CSCI 2132: Software Development

Lab 9: Make and GDB

Synopsis

In this lab, you will:

- Learn to compile a multi-file program using `make`.
- Learn to use the `gdb` debugger to find bugs in your code.

Contents

Overview

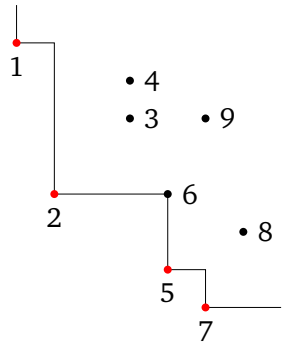
In this lab, you will use the `make` tool to make compiling programs consisting of multiple source files much easier. You will use such a program with some bugs in it to practice using `gdb` to locate these bugs.

The Story: Computing Skylines

The narrative that will guide today's lab is that we want to compute the skyline of a 2-dimensional point set. As a motivating example, consider that there is a rock concert in town and you want to help concert-goers to choose the hotel they will stay at after the concert. Our concert-goers are not very demanding in terms of the services the hotel offers, but given that the concert tickets were rather pricey, they do not want to spend all that much money on the hotel, and they also want to be able to get back to the hotel quickly after the concert, so distance from the concert venue also matters.

For each candidate hotel you may want to offer to the concert-goers, you know its distance from the concert venue and the nightly rate for a standard room. Now, if hotel A is closer to the conert venue than hotel B *and* hotel A is also cheaper than hotel B, then none of the concert-goers will be interested in booking hotel B; hotel B is simply not a reasonable choice. However, if hotel A is closer to the concern venue than hotel B but hotel B is cheaper than hotel A, then you cannot easily decide which hotel a given concert-goer will prefer: low price may be more important than distance for some while others may prefer to pay a little more if this allows them to stay closer to the concert.

If you represent each hotel H as a point $p_H = (x_H, y_H)$, where x_H is the distance from the concert venue and y_H is the price, a point p_A is said to *dominate* another point p_B if both $x_A \geq x_B$ and $y_A \geq y_B$. In this case, hotel A is of no interest to the concert-goers. Given a set of candidate hotels, you want to select the set of hotels that are not ruled out by this condition. This set of candidate hotels is the *skyline* of the point set corresponding to all hotels, where the skyline of a point set P is the set of points in P that do not dominate any other point in P. An example is shown in the following figure. The skyline points are red. Non-skyline points are black. It is easily checked that no skyline point dominates any other point and each non-skyline point dominates at least one skyline point.



Here is a simple algorithm to compute the skyline of a point set: First sort the points by increasing x-coordinates; break ties by increasing y-coordinates. Then add the first point in the sorted sequence to the skyline and store its y-coordinate in a variable y_{min}. For every subsequent point $p = (x, y)$, add p to the skyline if $y < y_{\text{min}}$; otherwise, do not add it to the skyline. If you add p to the skyline, also update $y_{\text{min}} = y$. You should follow this strategy on the figure above to verify that this does indeed compute the skyline of the point set (the red points). For your convenience, the points are numbered in the order the algorithm inspects them.

Step 1: Organizing the code

You should implement the function for computing the skyline of a point set as a library function that can also be used by other programs that need to solve this problem. Thus, you place your skyline implementation into a file `skyline.c` and declare a prototype of the skyline function in a file `skyline.h` that can be included by any program that wants to use your skyline function. You also write a test program that can be used to test the correctness of the skyline function. You put this test code into a file `test_skyline.c`. Finally, the point representation you use is something that could be useful in completely different programs that work with point sets, so you should implement the point representation in its own header file `point.h`. In summary, you will have the following four files. Create these files with the exact content shown here. Do not correct any mistakes in them that may be obvious to you; these are the mistakes you will find using gdb later in this lab.

