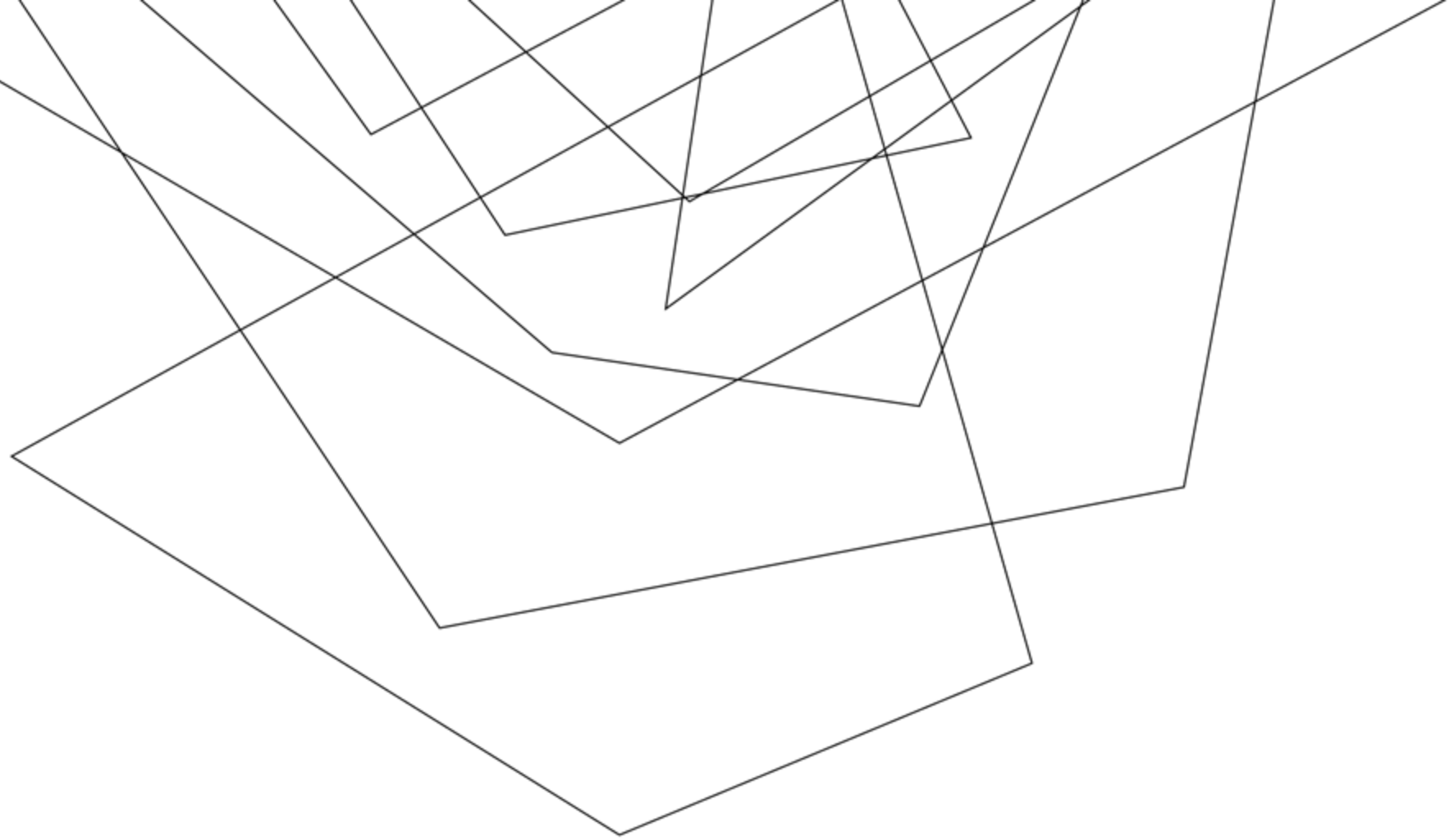

CMPT 225 D100 LAB07

TA

TOPICS FOR TODAY

Hash Table

- Basic concept (refer to lecture slides for details)
- C++ implementation & usage

Hashing

- We calculate a **special value** from the key of the data and use it as the index in an array to which we store the data, for example:
  - Course code (as string) & Term offered (as string)
  - Student ID (as string) & Contact info (as object)
- The array is called hash table
- The process of determining where in the hash table the data should be stored is called hashing

0	
1	
2	(“CMPT125”, ”Fall2018”)
3	
4	
5	
6	
7	(“CMPT125”, ”Fall2020”)
8	(“CMPT409”, ”Fall2019”)
9	(“CMPT225”, ”Spring21”)
10	
11	
12	(“CMPT125”, ”Fall2019”)
13	(“CMPT805”, ”Spring2019”)

Collision-Handling

- Collision happens when the hash function returns the same value with a different key during insert, resulting in attempting to insert a data to a slot that is occupied by a previously inserted data (different key but same value return by the hash function)
- There are a few schemes (strategies) to handle collisions:
  - Separate Chaining – each slot has an internal data structure (usually a list) that accepts colliding data (pros: simple to implement, cons: needs extra space)
  - Open Addressing – look for other available slots in the hashtable to store the colliding data (pros: doesn't need extra space, cons: more complex operations)
    - Within this scheme there are several options: linear probing, quadratic probing, and double-hashing

Hashing in C++

- The C++ STL offers a hash table implementation called unordered_map
  - Supports standard operations (e.g., insert, erase, find)
  - Provides some other useful/informative methods (e.g., load_factor, direct access)
- Available from the <unordered_map> library https://en.cppreference.com/w/cpp/container/unordered_map
- Used as a template
  - At minimal, provide the key type (typically a string) and the value type
  - Can also use custom key type (will also need a custom hash function class)

Implementation Example

- Suppose we want to store a hash table of recipe ingredients:
  - Key (as string) = name of the ingredient
  - Value (as struct object) = quantity and unit
- We can declare an unordered_map object as follow:

```
//create an unordered_map object storing a recipe
unordered_map<string, Measurement> shortBreadCookies;
```

1 cup (226 g) unsalted butter softened
¾ cup (95 g) powdered sugar
1 teaspoons vanilla extract
2 cups (250 g) all-purpose flour
2 Tablespoons cornstarch
½ teaspoon table salt

```
struct Measurement {
    float quantity;
    std::string unit;
};
```

Custom Key Type

- The unordered_map template class also accepts custom key types but you must provide the following:
  - Definition of the custom key class
  - Implementation of operator== of the custom key class
    - This allows unordered_map to figure out if the same key is used by an existing item
  - Definition of the custom hash function class
    - It only needs to implement 1 method: operator()
    - This is used by the unordered_map to calculate the hash value

```
//create an unordered_map object using a custom key
unordered_map<Ingredient, Measurement, HashFunctionForIngredient> recipe;
pair<Ingredient, Measurement> butter(Ingredient("no name", "unsalted butter"), {1, "cup"});
recipe.insert(butter);
```

Exercise

- Download the **ingredient.hpp**, **measurement.hpp** and **hashtableDemo.cpp** files
  - Complete the code in hashtableDemo.cpp (read the other files to understand the rest)
    - Look for the comment: //***your code***
  - Use the comments to help you
  - Try one more way to insert another ingredient:
    - shortBreadCookies["melted chocolate"] = {1, “cup"};
  - Try the erase method
  - Create another demo file doing the same thing but use the Ingredient class as the custom key and the HashFunctionForIngredient as the custom hash function
    - You can choose what brand to use… no name is always an option!
- The code is mostly done! Your job is to finish it, but you should also familiarize yourself with the provided code