fit method.

RatioW and RatioH cards constant

RatioW and RatioH are used to force a ratio on the horizontal and vertical axis of a card. For exemple, the default handler VCARDS and HCARDS use a ratio of 2/3 : RATIOW = 2, RATIOH = 3.

If any of the ratio is 0, no ratio is forced.

DEFAULT cards constant

This constant defines the card that will be used at the position 0 of a stack (the empty pile drawing).

SYNC cards constant

If this constant is defined, it must refer to a defined card handler (CARDS). It assures that this handler will have the same geometrical attributes (W,H and Ratio) as the one it is sync to.

INIT cards procedure

When the game is first executed (loaded), this method is called for each card handler.

It usually consists of series of FACE statement which defined the visual of each card of the handler.

CARDS Example

```
//****default card handlers
Cards HCARDS is
  W := 0;
  H := 0;
  RATIOW := 2;
  RATIOH := 3;
  DEFAULT := EmptyCard;
  //******************
  Init is
    Face 0..255 is NORMAL HORIZONTAL;
end HCARDS;

Cards VCARDS is
  W := 0;
  H := 0;
  RATIOW := 2;
  RATIOH := 3;
  DEFAULT := EmptyCard;
  SYNC := HCARDS;
  //******************
  Init is
    Face 0..255 is NORMAL VERTICAL;
end VCARDS;
```

for exemple to redefine the default card set backs :

```
CARDS vcards IS
  Init IS
    BEGIN
    FACE 0..255 IS NORMAL VERTICAL;
    FACE 52..103 IS BITMAP 'uglyback.bmp';
    END;
END vcards;
```

Expressions

expression ::= expression ('+'|'-'|'*'|'/'|AND|OR|MOD|'<<'|'>>') expression | expression ('='|'<>'|'<='|'>=') expression | ('+'|'-'|NOT) expression | [*id*] ! | [*id*] '[' expression ['..' expression] ']' | *id* | *integer* | '(' expression ')'

Expressions are evaluated from left to right in short-circuit (meaning that when the value of a boolean expression is determined, evaluation stops).

This is the priority of the operators. Operators on a same line have the same priority and are evaluated from left to right (left associative).

Hi

unary NOT + -
AND MOD * / >> <<
OR + -
= <> <= >=

Low

Instructions

:= Assignation
id Procedure call

Add stack instruction
Begin
Break
Clear helpbox instruction
Draw stack instruction
Face cards instruction
Flash stack instruction
If
Inverse stack instruction
Move stack instruction
Pull stack instruction
Remove stack instruction
Return
Shuffle stack instruction
Turn stack instruction
Wait helpbox instruction
While
With

:= assignation

statement ::= **left_value ':=' expression**
left_value ::= ***id*** **| stack_pos |** ***id*** **'.'** ***id***

Assign the value of the expression to the memory cell described by the left_value.

Left_value is a variable, a position in a stack array or a variable of an instance.

Procedure Call

statement ::= ***id*** **[parms] |** ***id*** **'.'** ***id*** **[parms]**

Let you call a global procedure or an instance procedure.

ADD stack instruction

statement ::= **ADD card_interval [TO stack_dst [pos]]**
stack_dst ::= ***id***

Add specified cards to a stack. If no position is specified the cards are added at the end.

BEGIN ... END instruction

statement ::= BEGIN (statement)* END

When bracketed in this way, any number of consecutive statements can be treated as a single statement.

BREAK instruction

statement ::= BREAK [PROCEDURE]

BREAK is used to get out of a block (in a WITH or in a procedure). To get out of a procedure when in a WITH body, use BREAK PROCEDURE. A BREAK inside a WHILE body acts like a BREAK PROCEDURE.

CLEAR helpbox instruction

statement ::= CLEAR

Closes the HelpBox if it is open.

statement ::= CLEAR *string* [AT *integer* ',' *integer* IS *integer* BY *integer*]

Opens the HelpBox giving the title *string*. If the four *integer* are present, they specify a position (x, y) and a size (w, h) on the playfield matrix.

