COMP6771
Advanced C++ Programming

Week 12
Extension Topics and Revision

2016

www.cse.unsw.edu.au/~cs6771

Multiple Inheritance

- Complicated and not widely used (i.e., reconsider your design!)
- Consider a class for an Amphibian vehicle:

```
1 class Amphibian : public Boat, public Car {
2   //
3 };
```

- `Amphibian` will support the public methods and contain the data members of both `Boat` and `Car`.
- `Amphibian`'s methods will have access to the protected data and methods in both `Boat` and `Car`.
- An `Amphibian` object can be upcast to either a `Boat` or a `Car`.
- Constructor occurs in order of listing (destruction in reverse).

Multiple Inheritance Example

```
1  class Boat {
2  public:
3    Boat() : anchorDropped{false} {}
4    void dropAnchor() { anchorDropped = true;}
5  private:
6    bool anchorDropped;
7  };
8
9  class Car {
10 public:
11   Car() : sunroofOpen{false} {}
12   void openSunroof() { sunroofOpen = true; }
13 private:
14   bool sunroofOpen;
15 };
16
17 class Amphibian : public Boat, public Car {
18
19 };
20
21 int main() {
22   Amphibian a;
23   a.dropAnchor();
24   a.openSunroof();
25 }
```

Name Ambiguity

What if both base classes have the same function name?

```
1  class Boat {
2  public:
3    Boat() : anchorDropped{false} {}
4    void dropAnchor() { anchorDropped = true;}
5    virtual void drive() {}
6  private:
7    bool anchorDropped;
8  };
9
10 class Car {
11 public:
12   Car() : sunroofOpen{false} {}
13   void openSunroof() { sunroofOpen = true; }
14   virtual void drive() {}
15 private:
16   bool sunroofOpen;
17 };
```

Name Ambiguity

The following won't compile:

```
1 int main() {
2   Amphibian a;
3   a.drive();
4 }
```

```
1 Amphibian.cpp: In function int main():
2 Amphibian.cpp:27:4: error: request for member drive is ambiguous
3   a.drive();
4     ^
5 Amphibian.cpp:14:16: note: candidates are: virtual void Car::drive()
6    virtual void drive() {}
7                 ^
8 Amphibian.cpp:5:16: note:                 virtual void Boat::drive()
9    virtual void drive() {}
```

We can resolve this either in user or class code.

Resolving Name Ambiguity

- User Code: Dynamic casting (upcasting):

```
1 dynamic_cast<Car&>(a).drive();
```

- User Code: Disambiguation Syntax:

```
1 a.Car::drive();
```

- Class Code: Define the drive function in the Amphibian class

```
1 class Amphibian : public Boat, public Car {
2 public:
3   void drive() {
4     Car::drive(); // Explicity call Car's version of drive
5   }
6 };
```

- Class Code: Using statement

```
1 class Amphibian : public Boat, public Car {
2 public:
3   using Car::drive;
4 };
```

Circular Dependencies

Consider a Image producer class (MakeImage.h):

```
1  #pragma once
2  #include "Display.h"
3
4  class Image {
5    int imgData[800][600];
6  };
7
8  class MakeImage {
9    Display& displayToSendImage;
10 };
```

And consider a Image display class (Display.h):

```
1  #pragma once
2  #include "MakeImage.h"
3
4  class Display {
5    void receiveImage(const Image& img);
6    Image lastReceivedImage;
7  };
```

Circular Dependencies Implementations

MakeImage.cpp

```
1 #include "MakeImage.h"
```

Display.cpp

```
1 #include "Display.h"
2
3 void Display::receiveImage(const Image& img) {
4   lastReceivedImage = img;
5 }
```

g++ -c Display.cpp

```
1 In file included from Display.h:3:0,
2                  from Display.cpp:1:
3 MakeImage.h:13:2: error: Display does not name a type
4   Display& displayToSendImage;
5   ^
```

Error! Display.h includes MakeImage.h for the Image class.
But MakeImage.h needs the class Display declared to compile the rest of the header file.

A (wrong) solution

- Forward declarations can often resolve dependency issues.
- Updated MakeImage.h

```
1  #pragma once
2  #include "Display.h"
3
4  class Display;
5
6  class Image {
7  private:
8    int imgData[800][600];
9  };
10
11 class MakeImage {
12   Display& displayToSendImage;
13   Image currImage;
14 };
```

- Display.cpp now compiles, but: g++ -c MakeImage.cpp

```
1  In file included from MakeImage.h:5:0,
2                   from MakeImage.cpp:1:
3  Display.h:7:26: error: Image does not name a type
4    void receiveImage(const Image& img);
5                            ^
6  Display.h:9:2: error: Image does not name a type
7    Image lastReceivedImage;
8    ^
```

A (still wrong) solution

- Forward declaration in Display.h

```
1  #pragma once
2  #include "MakeImage.h"
3
4  class Image;
5
6  class Display {
7  public:
8    void receiveImage(const Image& img);
9  private:
10   Image lastReceivedImage;
11 };
```

- g++ -c MakeImage.cpp

```
1  In file included from MakeImage.h:3:0,
2                   from MakeImage.cpp:1:
3  Display.h:11:8: error: field lastReceivedImage has incomplete type Image
4    Image lastReceivedImage;
5          ^
```

- Problem! we want a full copy (not reference) of Image but it hasn't been fully declared yet!