point.h

```
#ifndef POINT_H
#define POINT_H

typedef struct {
    float x, y;
} point_t;

#endif // POINT_H
```

skyline.h

```
#ifndef SKYLINE_H
#define SKYLINE_H

#include <point.h>

// Compute the skyline of a point set.
//
// Input:
// - points: pointer to an array of points
// - n:      pointer to an integer storing the number of points
//
// Output:
// - The integer referenced by n stores the number of points in the skyline.
// - The points in the points array are rearranged so that
//   - The skyline points are the first n points,
//   - The skyline points are sorted by their x-coordinates,
//   - The non-skyline points appear from position n onwards, in no particular
//     order.
void compute_skyline(point_t *points, size_t *n);

#endif // SKYLINE_H
```

In the following implementation file `skyline.c`, the `#include <skyline.h>` statement is technically not needed. However, it is good practice to always include the header file in the corresponding implementation file because it allows the compiler to check that the declaration of each function in the header file matches its definition in the implementation file.

skyline.c

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

int xy_cmp(const void *p, const void *q) {
    const point_t *pp = p; const point_t *qq = q;
    if (pp->x == qq->x) return (pp->y == qq->y) ? 0 : ((pp->y > qq->y) ? 1 : -1);
    else                return                        (pp->x > qq->x) ? 1 : -1;
}

void point_swap(point_t *p, point_t *q) {
    point_t tmp = *p; *p = *q; *q = tmp;
}

void compute_skyline(point_t *points, size_t *n) {
    size_t m;
    float y_min;
    qsort(points, *n, sizeof(float), xy_cmp);
    y_min = points[0].y;
    for (int i = m = 1; i < *n - 1; ++i) {
        if (points[i].y < y_min) {
            y_min = points[i].y;
            point_swap(points + i, points + m++);
        }
    }
    *n = m;
}
```

Your test code will not perform any verification of its input. That's okay because you use it only as a test harness, not in a production environment.

test_skyline.c (1/2)

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

int main(int argc, char **argv) {
    int result   = 0;
    size_t nin   = 0; size_t nout   = 0;
    size_t incap = 8; size_t outcap = 8;
    point_t *input  = malloc(incap  * sizeof(point_t));
    point_t *output = malloc(outcap * sizeof(point_t));
    point_t p;
    if (argc != 3) exit(2);
    FILE *in  = fopen(argv[1], "r");
    FILE *out = fopen(argv[2], "r");
    if (in && out) {
        while (fscanf(in, "%f%f", &p.x, &p.y) == 2) {
            if (nin == incap) {
                incap *= 2;
                input = realloc(input, incap * sizeof(point_t));
            }
            input[nin++] = p;
        }
        while (fscanf(out, "%f%f", &p.x, &p.y) == 2) {
            if (nout == outcap) {
                outcap *= 2;
                output = realloc(output, outcap * sizeof(point_t));
            }
            output[nout++] = p;
        }
        compute_skyline(input, &nin);
        if (nin == nout) {
            for (int i = 0; i < nin; ++i) {
                if (input[i].x != output[i].x || input[i].y != output[i].y) {
                    fprintf(stderr, "Wrong outptu point at position %d: "
                                    "expected (%f, %f), found (%f, %f)\n",
                            i, output[i].x, output[i].y, input[i].x, input[i].y);
                    result = 1;
                }
            }
        } else {
            fprintf(stderr, "Wrong output size: expected %lu, found %lu\n",
                    nout, nin);
            result = 1;
        }
    }
```

test_skyline.c (2/2)

```
    if (in)  fclose(in);
    else     result = 2;
    if (out) fclose(out);
    else     result = 2;
    free(input);
    free(output);
    return result;
}
```

Step 2: Using `make` to build the code

This project is still small enough that you could recompile all the files in it as needed to rebuild the program every time you change the C code. With bigger projects, this gets tedious. In addition, one advantage of splitting the input into multiple source files is that you do not need to recompile the whole source when changing only one file. You only need to recompile the files that have changed or depend on a changed file. A `Makefile` allows you to specify these dependencies using build rules. Each such rule specifies

Target: the file to be produced using this rule.
Prerequisite(s): the files that are needed as input to this rule to produce the target.
Recipe: the steps to be executed to produce the target from the prerequisite(s).

The `make` tool reads the `Makefile` and decides which targets to recompile and in which order, based on the following rules:

- If the target of a rule is older than any of the prerequisites of the rule, the rule is applied.
- If a prerequisite of a rule is the target of another rule, then that other rule is applied first to produce the target if necessary.

