• Lecture Outline - Multiple Inheritance and RTTI
  • Stroustrup Ch 21, 22.
  • Note: His diagrams are not UML diagrams!

• Basic concept of multiple inheritance
  • Inherit from more than one class
  • class Derived : public Base1, public Base2 {
    • Derived object has Base1 member variables followed by Base2 member variables
    • Has both sets of member functions
    • Base constructors run in declaration order, destructors in reverse declaration order

• Mixin Multiple inheritance
  • Mixin multiple inheritance is no problem, and can be very useful at times
    • concept: base classes are orthogonal - have nothing in common
      • no tricky ambiguity or duplication issues
    • Simply add some functionality or an interface to derived classes.
      • class Base1 {
        public:
        void defrangulate();
      };
      • class Base2 {
        public:
        void transmogrify();
      };
      • class Base3 {
        public:
        void munge();
      };
      • Derive from combinations of class to get combinations of functionality
        • class D1 : public Base1, public Base3 {
          can defrangulate and munge
        };
        • class D2: public Base2, public Base3 {
          can transmogrify and munge
        };
        • class D3; public Base1, public Base2, public Base3 {
          can do all three
        };
    • An example later - allows you to connect together two different class systems
      • A GUI toolkit class hierarchy and a whole functionality system
      • Model-View-Controller and Observer patterns

• Use for capability classes and queries solution to the fat interface problem
  • Class has a base class with a common interface and (possibly) implementation
  • Each capability class is an interface class describing a way of talking to an object
  • Class inherits also from one or more capability classes
  • A dynamic_cast is used to find out whether an object implements a particular interface

• Implications of multiple inheritance
  • If two base classes have a member of the same name, you have to disambiguate which one you mean in order to access it.
MultipleInheritanceRTTI.oo3  11/7/13 1:06:27

- class Base1 has void foo();
  class Base2 has void foo();
  class D inherits from both
  D d;
  d.foo() // ambiguous - you have to say which one you want
  d.Base2::foo();

- Confusing: Function overloading won't sort it out - compiler looks for the function name first, and balks immediately if the name is ambiguous.

- If a class has two bases with virtual functions in each base, a virtual call can only access the virtual functions in the one base corresponding to the pointer.
  - class Base1 has virtual void vf1();
  - class Base2 has virtual void vf2();
  - Base1 * p1; Base2 * p2;
  - p1->vf2();  // compile error - p1 is a Base1 *, no information about Base2
  - p2->vf1();  // compile error
  - If you have the Base1 pointer and you need to access a Base2 virtual function, use dynamic_cast to do a cross-cast.

- Run-time adjustments to the addresses in the pointers can be necessary - so some run-time cost of using MI.
• Duplicated bases - diamond shaped inheritance - produces ambiguities.
  ● "Dreaded diamond" inheritance pattern
  ● Base, D1 isa Base, D2 isa Base, DD is a D1 and isa D2
    ● Duplicated Base members - ambiguity - has two Base subobjects
    ● have to explicitly qualify which one you mean in the D1 and D2 classes
Example code

```cpp
#include <iostream>

using namespace std;

class B {
public:
    B(int i_) : i(i_)
    {
        cout << "B " << i << " ctor" << endl;
        void print() {cout << "B has " << i << '\n' << endl;}
    private:
        int i;
    };

class D1 : public B {
public:
    D1(int i_, int j_) : B(i_), j(j_)
    {
        cout << "D1 " << j << " ctor" << endl;
    private:
        int j;
    };

class D2 : public B {
public:
    D2(int i_, int j_) : B(i_), j(j_)
    {
        cout << "D2 " << j << " ctor" << endl;
    private:
        int j;
    };

class DD : public D1, public D2 {
public:
    DD(int i_, int j_, int k_, int m_) : D1(i_, j_), D2(i_*10, k_), m(m_)
    {
        cout << "DD " << m << " ctor" << endl;
    private:
        int m;
    };

    int main()
    {
        B b(0);
        b.print();

        D1 d1(1, 11);
        d1.print();

        D2 d2(2, 22);
        d2.print();

        DD dd(3, 22, 33, 111);
        // dd.print(); // error - it's ambiguous!
        dd.D1::print();
        dd.D2::print();

        cout << "done" << endl;
    }
}
```

/*
B 0 ctor
B has 0
MultipleInheritanceRTTI.oo3

Duplicated bases - diamond shaped inheritance - produces ambiguities.

