CP Lab 4: Simple graphics with `descartes`

Note

We know that quite a few of you have been having some difficulty with Labs 2 and 3. If that's you, then please don't try to do this lab now (in the lab sessions Thu 12 Oct, Mon 16 Oct and Tue 17 Oct), but instead continue to work on Labs 2 and 3.

The schedule for the course has a catch-up week following this one, to give you more time to complete Labs 2–4.

If you are getting through the labs happily, we will provide you with some extra entertainment in the 'catch-up' week labs.

Instructions

The purpose of this Lab is to introduce you to simple graphics programming with the `descartes` graphics resource. You will get extra programing experience with *conditional statements* (the `if`-statement) and with *iteration* or *loops* (`for` and `while`). We will be building on what you have done in the previous lab and in your classes.

All Labs for CP will take place on the DICE machines in AT5.05 (Thursday and Tuesday) or AT6.06 (Monday). For working alone, you may use any of the DICE labs in AT, as long as they are not booked out by a class. You have 24-hour access to AT levels 3,4,5: if your card/pin does not allow you in, use the ITO contact form on our course webpage to contact the ITO.

You must complete all lab sheets in order to learn C programming effectively (and have a good chance of passing the exam in December). Lab sheets not finished during the lab must be finished in your own time.

Aims

- Show you how to work with a library which is not a standard library supported by DICE (the `descartes` library).
- Get experience of working with *types* different to (and more interesting than) the in-built types of C.
- Introduce you to the `descartes` environment for basic graphics, and show you the various functions declared in `descartes.h`
- Offer you the option of following a `descartes` tutorial to write a simple program `onepoint.c` to accept one point and print the details of that point.
- Get you to write three programs based on `descartes`.
- Give you extra experience with the `if`-statement, and with *iteration* (`for` and `while`).

Stage A Getting Started

- As usual we start by bringing up a terminal window (you have plenty of experience doing this from labs 1, 2 and 3). Once you have a terminal up, return to your ‘home directory’ using

  ```
  cd↲
  ```

  and then create a new directory for this lab called `lab4`:

  ```
  mkdir lab4↲
  ```

  You may then move into this new directory using `cd`:

  ```
  cd lab4↲
  ```

- In this lab, you will be working with the `descartes` graphics functions. There is a ‘header file’ `descartes.h` and ‘executable’ `descartes.o` for these functions, and an example program `sqDraw.c` on the course webpage:

  `http://www.inf.ed.ac.uk/teaching/courses/cp/descartes.html`

  Please download these files (and the example file `sqDraw.c`), and save them *inside* the `lab4` directory. If you are working in the labs (not on your own machine), you can instead of downloading them just copy them directly by:

  ```
  cp /group/teaching/cp/Descartes/* .↲
  ```

  (the ‘`*`’ means ‘all files’, and ‘`.`’ is the current directory.

- This lab has associated template files. You will need to copy these files into your directory. Go to the course webpage again, and scroll down to find the timetable entry for Lab 4. Click on the ‘lab4 files’ link, and download each of the three files to your `lab4` directory.

  Alternatively, just copy them directly:

  ```
  cp /group/teaching/cp/lab4/* .↲
  ```

Stage B descartes tutorial

This is a tutorial on `descartes`. If you are confident, you can just read through this, have a look at the example program `sqDraw.c`, and go on to Stage C, Stage D and Stage E straight away. If you would like a more detailed introduction, please work through this tutorial on your machine.

Be careful: in this tutorial, we will *look at* code that *we* have written, but which you should not write yourself; and we will show you examples of code *you* might write. In the past, students have often confused these. We will try to keep these distinct by italicizing and indenting *our* code:

```
    /* This is a piece of our code - you should never be writing
       this yourself */
```

as opposed to

```
/* This is code that shows you how to write something */
```

About descartes

descartes is a *library* which extends the C language environment with *types* and *functions* for graphics, just as stdio provides types and functions for input/output. Unlike stdio, the descartes library is provided by us, not built in to the standard system.

The types and functions of descartes deal with simple geometric objects such as points and line segments, which can then be drawn on the screen. Points on the screen (in your program's graphics window) are described by specifying the x- and y-coordinates, both integers.

The descartes.h header file provides a type point_t for representing points, and functions GetPoint, XCoord and YCoord, for performing certain operations involving points. We have made the convention, that the names of types will start with a lower case letter and end with _t, while functions start with an upper case letter.[1]

If you look at the descartes.h file, you will see that it starts with three declarations each starting typedef struct. You don't yet know what this means, *but you don't need to know!*[2] All you need to know is that there are types point_t, lineSegment_t and rectangle_t.

