C++ course – Exercises Set 10

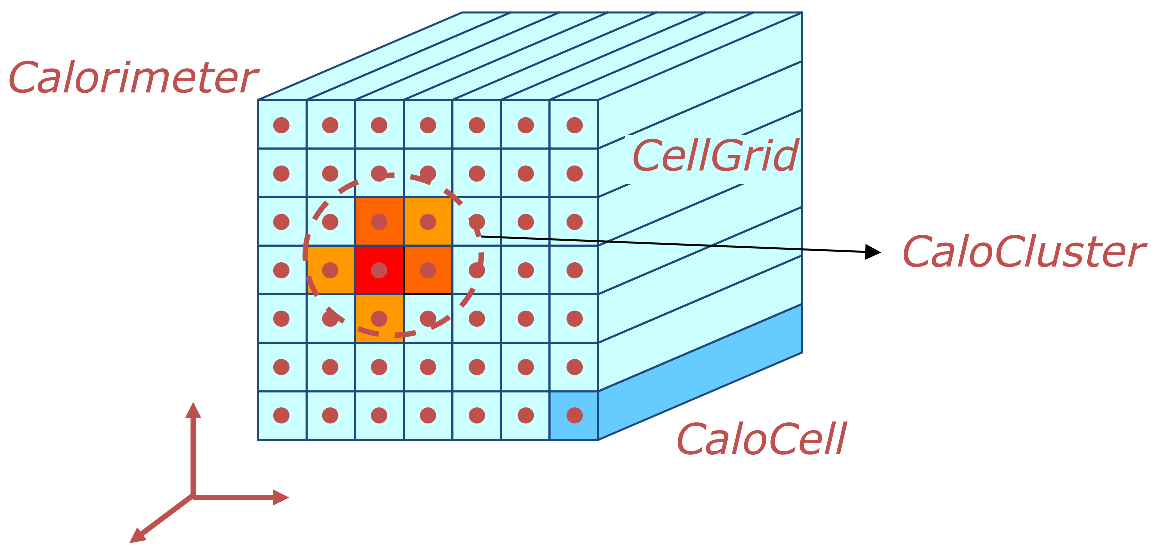
Wouter Verkerke (Jan 2021)

Exercise 10 – Putting together the Calorimeter example

The goal of this exercise is to write a program that reads data from a `Calorimeter` *from a file to a* `Calorimeter` *object and (optionally) to reconstruct the energy deposits in the calorimeter as 'clusters' of cells.*

- *The starting point for this exercise is the solution of exercise 4.1: the classes* `Calorimeter`*,* `Point`*,* `CaloGrid` *and* `CaloCell`*. These files are provided in ex10/input and will be your starting point. A final optional 4th design iteration builds a clustering algorithm from the calorimeter cells (with extra points awarded)*

- In the directory ex10/input is also a data file `calo.dat` that contains the data that you will reconstruct. The remainder of this exercise contains step-by-step instructions to accomplish this. Since this exercise is much larger than any of the previous, the approach to the final solution is broken down into a number of 'design iterations', each of which incorporates a few new features and builds on the previous cycle.



Design iteration 1 – Create dummy class `CaloReader`

- Setting up: Copy the files from ex10/input. Create a small main program in which you create a class Calorimeter and verify that everything compiles OK.

- *Iteration goal: The goal of this design iteration is to create a new class `CaloReader` that can read in the file calo.dat and that in the next design cycle will be able to create a `Calorimeter` object from the specifications in the input file*

- Create a class `CaloReader` with
    - A constructor taking a `const char*` that indicates the input file name
    - A destructor (empty for now)
    - A data member of type `Calorimeter*` named `_calo`
    - A data member of type `ifstream` named `_file`
    - Initialize the `_calo` pointer to 0 in the constructor and pass the file name in the `const char*` argument of the `CaloReader` constructor to the `_file` constructor. In this way the file will be automatically opened upon the construction of a `CaloReader` object

