Window Creation

New windows are created with a blank white background and a black one pixel wide pen. The initial pen position is at the centre of the window, and the initial heading is 0° (up). If no window is created, all drawing operations apply to the whole screen, writing over whatever windows or icons are already there. Exitting restored the background to its initial state.

All graphical operations use an (x, y) co-ordinate system in which (0, 0) is the top left corner of the window (or of the screen if there is no window). The position (10, 20) always indicates the point 10 pixels from the left of the window and 20 pixels down from the top of the window, regardless of where the window appears on the screen.

`make_window(`*w*`, `*h*`);`

Creates a new graphics window, w pixels wide and h pixels tall, centred on the primary monitor. W and h must both be integers.

`make_window(`*w*`, `*h*`, `*x*`, `*y*`);`

Creates a new graphics window, w pixels wide and h pixels tall. The left edge of the window is x pixels from the left edge of the screen, and the top of the window is y pixels down from the top of the screen. W, h, x, and y must be integers.

Pen Movement and Line Drawing

These functions come in pairs. The pen movement functions have names beginning with `move_`, and the line drawing functions have names beginning with `draw_`. Both functions in each pair behave exactly the same, except that the `draw_` version draws a straight line using the current pen settings from the pen's original position to its new position; the `move_` versions produce no visible effects.

`move_to(`*x*`, `*y*`);`
`draw_to(`*x*`, `*y*`);`

The new position of the pen is the point (x, y), x pixels from the left edge of the window and y pixels down from the top edge. X and y may be any numeric values, not necessarily integers.

`move_relative(`*dx*`, `*dy*`);`
`draw_relative(`*dx*`, `*dy*`);`

The new position of the pen is the point dx pixels to the right of, and dy pixels below the pen's starting position. X and y may be negative to indicate movement to the left or upwards respectively. X and y may be any numeric values.

```
move_distance(d);
draw_distance(d);
```

The new position of the pen is d pixels away from its starting position, in a direction determined by the pen's current heading. The pen's heading is 0° initially, which corresponds to straight up, but may be changed by the functions below. The distance d may be any numeric value.

Headings or directions are oriented so that Positive angles are clockwise, negative angles are anticlockwise, 0° is up (negative y), 90° is right (positive x), 180° is down (positive y), and 270° is left (negative x). Of course, intermediate directions such as 0° behave as would be expected. The functions for setting the pen's heading all have two versions, one measuring the direction in degrees, and one in radians. The two functions behave identically apart from that.

Note that the degrees versions will produce more accurate results when the angle can be measured exactly in degrees. 90 can be represented with perfect accuracy in the computer's hardware, but ½π can not. The pen's heading only has an effect on the `move_distance` and `draw_distance` functions, and only the functions listed in this section below can change the heading - moving the pen does not change its heading.

```
set_heading_degrees(a);
set_heading_radians(a);
```

Set the pen's heading to exactly the angle a.

```
turn_left_by_degrees(a);
turn_left_by_radians(a);
```

The angle a is subtracted from the pen's heading, causing an anticlockwise change of direction.

```
turn_right_by_degrees(a);
turn_right_by_radians(a);
```

The angle a is added to the pen's heading, causing a clockwise change of direction.

Drawing Single Points

```
draw_point(x, y);
```

The pen moves to (x, y) and makes a single dot at that position. A large pen width will cause a large solid circle to appear. The pen's width gives its diameter, not its radius. X and y may be any numeric values.

`draw_point();`

The pen makes a single dot, as described above, at its current position.

`set_pixel_color(`*x*`, `*y*`, `*c*`);`

The single pixel at position (x, y) is set to colour c. This operation pays no attention to the pen's current colour, size, or position. It also leaves those values unchanged. `Set_pixel_color` is faster than using `draw_point` with a one pixel pen. X and y may be any numeric value; c must be a proper colour representation.

`set_pixel_color(`*c*`);`

The single pixel at the pen's current position is set to colour c. This operation pays no attention to the pen's current colour or size. It also leaves those values unchanged.