After those, you will see *prototypes* (declarations of functions – see the end of lecture 7) for the functions provided by descartes. Each prototype tells you the type of the arguments and result, and is preceded by a short comment explaining what the function does.

For example,

```
/*
 * Waits until the user clicks the mouse,
 * then returns the point that the cursor is indicating.
 */
point_t GetPoint();
```

tells you about a function which you could you use like this:

```
point_t mypoint; /* declare a variable holding a point */
mypoint = GetPoint(); /* set it to where the user clicks */
```

Note that because you don't (officially) know what's inside a point_t, you have to use the provided functions to build points and look inside them. For example:

```
/*
 * Returns the y-coordinate of the point given as argument.
 */
int YCoord(point_t p);
```

is the function you use to find the y-coordinate of a point, which you could use like this:

```
if ( YCoord(mypoint) > 250 ) {
  printf("mypoint is in the top of the window\n");
}
```

[1]Consistency is important in a program. Type conventions provide a consistent way of naming things and so make a program easier to read. There is no correct type convention, though the one we use is quite common. Note that the system libraries follow a different convention, as printf demonstrates.

[2]In a modern language, you would not be able to see these type definitions – you don't 'need to know', so you don't get to know. This is the principle of 'abstraction', or 'information hiding'. C predates the wide development of that idea, so it doesn't let us hide information (other than by playing dirty tricks).

Opening and closing the graphics window `descartes` uses a small (500×500) window on the screen for you to draw in. Every `descartes` program starts by calling a function to initialize the graphics, like this:

```
OpenGraphics();
```

This will do some internal housekeeping, and create the drawing window on the screen. At the end of your program, just before the `return`, you should call

```
CloseGraphics();
```

which will wait for the user to right-click, and then close the graphics window. (Waiting for the user to right-click lets them inspect what you've drawn!)

Using points Points are represented with the `point_t` type. You can create a point either by reading one from the user with `GetPoint`, or by creating one directly from its coordinates with `Point`, like this:

```
point_t p;
p = Point(100,200); /* point with x coord 100 and y coord 200 */
```

You can retrieve the *x*- and *y*-coordinates with `XCoord` and `YCoord` as already illustrated.

Lines and rectangles `descartes` also has types for lines and rectangles. We will introduce these during the main part of the practical.

Compiling `descartes` programs To use `descartes`, you have to include the library header file in your program. This should be done after the standard headers, like this:

```
#include <stdlib.h>
#include <stdio.h>
#include "descartes.h"
```

Note the use of '`" "`' instead of '`< >`'. This tells `gcc` to look for the header file in the current directory, instead of in the standard system libraries.

You also have to give some additional arguments to `gcc`. If your program is `onepoint.c` (as below), you compile it like this:

```
gcc -Wall onepoint.c descartes.o -lSDL -lm↵
```

Here, `descartes.o` is the *object file*, which you copied into the current directory, and which is where the compiled `descartes` code is. `descartes` uses a system graphics library, so you have to add `-lSDL`; and it also uses the maths library, which is `-lm`.

Accepting one point

In this tutorial, you will write a tiny program called `onepoint.c` which accepts a single point *p*, indicated by the user clicking the left mouse button while the cursor is within the *graphics window*, and outputs the *x* and *y* coordinates of the point entered.

You may wish to copy your template file from previous labs, for example by doing

```
cp ../lab2/template.c onepoint.c↵
```

or you can just write the new program from scratch. Either way, open it for editing via emacs:

```
emacs onepoint.c &↲
```

Then follow this tutorial, editing `onepoint.c` as you go.

For our simple task, what needs to be done should be clear: call `GetPoint` to capture the point indicated by the user, then apply the functions `XCoord` and `YCoord` to extract the *x*- and *y*-coordinates of the point, and finally use `printf` to write those coordinates to the screen. It is necessary to store the point between the time it is captured using `GetPoint`, and processed by `XCoord` and `YCoord`. For this purpose, we introduce a variable `p` by way of the declaration

```
    point_t p;
```

First make sure your program has the standard headers and the `descartes.h` header, and the `main` routine as usual. Then add this declaration, together with two function calls opening the graphics package at the beginning and closing it at the end.

The variable `p` can be initialised using the *assignment* statement

```
p = GetPoint();
```

The meaning of this is: evaluate the expression to the right of the equals-sign (in this case, call the function `GetPoint`), and assign the result (in this case, the point indicated by the cursor at the instant the mouse is clicked) to the variable on the left of the equals-sign (in this case `p`).

