Function Pointers and RAII

The goals of this lab are to practice your C and C++ skills while working with function pointers, member functions and while applying the principle of RAII – ownership-based resource management in C++.

1 Function pointers

Write a C program that sorts an array of names. The names are implemented using the following struct:

```
struct name {
     const char * first_name;
     const char * last_name;
};
typedef struct name name;
```

The names should be first sorted alphabetically by first name and then printed.
Then the names should be sorted and printed alphabetically by last name.

Use `qsort` (https://en.cppreference.com/w/c/algorithm/qsort) for sorting, by writing a comparison function that is passed as function pointer to `qsort`, and you can use `strcmp` (https://en.cppreference.com/w/c/string/byte/strcmp) for comparing two strings.

Start from this program:

```
int main() {
     name names[4] = {
        {"Grace", "Hopper"},
        {"Dennis", "Ritchie"},
        {"Ken", "Thompson"},
        {"Bjarne", "Stroustrup"},
    };
    // sort array using qsort by first name
    // print array
    // sort array using qsort by last name
    // print array
}
```

2 Writing RAII code

Implement a buffer with RAII that can grow beyond the size initially allocated.
The buffer should be used as follows:

```
int main() {
    buffer b(3); // allocate space for 3 ints

    b.add(1);
    b.add(2);
    b.add(3);
    b.add(4); // this call must allocate new memory
}
```

Buffer should follow the RAII model, i.e. it should allocate memory in the constructor and free memory in the destructor as discussed in today's lecture. The buffer struct should be based on the following code snipped:

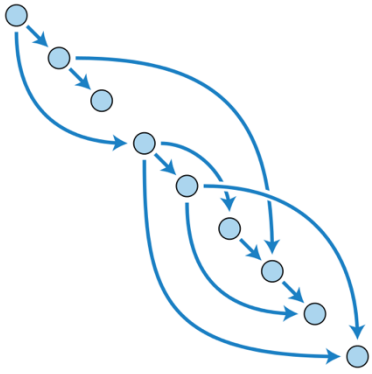
```
struct buffer {
  int * ptr;
  // … maybe store some meta data

  // constructor
  buffer(int initial_size) {
     // … allocate enough memory for the given initial_size
  }

  ~buffer() {
     // … free memory
  }

  void add(int element) {
    // check if there is enough room left.
    // If not, allocate new memory with enough space and
    // copy all elements from the old memory over.
    // Ensure to not leak memory!
  }
};
Typedef struct buffer buffer;
```

3 Using RAII



Look at the picture on the right of a **directed acyclic graph (DAG)**. In a DAG no cycles of edges are allowed, and all edges are directed. Think about how to model ownership here.

Design a `node` structure which uses `std::unique_ptr` or `std::shared_ptr` to express unique or shared ownership of a node. Only one option is appropriate there. Think about why.

A `node` should store a single value and many pointers to neighboring nodes. You can use an `std::vector` for storing the multiple pointers. When creating a `node` you should initialize the value. A `add_edge_to( node_ptr )` member function should be implemented to create the graph structure.

You can use the following main function for testing after replacing `??????` with either `unique` or `shared`.

```
int main() {
  std::??????_ptr<node> a = std::make_??????<node>("a");
  std::??????_ptr<node> b = std::make_??????<node>("b");
  std::??????_ptr<node> c = std::make_??????<node>("c");
  std::??????_ptr<node> d = std::make_??????<node>("d");
  std::??????_ptr<node> e = std::make_??????<node>("e");
  std::??????_ptr<node> f = std::make_??????<node>("f");

  a->add_edge_to(b);
  a->add_edge_to(d);
  b->add_edge_to(c);
  b->add_edge_to(d);
  c->add_edge_to(e);
  d->add_edge_to(f);
  e->add_edge_to(f);
}
```