

Getting started in C

This is an assessed practical in four parts, A–D. Each part of the practical guides you through the construction of a short program. To obtain credit for a part, you must submit the program electronically via the `submit` command from *your own* DICE account.

The deadline for completion and electronic submission of Practical Exercise 1 is 11am, Tuesday 20th October (week 5). In the absence of evidence of medical or other difficulties, work submitted after the deadline will attract zero credit.

Aims

This practical exercise aims to:

- develop further your skill in programming on the DICE system, and familiarise you with the “edit-compile-run” cycle;
- introduce you to some important programming concepts, such as *type*, *variable*, *expression* and *conditional execution*;
- and, above all, instill confidence by guiding you through the process of writing small programs.

Assessment

The maximum credit that can be obtained for this practical is 20 marks (out of a total of 100 for the whole of the total coursework mark for the entire course). Each of the following 4 tasks gains **5 marks**.

- Imperial-to-Metric distance converter.* Write a program that prompts the user for a number of miles, and also a number of yards, and outputs the equivalent distance in kilometres to the screen.
- Segment-Length Calculation.* Write a program which interacts with a basic graphics display to read in two points from the plane, and then returns the length of the segment connecting those points.
- Rectangle Statistics Program.* Write a program that interacts with a basic graphics display to read two points from the plane, then draws the implied rectangle, computes the length of its diagonal, and classifies its shape.
- Polygon Perimeter Calculator.* Write a program which takes a sequence of points from an interactive graphics tool and computes the perimeter of the polygon defined by those points.

Try to complete all four parts, keeping in mind that this will bring improved programming skills (useful later!) as well as the marks for practical 1. But don't spend *excessive* time on the practical - if you find yourself getting hugely stuck on some part, look for help (in the first instance, from the 'InfBase' demonstrators on level 5 in the evenings; or alternatively from your lecturer or tutor).

Credit is given on the basis of the *electronically submitted* programs (and note we *always* require *source code*, ie, the .c files).

If you don't submit something, you get no credit for it.

Notes

- Read through this document *before* you reach the keyboard, and work out in advance what you need to do. This document gives a significant amount of help with your tasks.
- It is perfectly acceptable to discuss the coursework specification with your classmates, to ask for help with understanding the exercise, or to ask for help with debugging. It is **never** acceptable to directly copy programs or fragments of programs from others.
- If you are genuinely stuck, seek help. These early exercises are more concerned with acquisition of skills than with testing your problem-solving ability.
- Please use `int` variables when appropriate, and use `float` variables for representing fractional numbers.

Electronic Submission

In this and future practicals, you need to *submit* files for marking. The procedure uses an electronic submission system which records the dates on which you submit the various items and saves a permanent record of your work for the external examiner at the end of the year. This saves on paper and prevents your valuable work from getting lost - *note that if you submit a file more than once, we only grade the final copy.*

The general form of the `submit` command for this practical is

```
submit cs1 cp1 P1 <file>
```

where *<file>* is the file to be submitted (`convert.c`, `segment.c`, `rectangle.c` or `polygon.c`). You should issue this command in the shell window, having moved to the directory which contains the file you want to submit.

Preparation

This practical has associated template files. You will need to copy these files into your directory so that you can modify them. First, make sure you have a directory to hold your practical work; Practical 0 gave information on how to do this.

To copy the templates, make sure you are in the directory that you have created for this practical. Then issue the following command from inside a console (or ‘terminal window’):

```
cp /group/teaching/cp1/Prac1/* ./
```

The first argument given to `cp (/group/.teaching/cp1/Prac1/*)` is the location we copy *from*, and the second argument (`./`) is the location we copy to - the current directory. The “*” at the end of the first argument is a ‘wild-card’ meaning ‘every file’ (in that directory).

Part A

Open the file `convert.c` in the `emacs` editor. The file only has a few lines in it, and you will need to fill in some basic things apart from writing the program. In Part A you must write a program which prompts the user to first input some (integer) number of miles, then some (integer) number of yards, and calculates the equivalent number of kilometres for this distance. In doing this, you need certain facts:

The number of *yards-per-mile* is 1760.0.

The number of *kilometres-per-mile* is 1.609.

