Daily Notes

Hasitha Dharmasiri

10/23/2013

1 Announcements

1.1 Project 4

- This assignment is *pledged*.
- Project was assigned Friday, October 25. It is due on **Friday, November 1**.
- Information is mostly posted, but item 5 may change.

1.2 Lecture Topic

The lecture topic is working with the parent/child (inheritance) relationship between classes in C++. Other features of C++ are also briefly covered. The vehicle for learning about class structure in C++ is through the `VecPPBase`, `VecPPDense` and `VecPPSparse` classes, provided in `vecppbase.hh`, `vecppdense.hh` and `vecppsparse.hh` files in the class `src` directory.

2 `const` Functions

Observe function `getLength()` inside `VecPPBase`. The declaration is

```
virtual int getLength() const = 0
```

The `const` function cannot changed the object that it is called upon. It is good programming practice to designate functions that do not modify the object as `const` so that the compiler can help you catch errors. The main takeaway: **If it can be const, make it const!**

2.1 Getters and Setters in C++

We need to be able to get and set individual elements inside a vector, so we overload the functional call operator `operator()`.

2.1.1 Changeable Reference

In order to set an individual element, the object must return a changeable reference. Therefore, we have a functional call operator overload function without a `const` keyword:

```
virtual double & operator()(int i) = 0;
```

This function will return a modifiable reference to element i.

2.1.2 Constant Reference

In order to get an element, the object does not need a changeable reference; a `const` one will do. Therefore we have a function call operator overload function with the `const` keyword:

```
virtual double const & operator()(int i) const = 0;
```

Notice that there are two `const` keywords in this declaration. It is good practice to include this overload in addition to the changeable reference overload to help identify errors that may occur when a value is changed when it shouldn't be.

2.1.3 Compiler Behavior with `const`

The compiler will actually use the most restrictive implementation of the overload function when deciding which version of the method to use. Therefore, the compiler will choose the `const` version of the method if it is called from the right-hand side (RHS) and the non-`const` version if called from the left-hand side (LHS).

3 Inheritance

To demonstrate subclass and class relationships, `VecPPDense` is declared a child of `VecPPBase` by using the following delcaration:

```
class VecPPDense: public VecPPBase
```

Another subclass of `VecPPBase`, `VecPPSparse` is also defined in a similar fashion.

`VecPPDense` stores data in a dense fashion (i.e. the vector data is stored in a one-to-one array). `VecPPSparse` stores a sparse vector, where there is an assumed reference value in the vector (most of the time, it is 0). Only the values that are not equal to the reference values are explicitly stored, along with their index (location) in the vector.

3.1 Getter and Setters in the Subclasses

Note that because the getter and setter overload functions in `VecPPBase` were defined as `virtual`, they must be defined in the subclasses. Both subclasses have to handle this differently:

- `VecPPDense` uses the exact same function definition for both the `const` and non-`const` versions of the functions. This is because accessing an element in an array is a simple process and modifying the element will not change the overall structure of the array. With simpler data structures, it may be required to have idential `const` and non-`const` versions. This leads to *code bloat*, widely considered a flaw in C++.
- `VecPPSparse` has a more complicated data structure. While accessing an element without modification only requires looking up the index and value of the element, accessing with modification requires different code for the non-`const` function. This code must be able to store new indices and values in the sparse vector data structure if the user changes an element from the reference value to the non-reference value.

3.2 Shared Functions

Notice that the subclasses do not have a definition for the `axpy` function. This is defined in the base class. When the compiler see as a call to `axpy` on an object of the subclass type, it will first check the subclass for an override of the function, and then it will check the base class for the function. As with the behavior on `const` and non-`const` functions, the compiler will always look for the most restrictive function to execute (the subclass is more restrictive than the base class).

Also, note that because `axpy` is defined in the base class, the user can `axpy` a dense vector with a sparse vector and vice versa.

4 Additional Topics

4.1 Type Promotion

C++ can automatically convert certain primitive types without requiring a cast. The safety of these conversions depends on the particular conversion. For example:

- **Integers can be converted to doubles safely**. The following code will result in the value of `2.0` being stored in `d`:

```
int i = 2;
double d = i;
```

- **Floats cannot be converted to doubles safely**. The following code will introduce error in `d`:

```
float f = 2.50;
double d = f;
```

  The `double` will contain the `float` information, but the less significant bits present in the longer `double` will be filled with trash.

4.2 Sparsity

We introduced a sparse vector class `VecPPSparse` in this lecture. Sparsity by definition indicates that most of the values in a structure (vector or matrix) are of a single reference values. In most linear algebra applications, this reference value is 0. Because of this, if a storage system is used that can only store the non-reference values and their locations, extremely large data structures can be created and used. One example involving large sparse matrices is solving the Netflix problem.

5 Conclusion

In this lecture, we used vector classes to show the power of **abstraction** in C++.