DRAW stack instruction

statement ::= **DRAW [** ***id*** **]**

Forces the redrawing of Stack *id*.

FACE cards instruction

statement ::= FACE card_interval IS card_statement
card_statement ::= NORMAL (HORIZONTAL | VERTICAL) | FACE number [SIDE SHADED] |
BITMAP string [SIDE SHADED]

This statement, used in a Card handler describe the visual aspect of a range of card. Three different card_statement can be used to describe the aspect :

NORMAL means to take it from the normal playing cards selected by the user. HORIZONTAL and VERTICAL means to use the set with the value to the left or at the top.

FACE is used to duplicates an entry already in the current handler. SIDE SHADED can be added to make the new entry shaded.

BITMAP mean to load the bitmap file string in the bitmap directory. SIDE SHADED can be added to make the bitmap shaded.

FLASH stack instruction

statement ::= FLASH stack_interval

Makes the cards in the interval flash three times.

IF instruction

statement ::= IF expression THEN statement [ELSE statement]

If the Boolean expression after IF is True, the statement after THEN is executed.

Otherwise, if the ELSE part is present, the statement after ELSE is executed.

INVERSE stack instruction

statement ::= INVERSE stack_interval

Inverse the order of the cards in the interval. Inversing an ascending sequence makes it a descending sequence.

MOVE stack instruction

statement ::= MOVE stack_interval TO stack_pos

Moves the contents of stack_interval to the stack_pos. The two stacks MUST be different.

Example :
A1=[13, 5, 8, 3, 2, 4]
A2=[1, 6, 9]

MOVE A1[2..4] TO A2[3];

A1=[13, 2, 4]
A2=[1, 6, 5, 8, 3, 9]

PULL stack instruction

statement ::= **PULL expression [FROM stack_src] [TO stack_dst]**
stack_src ::= ***id***
stack_dst ::= ***id***

Takes the last *n* (expression) cards of the source stack and add them at the end of the stack destination.

If stack_dst or stack_src is not specified then the context must specify them.

REMOVE stack instruction

statement ::= REMOVE stack_interval

Removes an interval of cards in a stack, destroying the cards.

RETURN instruction

statement ::= RETURN expression

It is used to get out of a function or a predicate and to specify the return value.

SHUFFLE stack instruction

statement ::= **SHUFFLE [stack_dst]**
stack_dst ::= ***id***

Shuffles the cards in a stack.

TURN stack instruction

statement ::= **TURN stack_interval SIDE (UP | DOWN | SHADED)**

Turn all cards in the interval on the specified side.

WAIT helpbox instruction

statement ::= WAIT *string id*

Adds a button titled *string* to the HelpBox, opening it if necessary. If the button is pressed, the procedure *id* is executed. *id* must be a procedure without argument and with no context object.

statement ::= WAIT expression *id*

Wait expression/60 seconds before calling the procedure *id*. The instructions following the wait are executed immediately. If you want to do a perfect timer, put the next wait at the start of the called procedure.

statement ::= WAIT expression

Wait expression/60 seconds before continuing the execution.

WHILE instruction

statement ::= WHILE expression DO statement

The statement after DO is executed repeatedly as long as the Boolean expression is True.

The expression is evaluated before the statement is executed, so if the expression is False at the beginning, the statement is not executed at all.

WITH instruction

statement ::= **WITH stack_dst DO statement FOR (stack_src (',' stack_src)* | group_var)**
stack_src ::= ***id***
group_var ::= ***id***
stack_dst ::= ***id***

WITH is an iterator construct. The statement is executed successively for each stack_src or each stack in the GROUP variable. stack_dst is used inside the statement to access the stack_src of the current iteration.