So far, we have read the point in, and stored it in the variable `p`. Now we just need to write out the coordinates of the point to the screen. This can be done with the `printf` function, which we have already been using throughout the course. For example, executing the statement

```
printf("The x-coordinate of that point is %d.\n", XCoord(p));
```

will print the *x*-coordinate of `p` to the terminal window.

Add a `printf` statement to your program to output a message of the form

```
You clicked at the point (x, y).
```

to the terminal window, where *x* and *y* are the coordinates of the point clicked on, namely the values of `XCoord(p)` and `YCoord(p)`.

Save your program within `emacs`, and compile and run it as described above:

```
gcc -Wall onepoint.c descartes.o -lSDL -lm↲
./a.out↲
```

This will start up the graphics window, and you can enter a point with a left-click of the mouse. (To close the graphics window and terminate the program, press the right mouse button in response to the prompt.)

Drawing a square

For examples of other parts of `descartes`, see the example program `sqDraw.c`: compile and run it.

Tutorial outcome

- ☐ I understand the basic types and functions of `descartes`.
- ☐ I understand how to ‘link’ to `descartes.o` to run my graphics programs.

Stage C Line Segment

In Stage C, we construct a program that allows the user to specify two points using the mouse, draws the line segment joining those points, and computes the length of the segment.

The program that you write for Stage C will be written in `segment.c`. You have been given the basic template for this program.

The 'grand plan' for `segment.c` will be:

(i) Obtain a point (say p) from the user;

(ii) Obtain a second point (say q) from the user;

(iii) Form the line segment pq with end-points p and q;

(iv) Draw the line segment on the drawing pad;

(v) Finally (iv) output the length of pq.

If you need any help with Steps (i) or (ii), please look back at the details of Stage B: you just need to declare two variables p and q of type `point_t` and initialise them one at a time using `GetPoint`. We now skip onto (iii)–(iv), by looking at some extra functions declared in the `descartes.h` file:

```
/*
 * Creates a line segment with given endpoints.
 */
lineSeg_t LineSeg(point_t p1, point_t p2);

/*
 * Returns the length of a line segment.
 */
double Length(lineSeg_t l);

/*
 * Draws a line segment.
 */
void DrawLineSeg(lineSeg_t l);
```

From this we see that Steps (iii) and (iv) are straightforward, given the functions `LineSeg` (which takes two points and returns the line segment joining those points), and `DrawLineSeg` (which takes a line segment and displays it). We just need to declare a new variable pq of type `lineSeg_t`, initialise it using the assignment

```
pq = LineSeg(p, q);
```

and then display it using a call to the function `DrawLineSeg`. (Do this now!)

Finally we use the function `Length` within a `printf` statement to print out a message of the form

```
The length of the line segment is 29.529646.
```

or whatever. Remember that the `printf` specification for `doubles` is `%lf` (or just `%f`).

Compile and run your program:

```
gcc -Wall segment.c descartes.o -lSDL -lm↵
./a.out↵
```

This will load up the terminal window, and you can enter your two points with left-clicks of the mouse. (To close the graphics window and terminate the program, press the right mouse button in response to the prompt.)

When you are happy that you have entirely debugged your program, make a copy of `a.out` for later use, by renaming it to `segment` using the Unix/Linux `mv` command.

Finally, **submit** your program as usual, using
`submit cp lab4 segment.c↵`

`segment.c` results

- ☐ Program compiles using `gcc`?

  If not, please read back over the instructions, and check all files are in the right place.
- ☐ Program accepts two points and draws the segment.
- ☐ The 'length' part is working?

  A good way to check this is by drawing a long vertical segment. The side length of the window is 500 units.
- ☐ Source file submitted?

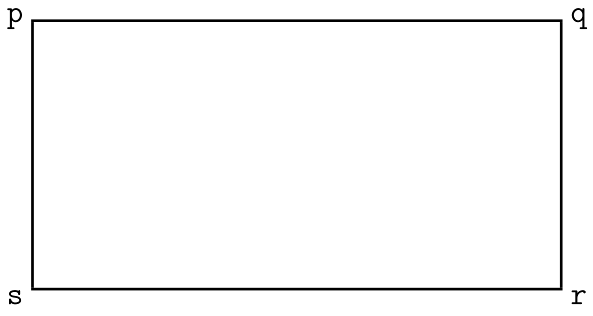
Stage D Drawing a rectangle

The topics explored for this task are *expressions* and *conditional execution* (the `if`-statement).

