C/C++ 2016/17 assessed programming exercise 1

There are 5 points for this exercise. Each point amounts to one percent of the total module mark.

Consider the following structure definition:

```
struct N {
  struct N *x;
  struct N *y;
  struct N *z;
  long data;
};
```

We can use this structure to build various data structures. In particular, we can build *trees*, where every node has at most three children. Note that we have a tree if it is not possible to reach a node along two *different* chains of pointers. For example, the following will never be the same if `p` points to a tree:

`p->x->y->z`

`p->z->x`

More generally, we could have a *graph* rather than a tree. In this case, the same node may be reached along two different pointer chains, and a node may even loop back to itself via some pointers, so that `p` and `p->y->x->z` could be the same. Each tree is also a graph, but not conversely.

Trees are a lot easier to deal with than more general graphs. In this exercise, you can gain some points if you can handle trees, but for full marks your solution has to handle more general graphs as well, including ones that contain pointer loops.

Part 1

Write a function

```
long sum(struct N *p)
```

that computes the sum of the `data` members of the nodes reachable from the parameter `p`. Two nodes may contain the same number, but each node should only be added once.

- 1 point is given if your implementation of `sum` works correctly on trees.
- 1 more point is given if it works correctly on graphs as well.

Part 2

Write a function

```
void deallocate(struct N *p)
```

that deallocates all the nodes reachable from the parameter `p`.

- 1 point is given if your implementation of `deallocate` works correctly on trees. Valgrind should not report any memory errors or leaks.
- 2 more points are given if it works correctly on graphs as well.

You may build additional data structures provided that these also get deallocated at the end. For example, you could construct a linked list that contains pointers to each node in the graph, without duplicates.

Your solution must be submitted as a file `freegraph.c` on Canvas that contains definitions of the functions above. It may contain additional helper functions, but no main function. The code should not do any IO.

If your code fails to compile on the Linux lab machines with

```
clang -Wall -Werror
```

then 0 points are given for the whole exercise.

Testing and marking

For background on Memcheck, see

`http://valgrind.org/docs/manual/mc-manual.html`.

You will need the following files:

`http://www.cs.bham.ac.uk/~hxt/2016/c-plus-plus/freegraphmain.c`
`http://www.cs.bham.ac.uk/~hxt/2016/c-plus-plus/freegraph.h`

You may also write your own test cases to make sure your solution is correct. Your code should not perform any output, as that will be interpreted as an error by our marking script.

Compile the code by typing the following into the command line, in a directory containing all the above files:

```
clang -Werror -Wall -g -o freegraph freegraphmain.c freegraph.c
```

To see allocation and freeing, run the compiled program with valgrind using:

```
valgrind --leak-check=full ./freegraph
```

If you give valgrind the `-q` option, it should not report anything, unless there are errors:

```
valgrind -q --leak-check=full ./freegraph
```

If you want to make the exercise more challenging, you could: not use recursion, only while; or deallocate the graph without allocating any new memory and rewire the graph itself instead.