A working solution

- Both classes are dependant on one another (and on Image)
- Put the Image class declaration in it's own .h file (Image.h)

```
1 #pragma once
2
3 class Image {
4 private:
5   int imgData[800][600];
6 };
```

- Display.h

```
1 #pragma once
2 #include "Image.h"
3
4 class Display {
5   void receiveImage(const Image& img);
6   Image lastReceivedImage;
7 };
```

- MakeImage.h

```
1 #pragma once
2 #include "Image.h"
3 #include "Display.h"
4
5 class MakeImage {
6   Display& displayToSendImage;
7   Image currImage;
8 };
```

- Now both MakeImage.cpp and Display.cpp compile!

Performance

- Writing code is easy. Writing fast code can be difficult.
- Sometimes a trade off between memory and execution time.

Things to keep in mind:

- References and pointers can minimize memory copies.
- Different STL containers have different performance for different operations.
  - What is the insertion complexity for a vector and a map?
  - What is the lookup time for an item in a vector and a map?
- The optimizer can dramatically improve performance. But we should also always think about how we can make our code go fast!

Designing the Graph Container

- Let's design a fast and a low memory Graph container.
- Begin with the Node objects, how should these be stored?
  - Should we create a wrapper class around the Node objects?
  - If we have a wrapper class should it store objects of type N or smart pointers around N?
  - What sort of container should we store raw Nodes or Node Wrapper objects in?
  - What are we going to do with the nodes in this container? Insert, delete, find?
- What about the Edge objects?
  - Do we need an Edge wrapper class?
  - Where should a Edge wrapper class be declared?
  - What is the interaction between the Edge and Node class?
  - Does the Graph class need direct access to Edges?
  - Should we store a big container of Edges in the Graph, or many small containers in each Node wrapper?

Graph with inner Node class

```
1  template <typename N, typename E>
2  class Graph {
3    // private inner class for the node data
4    class Node {
5      // the data stored on this node
6      std::shared_ptr<N> nodeData;
7      // the list of edges
8      mutable std::list<std::pair<std::weak_ptr<Node>,E>> edges;
9    };
10
11   // ...
```

- `nodeData` is a shared_ptr as the iterator uses weak_ptr's over these N objects.
- An edge is a `std::pair<std::weak_ptr<Node>,E>` so Node objects will have to be stored as shared_ptr
- The edge weight object E is stored by value.
- The edges list is mutable so when an expired Node is found it can be removed (more on this later)

Storing the Nodes in the Graph

- What container should store Node objects, vector or a Map?
- A vector has bad performance for finding data O(N) and deleting (due to shuffling) so a Map is better.
- A map, by definition, also doesn't have duplicates.
- Should the map's key/values be:

```
1 std::map<N, std::shared_ptr<Node>> nodeMap;
```

  Lookup is log(N) but we're now storing the N objects twice - wasting memory

- What about:

```
1 std::map<std::shared_ptr<N>, std::shared_ptr<Node>> nodeMap;
```

  We are no longer wasting memory, but now our lookup is back to O(N) as we have to iterate through the map

- Is there another way?

Node Wrapper with Operator Overloading

- We can wrap the smart pointer and expose the operator< function to enable lookup based on the value of N

```
1  class nodePtrWrapper {
2    std::shared_ptr<N> n; // the shared pointer to the node data
3  public
4    // the friend less than operator is defined inline to make the compiler happy.
5    friend bool operator<(const nodePtrWrapper& lhs, const nodePtrWrapper& rhs) {
6      return *(lhs.n) < *(rhs.n);
7    }
8  };
```

- Our map is now:

```
1  std::map<nodePtrWrapper, std::shared_ptr<Node>> nodeMap;
```

  This makes look up fast again (we simply create a nodePtrWrapper object over the N item we want to find!)

- (Note: we could also combine this with the previous Node class and use a std::set)

Lookup using the Node wrapper classes

```
1 template<typename N, typename E>
2 std::shared_ptr<typename Graph<N,E>::Node> Graph<N,E>::getNodePtr(const N& nd) const {
3   try {
4     auto nodeIt = nodeMap.at(nodePtrWrapper(std::make_shared<N>(nd)));
5     return nodeIt;
6   } catch (...) {
7     return nullptr;
8   }
9 }
```

- To lookup items we create a nodePtrWrapper over a new shared_ptr and see if we can find it in the nodeMap!
- If the nodeMap already contains a shared_ptr over an N object of the same value the two shared_ptrs will match (due to the operator overloading), even though they are different ptrs!
- We can then efficently lookup in our graph

Deleting a node is simple

```
1 template<typename N, typename E>
2 void Graph<N,E>::deleteNode(const N& nd) noexcept {
3   try {
4     nodeMap.erase(nodePtrWrapper(std::make_shared<N>(nd)));
5   } catch (...) {
6     // erase shouldn't throw an exception
7     return;
8   }
9 }
```

- Create a nodePtrWrapper and a shared_ptr to delete a node!
- Because of the Map structure the delete is very fast!
- As the Node class stores Edges as weak_ptrs we don't clean them up now.
- Instead when we discover they are expired we clean them up e.g., when creating an iterator or printing.