Interval

stack_interval ::= [*id*] range
stack_pos ::= [*id*] pos

Indicates an interval in a stack or a position. The allowed range is [0..!] for any stack.

If the stack (*id*) is not specified then the context must specify it.

range ::= '[' expression '..' expression ']' | '[' expression ']'
pos ::= '[' expression ']'

In range, if the second option is used, it describes a range of one card.

card_interval ::= expression .. expression | expression

Describes an interval of cards. For example, a flush in heart could be : NINE+HEART .. KING+HEART.

PROCEDURE

proc_def ::= **[object_name] PROCEDURE *id* [parms] IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A procedure is a program part that performs a specific action, often based on a set of parameters.

The procedure header specifies the identifier for the procedure and the formal parameters (if any).

A procedure is activated by a procedure call statement.

Since a procedure must be declared before being used and sometimes circular references would be useful, you can declare a procedure before it is defined like this :

procedure D1(it : stack);

A definition can be preceded by an object name if the procedure should be executed in the context of an instance of the object.

WRITE special helpbox procedure

statement ::= **WRITE '(' expression (',' expression)* ')'**

Writes a serie of expressions (of type CARD, INTEGER or STRING) in the HelpBox, opening it if necessary.

FUNCTION

func_def ::= **[object_name] FUNCTION *id* [parms] ':' types IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A function is a program part that computes a value of type types, often based on a set of parameters.

The function header specifies the identifier for the function and the formal parameters (if any).

A function is activated in an expression.

Since a function must be declared before being used and sometimes circular references would be useful, you can declare a function before it is defined like this :

function Higher(it : stack): card;

A definition can be preceded by an object name if the function should be executed in the context of an instance of the object.

function random(n : integer): integer;

Does : computes a random integer between 0 and n-1
parameters : n
Returns : An integer between 0 and n-1

PREDICATE

pred_def ::= **[object_name] PREDICATE *id* [parms] IS [var_def] statement ';'**
parms ::= **'(' var_list (';' var_list)* ')'**

A predicate is a program part that computes a boolean value, often based on a set of parameters.

The predicate header specifies the identifier for the predicate and the formal parameters (if any).

A predicate is activated in an expression.

Since a predicate must be declared before being used and sometimes circular references would be useful, you can delacre a predicate before it is defined like this :

predicate Empty?(it : stack);

By convention a predicate name should end with a question mark.

A definition can be preceded by an object name if the predicate should be executed in the context of an instance of the object.

There are three special predicates you can define :

- Win?
- Loose?
- Integrity?

Include files

- deck.cdh
- function.cdh
- poker.cdh
- predicat.cdh
- rightbut.cdh
- sequence.cdh
- stack.cdh
- system.cdh

deck.cdh

```
stack procedure OneDeckDown;
stack procedure OneDeckUp;
```

stack procedure OneDeckDown;

Does : Add one full deck (52 cards), face down, to a stack
parameters :
Returns :

stack procedure OneDeckUp;

Does : Add one full deck (52 cards), face up, to a stack
parameters :
Returns :

function.cdh

```
function min(a, b : integer): integer;
function max(a, b : integer): integer;
function CSide(as : card):integer;
```

function min(a, b : integer): integer;

Does : computes the smallest of two values
parameters : two integers to compare
Returns : a if smaller than b else b

function max(a, b : integer): integer;

Does : computes the biggest of two values
parameters : two integers to compare
Returns : a if bigger than b else b

function CSide(as : card):integer;

Does : computes a value to add to a card side up to turn it on another side
parameters : SHADED or DOWN
Returns : the value to be added

poker.cdh

stack predicate Straight?;
stack predicate Flush?;
stack predicate OnlyTwo?;

stack predicate Straight?;

Does : for a stack with five cards, check if the cards make a straight (five cards in sequence)
parameters :
Returns : TRUE if a straight else FALSE

stack predicate Flush?;