In your project, you have the following dependencies. The output program `test_skyline` is produced from the compiled object files `test_skyline.o` and `skyline.o`. As a `Makefile` rule this looks like this:

```
test_skyline: test_skyline.o skyline.o
	gcc -o $@ $^
```

Pay attention: the indentation of the recipe `gcc ...` must be achieved using a tab character, not using spaces!

For this rule, `test_skyline` is the target, `test_skyline.o` and `skyline.o` are the prerequisites, and `gcc -o $@ $^` is the recipe. You could have written `gcc -o test_skyline test_skyline.o skyline.o` as the recipe. However, if you ever change the target or sources of the rule, you would then also have to change the recipe. Instead, you use two special variables available in every `Makefile` recipe: `$@` always expands to the target of the current rule, `$^` expands to the list of prerequisites of the current rule. There is a third special variable, `$<`, which expands to only the first prerequisite. You will use this one soon.

Now, the `skyline.o` file needed by this rule is produced by compiling `skyline.c`. Are there any other files that may require you to recompile `skyline.c`? Since `skyline.c` includes both `point.h` and `skyline.h`, it is wise to recompile it every time one of the three files `skyline.c`, `point.h` or `skyline.h` changes. You express this using the following rule:

```
skyline.o: skyline.c skyline.h point.h
	gcc -c $<
```

This says that `skyline.o` can be produced by compiling `skyline.c`. (Remember, `$<` refers only to the first prerequisite.) However, by listing `skyline.c`, `skyline.h`, and `point.h` as prerequisites, `skyline.o` is recreated every time one of these three files is changed.

`test_skyline.o` has similar dependencies:

```
test_skyline.o: test_skyline.c skyline.h point.h
	gcc -c $<
```

This gives your first `Makefile` for your project:

Makefile

```
test_skyline: test_skyline.o skyline.o
	gcc -o $@ $^

skyline.o: skyline.c skyline.h point.h
	gcc -c $<

test_skyline.o: test_skyline.c skyline.h point.h
	gcc -c $<
```

You can now (try to) build the program using

```
$ make test_skyline
gcc -c test_skyline.c
test_skyline.c:3:21: fatal error: skyline.h: No such file or directory
 #include <skyline.h>
                     ^
compilation terminated.
make: *** [test_skyline.o] Error 1
```

This failed because you have not told gcc where to find `skyline.h`. You will fix this in the next step.

Step 3: A more comprehensive `Makefile`

`make` and `gcc` are designed to work very well together. For example, `gcc` can automatically generate the set of prerequisites that each target depends on and then this list of dependencies can be included in the `Makefile`. This is where the true power and convenience of `Makefiles` start to become apparent, but it is also well beyond the scope of an introductory course such as CSCI 2132. You are encouraged to search the web for collected wisdom of advanced usage of `make`. Here, you will at least explore some more common patterns often used in `Makefiles`.

First, observe that the rules for `skyline.o` and `test_skyline.o` are essentially identical: the recipe is the same and each rule produces a file `XXX.o` from a file `XXX.c` and files `skyline.h` and `point.h`. Make's pattern syntax allows you to replace these with a single rule:

```
%.o: %.c skyline.h point.h
        gcc -c $<
```

Next, you make sure that `gcc` uses the right flags to compile your code correctly. The two rules for `test_skyline` and `%.o` both use `gcc` but in different roles. The `test_skyline` rule uses only the linking phase of `gcc` because its input consists only of already compiled object files. The behaviour of the linker can be controlled using command line flags. More complex `Makefiles` often have many rules in them, so it is good practice to defined these flags as a variable and then use this variable in every rule that uses `gcc` as a linker. The link flags are commonly stored in a variable `LDFLAGS`, but this is only convention; any name is fine. For now, modify your `Makefile` as follows:

`Makefile`

```
LDFLAGS=

test_skyline: test_skyline.o skyline.o
        gcc $(LDFLAGS) -o $@ $^

%.o: %.c skyline.h point.h
        gcc -c $<
```

Currently, there are no special `LDFLAGS`, but this will change soon.