Example code

```cpp
#include <iostream>
using namespace std;

class B {
public:
    B(int i_) : i(i_) {cout << "B " << i << " ctor" << endl;}
    void print() {cout << "B has " << i << '
' << endl;}
private:
    int i;
};

class D1 : public B {
public:
    D1(int i_, int j_) : B(i_), j(j_) {cout << "D1 " << j << " ctor" << endl;}
private:
    int j;
};

class D2 : public B {
public:
    D2(int i_, int j_) : B(i_), j(j_) {cout << "D2 " << j << " ctor" << endl;}
private:
    int j;
};

class DD : public D1, public D2 {
public:
    DD(int i_, int j_, int k_, int m_) : D1(i_, j_), D2(i_*10, k_), m(m_) {cout << "DD " << m << " ctor" << endl;}
private:
    int m;
};

int main() {
    B  b(0);
    b.print();
    D1 d1(1, 11);
    d1.print();
    D2 d2(2, 22);
    d2.print();
    DD dd(3, 22, 33, 111);
    // dd.print(); // error - it's ambiguous!
    dd.D1::print();
    dd.D2::print();
    cout << "done" << endl;
}
```

```cpp
B 1 ctor
D1 11 ctor
B has 1

B 2 ctor
D2 22 ctor
B has 2

B 3 ctor
D1 22 ctor
B 30 ctor
D2 33 ctor
DD 111 ctor
B has 3

B has 30

done

*/
```
• virtual inheritance: Heavyweight solution to the "dreaded diamond" problem.
  • Remove ambiguity of Base subobject, but complicates object construction.
  • Use is not required in this course - but you might be able to try it out if you want.
  • virtual inheritance - construct a single shared subobject - very different from normal
    • complicates the addressing, causes some run-time overhead
    • has to be anticipated in the class design - not necessarily possible to retrofit the design
    • don't use if the design can be modified to make it unnecessary
  • D1 virtually inherits from Base, D2 virtually inherits from Base, DD isa D1 and isa D2
    • B becomes a virtual base class.
  • DD gets only one Base subobject
    • gets a pointer to a single chunk of memory for the base subobject
    • No longer any ambiguity about base object members
• Initialization problem: Does D1 or D2 initialize the Base subobject? Who controls it? Ambiguity is not allowed!
  • Answer: The leaf class has to invoke the base constructor of the virtual base class
    • violation of the normal immediate-base-only constructor rule
    • Any other attempts to invoke the base constructor are ignored!
    • Notice that depends on the type of object declared - most derived one gets to say what happens
  • See example - derived from bottom of diamond gets to initialize the base subobject
    • The D1, D2 initializations of Base do nothing!
• Run time cost: single base subobject is referred to by indirection - through a pointer - means a bit of run-time cost compared to a non-virtual inheritance.
• Complexity and overhead leads to recommendation to avoid using virtual inheritance except where it is actually needed - try to avoid!
• Example code and execution
  • #include <iostream>
    using namespace std;

class B {
public:
    B() : i(0) {
        cout << "B " << i << " default ctor" << endl;
    }
    B(int i_) : i(i_) {
        cout << "B ctor arg is " << i << endl;
    }
    void print() {cout << "B has " << i << endl;}
private:
    int i;
};

class D1 : virtual public B {
public:
    D1(int i_, int j_) : B(i_), j(j_) {
        cout << "D1 ctor args are " << i_ << " for B, and " << j_ << endl;
    }
    void print() {cout << "D1 has " << j << endl;}
private:
    int j;
};

class D2 : virtual public B {
public:
    D2(int i_, int j_) : B(i_), j(j_) {
        cout << "D2 ctor args are " << i_ << " for B, and " << j_ << endl;
    }
    void print() {cout << "D2 has " << j << endl;}
private:
    int j;
};

class DD : public D1, public D2 {
public:
    // DD(int i_, int j_, int k_, int m_) : D1(i_, j_), D2(i_*10, k_), m(m_) // error - tries to call B() constructor if not defined
    DD(int i_, int j_, int k_, int m_) : B(99), D1(i_, j_), D2(i_, k_), m(m_) {
        cout << "DD ctor args are " << i_ << " for D1, and " << j_ << " for D2, and " << k_ << " for D3, and " << m_ << endl;
    }
    void print() {cout << "DD has " << m << endl;}
private:
    int m;
};
class DDD : public DD {
public:
  // DDD(int i_, int j_, int k_, int m_, int n_) : DD(i_, j_, k_, m_), n(n_) // error - tries
to call B() constructor if not defined
  DDD(int i_, int j_, int k_, int m_, int n_) : B(999), DD(i_, j_, k_, m_), n(n_)
  {cout << "DDD ctor args are " << i_ << ", " << j_ << ", " << k_ << ", " << m_ << "" for DD, and " << n_ << endl;}
  void print() {cout << "DDD has " << n << endl;}
private:
  int n;
};