Colour Representations

The RGB system

RGB matches the way computer monitors show colours. Any colour that can be displayed is broken down into intensities of the three human primary colours Red, Green, and Blue (hence the name). The individual intensities may be given either as integers in the range 0 to 255 or as floating point numbers in the range 0.0 to 1.0. As examples,

R=0	G=0	B=0	represents solid black				
R=1.0	G=0.0	B=0.0	or	R=255	G=0	B=0	represents bright red
R=1.0	G=0.6	B=0.0	or	R=255	G=150	B=0	represents orange
R=1.0	G=1.0	B=0.0	or	R=255	G=255	B=0	represents yellow
R=0.7	G=0.7	B=0.7	or	R=179	G=179	B=179	represents dull grey
R=0.7	G=0.7	B=0.8	or	R=179	G=179	B=205	is a blueish grey

The HLS system

HLS matches the way paints are mixed and colours are printed. A colour is represented by three attributes:

Hue gives the basic colour, red is 0.0 or 1.0, green is 0.3333, and blue is 0.6667, and intermediate values are as expected: yellow is 0.1667, orange is 0.125, cyan (half way between green and blue) is 0.5, and magenta (half way between blue and red) is 0.8333.

Lightness gives the overall brightness, 0.0 reduces every colour to black, 1.0 bleaches every colour to white. Solid primary colours are at 0.5.

Saturation describes how colourful the colour is. 0.0 reduces all colours to grey, 1.0 makes them as colourful as possible. A light blueish grey would have a Hue of 0.6667 for blue, and a Lightness of about 0.75 to make it a light grey, and a Saturation of 0.1 or 0.2 so that it is definitely more grey than colour.

Of the graphics functions that deal with colour, some accept RGB or HLS components directly, as three separate inputs, but some (such as `set_pixel_color`, above) expect to be told a colour as a single value. Colours may be compressed into single values in one of three ways:

`int make_color(`*r*`, `*g*`, `*b*`)`	where r, g, b are floating point between 0.0 and 1.0
`int make_color_int(`*r*`, `*g*`, `*b*`)`	where r, g, b are integers between 0 and 255
`int make_color_hls(`*h*`, `*l*`, `*s*`)`	where h, l, s are floating point between 0.0 and 1.0

Additionally, 26 predefined names for colours are also available:

`color::black`,	`color::dark_grey`,	`color::grey`,
`color::light_grey`,	`color::white`,	`color::dark_red`,
`color::red`,	`color::light_red`,	`color::pink`,
`color::dark_blue`,	`color::blue`,	`color::light_blue`,
`color::cyan`,	`color::dark_green`,	`color::green`,
`color::light_green`,	`color::lime_green`,	`color::yellow`,
`color::sodium_d`,	`color::orange`,	`color::brown`,
`color::indigo`,	`color::mauve`,	`color::violet`,
`color::purple`,	`color::magenta`.	

The following four statements all have exactly the same effect:

```
set_pixel_color(120, 45, color::orange);
set_pixel_color(120, 45, make_color(1.0, 0.6, 0.0));
set_pixel_color(120, 45, make_color_int(255, 150, 0));
set_pixel_color(120, 45, make_color_hls(0.1333, 0.5, 1.0));
```

Pen Settings

`set_pen_color(`*c*`);`
`set_pen_color(`*r*`, `*g*`, `*b*`);`
`set_pen_color_int(`*r*`, `*g*`, `*b*`);`
`set_pen_color_hls(`*h*`, `*l*`, `*s*`);`

Changes the pen colour for all subsequent drawing operations. Colours are as described above.

`set_pen_width(`*w*`);`

Changes the pen width for all subsequent drawing operations. The width w is the diameter of the pen tip in pixels.

`set_pen_width_color(`*w, c*`);`

Combines `set_pen_width` and the first version of `set_pen_color` into a single operation. It is slightly faster to change width and colour together if they are both to be changed.

Filling Areas

`clear();`

Erases the contents of the window, blanking everything to white.

`fill_rectangle(`*x*`, `*y*`, `*w*`, `*h*`);`

Fills the entire rectangular area of width w and height h, with top left corner at (x, y) with the current pen colour. The pen returns to its starting position.

`fill_rectangle(`*x*`, `*y*`, `*w*`, `*h*`, `*c*`);`

Fills the entire rectangular area of width w and height h, with top left corner at (x, y) with the colour c. The pen returns to its starting position, and the pen's colour is not changed.