The first thing to do in writing any program is getting the basic structure (header files, `main` etc) of the program correct. Therefore as a first step, we consider the simplest ANSI C program imaginable:

```
#include <stdlib.h>

int main(void)
{
    return EXIT_SUCCESS;
}
```

Figure 1: The simplest C program

The program in Figure 1 is an ideal starting set of statements for `convert.c` (or any program). It takes care (in the most basic possible way) of the three initial issues for every C-program:

- **Libraries:** One design feature of the C programming language is that some very common operations, such as performing input and output, are not supported directly by primitives in the language, but rather broken down into simpler operations and stored as “functions” in an external “library.” Even the most basic program will need to use some of these externally defined functions. The `#include` “directive” at the first line of the program makes available the many standard functions from the library `stdlib`.
- **main** Every C program must contain a function `main`. This is where execution of the program begins. We haven’t discussed functions much at this point. Later in the course we will see C programs that contain many functions; however, the very simple programs we shall encounter in the current exercise contain only a single function `main`. The part contained in curly-parentheses `{` and `}` is the body of the `main`.
- **return:** As discussed in class, every function *must* complete with a `return` statement on every execution. For the `main` function in Figure 1, the body consists of a `return` statement and nothing else! The value returned is the special value `EXIT_SUCCESS`, indicating that the program has terminated successfully.

To start work on `convert.c`, type the program of Figure 1 into the `emacs` buffer `convert.c` and save it. Compile the program by issuing the command `make convert` in the shell window.¹ If your typing was accurate, the compilation will be successful, otherwise you will need to correct the program using the editor and try again. Once you are successful, the compiled version of the program will appear in a new file named simply `convert`. (If you type `ls`, the file may appear highlighted in green; this signifies that the file is “executable”.) Now run (execute) the resulting compiled program by typing `./convert` in the shell window. The program runs successfully... but nothing will be output. This is because the program does not do anything at the moment!

We can make the program slightly more interesting by taking one step in the direction of completing our programming task. Clearly, if we want to take in a number-of-miles from the user of our program, we will need to have a `printf`-statement (asking for number-of-miles) and a `scanf`-statement (to grab the number that the user types). Here is a strand of code which asks the user for the number of miles and reads that number into a variable called `miles`:

```
printf("Input the number of miles: ");
scanf("%d", &miles);
```

¹Note that this is a different way of compiling than I have explained in class. In typing `make convert`, you are making use of a “makefile” which has been set up to make working easier. It provides the more complicated commands required to compile programs B, C, D in conjunction with the graphics primitives on the system. It also has the advantage of keeping the 4 executable files stored under individual names (rather than as `a.out`).

`printf` is an output function we saw in class. It is a flexible function that can be used to output many kinds of values: here it is used to output a string, i.e., a sequence of characters. We don't end the string with a `\n` (the “newline character”) because we want the user to type the number to the right of the sentence. `scanf` is an input function we have seen in class. It also can read various kinds of values, according to the character used after the `%`. Note that to *read into* a variable, we must use `&miles` (where `miles` is the variable name) rather than just `miles` (which is what would be used for output with `printf`). In inserting this fragment into our “shell” for `convert.c`, we have two issues:

- To use `printf` and `scanf` we need to include the `<stdio.h>` library (The `stdio` library contains various items concerned with input and output.)
- We have used a variable named `miles` (presumably an integer variable), but never defined it.

Therefore to include the code-fragment above to our basic program, we will also need to include `<stdio.h>` and will have to define `miles`, in the following way:

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

int main(void)
{
    int miles;

    printf("Input the number of miles: ");
    scanf("%d", &miles);
    return EXIT_SUCCESS;
}
```

Figure 2: Reading part of the input for `convert.c`

Now try updating your program `convert.c` to the one shown in Figure 2.

Notice that all our statements are terminated with semi-colons.² Ensure that your indentation is even within the body of the function `main`; as the program becomes longer, it will be important to format it carefully to elucidate the structure. Now compile (by typing `make convert`) and run (using the command `./convert.c`) the program. On this run, you should see the request for input appear on the screen, and should be able to input an integer to the program!

However, there is still a lot of work to be done. The task of converting miles and yards to kilometres is a generalized version of the “marathon” program that we saw (and discussed) in the lecture on Thursday, 1st October. The difference is

²If you are familiar with Pascal, note that semicolon is used as a terminator rather than as a separator when working with C.