int main()
{
  cout << "\n  B  b(0);" << endl;
  B  b(0);
  b.print();

  cout << "\nD1 d1(10, 11);" << endl;
  D1 d1(10, 11);
  d1.B::print();
  d1.print();

  cout << "\nD2 d2(20, 22);" << endl;
  D2 d2(20, 22);
  d2.B::print();
  d2.print();

  cout << "\nDD dd(30, 32, 33, 333);" << endl;
  DD dd(30, 32, 33, 333);
  dd.B::print();
  dd.D1::print();
  dd.D2::print();
  dd.print();

  cout << "\nDDD ddd(40, 42, 43, 444, 4444)" << endl;
  DDD ddd(40, 42, 43, 444, 4444);
  ddd.B::print();
  ddd.D1::print();
  ddd.D2::print();
  ddd.DD::print();
  ddd.print();
  cout << "done" << endl;
}

Demo of non-default construction
B  b(0);
B ctor arg is 0
B has 0

D1 d1(10, 11);
B ctor arg is 10
D1 ctor args are 10 for B, and 11
B has 10
D1 has 11

D2 d2(20, 22);
B ctor arg is 20
virtual inheritance: Heavyweight solution to the "dreaded diamond" problem.

Example code and execution

```cpp
#include <iostream>
using namespace std;

class B {
public:
    B() : i(0) {
        cout << "B " << i << " default ctor" << endl;
    }
    B(int i_) : i(i_) {
        cout << "B ctor arg is " << i << endl;
    }
    void print() {cout << "B has " << i << endl;}

private:
    int i;
};

class D1 : virtual public B {
public:
    D1(int i_, int j_) : B(i_), j(j_) {
        cout << "D1 ctor args are " << i_ << " for B, and " << j_ << endl;
    }
    void print() {cout << "D1 has " << j << endl;}

private:
    int j;
};

class D2 : virtual public B {
public:
    D2(int i_, int j_) : B(i_), j(j_) {
        cout << "D2 ctor args are " << i_ << " for B, and " << j_ << endl;
    }
    void print() {cout << "D2 has " << j << endl;}

private:
    int j;
};

class DD : public D1, public D2 {
public:
    // DD(int i_, int j_, int k_, int m_) : D1(i_, j_), D2(i_*10, k_), m(m_) // error - tries to 
    // call B() constructor if not defined
    DD(int i_, int j_, int k_, int m_) : B(99), D1(i_, j_), D2(i_, k_), m(m_) {
        cout << "DD ctor args are " << i_ << ", " << j_ << " for D1, and " << i_ << ", " << k_ << " for D2, and " << m_ << endl;
    }
    void print() {cout << "DD has " << m << endl;}

private:
    int m;
};

class DDD : public DD {
public:
    // DDD(int i_, int j_, int k_, int m_, int n_) : DD(i_, j_, k_, m_), n(n_) // error - tries 
    // to call B() constructor if not defined
    DDD(int i_, int j_, int k_, int m_, int n_) : B(999), DD(i_, j_, k_, m_), n(n_) {
        cout << "DDD ctor args are " << i_ << ", " << j_ << ", " << k_ << ", " << m_ << " for DD, and " << n_ << endl;
    }
    void print() {cout << "DDD has " << n << endl;}

private:
    int n;
};

int main() {
    cout << "B  b(0);" << endl;
    B  b(0);
    b.print();
    cout << "D1 d1(10, 11);" << endl;
    D1 d1(10, 11);
    d1.print();
    cout << "D2 d2(20, 22);" << endl;
    D2 d2(20, 22);
    d2.print();
    cout << "DD dd(30, 32, 33, 333);" << endl;
    DD dd(30, 32, 33, 333);
    dd.print();
    cout << "DDD ddd(40, 42, 43, 444, 4444)" << endl;
    DDD ddd(40, 42, 43, 444, 4444);
    ddd.print();
    cout << "done" << endl;
}
```

/*
Demo of non-default construction
*/
But can get ambiguity about virtual functions.