Similarly, the preprocessor and the compilation stage can be influenced by their own flags. It is common to store these in a variable `CFLAGS`. Here, you need to fix two problems: Currently, the preprocessor does not find the `skyline.h` file because it is not stored in a "system location". You can add to the search path of header files using the `-I` flag. Here you want to add the current directory, using the option "`-I.`". The other flag you need to set chooses the C standard. You want C99 here, so you can define loop variables in a `for`-loop. This gives the following updated `Makefile`:

Makefile

```
CFLAGS=-std=c99 -I.
LDFLAGS=

test_skyline: test_skyline.o skyline.o
        gcc $(LDFLAGS) -o $@ $^

%.o: %.c skyline.h point.h
        gcc $(CFLAGS) -c $<
```

Finally, you want to add a few convenience targets. The first two, `all` and `clean` are fairly common. All of these are what `make` calls "phony targets". The corresponding rule is invoked every time you build this target no matter whether a file with the same name exists and is newer than the rule's prerequisites. To declare a target phony, you use the following syntax:

```
.PHONY: all clean test1 test2 test3
```

The `all` target is usually the first target in a `Makefile` to ensure that `make` and `make all` behave the same and simply build all real targets in the `Makefile`. Here, you have only one real target to build, namely `test_skyline`, so you add this as a dependency for `all` and do not specify any recipe for the rule because there is nothing to be done to build the `all` target; the only point of the rule is to ensure that `all`'s dependencies are built if they are out of date:

```
all: test_skyline
```

The `clean` target deletes all files that are not source files. A common pattern to obtain a "clean build" by recompiling everything from scratch is to use `make clean && make all`. The rule for the `clean` target is

```
RM=rm -f
OBJFILES=test_skyline.o skyline.o

clean:
        $(RM) test_skyline $(OBJFILES)
```

The use of `rm -f` is important here because it prevents `rm` from failing if the file to be deleted does not exist. By storing the object files to be deleted in a variable `OBJFILES`, you can use the same variable to specify the dependencies of `test_skyline`:

```
test_skyline: $(OBJFILES)
        gcc $(LDFLAGS) -o $@ $^
```

Finally, you create three more targets `test1`, `test2`, and `test3` so you can run your program on three standard test instances using `make test[123]`:

```
test1: test_skyline input1.txt output1.txt
	./test_skyline input1.txt output1.txt

test2: test_skyline input2.txt output2.txt
	./test_skyline input2.txt output2.txt

test3: test_skyline input3.txt output3.txt
	./test_skyline input3.txt output3.txt
```

Your final Makefile should look like this now:

Makefile

```
CFLAGS=-std=c99 -I.
LDFLAGS=

RM=rm -f
OBJFILES=test_skyline.o skyline.o

.PHONY: all clean test1 test2 test3

all: test_skyline

clean:
	$(RM) test_skyline $(OBJFILES)

test1: test_skyline input1.txt output1.txt
	./test_skyline input1.txt output1.txt

test2: test_skyline input2.txt output2.txt
	./test_skyline input2.txt output2.txt

test3: test_skyline input3.txt output3.txt
	./test_skyline input3.txt output3.txt

test_skyline: $(OBJFILES)
	gcc $(LDFLAGS) -o $@ $^

%.o: %.c skyline.h point.h
	gcc $(CFLAGS) -c $<
```

Try to run

```
$ make all
gcc -std=c99 -I. -c test_skyline.c
gcc -std=c99 -I. -c skyline.c
gcc  -o test_skyline test_skyline.o skyline.o
$ make all
make: Nothing to be done for `all'.
$ touch skyline.c
$ make all
gcc -std=c99 -I. -c skyline.c
gcc  -o test_skyline test_skyline.o skyline.o
$ make clean
rm -f test_skyline test_skyline.o skyline.o
$ make all
gcc -std=c99 -I. -c test_skyline.c
gcc -std=c99 -I. -c skyline.c
gcc  -o test_skyline test_skyline.o skyline.o
```