Text in the Graphics Window

`write_string(`*x*`);`

The value of x is displayed in text immediately to the right of the current pen position. The pen's y co-ordinate sets the base-line (the bottom of characters that have no descenders) of the text. The pen's position is moved right to the end of the displayed text.

The value of x may be a string, an integer, a floating point value, a single character, or a boolean.

`write_string(`*x*`, `*direction*`);`

The value of x is displayed exactly as with the plain `write_string`, except that the direction parameter modifies its position relative to the pen's position. This option does *not* write the text at an angle, for that, the `set_font_angle` functions should be used.

The direction parameter may have any one of the following nine predefined values:

```
direction::center       direction::north        direction::south
direction::west         direction::east         direction::north_west
direction::north_east   direction::south_west   direction::south_east
```

Specifying `direction::north_east` has the same effect as using the plain `write_string` function. The effect is best illustrated by example:

```
set_pen_color(color::red);   move_to(300, 300);   draw_point();
set_pen_color(color::black);
move_to(300, 300);    write_string("North", direction::north);
move_to(300, 300);    write_string("South", direction::south);
move_to(300, 300);    write_string("West", direction::west);
move_to(300, 300);    write_string("East", direction::east);
```

produces this arrangement:


```
move_to(300, 300);   write_string("Center", direction::center);
set_pen_color(color::red);   move_to(300, 300);   draw_point();
```

produces this:

and the first example modified to use the other four directions produce this:

The pen is always left in such a position that another call to `write_string` without any direction parameter would produce text that continues from where the first text ended, with the same alignment.

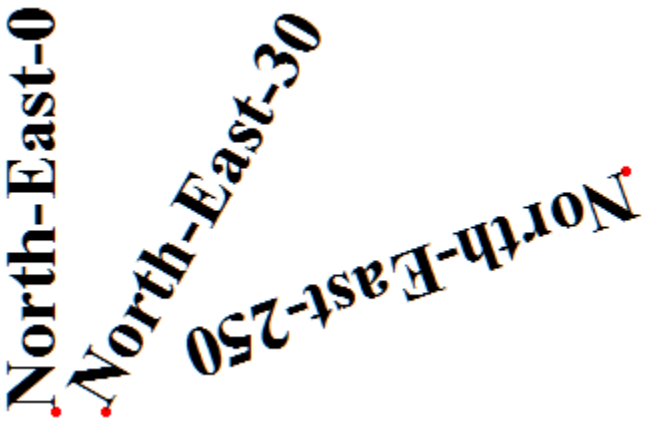
The `north_west` and `north_east` directions align the pen's position with the base-line of the text written. This means that printing with different fonts of different sizes will line up properly.

If the current font is set at an angle, the directions are aligned with the font, not with the screen. So, for example, north_east always starts with the pen position at the base-line of the text.

```
set_pen_color(color::red);    move_to(50, 300);     draw_point();
set_font_angle_degrees(0);    set_pen_color(color::black);
write_string("North-East-0", direction::north_east);

set_pen_color(color::red);    move_to(75, 300);     draw_point();
set_font_angle_degrees(30);   set_pen_color(color::black);
write_string("North-East-30", direction::north_east);

set_pen_color(color::red);    move_to(335, 180);     draw_point();
set_font_angle_degrees(250);  set_pen_color(color::black);
write_string("North-East-", direction::north_east);
write_string(250);
```



`write_string(`*x*`, false);`
`write_string(`*x*`, `*direction*`, false);`

Exactly as above, except that the pen's position is not updated after the text is displayed.

`write_char(...);`

Exactly the same as `write_string`, except that the first parameter must be an integer representing a single character code, and unlike `write_string`, the whole of the unicode character set is available. `write_string` only has access to the basic extended ASCII character set.

`write_new_line();`

Moves the pen position to the left edge of the window, and down by a number of pixels equal to the line height of the largest font used by a `write_`... function since `write_new_line` was last called.

`write_new_line(`*x*`);`

Exactly the same as above, except that the pen is left x pixels from the left edge.

As the `write_string` function takes many different types of parameter, occasionally type ambiguities arise. The following type-specific functions are also available when needed.

