CS 008
Lecture notes
3/7/24

1 Outline

- Pointers & Dynamic Arrays
- New data structure (sequence)!

2 Pointers

One of the most important concepts in C++ involves **pointers**. Pointers point to specific **address** in memory and expect a specific data type to be there. One consequence of pointers is the **Dynamic Array**.

2.1 Pointers

Definition:

To **declare** a pointer, you can use the * symbol in front of the name of the variable.

```
int *pointername;
```

To **assign** a pointer, you can use the & to set it to the **address** of a variable. Pointer assignment:

```
int *pointername = &pointtome;
```

You can get and change a value that the pointer is pointing to by **dereferencing** the pointer using the * symbol.

```
*pointername = 43;
```

*pointer gets the value of the object it is pointing to. You can change this value too.

2.2 Dynamic variables

We can use a pointer to initialize a variable. You can create a pointer of the proper type and then using the **new** character we can allocate a **dynamic variable**. Dynamic variables can only be accessed by a pointer.

Example:

```
int* pointername = new int[100];
```

Using [], you can allocate a nameless array. Since the pointer points to a dynamic array, we can **access** and **assign** using [].

Using the symbol [], you can allocate a dynamic array of a datatype. Similarly to the arrays we know, the dynamic arrays can be allocated with a certain amount of elements. Using **new** opens up the possibility for user-defined lengths on class members.

2.3 Delete

Deallocation is critical! When you use **new** somewhere, you need to use **delete** somewhere else.

Warning:

Failure to deallocate dynamic variables will result in a **memory leak**. If it's bad enough, your machine will run out of memory and your program will crash.

Note:

When deleting dynamic arrays, be sure to use `delete[]` otherwise you will not delete the entire array.

2.4 Dynamic members

When defining a class with dynamic members, you must add:

- A class copy constructor (`Sequence(const Sequence& other_sequence);`)
- An overloaded assignment operator (`void operator=(const Sequence& other_sequence);`)
- A class destructor (`~Sequence();`)

When working with dynamic variables, one of the things you need to do when defining a class with dynamic members, you must create a class copy constructor.

Example:

```
// Example class
 class bag {

   // EXAMPLE

   int used;
   int* data_ptr = new int[30];

   // EXAMPLE

 }

  bag bag1 = ...;
  bag bag2(bag1);
```

You need to have a class copy constructor since the copy might point to the same dynamic variables the original was pointing to. The default class copy constructor only creates a *shallow* copy of the original. It grabs the values in the original member variables and then creates an identical copy. However, if they are both identical the data pointer in your copy object will point to the same piece of memory in your original object. To combat this, you create your own copy constructor and copy the contents of a dynamic array to a new dynamic array for the copy of the object.

Warning:

You must define a copy constructor, assignment operator and a class destructor to avoid conflicting pointers. If you don't include these elements, your code may still compile, but your memory management will be completely messed up.

Bad errors will happen.

3 Sequence class

The sequence class is another **container** class. It is also very similar to the bag class seen earlier but unlike the bag class, it has a sense of ordering. There can be a 'first' element, 'second' element, ..., and 'last' element.

In constrast, the bag class is only a pile of elements. There is no specific ordering of elements in the bag.

Question:

What is the dimensionality of the data in the Sequence class?

Answer: The Sequence class is one-dimensional. Compared to the bag class, the Sequence class has a sense of ordering. The position of the elements in the Sequence class can be described by a number unlike a bag.

3.1 Misc.

Everytime you create a header file, you need to create a **header guard**. Header guards are used to avoid duplicate definitions.

Example:

```
#ifndef UNIQUE_NAME
#define UNIQUE_NAME

// contents

#endif
```

The first line of the header file checks if the UNIQUE NAME has already been defined. The second line then defines UNIQUE NAME and the contents of the file. When there is another header file trying to define UNIQUE NAME again, the contents of the header will be ignored.

Example:

```
#ifndef SEQUENCE_H
#define SEQUENCE_H
#include <stddef.h>

class Sequence {
  // ...

 public:

    typedef int value_type;
    typedef size_t size_type;

    static const size_t CAPACITY = 30;

    Sequence();

    //...

};
#endif
```