and observe how the second `make all` does not recompile anything because all files are up-to-date. The third `make all` compiles `skyline.c` and then produces `test_skyline` but does not compile `test_skyline.c`. The reason is that `touch skyline.c` gave `skyline.c` a new time stamp, so it is now newer than `skyline.o` and `make` correctly triggers the rule to build `skyline.o` from `skyline.c`, which in turn triggers the rule to rebuild `test_skyline` from `test_skyline.o` and `skyline.o` because `skyline.o` is now newer than `test_skyline`. `make clean` deletes all the object files, so the fourth and final `make all` has to rebuild all files `skyline.o`, `test_skyline.o`, and `test_skyline` from scratch.

`make` helpfully prints the sequence of steps it performs, which can be very useful when debugging `Makefiles`. This output can be suprressed by running `make -s` (silent).

Step 4: Test your code

First create some test instances. The six input and output files referenced in your `Makefile` rules in Step 3 should look like this:

`input1.txt`

```
1.0 8.0
2.0 7.0
4.0 5.0
7.0 2.0
8.0 1.0
```

`output1.txt`

```
1.0 8.0
2.0 7.0
4.0 5.0
7.0 2.0
8.0 1.0
```

`input2.txt`

```
3.0 4.0
3.0 2.0
1.0 3.0
1.0 2.0
8.0 7.0
```

`output2.txt`

```
1.0 2.0
```

`input3.txt`

```
7.0 1.0
2.0 8.0
7.0 2.0
3.0 5.0
3.0 6.0
4.0 9.0
```

`input3.txt`

```
2.0 8.0
3.0 5.0
7.0 1.0
```

Create these files and then run your first test:

```
$ make test1
./test_skyline input1.txt output1.txt
Wrong output size: expected 5, found 1
make: *** [test1] Error 1
```

Hmm, this did not go as expected. The computed skyline has only one point but the expected output has 5 points.

Step 5: Fix your code

Next, you will practice using gdb to find the error. To this end, you have to prepare your code appropriately. In particular, you have to ensure that the compiler includes source code information in the program it produces. The `-g` flag does this if it is provided to both the compiler and the linker. gdb uses this source code information to help you interact with your code (e.g., refer to specific lines in the source code). Edit your `Makefile` so the first two lines look like this:

```
CFLAGS=-std=c99 -I. -g
LDFLAGS=-g
```

This ensures that both the compiler and the linker are run with the `-g` flag. Now recompile your code:

```
$ make clean && make all
rm -f test_skyline test_skyline.o skyline.o
gcc -std=c99 -I. -g -c test_skyline.c
gcc -std=c99 -I. -g -c skyline.c
gcc -g -o test_skyline test_skyline.o skyline.o
```

Now you should normally systematically search for the spot in your program where things start to go wrong. To narrow your search, you consult the oracle this time and she tells you that the `compute_skyline` function is the one that's misbehaving. So, the first thing you do is inspect this function's output. You could use `printf`-style debugging as discussed in class. Here, you will practice using a debugger, gdb. Load your program into gdb using

```
$ EDITOR=emacs
$ gdb test_skyline
```

The `EDITOR=...` line tells gdb (and other programs started from the shell) that you want to use `emacs` as your editor whenever an editor is needed. This will become important later because you will edit and recompile your code without quitting gdb.

As a first exercise, run your program from within gdb. The command to do this is

```
(gdb) run input1.txt output1.txt
```

`(gdb)` is the prompt gdb presents to you, in order to distinguish this from a standard shell prompt. The command essentially says: Run whichever program was loaded either as a command line argument when starting gdb (as you did here) or using the `file` command of gdb and pass `input1.txt` and `output1.txt` as command line arguments to this program.

The output looks something like this:

```
Starting program: /users/faculty/nzeh/lab9/test_skyline input1.txt output1.txt
Wrong output size: expected 5, found 1
[Inferior 1 (process 27178) exited with code 01]
```

This tells you that gdb started your program and with which command line arguments. Then comes the output of your program, just as when you ran it from the shell. Finally, gdb tells you that your program exited with exit code 1.

