CPSC 5011: Object-Oriented Concepts

Lecture 9: More Polymorphism, Heterogeneous Collections

Polymorphism

- General Definition
  - Same name
  - Meaning depends on type
  - Choice of function resolved by
    - Type(s) of passed parameters
    - Type of `this` pointer (object through which function invoked)
- Benefits
  - Uniform interface
  - Flexibility
  - Code reuse
  - “automatic” type resolution
    - ⇒ Extensibility
    - ⇒ Maintainability

Three Types of Polymorphism

- *Ad hoc Polymorphism* (*Overloading*)
    - Operator or procedure works on arguments of different types
    - Several different functions, all with same name
        - Function signature used to resolve call
    - Function to invoke determined at compile-time
- *Parametric Polymorphism* (*Generics*)
    - Parametric overloading
        - Methods or procedures used in same context
        - share a name
        - disambiguated by the number and type of arguments supplied
    - Explicit parameter used to denote (generic) type
        - type placeholder
    - Different versions of definition created when actual type provided

Three Types of Polymorphism

Inclusion – Inheritance of Interface (Subtyping)

- Uniform interface
- Different behavior through polymorphic object
  - Handle can contain reference to different subtypes
  - Dynamic binding
- Operator or method applied to different objects from different subclasses in an inheritance hierarchy
  - Parent-child relationship required
    => substitutability
  - Sibling relationships not relevant
- **Procedure invoked determined at run-time**
  - Flexible
  - Overhead

More on Polymorphism

- Dynamic Binding
  - Postponement of function call resolution until run-time
- Requires *virtual functions*
  - Polymorphic methods
  - Overridden in derived class(es)
    - Same signature
  - Type of `this` pointer not known until run-time
  - Function to invoke determined at run-time
  - Compiler uses hidden virtual function table (vtab) to retrieve function address at run-time

Once Virtual Always Virtual

- Address of each virtual function placed in class vtab
  - “vtab” = virtual function table = jump table
- When defined, descendant class inherits copy of parent’s vtab
  - Virtual functions have same offset in all hierarchy vtabs
  - Every virtual function remains virtual for descendants.
- Each overridden function will cause an address update to the corresponding entry in the descendant vtab.

Java versus C++ Polymorphism

- Java: dynamic binding is default
- C++: static (compile-time) binding is default
  - Dynamic (run-time) binding must be explicitly chosen
  - Requires
    - declaration of virtual function in base class AND
    - use of base class pointers in application code

C++ : Dynamic (run-time) Binding

```
// C++ dynamic (run-time) binding: 3 requirements
//   pointer/reference typed to BASE class of inheritance
//   hierarchy virtual functions declared in BASE class
//   => late (dynamic) binding call to virtual member
//   function through BASE class pointer
class Base {
public:
   virtual int corners() { return 4; }
   virtual void show() { cout << "Base"; }
};
class First: public Base {
public:
   virtual void show() { cout << "First"; Base::show(); }
};
class Second: public Base {
public:
   virtual void show() { cout << "Second";
    Base::show(); }
};
class Third: public Base {
public:
   virtual void show() { cout << "Third"; Base::show(); }
};
class Home: public Base {
public:
   virtual int corners() { return 5; }
   virtual void show() { cout << "Home"; Base::show(); }
};
```

```
int main() {
   First    F;
   Second   S;
   Third    T;
   Home     H;
   Base *   Bptr;

   Bptr = &F;
   Bptr->show();
   Bptr->corners();
   Bptr = &S;
   Bptr->show();
   Bptr->corners();
   Bptr = &T;
   Bptr->show();
   Bptr->corners();
   Bptr = &H;
   Bptr->show();
   Bptr->corners();

   // what if show() were not
   // virtual in Base class?
   // what if corners() were not
   // virtual in Base class?

   return 0;
}
```

C++ : Utility of Polymorphism

```
// Utility of polymorphism
//  uniform manipulation of
//  heterogeneous objects
//  handle a variety of objects
//  without knowledge of subtype
class BaseStack {
…
public:
   BaseStack()      {    …    }
   void push(Base * item)   {    …    }
   Base * pop()         {    …    }

};

// ANY CONTAINER (including ARRAY)
// that holds base class pointers
// (references)
// EASILY supports polymorphic behavior
```

```
// significant benefit of polymorphism:
// support of heterogeneous collections
int main() {
   First   F;
   Second  S;
   Third   T;
   Home       H;
   BaseStack   BStack;

   BStack.push(&F);
   BStack.push(&S);
   BStack.push(&T);
   BStack.push(&H);

   Base *  temp;

   // static type of object popped off
   // stack is Base * value printed
   // dependent on actual (dynamic) type
   while (temp = BStack.pop())
      temp->show();

   return 0;
}
```

Heterogeneous Collections

- Tied to interface of base class in class hierarchy
  - Available functionality defined by public functions of base class
- Contains assortment of objects
  - Each object some (sub)type of class hierarchy
  - No required order or frequency of (sub)types
- Elements of collection are actually address holders
  - Pointer in C++
- Dynamic binding
  - Postpones function resolution until run-time so actual subtype used
  - Great flexibility and extensibility!!
- Class hierarchy uses virtual functions
  - Variant behavior (based on subtype)
- Traversal of heterogeneous collection
  - Yields streamline execution of varying functionality
  - Masks subtype construction and evaluation
- Type extensibility supported

