Topics Covered

- Reading: 654-672, 731-774
- Multiple Inheritance
  - Syntax for Multiple Inheritance [MI]
  - Calling inherited methods
  - Overriding Inherited methods
  - Dealing with ambiguity and virtual base classes
- C++ Exception handling and Exception classes
- RTTI (Runtime Type Identification)
- Homework

Multiple Inheritance I

- In C++ a child (derived) class can have more than one parent (base) class
  - This is called multiple inheritance
  - Can be public, private, protected, or mixed
- Cat
  - Cat();
  - purr();
  - speak();
- Fish
  - Fish();
  - speak();
  - swim();
- CatFish
  - CatFish();
Multiple Inheritance II

- Syntax is similar to single inheritance
  - Separate parents with commas
    ```
    class CatFish : public Cat, public Fish {...}
    ```
- Child inherits members from each parent
  - CatFish can both swim() and purr()
    ```
    CatFish cf;
    cf.swim(); // from Fish ancestor
    cf.purr(); // from Cat ancestor
    ```

Multiple Inheritance III

- MI can introduce potential ambiguities
  - When ancestors have identically named members
    ```
    cf.speak(); // does not compile
    ```
  - Catfish can't decide between competing parents
    - Note: will not simply use closest method
  - Two solutions: new method, scope resolution
    ```
    cf.Cat::speak(); // Not ambiguous
    ```
    ```
    void CatFish::speak() { /* new method */ }
    ```

Real-life MI

- MI is often used in real code
  - Example: the stream classes
    ```
    ios : formatting
    istream : input methods
    ostream : output methods
    istream : input & output
    ```
    ```
    ios
    General properties
    + pointer to streambuf
    ```
    ```
    ostream
    Output methods
    istream
    Input methods
    ```
    ```
    istream
    Input & Output
    ```
Virtual Base Classes I

- What happens when both parents have a common ancestor, like the iostream class [ios]?
  - Let's add an Animal class to our CatFish hierarchy, and see
    - CatFish02.cpp

Virtual Base Classes II

- We'll store the name of each subclass in Animal
  - Store as a C++ string so we don't manage memory
  - Add a getName() method to retrieve it
  - Doesn't need to be virtual
- Add a speak() method to CatFish [ambiguity]
  - Call Cat::speak() and Fish::speak(). Results?
- Call getName() through a base pointer
  - Obviously two different Animal subobjects

Virtual Base Classes III

- In reality, Cat and Fish each have their own copy of Animal, so the real hierarchy looks like:
Virtual Base Classes IV

- This is normally undesirable. To fix it: `CatFish03`
  - Use keyword `virtual` when deriving from `Animal`
    ```
    class Fish : virtual public Animal {}; 
    class Cat : public virtual Animal {}; 
    ```
  - Either virtual or public can go first
  - Only one copy of `Animal` will exist when new classes [like `CatFish`] are derived from `Cat` and `Fish`

Virtual and NonVirtual

- Don't use `virtual` when deriving MI subclasses
  ```
  class CatFish : public Cat, public Fish {}; 
  ```
  - Would use `virtual` if you wanted to derive new subclasses from `CatFish`, and you wanted to ensure a single copy of `Cat` or `Fish` was used
- Suppose you create a `CatFishBird`?
  - Bird virtually derived? One copy of `Animal`
  - Bird non-virtual? One for `Cat` & `Fish`, one for `Bird`

VBC Constructors I

- Derived classes call immediate parent constructor
  - Parent passes arguments to grandparent, etc.
  - Assume a `CSStudent` is-a `Student` is-a `Person`
    ```
    Student(const char * nm) : Person(nm) {} 
    CSStudent(const char nm) : Student(nm) {} 
    ```
  - This would normally be illegal
    ```
    // Person is a grandparent, not parent 
    CSStudent(const char * nm) : Person(nm) {} 
    ```
VBC Constructors II

- Rules are different for virtual base classes
  - Suppose we wanted similar constructors for Animals
    