Now, to find out what's wrong with `compute_skyline`, you first want to inspect its output. To this end, you need to set a *breakpoint* after the place where `compute_skyline` is called. A breakpoint is a location in the program where the debugger stops the program every time this location is reached in the program's execution. This allows you to examine the contents of variables, execute individual steps one at a time or simply resume the execution if you are satisfied with the current state of the program.

Type

```
(gdb) list main
```

and press enter until you see the line where `compute_skyline` is called. Set a breakpoint on the next line number. If your code looks *exactly* as above, then this should be line 31:

```
(gdb) break 31
Breakpoint 1 at 0x40097c: file test_skyline.c, line 31.
```

Now start the program again:

```
(gdb) run input1.txt output1.txt
Starting program: /users/faculty/nzeh/lab9/test_skyline input1.txt output1.txt

Breakpoint 1, main (argc=3, argv=0x7fffffffe358) at test_skyline.c:31
31          if (nin == nout) {
```

This tells yout hat the program execution was interrupted at breakpoint 1, in line 31 of `test_skyline.c`, which contains the code `if (nin == nout) {`. Now you can inspect the contents of variables:

```
(gdb) print nin
$1 = 1
(gdb) print input[0]
$2 = {x = 1, y = 2}
```

`$1`, `$2`, `...` refer to the values you inspect, numbering them in order. So, for some reason, the size of the computed skyline is 1, even though you expected all five input points to be in the skyline. More worrisome is the fact that the one point in the computed skyline is the point $(1, 2)$, which is not part of the input at all!

So get ready to investigate where things go wrong. You set a breakpoint on the call to `compute_skyline` itself:

```
(gdb) break 30
Breakpoint 2 at 0x400969: file test_skyline.c, line 30.
```

and then restart the program:

```
(gdb) run input1.txt output1.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /users/faculty/nzeh/lab9/test_skyline input1.txt output1.txt

Breakpoint 2, main (argc=3, argv=0x7fffffffe358) at test_skyline.c:30
30          compute_skyline(input, &nin);
```

As a sanity check, inspect that the input looks as expected:

```
(gdb) print nin
$3 = 5
(gdb) print input[0]
$4 = {x = 1, y = 8}
(gdb) print input[1]
$5 = {x = 2, y = 7}
(gdb) print input[2]
$6 = {x = 4, y = 5}
(gdb) print input[3]
$7 = {x = 7, y = 2}
(gdb) print input[4]
$8 = {x = 8, y = 1}
```

All is in order.

In order to inspect where `compute_skyline` misbehaves, step into the function using `step`:

```
(gdb) step
compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:17
17      qsort(points, *n, sizeof(float), xy_cmp);
```

This tells you that you are now inside the `compute_skyline` function at line 17 of the `skyline.c` file and you are about to call `qsort`. You *do not* want to step into `qsort` because you can trust that its implementation is correct (and you have no source code information available for it anyway). You want to advance to the line *after* calling `qsort` and inspect the data at this point. You do this using `step`'s sibling `next`. The difference is that `step` steps *into* the function call if the current instruction is a function call. That is, it advances to the first instruction inside the function as above, where you stepped into the `compute_skyline` function from the `main` function. `next` steps *over* the function call, that is, it advances to the first instruction after the function call returns. This is what you want here:

```
(gdb) next
18      y_min = points[0].y;
```

As desired, you are now stopped at the next line after the call to `qsort`. Check again that the data is still intact. It was sorted by x-coordinates before, so `qsort` should not have done anything to it.

```
(gdb) print *n
$9 = 5
(gdb) print points[0]
$10 = {x = 1, y = 2}
(gdb) print points[1]
$11 = {x = 4, y = 7}
(gdb) print points[2]
$12 = {x = 8, y = 5}
(gdb) print points[3]
$13 = {x = 7, y = 2}
(gdb) print points[4]
$14 = {x = 8, y = 1}
```

You still have 5 points but the first three points are not in the original input. Something got messed up and it is the call to `qsort` that misbehaves, so have a closer look. Of course, you see the mistake right away. You tell `qsort` that the element size is `sizeof(float)` but the array elements are points, so `qsort`'s pointer arithmetic to access the elements to be sorted cannot work correctly.