The goal is to write a program that allows the user to enter a rectangle using the mouse, computes the length of the diagonal of the rectangle, and classifies the rectangle as *tall*, *wide* or *almost square*.

Start this task by opening file `rectangle.c` (supplied to you) with `emacs`.

A rectangle has four vertices say `p`, `q`, `r` and `s` of type `point_t`. These vertices define four edges, which may be represented by four variables, say `pq`, `qr`, `rs` and `sp` of type `lineSeg_t`. Even though a rectangle has four vertices, we can define the rectangle just by giving two opposing vertices. For example, consider the following rectangle. If we know vertices `p` and `r`, then we can calculate `q` and `s`.



For `rectangle.c`, the user will specify the vertices `p` and `r` by positioning the mouse and clicking within the graphics window, once for either vertex. The program must accept these two clicks, draw the implied rectangle, calculate its diagonal length, and classify it (as Tall, Wide, or Almost Square). For the 'drawing' part, you should add the following declarations and code to the skeleton of `rectangle.c`:

1. Add variable declarations for the 4 vertices and edges of the rectangle.
2. Read in vertices `p` and `r` using two calls to `GetPoint`.
3. Add two assignment statements computing vertices `q` and `s`. Notice that the *x*-coordinate of vertex `q` is the same as that of vertex `r`, and its *y*-coordinate is the same as that of vertex `p`. Using this fact, and functions `XCoord`, `YCoord` and `Point`, we can compute `q` (and also compute `s`).
4. Add four assignment statements to compute the four edges of the rectangle.
5. Draw the rectangle, using four calls to `DrawLineSeg`.

Some of the work required here is similar to Stage C. It is a good idea to compile your program at this point (see below for details) and check that it is doing the drawing.

Our next task is to determine the length of the diagonal of the rectangle we have just drawn. This can be done using the `Length` function with the correct two points. Notice that the diagonal length is not, in general, a whole number, so we must use a double variable to represent it. Then add a `printf` statement that outputs the length of the diagonal to the terminal window, for example:

```
The diagonal of the rectangle has length 88.025681.
```

At this stage you might like to compile and run the program again to test it.

The final stage will need you to use the `if`-statement. The goal is to classify the rectangle that has been input as *tall*, *wide* or *almost square*. We define a rectangle as *tall* if its height is at least 25% greater than its width, *wide* if its width is at least 25% greater than its height, and *almost square* otherwise. That is, a rectangle is *tall* precisely when $h \geq 1.25 \times w$, *wide* precisely when $w \geq 1.25 \times h$ and *almost square* otherwise. Note that the equivalent C expressions will look something like:

```
h >= 1.25 * w
```

Add an `if`-statement to your program which prints an appropriate message, depending on which of the three categories the rectangle belongs. Please look back on slides for Lectures 4 and 5 if you want to revise your knowledge of the `if`-statement.

Compile and run your program as usual with

```
gcc -Wall rectangle.c descartes.o -lSDL -lm↵
./a.out↵
```

from the Terminal prompt. This will load up the terminal window, and you can enter your two points with a left-clicks of the mouse. (To close the graphics window and terminate the program, press the right mouse button in response to the prompt.)

When you are happy that you have entirely debugged your program, make a copy of `a.out` for later use, by renaming it to `rectangle` using the Unix/Linux `mv` command.

Finally, **submit** your program with
```
submit cp lab4 rectangle.c↵
```

`rectangle.c` **results**

☐ Program compiles using `gcc`?

If not, please read back over the instructions, and check all files are in the right place.

☐ Program is drawing the rectangle properly, after being given only 2 points?

- ☐ The 'diagonal length' part is working?
- ☐ The classification of rectangle type is working?

  Try a few tests of this - it can take delicate tests to be sure the Almost Square option is exactly correct.
- ☐ Source file submitted?

Stage E Drawing a polygon

The final stage of this lab is intended to give you another chance to work with *iteration*, via the `while`-loop.

Your mission is to write a program called `polygon.c` which allows the user to input an arbitrary number of clicks on the graphics window, to draw (one at a time) the 'sides' of the polygon represented by that series of clicks, and then – after a middle-click is given (to indicate the user is finished) – to output the total length of the perimeter of the polygon.

Using the mouse, the user indicates the vertices of the polygon, in the order in which they occur on the perimeter. The final vertex is indicated by clicking the *middle* mouse button. From the program's point of view, this action returns a point with *negative* coordinates, which is interpreted as a terminator for the input.