Does : for a stack with five cards, check if the cards make a flush (five cards of the same kind)
parameters :
Returns : TRUE if a flush else FALSE

stack predicate OnlyTwo?;

Does : for a stack with five cards, check if the cards are of only two different value. This includes a full house and a four of a kind.
parameters :
Returns : TRUE if has only two value else FALSE

predicat.cdh

```
predicate FollowRankWrap?(c1, c2 : Card);
predicate FollowRankWrapN?(n : integer; c1, c2 : Card);
predicate FollowRank?(c1, c2 : Card);
predicate FollowRankN?(n : integer; c1, c2 : Card);
predicate FollowSuit?(c1, c2 : card);
predicate AlternateColor?(c1, c2 : Card);
predicate SameSuit?(c1, c2 : Card);
predicate SameRank?(c1, c2 : Card);
predicate SameCard?(c1, c2 : Card);
predicate SameColor?(c1, c2 : Card);
predicate IsSideDown?(c1 : card);
predicate IsShaded?(c1 : card);
predicate Smaller?(c1, c2 : Card);
predicate IsAce?(c1 : card);
predicate IsKing?(c1 : card);
```

predicate FollowRankWrap?(c1, c2 : Card);

Does : Check if two cards are following each other in a sequence, wrapping around to the Ace when the king is reached
parameters : c1 is the lower card, c2 the higher one
Returns : TRUE if they follow

predicate FollowRankWrapN?(n : integer; c1, c2 : Card);

Does : Check if two cards are n cards away from each other in a sequence, wrapping around to the Ace when the king is reached
parameters : c1 is the lower card, c2 the higher one
Returns : TRUE if they are spaced n cards away

predicate FollowRank?(c1, c2 : Card);

Does : Check if two cards are following each other in a sequence
parameters : c1 is the lower card, c2 the higher one
Returns : TRUE if they follow

predicate FollowRankN?(n : integer; c1, c2 : Card);

Does : Check if two cards are n cards away from each other in a sequence
parameters : c1 is the lower card, c2 the higher one
Returns : TRUE if they are spaced n cards away

predicate FollowSuit?(c1, c2 : card);

Does : Check if two cards are following each other in a sequence and they are the same kind
parameters : c2 is the lower card, c1 the higher one
Returns : TRUE if they follow and are the same kind
comments : c1 and c2 are inversed from the other predicate, it will be correted

predicate AlternateColor?(c1, c2 : Card);

Does : Check if two cards are of different color (one black and one red)
parameters : the two cards to check
Returns : TRUE of they are different color

predicate SameSuit?(c1, c2 : Card);

Does : Check if two cards are of the same kind (suit)
parameters : the two cards
Returns : TRUE if they are of the same suit

predicate SameRank?(c1, c2 : Card);

Does : Check if two cards are of the same value (rank)
parameters : the two cards
Returns : TRUE if they are of the same value

predicate SameCard?(c1, c2 : Card);

Does : Check if two cards are the same value and kind
parameters : the two cards
Returns : TRUE if they are of the same value and kind

predicate SameColor?(c1, c2 : Card);

Does : Check if two cards are the same color (black or red)
parameters : the two cards
Returns : TRUE if they are of the same color

predicate IsSideDown?(c1 : card);

Does : Check if a card is turned side down
parameters : the card to check
Returns : TRUE if it is side down

predicate IsShaded?(c1 : card);

Does : Check if a card is shaded
parameters : the card to check
Returns : TRUE if it is shaded

predicate Smaller?(c1, c2 : Card);

Does : Check if one card is smaller than the other one in value (Ace<King)
parameters : c1 is the small one, c2 is the bigger one
Returns : TRUE is c1 is smaller than c2

predicate IsAce?(c1 : card);

Does : Check if the card is an ace
parameters : the card to check
Returns : TRUE if it is an ace

predicate IsKing?(c1 : card);