Start `emacs` from within `gdb` by running

```
(gdb) edit
```

This conveniently opens `emacs` on the line where the program is currently stopped. Change the line before it, which is the `qsort` invocation, to look like this:

```
    qsort(points, *n, sizeof(point_t), xy_cmp);
```

Then exit `emacs`, which puts you back at the `gdb` prompt and rebuild your program:

```
(gdb) make all
gcc -std=c99 -I. -g -c skyline.c
gcc -g -o test_skyline test_skyline.o skyline.o
```

For now, assume that this fixed all your problems (it didn't). Delete the breakpoint at line 30 in order to run the program until after the call to `compute_skyline`.

To delete a breakpoint, you need to refer to it by its number. If you do not remember these numbers, you can get a list of all current breakpoints using

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040097c in main at test_skyline.c:31
2       breakpoint     keep y   0x0000000000400969 in main at test_skyline.c:30
```

Okay, so this was breakpoint 2. Delete it using `delete`:

```
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040097c in main at test_skyline.c:31
```

Only breakpoint 1 is left. Now restart the program:

```
(gdb) run input1.txt output1.txt
```

Again, answer `y` to the question whether you want to restart the program from the beginning. The program once again stops on line 31. Inspect the output now:

```
Breakpoint 1, main (argc=3, argv=0x7fffffffe358) at test_skyline.c:31
31         if (nin == nout) {
(gdb) print nin
$15 = 4
```

Hmm, that's better, but you have 5 input points and they should all be part of the skyline for this input.

Bring back the breakpoint on line 30, restart the program and step into the call to `compute_skyline`:

```
(gdb) break 30
Breakpoint 3 at 0x400969: file test_skyline.c, line 30.
(gdb) run input1.txt output1.txt
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /users/faculty/nzeh/lab9/test_skyline input1.txt output1.txt

Breakpoint 3, main (argc=3, argv=0x7fffffffe358) at test_skyline.c:30
30         compute_skyline(input, &nin);
(gdb) step
compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:17
17     qsort(points, *n, sizeof(point_t), xy_cmp);
```

Step over `qsort` again and check that your fix of `qsort` actually worked, something you should have done before:

```
(gdb) next
18     y_min = points[0].y;
(gdb) print *n
$16 = 5
(gdb) print points[0]
$17 = {x = 1, y = 8}
(gdb) print points[1]
$18 = {x = 2, y = 7}
(gdb) print points[2]
$19 = {x = 4, y = 5}
(gdb) print points[3]
$20 = {x = 7, y = 2}
(gdb) print points[4]
$21 = {x = 8, y = 1}
```

So far, so good.

So, given that you have the correct point set before the main loop of `compute_skyline`, something goes wrong in the loop, at the very least with the manipulation of `m`. To investigate this, set three breakpoints, one directly on the `for`-expression, another on the line that updates `y_min`, and a third on the line that updates `*n` right after the loop. If you entered the code in Step 1 exactly, they are in lines 19, 21, and 25 of the file `skyline.c`:

```
(gdb) break 19
Breakpoint 4 at 0x400c4d: file skyline.c, line 19.
(gdb) break 21
Breakpoint 5 at 0x400c81: file skyline.c, line 21.
(gdb) break 25
Breakpoint 6 at 0x400cf3: file skyline.c, line 25.
```

By observing the order in which the program hits these breakpoints and the values of various variables at each breakpoint, you will be able to figure out what goes wrong.

Continue the execution until you stop at the next breakpoint:

```
Continuing.

Breakpoint 4, compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:19
19     for (int i = m = 1; i < *n - 1; ++i) {
```

Now you want to know the values of `i`, `m`, `y_min`, and the current point `point[i]` at each breakpoint. Typing `print` to do so will become tedious. Use `display` instead, which ensures that these values are shown every time the program execution stops.