- Check in the constructor that the file opened OK by checking the return value of `_file.fail()`. If the file is not OK throw an exception of the type `CaloReaderException` in the constructor.
    - To be able to do that, first write a class `CaloReaderException` with a constructor that takes a `const char*` argument, a data member `std::string _what`, in which you should store the argument of the constructor, a `const char* what()` function that returns the contents of `_what` (hint: use `_what.c_str()` to obtain the `const char*` pointer to the contents of a `std::string`).
    - Now continue with the constructor of class `CaloReader`. If `_file` did not open the given file OK, throw a `CaloReaderException` object as exception and give the thrown object a descriptive error message that `main()` can print.
        - *Optional*: Write a global function `std::ostream& operator<<(std::ostream& os, const CaloReaderException& cre)` to simplify the printing of a `CaloReaderException` object

- Adapt your main program
    - Put the code that creates the `Calorimeter` object in a `try/catch` block and catch any `CaloReaderException` that may occur. If that happens print out an error message (from the exception) and terminate the program.
    - Test your program twice: once by giving `CaloReader` a file that doesn't exist and once by giving it a file that does exist.

Design iteration 2 – Read the file header, create the Calorimeter

- *Iteration goal: You will now complete the constructor of `CaloReader` by including code that reads the header portion of the file (which describes the layout of the calorimeter) and code that creates a `Calorimeter` object according to those specifications*

- Create a `Calorimeter` object
    - First have a look at `calo.dat` and look at its structure. The header portion consists of the lines between and `BEGIN_CALO_DEF` and `END_CALO_DEF`.
    - Add code to the constructor: create a `std::string` word to hold a word. Read the first word from `_file`. If it is not `BEGIN_CALO_DEF` throw an exception containing a descriptive error message (hint: you can use `if (word==“blah”))` as `std::string` implements `operator==()`.
    - Read another word from `_file`. Check that it reads “`SIZE`”. If not throw an exception (with message). Otherwise, read in two integers: `size_x` and `size_y` as the fields after the word `SIZE` give the size of the calorimeter in x and y respectively.
    - Create a `Calorimeter` object of the correct size using `new` and store the pointer in `_calo`.

- Read in the cell mapping.
    - The data describing the content of the calorimeter (later on in the file) consists of pairs of readout-ID numbers and energy values.
      To be able to use that data we must first know what the position the `CaloCell` with a given read-out ID has in the calorimeter. In other words, we need a mapping from readoutID → (int x, int y).
      This information is also stored in the header section of `calo.dat`: each line starting with `POSITION` is followed by a readoutID and the corresponding x and y position on the grid.

    - Add code to the constructor that reads in all the `POSITION` lines and modifies each corresponding cell in `_calo` to contain the readout ID that corresponds to that position:
      My suggestion is to do it as follows:
      First, read in the next word, then create a `while (word==“POSITION”)` loop that keeps looping until you have read in a word that is not “`POSITION`”. In the loop, read in `int readoutID`, `int ix` and `int iy` from `_file`. Then, get a pointer to the corresponding cell in `_calo` from `_calo->grid().cell(ix,iy)` and change the readout ID of that cell to `readoutID`. Finally, read in the next word in `word`. If it is not “`POSITION`”, the loop will terminate.
      If all is right, you are at the end of the header section of calo.dat. (Hint: check that the final word is “`END_CALO_DEF`”).

- Verify your code
    - Add code that prints the layout of readoutIDs in the calorimeter that the constructor of `CaloReader` put there.
    - First add a '`Calorimeter& calo()`' accessor to `CaloReader` that returns a reference to the `_calo` pointer in `CaloReader`.
    - Next, add a routine
      `void dumpReadoutMap(std::ostream& os = std::cout) const`
      that prints out all readout IDs on the terminal. Print the IDs in the correct layout, i.e print all ids with x=0 and y=0…ny on one line, all ids with x=1 on the next etc… Use the `setw()` manipulator to give each printed readout ID a fixed width so that the printout looks regular.
    - Add code to `main()` that dumps the readout map, e.g.

      ```
      CaloReader r("calo.dat") ;
      r.calo().dumpReadoutMap() ;
      ```

      and verify that all cells have a readout ID assigned.