Does : Check if the card is a king
parameters : the card to check
Returns : TRUE if it is a king

rightbut.cdh

```
procedure RButton;
stack procedure PollLeftTo(gr : group);
stack procedure PollLeftToNE(gr : group);
```

procedure RButton;

Does : Help text for usual right button quick moves
parameters :
Returns :
comments : in help : WAIT 'Right Button' RButton ;

stack procedure PollLeftTo(gr : group);

Does : You call this procedure after putting cards on the cursor stack. It will call every SelectLeftFrom method of the stacks in the group until one grabs all the cards or all are checked.
parameters : The group of stack to check
Returns :

stack procedure PollLeftToNE(gr : group);

Does : You call this procedure after putting cards on the cursor stack. It will call every SelectLeftFrom method of the stacks in the group until one grabs all the cards or all are checked. Stacks on the group with no cards on them will not be checked.
parameters : The group of stack to check
Returns :

sequence.cdh

stack predicate KingToAceSuit?;

stack predicate KingToAceSuit?;

Does : Check a stack to see if it contains a sequence of cards from King to Ace
parameters :
Returns : TRUE is it is a full sequence

stack.cdh

stack procedure MoveFirstFrom(c1 : card; src : stack);
stack procedure MoveAllFrom(c1 : card; src : stack);

stack procedure MoveFirstFrom(c1 : card; src : stack);

Does : Grabs the first card to match the one wanted in the source stack
parameters : c1 is the card wanted, src is the stack to check
Returns :

stack procedure MoveAllFrom(c1 : card; src : stack);

Does : Grabs all the cards that match the one wanted in the source stack
parameters : c1 is the card wanted, src is the stack to check
Returns :

This file is automatically included at the start of the compilation.

Files

***.cdl**
source files

***.cvc**
executable files

***.csg**
saved player game files

cws.ini
information about the setup of CardsWorkShop. resides in the main windows directory.

Copyright for CardsWorkShop

All the source games are copyrighted David Jean, 1994. some are copyrighted Charles-E. Jean, 1994 or Jean-Pierre Grenier, 1994.

We are holding no copyright to the graphics in *.bmp files.

We hold no right on *.cdl files you create. Digital copies are perfect, go on and multiply your work... but please not our.

CardWorkShop V1.7c is :

Internet : david.jean@dmi.usherb.ca

Registration

Why should I register?

To get CWS-Library-I, a compilation of 25 ready-to-run with sources solitary card games

To know about Version 2.0 and CWS-Library-II before everybody else

How much will that cost me?

17.95$ U.S, or 19.95$ cdn, p&h included.
oversea, add 3.00$ cdn for p&h.
no c.o.d. please.

Specify if you want a 3.5" or 5.25" disk.

You can ease your job by printing the file order.frm.

Where

SynHeme
C.P. 206
Sherbrooke, Québec, Canada
J1H 5H8

TroubleShooting

Playing

Programming

Stacks don't appear when I run my program...

If no stacks appear on running, make sure that every stack has a W and H greater than 0, inherited or specified.

Make sure the width and height of the playfield is less than 320 by 200.

References for CardsWorkShop

[1] Tarpel, C., Toutes les réussites et jeux de patiences, Guy Le Prat, 1975
[2] Brown, Douglas, 150 Solitaire Games, Harrow Books, 1972
[3] Les règlements officiels des jeux de cartes, International Playing Card Company Limited, 1977
[4] Morehead, Albert H., The pocket book of games, Pocket Books, 1944
[5] Berloquin, Pierre, Les réussites les plus passionnantes, Marabout, 1980
[6] Freha, Pierre, Jouer auz réussites, de Vecchi, 1991
[7] Bezanovska et Kitchevats, Le livre des patiences, Homme, 1987

[8] Wirth, Niklaus, Algorithm + Data Structures = Program
[9] Aho, Sethi, Ullman, Compilateurs, Principes, techniques et outils, InterEditions