    Animal(const char * n): name(n) {}
    Cat(const char * n): Animal(n) {}
    Fish(const char * n): Animal(n) {}
    CatFish(const char * n): Cat(n), Fish(n) {}
  
- With non-virtual inheritance, this is no problem
  - We initialize both our Cat side and our Fish side
  - With virtual inheritance, there is only one name field

VBC Constructors III

- With virtual base classes, all subclasses may, and must, call the virtual class constructor
  - Intermediate constructor calls are just ignored
    
    Fish charlie("Charlie"): // OK
    Cat garfield("Garfield"): // OK
    CatFish king("KingKat"): // Not OK
  
  - Neither charlie nor garfield will call the Animal constructor for king
  - All other Cat and Fish construction will occur

VBC Constructors IV

- This means that virtual MI subclasses must call their immediate base class constructors, as well as the virtual base class constructor
  
  Animal(char * n): name(n) {}
  Cat(char * n, int lv=9): Animal(n), lives(lv) {}
  Fish(char * n, int x): Animal(n), age(x) {}
  CatFish(char * n, int x, int lv=9)
    : Animal(n), Cat("none", lv), Fish("none", x) {}

  - Note that the char * arguments are passed to Fish and Cat, but not relayed to Animal
VBCs Ambiguity & Dominance I

Ambiguity rule is simple if no virtual inheritance
- Two methods of same name are ambiguous

```cpp
void CatFish::doIt()
{
    speak(); // ???
    dance(); // ???
    Cat::dance(); // OK
}
```

VBCs Ambiguity & Dominance II

VBCs guarantee only one "copy" of base methods, so methods in derived class "dominate" and can be used without qualification

```cpp
void CatFish::doIt()
{
    speak(); // Cat::speak()
    Animal::speak();
}
```

Homework

- Both are due next Sunday, April 25th
- Homework 20: Multiple Inheritance and Templates
  - Use Array<T> and RCArrary<T> in hmwk20.cpp
  - Add BoundedArray<T> and BoundedRCArrary<T>
  - Note that upper bounds is outside array
- Homework 21: Multiple Inheritance
  - Add a ContractEmployee to Homework 13
  - Watch your memory
Exception Handling I

- Sometimes, runtime errors are caused by unusual situations, not programming errors
- For example: writing data to a file
  - Most of the time things go uneventfully, but...
    - The disk may be full
    - There may be a hardware error
    - The file may have been changed to read-only
- Fragile code ignores the possibility of problems
- Robust code anticipates such problems

Exception Handling II

- Traditional method uses "completion codes"

```
Get a file name
Open the file
  If there is no error opening the file
    Then read some data
      If there is no error reading the data
        Then process the data
          Write the data to output
            If there is no error writing the data
              Then close the file
                Then return OK
            Else ...
        Else ...
    Else ...
Else ...
```

Exception Handling III

- Things to notice with completion codes
  - Almost every step may fail, except "PROCESS DATA"
    - Program just be prepared to deal with failure
  - Very difficult to determine "normal" actions
    - So concerned with things that can go wrong it is hard to tell if things are done in the right order
  - Difficult to use if library function contains such code
    - Function must return several error codes so that user can tell what went wrong
Exception Handling IV

Using exceptions, code looks like this:

```cpp
Try to do each of these things:
    Get a filename
    Open the file
    Read some data
    Process the data
    Write the data
    Close the file
Return
    If an error occurred when opening the file then …
    If an error occurred when reading the data then …
    If an error occurred when writing the data then …
    If an error occurred when closing the file then …
```

Exception Handling V

Compare methods of handling "bad things"

Code that uses the exception strategy:
- Is shorter and easier to read
- Makes the normal logic the main focus of the method. With completion codes, the error-handling code obscures the normal actions
- Allows use to decide whether to handle the problem or defer handling the error on a case-by-case basis