```
[rydell]mcryan: ./convert
Input the number of miles: 5
Input the number of yards: 0

5 miles and 0 yards equals 8.045000 kilometres
```

Figure 3: Example run of a (correct) `convert.c` implementation.

that instead of working with an exact distance (the marathon distance of 26 miles and 385 yards), we allow the user to present any number of miles and of yards to the program, and require the equivalent number of kilometres to be computed. Because the number of miles and yards may vary (being given as input), we must define **variables** to store these values. In Figure 2 we have given an example of the definition of the `miles` variable and of its use as a place to store the inputted number-of-miles. The following steps remain to be done, to complete Task A:

- (i) A variable (of type `int`) for the number of yards needs to be defined, and a code fragment written to request-and-read-in the yards from the screen (as was done for “miles”);
- (ii) We need a `float` variable to store the number of kilometres (once it has been computed), and we must output that number (once computed);
- (iii) As was done for the basic “marathon” program of 1st October 09, we should define *global constants* (which sit above the `main` function) to represent the number of yards-per-miles, and number of kilometres-per-fractional-mile. This will involve the `const float` declaration.
- (iv) The code which does the converting must be written. It should be possible to convert the “marathon” code for this.
- (v) Please take care when working with both the `int` and `float` type in the same arithmetic expression. It is best to explicitly perform *casting* using `(float)` rather than leave it to the compiler.

We conclude this section, with an example of one run of a correctly-implemented `convert.c`, as shown in Figure 3. Please use this as a comparison for your own program, both to judge correctness of your calculations, and as a guide to formatting the input and output statements.

Obtaining credit *Once you are sure that your program works, submit the file `convert.c` using the `submit` command, as explained on page 2.*

Part B

Textual input and output is just one mode of interaction with a program; another is graphical. In Part B, 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 is required to manipulate various kinds of data — numbers, points and line segments — and provides a working introduction to the important notions of *type* and *variable*.

The program that you write for Part B will be written in `segment.c`. You can compile this program using the compile command `make segment`, and run the program by typing `./segment` in the shell window. In addition to the two `#include` directives that will be familiar from Part A, we now need a third: `#include "descartes.h"`. `descartes.h` provides a small set of types and functions relating to simple geometric objects such as points and line segments. Points in the plane are described by specifying the x - and y -coordinates, both integers.

As a first step, let us write a program that takes as input a single point p — indicated by the user clicking the left mouse button while the cursor is within the *CP1 graphics* window — and outputs the x - and y -coordinates of p . The `descartes.h` library provides a type `point_t` for representing points, and functions `GetPoint`, `XCoord` and `YCoord`, for performing certain operations involving points. By convention, the names of types start with a lower case letter and end with `_t`, while functions start with an upper case letter.³

The purpose and mode of use of the functions `GetPoint`, `XCoord` and `YCoord` may be gleaned from file `descartes.h`, an extract of which is presented in Figure 4. For each function, there is a comment, enclosed in `/*` and `*/`, that states informally what the function is intended to do, and a one-line “prototype” that indicates the arguments of the function and its result. Thus the function `XCoord` takes as argument a point `p` (of type `point_t`) and produces as result the x -coordinate of `p` (of type `int`). The specification of the function `YCoord` can be read off in an analogous way. The function `GetPoint` takes no argument but returns a point (of type `point_t`). The function waits for the user to position and click the left button of the mouse, and returns the point indicated by the cursor, which appears as an arrow in the graphics window.

In outline, what needs to be done should be reasonably 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;
```

³Consistency 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 libraries follow a different convention, as `printf` demonstrates.