Since there is no a priori bound on the number of sides the polygon will have, we are forced to use a *loop*, in this instance, the `while`-loop. The basic structure of the loop is provided for you. Figure 1 gives a preview of what will appear when you open `polygon.c` with `emacs`.

The idea is that the body of the `while`-loop will be executed once for each edge of the polygon (except the final edge which closes it), which enables us to process each edge of the polygon in turn. The key variables `curr` and `prev`, of type `point_t`, hold the 'current' and 'previous' vertices; the edge being processed is the one which joins `prev` to `curr`.

You must add a call to `DrawLineSeg` within the body of the loop, to draw the edge currently being processed. This will results in displaying all edges except the last (the one which closes the polygon by connecting the final vertex back to the first). Add the necessary code *after* the loop to draw the final edge, and test that the program works (see below).

Aside from displaying the polygon in the graphics window, we are required to compute the total length of the perimeter. To do this we introduce a floating point variable, say `cumulativeLength` (of type `double`), that accumulates the sum of the lengths of the edges as they are read in. Naturally, this variable should be initialised to zero, which can be done at the point of declaration thus:

```
    double cumulativeLength = 0.0;
```

Now add an assignment statement to the body of the `while`-loop, to update the variable `cumulativeLength` appropriately. Noting that we have still to take account of the final edge, add a further assignment statement after the `while`-loop to increase `cumulativeLength` by the length of that final edge. Finally add a `printf` statement to print out a message informing the user of the length of the perimeter of the polygon.

Compile and run your program as usual with (within the `lab4` directory)

```
gcc -Wall polygon.c descartes.o -lSDL -lm↵
./a.out↵
```

This will load up the terminal window, and you can enter your two points with a left-clicks of the mouse. (To close the graphics window and terminate the program, press the right mouse button in response to the prompt.)

When you are happy that you have entirely debugged your program, make a copy of `a.out` for later use, by renaming it to `polygon` using the Unix/Linux `mv` command.

Finally, **submit** your program with

```
submit cp lab4 polygon.c↲
```

`polygon.c` results

- ☐ Program compiles using `gcc`?

  If not, please read back over the instructions, and check all files are in the right place.
- ☐ Program is drawing the polygon properly?

  Even joining the final point to the first one?
- ☐ The ‘circumference length’ part is working?
- ☐ Source code submitted?

Julian Bradfield after *Mary Cryan and many previous CP1 lecturers, Oct 2016*

```
#include <stdlib.h>
#include <stdio.h>
#include "descartes.h"

int main(void)
{
   point_t   curr,  /* Current point */
             prev,  /* Previous point */
             init;  /* Initial point */
   lineSeg_t l;     /* Line segment joining prev and curr */

   OpenGraphics();
   prev = GetPoint();
   curr = GetPoint();
   init = prev;   /* this stores the first vertex in the
                     variable "init": we need to remember it,
                     to be able to join it to the last vertex
                     when it is entered */

   while (XCoord(curr) >= 0) {
      /*
       * Process the current edge.
       * The current edge joins the previous
       * vertex "prev" to the current vertex "curr".
       */
      prev = curr;
      curr = GetPoint();
   }

   /*
    * Now tackle the last edge - remember, the first vertex
    * is stored in the variable "init"
    */

   CloseGraphics();
   return EXIT_SUCCESS;
}
```

Figure 1: The skeleton program for `polygon.c`

```
/* Opens and initialises the graphics window */

void OpenGraphics(void);

/* Closes the graphics window */

void CloseGraphics(void);

/* Waits until the user clicks the mouse,
 * then returns the point that the user is indicating.*/

point_t GetPoint(void);

/* Creates a point with given coordinates. */

point_t Point(int x, int y);

/* Returns the x-coordinate of the point given as argument. */

int XCoord(point_t p);

/* Returns the y-coordinate of the point given as argument. */

int YCoord(point_t p);

/* Creates a line segment with given endpoints. */

lineSeg_t LineSeg(point_t p1, point_t p2);

/* Returns one endpoint of a line segment... */

point_t InitialPoint(lineSeg_t l);

/* ... returns the other endpoint. */

point_t FinalPoint(lineSeg_t l);

/* Returns the length of a line segment. */

double Length(lineSeg_t l);

/* Draws a line segment. */

void DrawLineSeg(lineSeg_t l);
```

Figure 2: `descartes` functions used in this Lab