C++ Exception Handling

C++ uses three keywords for exception handling
- `throw` is used to "trigger" a specific exception
  » This may be in your code or library code
- `try` is used to bracket code or functions that may throw an exception
  » Used when executing the code
- `catch` is used when an exception occurs
  » You may supply code to quit or recover
  » You may have code for different types of errors
Using throw

- Exceptions are generated by using throw
  - Code following the throw is skipped: [Exception01]

```cpp
void riskyBusiness()
{
    char ch;
    cout << "Invest in tech stocks?" << endl;
    cin >> ch;
    if (ch == 'y') throw "Are you crazy?";
    cout << "OK, here's my savings!";
}
```

Using try

- Exceptions are trapped by using try:

```cpp
try {
    riskyBusiness(); // May throw exception
    cout << "All right, you've got some sense";
}
```

- The keyword try followed by braces
  - Braces are required part of the syntax
  - Statements or functions that may throw an exception.
  - Statements following the exception are skipped

Using catch

- The catch block looks a little like a function
  - Must follow immediately after try or another catch
  - Body of catch block delimited by braces
  - catch block argument list contains type & name

```cpp
  catch (const char * msg)
  {
      cerr << msg << endl;
      cerr.flush();
      exit(1);
  }
```
Multiple Exceptions I

- A single function may generate several errors
  - You may pass the wrong filename
  - You may not have permission to read the file
  - You may have some kind of reading error
  - You may not be able to write to the file

- You can handle this by throwing different types
  
  ```
  if (ch == 'n') throw "Missing an opportunity!!!";
  if (ch == 'y') throw 995345.0;
  cout << "Your answer was : " << ch << ". ";
  ```

Multiple Exceptions II

- Calling the function, use multiple catch blocks
  
  ```
  try {
    riskyBusiness();
  } catch (const char * e1) { ... }
  catch (double e2) { ... }
  catch (...) { }
  ```

- You can use the ellipsis ... to catch all the others

- Examples : Exception2, Exception3

Exception Classes

- Usually, instead of throwing primitive types, you'll create a special exception class
  - Often, you'll subclass the Std C++ class exception
  - Override the virtual method what()
  - Example : Exception4.cpp

- Create and throw an instance of your class
  
  ```
  if (name == "bozo")
    throw NamingException("No bozos allowed");
  ```
**Uncaught Exceptions**

- The user doesn't have to catch your exceptions
  - If there is no try block, then the uncaught exception calls a method named terminate()
    » This normally just calls abort()

- You can have terminate() call your own function
  - Write a function that takes and returns no arguments
  - Pass your function to set_terminate()
  - Example: Exception5.cpp

**Unexpected Exceptions**

- Adding an exception specification to prototype
  - Allows compiler to issue warnings [Exception6]
  - Allows runtime to catch unexpected errors
    ```
    void getname() throw (NamingException&)
    ```

- Catching an unexpected exception
  - Normal behavior is to call unexpected() function
    » Normally calls terminate()
  - You can modify by calling set_unexpected()

**Standard Library Exceptions**

- Not really well integrated into standard library
  - out_of_range used for standard at() method
  - bad_alloc now used for failures of new operator
    ```
    #include <new>
    try {
    /* allocate memory with new */
    }
    catch (bad_alloc& e) { ... }
    ```
  - Can eliminate by using new(throw)
Why OOP?

Consider following "C" code for a simulation:

```c
struct item { int type; ...
struct Doctor { int type; ...
struct Patient { int type; ...

void Draw(void * p) {
    item * ip = (item *) p;
    switch(item->type) {
        case DOCTOR: // Draw a doctor
        case PATIENT: // Draw a patient
    }
}
```

What's the problem with using a type ID?
- Compiler cannot check validity of cast
  - Easy to make a mistake
- Code is not easily extensible
  - Each function has to have its own switch
  - Adding Nurses or Janitors will break the code
- Better solution is virtual functions
- Some designs require more

The Problem with Virtual Birds I

Consider the following class hierarchy

```
class Bird
    virtual void speak();
    virtual void fly();

class Robin
class Hawk
class Penguin
```
The Problem with Virtual Birds II