`write_char(...);`

`write_int(...);`
`write_float(...);`
`write_double(...);`
`write_bool(...);`

They all work exactly as `write_string` works, and they all a direction and/or false to prevent pen movement. The only difference is that the first parameter must be of the type indicated by their names.

Font Settings.

Text is always displayed using the current pen colour, and is positioned around the pen's current position, but the pen's width and heading have no effect on text.

`set_font(`*name*`);`

The name must be a string, and must match one installed on the system, such as "Times New Roman", "Ariel", and "Courier New". If the chosen font does not exist, an existing font will be substituted.

`set_font_size(`*n*`);`

N must be an integer, and is measured in pixels. N is the height of a whole line of single-spaced text, which is a little larger than the real height of the characters. Setting the font size has no effect on the other font settings.

`set_font_style(`*s*`);`

S must be one of the pre-defined constants

```
style::italic       style::underlined    style::crossed_out
style::normal       style::bold
style::very_light   style::light         style::very_bold
```

Not all fonts have an italic version, and very few have a very_light, light, or very_bold version. If the desired style is not available for the current font, the closest matching style will be substituted. Styles may be combined by adding constants together, for example

```
set_font_style(style::bold + style::italic);
```

selects a bold italic style. Setting the font style has no effect on the other font settings.

`set_font_angle_in_degrees(`*a*`);`
`set_font_angle_in_radians(`*a*`);`

A must be a numeric value. All the characters of the font are rotated before displaying (90° is the regular orientation, set by default. Angles increase in a clockwise direction), Under windows, the angles are not precise, but very close. Setting the font angle has no effect on the other font settings.

`set_font(`*name*`, `*size*`);`
`set_font(`*name*`, `*size*`, `*style*`);`
`set_font(`*name*`, `*size*`, `*style*`, `*angle*`);`

These three combined functions have the same effect as using the individual functions described above in sequence. Changing fonts does require some effect by the windows system, so a program that makes one big change rather than a sequence of smaller ones will be slightly more efficient.

Measuring Text for Display.

The `measure_string` function works out the area (in pixels) that would be occupied if the string were displayed using the `write_string` function. The area is always a rectangle that contains all of the pixels that would be set. *S* must be string, *w* and *h* must be int variables.

`measure_string(`*s, w, h*`);`
`measure_char(`*c, w, h*`);`

`Measure_string` and `write_string` only have access to characters in the standard extended ASCII set. `Measure_char` performs the same task except that it can only measure a single character, but it can be chosen from the entire Unicode range provided by the current font. The first parameter to `measure_char` must be an int giving the character code.

The size calculated by `measure_string` is not the *smallest* rectangle that would enclose the written text. Windows fonts contain blank padding around their characters, and there is no efficient way to exclude all of the empty space from the calculations. The size given is whatever the font designer thought would be the best sized rectangle for displaying the given text.

These functions use the currently selected font at the time they are called. If they are used before a window has been created or before a change of font, they can give inappropriate results.

Some aspects of size apply to a whole font, and do not depend on the string that is to be displayed. The most important of them may be accessed thus:

`get_font_size(`*height, ascent, descent, intlead, extlead*`)`

All five parameters must be int variables. *Height* is the value that `measure_string` will always give for any text in this font. It should be equal to *ascent* plus *descent* plus *intlead*. *Ascent* is the maximum number of pixels above the base-line occupied by normal characters in this font, excluding accents. *Descent* is the maximum number of pixels below the baseline occupied by normal characters. *Intlead* is the "internal leading": space added above the ascent that may be occupied by accents. *Extlead*, "external leading" is the number of pixels of extra space that the font designer thinks should be left between two lines of text so that they don't run together too closely. (*Leading* refers to the very thin strips of lead (the heavy metal) that used to be used in printing to make space between letters).

Retrieving Pen Information.

Text is always displayed using the current pen colour, and is positioned around the pen's current position, but the pen's width and heading have no effect on text.

int `get_pen_color()`

Returns the current pen colour, compressed into a single integer value as created by the `make_color` functions, and as used by the `set_pixel_color` and single-parameter `set_pen_color` functions. These values are the same as the named color constants such as `color::red`. The individual red, green, and blue components of a colour may be found by using the `split_color` function.

int `get_pen_width()`

Returns the current pen width measured in pixels.

double `get_x_position()`
double `get_y_position()`

Return the x or y component of the pen's current position, as a floating point number.