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

/*
 * Creates a point with given coordinates.
 */
point_t Point(int a, int b);

/*
 * 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);
```

Figure 4: Extract from `descartes.h`

The skeleton program already contains 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 thought of as a box which is capable of containing a value of type `point_t`. It may 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`). Add the above assignment statement to the program immediately after the function call that opens the graphics window. (The position is marked by a comment, which you will need to remove.) Note that the parentheses in `GetPoint()` are essential, even though `GetPoint` does not take an argument.

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 accomplished using the `printf` function, which we have already encountered. An example should give the general idea of how it can be used in this context. Executing the statement

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

will cause the message

```
The x-coordinate of that point is 172.
```

to appear in the shell window (assuming the x -coordinate of p is indeed 172). Roughly, the effect of this statement is to cause the *control string*

```
"The x-coordinate of that point is %d.\n"
```

to be written to the shell window. However, when a *conversion specification* such as `%d` is encountered, a value specified by one of the subsequent arguments is substituted—in this instance there is just one, namely `XCoord(p)`. There are many possible conversion specifications; in this instance, `%d` indicates that a decimal (whole) number is to be written. A control string may contain many conversion specifications, each corresponding to a separate argument, the order of the control specifications matching that of the arguments.

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

```
You clicked at the point (172, 87).
```

to the shell window, assuming `XCoord(p)` is 172 and `YCoord(p)` is 87. Test (i.e., compile and run) your program to make sure that what you have written so far works. Recall that compiling is done by typing `make segment` and running by typing `./segment`. (To close the graphics window and terminate the program, press the right mouse button in response to the prompt.)

Returning to the grand plan, we still need to (i) obtain a second point (say q) from the user, (ii) form the line segment pq with end-points p and q , (iii) draw the line segment on the drawing pad, and finally (iv) output the length of pq . Step (i) is just a repeat of what we have already done, so let's get it out of the way immediately: introduce a new variable q of type `point_t`, initialise it using `GetPoint`, and write out its coordinates using `printf`.

Referring again to `descartes.h` (see Figure 5) we see that Steps (ii) and (iii) 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. Thus 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 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. There is one additional technical fact you need to know. The result of the function `Length` is a fractional, or “floating-point” number (of type `float`) and not a whole number (of type `int`). The conversion specification appropriate to floating-point numbers is `%f`.

```

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

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

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

```

Figure 5: A further extract from `descartes.h`

Obtaining credit When you are convinced that your program is functioning correctly, submit the file `segment.c` using `submit`, as explained on page 2.

Things to consider (a) Roughly, what is the unit of length for x - and y -coordinates when points are shown on the drawing pad? Experiment to determine this quantity. (b) Suppose we had tried to avoid using a variable `p` by writing:

```
printf(" ... ", XCoord(GetPoint()), YCoord(GetPoint()))
```

Would this code have the correct effect? If not, why not.

Part C

The topics explored in this part are *expressions* and *conditional execution*. 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*.

Set up Part C by opening file `rectangle.c`. You can compile this program using the compile command `make rectangle` and run the program by typing `./rectangle` in the shell window.

A quadrilateral is defined by its four vertices, which we may represent by four variables, 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`. Naturally, the edge `pq` joins vertices `p` and `q`, and so on. By giving sensible and consistent names to variables we make the program easier both to write and understand.



Figure 6: A rectangle

A rectangle is a particular type of quadrilateral. Assuming the edges of a rectangle are parallel to the sides of the computer screen, we can define a rectangle by any two diagonally opposing vertices. For example, consider the rectangle in Figure 6. If we know vertices **p** and **r**, then we can calculate **q** and **s**.

The idea is that the user will specify the vertices **p** and **r** by positioning the mouse and clicking within the graphics window, once for either vertex. Starting with the skeleton program provided:

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**.

Most of the work required here is similar to Part B, and should not cause too much trouble. Test your program to make sure that everything so far works.

Our next task is to determine the length of the diagonal of the rectangle we have just drawn. This can be done by translating the formula from Pythagoras's Theorem into C code,⁴ but there is an easier way; can you see it? Notice that the diagonal is not, in general, a whole number, so we must use a floating-point

⁴If you go this way, note that x^2 can be written $x \times x$. There is a library function called **sqrt** for computing square roots. In order to be able to use this, add a fifth directive, **#include <math.h>** to the four which are already in place. The function **sqrt** returns an approximation to the square root of its argument, so that, for example, **sqrt(2.0)** is some floating point number close to 1.414.

variable to represent it. Now add a `printf` statement that outputs the length of the diagonal to the shell window, for example:

```
The diagonal of the rectangle has length 88.02568.
```

At this stage you might like to compile and run the program to make sure that what you have written so far is working correctly.

The final stage introduces the notion of *conditional execution* in the form of the `if`-statement. The goal is to classify the rectangle that has been input as *tall*, *wide* or *almost square*. As a warm-up, we'll attempt the simpler task of distinguishing between *tall* and *wide*. Provisionally, let's say that a rectangle is *tall* if its height is greater than its width. Assuming `h` (respectively `w`) is an integer variable containing the height (respectively width) of the rectangle, this effect can be achieved by an `if`-statement of the form:

```
if (h > w) {
    /* Write a suitable message to the Program I/O window. */
} else {
    /* Write a suitable message to the Program I/O window. */
}
```

Observe that the condition

```
h > w
```

is enclosed in parentheses; don't omit these, as they are required by the *syntax* of C. In general, the condition governing an `if`-statement is an expression that can be interpreted as having value **true** or **false**. The simplest such expressions are constructed by comparing two arithmetic expressions using one of the *relational operators* of C, such as `<=`, `<`, `>`, `==`; the meaning of these should be fairly clear, but note that equality is denoted by `==`, to avoid confusion with assignment `=`.

```
if (/* Condition */) {
    /* Action 1 */
} else {
    /* Action 2 */
}
```

To decide between more than two alternatives, we may *nest* the construction as suggested in Figure 7. The meaning of this nested conditional is: Test *Condition 1*, *Condition 2*, and so on in turn, until the first true condition, say *Condition i*, is found; then execute *Action i*. If all the conditions are false, execute the *Default* action. Note that precisely one *Action* is performed, whatever the circumstances.

Now 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 $h > 1.25 \times w$, *wide* precisely when $w > 1.25 \times h$ and *almost square* otherwise. Note that the equivalent C expressions will look something like:

```

if (/* Condition 1 */) {
    /* Action 1 */
} else if (/* Condition 2 */) {
    /* Action 2 */
}
    /* ... and so on until ... */{

} else {
    /* Default action */
}

```

Figure 7: A nested if-statement

```
h > 1.25 * w
```

Add a nested `if`-statement to your program which prints an appropriate message, depending on which of the three categories the rectangle belongs.

Obtaining credit When your program seems to be working correctly, submit the file `rectangle.c` using the `submit` command, as described on page 2.

Things to consider (a) How should test examples be chosen to rigorously test the program?

Part D

The final part of the practical is intended to illustrate *repetitive execution*, as embodied in the `while`-loop. The program you are required to write should allow the user to input a polygon with an arbitrary number of sides, and then 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 some *iterative statement*, in this instance, the `while`-loop. At this stage, you are not expected to construct a `while`-loop from scratch, so the basic structure of the loop is provided for you. Figure 8 gives a preview of what will appear when you open `polygon.c`. You can compile the program by using the compile command `make polygon` and run the program by typing `./polygon` in the shell window.

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`.

It is important to appreciate that in order to complete the program it is not necessary to understand exactly *how* the variables `curr` and `prev` acquire the claimed values. All the essential information is encapsulated in the comment. This illustrates an important point. It is not possible to conceive at one instant the entire solution to some complex task, and hence it is essential to break that complex task into simple tasks that can be tackled in isolation. If these simple tasks are to have truly independent existences, it is crucial that each has a simple and well-defined interface to the others. The comment in the loop is an attempt to specify such an interface. Of course, the current exercise is sufficiently simple that it *can* be held in the mind all at one time, but we shall soon encounter exercises for which this is not so.

It is possible to compile and run the program as it stands; this would allow the user to input the polygon using the graphics window, but there would be little point, as there would be no visible effect. Add a call to `DrawLineSeg` within the body of the loop, to draw the edge currently being processed. This will display all edges except the last, which closes the polygon by connecting the 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.

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 `float`), 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:

```
float 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.

Obtaining credit *After testing the program, submit the file `polygon.c` using the `submit` command, as described on page 2.*

Things to consider (a) What are suitable examples for testing? How can we verify that the answer given by the program is (approximately) correct? (The experiments following Part B may be useful here.) (b) After performing any finite sequence of experiments, can we be *certain* that the program is performing correctly?

```
#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 8: The skeleton program for Part D