C++: Polymorphic Object Creation

```
// Function that evaluates environment, possibly file input
//  generates an object of some type from class hierarchy
//   => can return address of any object from class hierarchy
//   => subtype of object allocated determined at run-time
// BASE pointer can hold address of ANY class hierarchy object
//   at compile-time:
//   => return pointer holding address generated at run-time
//   => cannot 'guess' what (sub)type of object allocated
FirstGen * GetObjAddr() {
   if (condA)          return new FirstGen;       // base
    else if (condB)    return new SecondGen;      // derived
    else if (condC)    return new ThirdGen;       // derived
    …
}   // ownership of object passed back
```

```
// initialization of heterogeneous collection:  subtype hidden
// at compile-time, do NOT know type of object  generated
FirstGen * bigPtrArray[100];
for (int k = 0; k < 100; k++)      bigPtrArray[k] = GetObjAddr();

// dynamic behavior
for (int k = 0; k < 100; k++)      bigPtrArray[k]-> simple();
…
// MEMORY MANAGEMENT: release heap memory before leaving scope
// deallocate dynamically allocated objects
for (int k = 0; k < 100; k++)      delete   bigPtrArray[k];
```

C++: Parameter Passing

```
// Polymorphism and Parameter Passing
//   => pass by value slices
//   => pass by reference (pointer) retains dynamic type
class Base {
public:
   virtual void msg() { cout << "Base" << endl; }
};

class Derived: public Base {
public:
   virtual void msg() { cout << "Derived" << endl; }
};
```

```
void passByValue(Base    x) {   x.msg();  }
void passByRef(Base& x) {   x.msg();  }

int main() {
   Derived   d;
   passByValue(d);    // d sliced to Base object
                      //    => Base::msg() invoked
   passByRef(d);      // reference holds address of d
                      //    => Derived::msg() invoked
   return  0;
}
```

C++: Internal Dynamic Memory

```
class Base {
public:
   Base() {
      ptrB = new int[10];
      cout << "Base allocates 10 integers" << endl;
   }
   ~Base() {
      delete [] ptrB;
      cout << "Base deallocates 10 integers" << endl;
   }
private:
   int * ptrB;
};
class Child: public Base {
public:
   Child() {
      ptrD = new int[100];
      cout << "Child allocates 100 integers" << endl;
   }
   ~Child() {
      delete [] ptrD;
      cout << "Child deallocates 100 integers" << endl;
   }
private:
   int * ptrD;
};
```

```
int main() {
   Base*        ptr = new Child;

   delete    ptr;
   // destructor invoked

   // Base class destructor
   // statically resolved
   //  destructor non-virtual
   //  => Derived destructor
   //      not invoked!!
   //  => MEMORY LEAK

   return  0;
}
```

Destructors

- Fire from Derived to Base
  - Reverse order of constructors
- Object goes out of scope
  - If object of type Base, *only* Base destructor invoked
- `delete` operator invoked thru handle
  - If handle of Base type, *only* Base destructor invoked
  - If handle of Derived type, *both* Derived and Base destructor invoked

Destructors

- Derived class allocates heap memory
- Base class pointer holds address of derived object
  - Heterogeneous collections
- Destructors statically resolved
  - Derived destructor not invoked through base handle
- **Application programmer FOLLOWS the RULES**

⇒ Dynamic resolution of destructor invocation

- ⇒ base class pointer holds address of derived object
- ⇒ type examined at run-time
- ⇒ Derived destructor invoked first then Base destructor invoked
- ⇒ Requires VIRTUAL DESTRUCTOR

C++: Internal Dynamic Memory (revisited)

```
// Polymorphism and Internal Dynamic Memory
//     Problem when pointer holds address of derived class object AND
//     derived class object has allocated heap memory => memory leak
//     Solution: make Base destructor virtual
//         => call to destructor is dynamically bound
class Base {
public:
   Base() {
      ptrB = new int[10];
      cout << "Base allocates 10 integers" << endl;
   }
   virtual ~Base() { // ONLY CHANGE
      delete [] ptrB;
      cout << "Base deallocates 10 integers" << endl;
   }
private:
   int*    ptrB;
};

// Child class definition unchanged - ONCE virtual ALWAYS virtual
```

```
int main() {         // application code the same
   Base * ptr = new Child;
   delete  ptr;      // no memory leak
   return  0;
}
```

Polymorphism: Advantages

- Flexibility
  - Function call resolution postponed until run-time
- Language Support => Stable software
  - Application code need not perform "manual" type checks
  - Base class need not check subtype
  - Compiler's responsibility to generate code to resolve call correctly
- Support of Heterogeneous collections
  - Uniform interface
- Substitutability
  - Possible to substitute child objects for parent objects
- Extensibility
  - Additional child classes may be added to inheritance hierarchy
  - ⇒ Additional overridden version of base virtual function
  - ⇒ Minimal change in application code
    - For example, consider adding a "`class Four`"
      - ⇒ Another version of `whoami()`
      - ⇒ Other than object construction, no change to application code

Polymorphism: Disadvantages

- Overhead
  - Run-time type check to resolve function call
  - Extra compiler-generated instructions (3-4)
  - Extra space to store function pointers
    - Virtual function table
- Performance impact
  - Efficient implementation of run-time resolution
  - **Function calls cannot be in-lined**

Java

- Cannot choose efficiency of static binding
- All calls dynamically bound
- Slow
  - typically, 2 to 3 times slower than C++

C++

- Provision of both static & dynamic binding
  - Code complexity, both types of calls present
  - Confusion
- Must anticipate need for dynamic binding
  - Function MUST be declared virtual in base class
- OR must use RTTI (run-time type identification)
  - Ugly, non-extensible fix (dynamic_cast)