```
(gdb) display i
1: i = 0
(gdb) display m
2: m = 140737354129744
(gdb) display y_min
3: y_min = 8
(gdb) display points[i]
4: points[i] = {x = 1, y = 8}
```

You can force the re-display of all of these “auto-display” values at any time by simply typing

```
(gdb) display
4: points[i] = {x = 1, y = 8}
3: y_min = 8
2: m = 140737354129744
1: i = 0
```

Now step through the next few breakpoints:

```
(gdb) cont
Continuing.

Breakpoint 5, compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:21
21              y_min = points[i].y;
4: points[i] = {x = 2, y = 7}
3: y_min = 8
2: m = 1
1: i = 1
(gdb)
Continuing.

Breakpoint 5, compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:21
21              y_min = points[i].y;
4: points[i] = {x = 4, y = 5}
3: y_min = 7
2: m = 2
1: i = 2
```

```
(gdb)
Continuing.

Breakpoint 5, compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:21
21              y_min = points[i].y;
4: points[i] = {x = 7, y = 2}
3: y_min = 5
2: m = 3
1: i = 3
(gdb)
Continuing.

Breakpoint 6, compute_skyline (points=0x603010, n=0x7fffffffe220) at skyline.c:25
25      *n = m;
3: y_min = 2
2: m = 4
```

As you can see, just pressing `Enter` after the first `cont(inue)` is the same as entering `cont(inue)` again. This is handy. Sadly, you did not hit the breakpoint on line 19 again. The reason is that `break 19` sets the breakpoint on the first machine instruction in this line, which is the initialization `int i = m = 1`, and this one is executed only once. In order to set the breakpoint on the comparison `i < *n - 1`, which would have been helpful to inspect, you would have had to inspect the assembly code of this line and set the breakpoint on the address of a specific machine instruction. `gdb` lets you do this, but this is quite a bit beyond this introductory lab. The above sequence of breakpoints still tells you quite a bit. Somehow, the `for`-loop has only 3 iterations, not 4 as expected. A closer look of course reveals the next error. The comparison should say `i < *n` or `i <= *n - 1`. Both are correct but their combination `i < *n - 1` ensures that you skip the last point. Use `gdb`'s `edit` and `make` commands to fix the problem.

Congratulations, you now have an (almost) correct skyline function. To verify this, exit `gdb`:

```
(gdb) quit
A debugging session is active.

Inferior 1 [process 19827] will be killed.

Quit anyway? (y or n) y
```

and run all three tests:

```
$ make test1
./test_skyline input1.txt output1.txt
nzeh@bluenose:~/lab9$ make test2
./test_skyline input2.txt output2.txt
nzeh@bluenose:~/lab9$ make test3
./test_skyline input3.txt output3.txt
```

No output is good. This means that the programs do not report any errors and `make` did not detect any non-zero exit code.

Step 6: Another subtle bug

I just said that the program is *almost* correct. Test one more corner case. Create two empty files `input4.txt` and `output4.txt`. You can use `touch` to do so, provided these files do not exist already:

```
$ touch input4.txt output4.txt
```

Add the following rule to your `Makefile`:

```
test4: test_skyline input4.txt output4.txt
./test_skyline input4.txt output4.txt
```

Now run

```
$ make test4
./test_skyline input4.txt output4.txt
Wrong output size: expected 0, found 1
make: *** [test4] Error 1
```

Hmm, you expected no point in the skyline (because there are no points in the input) but you somehow computed a skyline with one point in it.

Use `gdb` to hunt down the error and correct it. When you are done, run all four tests. You should see the output

```
$ make test1
./test_skyline input1.txt output1.txt
nzeh@bluenose:~/lab9$ make test2
./test_skyline input2.txt output2.txt
nzeh@bluenose:~/lab9$ make test3
./test_skyline input3.txt output3.txt
nzeh@bluenose:~/lab9$ make test4
./test_skyline input4.txt output4.txt
```

that is, all tests should pass.

Step 7: Commit your work

Commit your work to SVN. The files you should commit are

```
lab9
├─ Makefile
├─ point.h
├─ skyline.c
├─ skyline.h
├─ test_skyline.c
├─ input1.txt
├─ input2.txt
├─ input3.txt
├─ input3.txt
├─ output1.txt
├─ output2.txt
├─ output3.txt
└─ output4.txt
```