- If Base declares a virtual function, and both D1 and D2 override it, and DD does not, then DD can't be compiled - the compiler reports more than one overriding definition, and it has no basis for resolving the ambiguity.

```cpp
#include <iostream>
using namespace std;

class B {
public:
  B() : i(-1) {cout << "B default ctor" << endl;}
  B(int i_) : i(i_) {cout << "B ctor arg is " << i << endl;}
  void print() {cout << "B has " << i << endl;}
  virtual void avf() {cout << "B avf called" << endl;}
private:
  int i;
};

class D1 : virtual public B {
public:
  D1(int i_, int j_) : B(i_), j(j_) {
    cout << "D1 ctor args are " << i_ << " for B, and " << j_ << endl;
  }
  void print() {cout << "D1 has " << j << endl;}
  void avf() override {cout << "D1 avf called" << endl;}
private:
  int j;
};

class D2 : virtual public B {
public:
  D2(int i_, int j_) : B(i_), j(j_) {
    cout << "D2 ctor args are " << i_ << " for B, and " << j_ << endl;
  }
  void print() {cout << "D2 has " << j << endl;}
  // following causes error in DD declaration unless DD also overrides it
  void avf() override {cout << "D2 avf called" << endl;} // declaration of DD error
private:
  int j;
};

class DD : public D1, public D2 {
public:
  DD(int i_, int j_, int k_, int m_) : D1(i_, j_), D2(i_*10, k_), m(m_) {cout << "DD ctor args are " << i_ << "", " << j_ << " for D1, and " << k_ << " for D2, and " << m_ << endl;}
  void print() {cout << "DD has " << m << endl;}
  // following required
  void avf() override {cout << "DD avf called" << endl;}
private:
  int m;
};
```
RTTI

- Compiler can generate information that identifies the class of an object
  - made available in the vtable, so doesn't take up any more space in the object
  - but is only available for classes that have virtual functions - only time a vtable is present.
  - makes sense because only if virtual functions are present is there any issue about the type of an object being unknown ....
  - the one place in the language where dynamic type rather than static type is involved

- One use is kinda handy - identifying the actual type of an object for debugging purposes:
  - typeid operator (an operator, not a function)
  - normal use:
    - typeid(<an object>) - returns a reference to a type_info object that has information about the exact type of the object - can be compared, ordered, etc.
    - A name() function returns a pointer to a C-string containing the type name in implementation-defined form.
    - use to print out the type of an object at run time:
      - Base * ptr;
      - ptr = a pointer to some object in the class hierarchy
      - if(trace)
        - cout << typeid(*ptr).name() << " being processed" << endl;
      - the string returned by name is implementation-defined, so might not be good for an end user to see, but certainly useful for debugging, analysis by developer.

- Sometimes the design is such that you need to interrogate an object to see what its type is:
  - Virtual functions are better, but sometimes they won't do the trick.
  - Fat-interface dilemma.
  - Crummy design situation that you can't avoid or fix.
    - You're stuck with branch-on-type programming.
  - Don't use your own type codes, use RTTI instead
  - Usual procedure is to use a dynamic_cast to determine the type of an object
Kinds of casts used in inheritance situations

- static_cast<> can cast up or down, but is not checked
  - upcast:
    - Base * p = static_cast<Base *>(derived_ptr);
    - upcast is redundant, explicit cast is poor style, since always valid - Substitution principle
    - Base * p = derived_ptr; // instead
  - downcast:
    - Derived * p = static_cast<Derived *>(base_ptr);
    - not checked, so undefined if base_ptr does not actually point to a Derived object

- dynamic_cast<> can cast up, down, and "sideways" (cross-cast, when multiple inheritance is involved)
  - Up-cast with dynamic_cast is redundant for same reasons as upcast with static_cast.
  - Downcast can be checked for validity
    - Derived * p = dynamic_cast<Derived *>(base_ptr)
      if(p)
        // p is guaranteed to point to a Derived
      else
        // p doesn't point to a Derived
    - can be used for compiler-supported branch-on-type programming if you are forced into it.
  - Dynamic casts used to solve fat interface problem:
    - Instead of fat interface, see if you can downcast safely; if so, can access the derived interface safely.
      - if(Derived * p = dynamic_cast<Derived *>(base_ptr))
        // use p to access something found only in Derived interface
      else
        // can't validly access Derived interface
  - Cross-cast example - check to see if object in one hierarchy (pointed to be Base1 * base1_ptr) is also in another (Base2).
    - Base1 * base1_ptr = some object's address;
    - Base2 * base2_ptr = dynamic_cast<Base2 *>(base1_ptr)
      if(base2_ptr)
        // base1_ptr (and now base2_ptr) indeed points to a Base2-derived object as well as a Base1-derived object.
      else
        // base1_ptr (and also base2_ptr) does not point to a Base2-derived object

- See capability class discussion ...