`get_position(`*x, y*`);`

X and y must both be variables, and of either integer or floating point type, but not a mixture of both. Sets the variables x and y to the pen's current position.

```
double get_heading_degrees()
double get_heading_radians()
```

Returns the pen's heading. Regardless of whether the heading was originally set in degrees or radians, these functions perform the appropriate conversion to provide a result in the units suggested by their names.

Colour Conversions.

```
split_color(c, r, g, b);
```

R, g, and b must be floating point variables. C must be the single-value compressed representation of a colour, as given by `make_color`, `get_pen_color`, the named `color::`... constants, etc.

The individual red, green, and blue components of the given colour are extracted as values between 0.0 and 1.0, and stored in the variables r, g, and b.

`Split_color` and `make_color` are not exact mutual inverses, because computer displays do not have an infinite variety of colours available. `Split_color` gives the actual values used by the display, which may be nearly ½% different from those set by the program.

```
split_color_int(c, r, g, b);
```

The same as `split_color`, except that r, g, and b must be integer variables, and the range of results is 0 to 255 instead of 0.0 to 1.0. This is the representation used internally by the windows system, so `Split_color_int` and `make_color_int` *are* not mutual inverses

```
rgb_to_hls(r, g, b, h, l, s);
```

Converts an RGB colour representation to the equivalent HLS representation. R, g, and b must be floating point numbers in the range 0.0 to 1.0. H, l, and s must be variables of type `double`. The range of results is 0.0 to 1.0.

```
hls_to_rgb(h, l, s, r, g, b);
```

Converts an HLS colour representation to the equivalent RGB representation. H, l, and s must be floating point numbers in the range 0.0 to 1.0. R, g, and b must be variables of type `double`. The range of results is 0.0 to 1.0.

Pixel Inspection.

```
int get_pixel_color()
```

Returns as a single integer, the colour of the nearest pixel to the pen's current position.

int get_pixel_color(*x*, *y*)

X and y must be numeric values. Returns as a single integer, the colour of the pixel at position (x, y) in the window. This does not change the pen's position.

Co-ordinate Calculations.

These few functions are of no use to people who remember trigonometry. None of them move the pen or modify any settings.

double distance_to(*x*, *y*)

Calculates the distance from the pen's current position to the point (x, y).

double direction_to_in_degrees(*x*, *y*)
double direction_to_in_radians(*x*, *y*)

Calculates the direction from the pen's current position to (x, y). Setting the heading to this direction, then drawing a line of the length given by `distance_to` would have the same effect as `draw_to(x, y)`.

double direction_from_to_in_degrees(x_1, y_1, x_2, y_2)
double direction_from_to_in_radians(x_1, y_1, x_2, y_2)

Calculates the direction from the position (x_1, y_1) to position (x_2, y_2).

Character-Mode Input and Output.

These functions all communicate with the user through the usually black "dos shell" window. For input operations, that window must be selected (clicked) before typing.

```
print(x);
```

The value of x is printed. It may be a string, an integer, a floating point value, a single character, or a boolean.

```
print_string(x);
print_int(x);
print_char(x);
print_double(x);
print_float(x);
print_bool(x);
```

These are type-specific functions that perform exactly as the generic `print` function, for use when type ambiguity arises.

```
print_hexadecimal(x);
print_binary(x);
```

X must be an integer. It is printed in base 16 or base 2.

```
new_line();
```

Starts a new line of output in the dos shell window.

```
string read_string();
int read_int();
double read_double();
double read_float();
bool read_bool();
```

Each function waits if necessary until a value has been typed, then returns that value as its result. Input is not available until a whole line has been typed (with ENTER at the end), but multiple values may be typed on a single line. Spaces, commas, and new-lines are taken as separators. The following are accepted as boolean values: true, t, false, f, yes, y, no, n, 1, 0; case is not significant.

```
string read_line();
```

A whole line of input is accepted, and returned as the result. Spaces and commas have no special meaning.

Window Manipulation

The window has a text caption in the (usually blue) bar at the top. This may be inspected and changed with these two functions. The caption must of course be a string.

```
string get_caption();
set_caption(s);
```

The size of the window may be found with any of the following three functions. The first two return as their result the width or height of the usable portion of the window, in pixels. The usable portion is the area that graphical operations can draw inside, it does not include the window's frame, scroll bars, caption, or menu bar.