- Virtual function is mandatory interface
  - Previous design says "Penguins can fly"
  - One solution: No-op the method:
    ```cpp
    void Penguin::fly() { /* nada */ }
    ```
  - Design is still incorrect, but this is a "quick fix"
  - Better design is to add a subclass of flying birds with a `fly()` method

The Problem with Virtual Birds III

- Flying and non-flying Birds
  - Class `Bird` with virtual `speak()`
  - Class `FlyingBird` with virtual `fly()`
  - Class `Robin`
  - Class `Hawk`
  - Class `Penguin`

The Problem with Virtual Birds IV

- How do we get this code to work?
  ```cpp
  Bird *bp = getRandomBird();
  bp->speak(); // All birds can speak
  bp->fly();  // Nope, penguins can't fly
  ```
- One solution is just to cast the pointer
  ```cpp
  ((FlyingBird *)bp)->fly();
  ```
  - Legal, but incredibly unsafe.
Enter dynamic_cast

- What we want is a way to say
  - "If a bird is a FlyingBird, then fly()"
- C++ adds operator to perform a safe downcast
  - Downcast is from a base ptr to derived ptr
  - Safe only if base ptr points to a derived object
    - Upcasting [derived ptr to base ptr is always safe]
  - dynamic_cast performs a safe downcast
  - Checks if base ptr points to derived object
    - If not, then it returns 0;

Syntax of dynamic_cast

- Here's a dynamic_cast example:

```cpp
typedef FlyingBird* FBPtr;
Robin r; Penguin p;
Bird * bp1 = r, *bp2 = p;
FBPtr fbp;
fbp = dynamic_cast<FBPtr> (bp1);
fbp = dynamic_cast<FBPtr> (bp2);
```

- First example succeeds, second does not
- Example: RTTI01.cpp

References and dynamic_cast

- Dynamic cast works with references as well
  - Reference cannot be 0 like pointers
  - Throws a bad_cast exception
  - Must include <typeinfo> header

```cpp
#include <typeinfo>
try {
Penguin& pr = dynamic_cast<Penguin&>(p);
} catch (bad_cast &) { … }
```
Using typeid() and type_info

- The typeid() returns type of an object
  - Argument to typeid() is
    - An object or a type [like with sizeof]
  - May need to activate RTTI on compiler
  - Must include standard <typeinfo> header
  - Return value is a type_info object
  - Throws a bad_typeid exception
- Example: RTTI02.cpp

Using typeid() and type_info

- Here’s how you can use typeid()
  - Compare typeid of object to a class
    - if (typeid(*bp) == typeid(Penguin)) ...
  - Compare typeid of two objects
    - if (typeid(*bp1) != typeid(*bp2)) ...
  - Call the name() method on type_info
    - cout << typeid(*bp).name() << endl;

Other C++ Casts I

- C++ adds four other new casting styles
  - Normal casts are error-prone and hard to see
  - static_cast works most like traditional cast
    - double ans = (double) a / b;
    - double ans = static_cast<double>(a)/b;
    - Cannot change the constness of an object
  - reinterpret_cast casts incompatible pointers
    - x = * (reinterpret_cast<int *>(pd));
Other C++ Casts II

- The `const_cast` changes `const` or `volatile` – Does not change any other characteristics

```cpp
class Widget { ... };  
class SWidget : public Widget { ... };  
void update(SWidget *psw);  
const SWidget sw;  
update(&sw); // Error  
update(const_cast<SWidget*>(&sw));  
Widget * pw = new SWidget;  
update(const_cast<SWidget*>(pw)); // Error
```

Homework

- Homework 22 : Exceptions
  - Modify `Fraction22.cpp`
  - Add `DivZero` class derived from exception
  - Override the `what()` function
  - Modify `Fraction` functions to throw exceptions

- Homework 23 : RTTI
  - The `BankAccount` class hierarchy
  - Using `type_info` and `dynamic_cast` with `BankAccount`