Design iteration 3 – Read event data from file and print it

- *Iteration goal: Add code to `CaloReader` that reads in an event from the file into the Calorimeter object*

- Preparations:
    - First, we do some pre-work that we're going to need later.
    - Add a function `CaloCell* findCellByID(int id)` to the class `Calorimeter` that finds a calorimeter cell with a given readout ID. You can keep the function very simple: just loop over all cells and return a pointer to the cell that matches the given readout ID.
    - Add a function void `clear()` to class Calorimeter that sets the energy of every `CaloCell` object in the calorimeter to zero.

- Add function bool `readEvent()` to class `CaloReader`.
    - Check that `_file` is still OK for reading (hint: check `fail()`). If it isn't return `false`. Clear the calorimeter using `_calo->clear()` ;
    - Read in a word, check that it is "`BEGIN_EVENT`". If it isn't, return `false`. Read in the next lines: first read `word` (it should be "`ENERGY`"), then an `int readoutID`, and finally a `double energy`.
      Use `_calo->findCellByID()` to get a pointer to the cell with the given `readoutID` and change its energy to `energy`. Keep reading lines until the first word of the line is no longer "`ENERGY`" (it should be "`END_EVENT`", check it, if it isn't return `false`). If all reading went OK, return `true`.

- Approach – verify the result
    - Add a function void `dumpEvent(std::ostream& os=cout)` to class `Calorimeter` and print out the energy of each cell in the same layout as was done for function `dumpReadoutID()`. Don't print out the energy, but

do the following: if the `energy` is < 0.5 print out a “.”, if the energy is between 0.5 and 2.0, print out a “`x`”, if it is >2, print out an “`X`”.
  - Add code to `main()` that calls a `readEvent()` and prints the energy contents of the `Calorimeter`:

```
reader->readEvent() ;
reader->calo().dumpEvent() ;
```

- If you coded `readEvent()` correctly, it should return `true` if an event was read correctly, and `false` if anything went wrong. This means you should be able to read in all events in the file if you code the following in main:

```
while(reader->readEvent0()) {
   reader->calo().dumpEvent() ;
}
```

- Verify that this works as intended. If it doesn’t, fix your code.

<u>Optional</u> **Design iteration 4** – Clustering

- *The goal of this design iteration is to write an algorithm that groups adjacent cells with energy together into ‘clusters’*

- *In this ‘advanced’ exercise I will only describe what you need to do in rough detail and leave it to you to figure out the details on how to accomplish your goal.*

- Algorithm - The idea of the algorithm of cell clustering is this:
  1. Find the cell with the highest energy. This is the starting point of the cluster.
  2. Add to this one-cell cluster any neighboring cell with `energy`>0. Do this recursively: i.e. add any neighbors of neighbors (with `energy`>0) until there are no further. The net result is that all adjacent cells with `energy>0` belong to the same cluster.
  3. Find again the cell with the highest energy not yet part of a cluster – and repeat step 2.
  4. Repeat step 3. Until there are no cells with `energy`>0 left that are not part of a cluster

  The net result is that all groups of adjacent cells with `e`>0 form a cluster.

- Class `CaloReco`
  - Modify class `CaloCell` to contain an extra integer that stores the ID of the cluster it is part of. Add an accessor and modifier function as well.
  - Now write a class `CaloReco` with a constructor (that takes a `Calorimeter&` argument) and a `Calorimeter*` data member. Store the pointer to the `Calorimeter` passed in the constructor in the data member.
  - Write a member function `findSeed()` that finds the highest energy cell.

- Write a member function `findClusters()` that 1) sets the `clusterID` of all cells to zero. 2) calls `findSeed()` to locate the highest energy cell and 3) calls `growCluster()` to expand the seed to a full cluster (function `growCluster()` explained next)
- Write a member function `growCluster(int ix, int iy, int clusid)` that takes the position of the seed cell as argument and that
  1) sets the cluster ID of that cell to `clusid`,
  2) finds all adjacent cells (nominally there are 8 cells, but there can be less if you are at an corner or edge) and
  3) calls `growCluster()` on all neighbors with (`energy>0 && clusID==0`).
  The feature that this function calls itself on its neighbors encodes the recursive aspect of the clustering algorithm in an elegant way.
- Change `findClusters()` such that it makes a loop calling `findSeed()` and `growClusters()`, incrementing the `clusterID` one at a time, until you run out of seeds (i.e. there no more cells with `energy>0` that have not been assigned to a cluster yet)