The third function provides both width and height together by setting its parameters, which must be int variables. It does not return a result.

```
int get_window_width();
int get_window_height();
get_window_size(w, h);
```

The size of the entire window, including frames, scroll-bars, captions, etc., may also be found if needed:

```
int get_full_window_width();
int get_full_window_height();
get_full_window_size(w, h);
```

In a similar manner, the size or resolution of the entire display may be found. These values do not depend upon any windows that may be displayed, they are physical properties of the monitor itself:

```
int get_screen_width();
int get_screen_height();
get_screen_size(w, h);

int get_pixels_per_inch();
```

The `get_pixels_per_inch` function returns a value that is not necessarily accurate. It is not determined by the true d.p.i. (dots per inch) of the monitor, but by a software setting in the "display" control panel. It is nearly always 100.

The size of the window may be changed at any time. The function `set_window_size` changes the window so that its usable area is *w* pixels wide and *h* high, the complete window will be rather larger. The second function, `set_full_window_size`, sets the size of the entire window, so the usable area will be smaller. The difference depends upon a multitude of system settings. For both functions, the width and height must be int values.

```
set_window_size(w, h);
set_full_window_size(w, h);
```

The position of the graphics window on the monitor may be accessed and changed:

```
int get_window_x_position();
int get_window_y_position();
get_window_position(x, y);
set_window_position(x, y);
```

X and Y positions for a window are measured in pixels from the left or top edge of the whole display to the left or top edge of the whole window (including its frame). For `set_window_position` *x* and *y* must be int values, and for `get_window_position` they must be int variables. Care should be taken when using `set_window_position`, as it is possible to move a window entirely outside the monitor's display area, which can make closing it difficult.

Window Visibility and the Console

The *Console* is the official name for the (usually black) window, also called the Dos Shell window, that is used for plain text (`read`, `print`) input and output.

Both the graphics window and the console may be made to temporarily disappear. This reduces screen clutter when input is being entered. Making a window disappear has no effect on its contents; when it reappears it will be unchanged and ready for use. The four functions are:

```
hide_window();
show_window();
hide_console();
show_console();
```

Additionally, the caption text for the console, and its size and the position may be changed. As with the graphics window, size and position are measured in pixels and must be int values.

```
set_console_caption(string);
set_console_position(x, y);
set_console_size(w, h);
```

When the `read_`... functions are used to get user input, the console must be *active*, which is usually arranged by the user clicking on it before typing anything. For convenience, or when writing a program for a non-computer person, the program may force the console to become the active window by using this function:

```
make_console_active();
```

Multiple Windows

A program may create as many graphics windows as its programmer desires, bearing in mind that windows consume a lot of memory, so an excessive number of them will cause trouble. The basic `make_window` function may be used repeatedly, each time it creates a new independent graphics window. Each window has its own "personal" pen, font, size, etc settings: operations on one window do not interfere with others.

Whenever a window is created, it becomes the *Current Window*. All operations are performed on the current window only. There are simple functions that allow switching between windows, making a different one current.

The `make_window` function actually returns a result, which is normally ignored. It is a reference to the new window that may be used in the future to make it current again. The type of this value is "window *":

```
window * window_one = make_window(500, 200, 50, 50);
window * window_two = make_window(500, 200, 50, 400);
```

Alternatively, a reference to the current window may be obtained at any time after it has been created:

```
window * old_window = current_window();
```

Once a reference to a window has been saved, that window may be made current again at any time, for example:

`select(`*window_one*`);`

Selecting a window only means that it is the window that will be used for all subsequent graphical operations, it does not change the *focus*. Focus is handled by the Windows operating system; whenever a keyboard key is typed or a mouse is clicked, the input is given to whichever window has focus. The `make_console_active` function switches focus to the console so that keyboard input can be handled in the normal way. If your program is to perform its own keyboard processing, focus must be switched to a graphics window. If this is done, the basic `read_`... functions and `cin` will not work.

`make_window_active(`*old_window*`);`

The `make_window_active` function transfers focus to a graphics window. This is what happens automatically when the user clicks within that window.