Advanced Programming Concepts with C++ CSI2372 – Fall 2019

Jochen Lang &
Mohamed Taleb
EECS

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

This lecture

OO

- **Object-oriented design**
  - Assignment Operator
    - Copy control
    - Copy control with hierarchies
  - Exceptions Ch. 18.1
  - Static attributes and methods, Ch. 7.6
  - Inline functions, Ch. 6.5.2
  - Friend operator, Ch. 7.2.1

Review: Copy Constructor vs. Assignment Operator

- Copy constructor creates a new object

```
Point2D pt1( 3.0, 4.0 );
Point2D pt(pt1);
```

  - Creates a new object **pt** by calling the copy constructor. **pt1** is a Point2D (same type than **pt**) which existed before the call.

- Assignment operator makes two existing objects the same

```
Point2D pt, pt1( 3.0, 4.0 );
pt = pt1;
```

  - Copies the content of an existing object **pt1** to another existing object **pt**

→ Both are synthesized by the compiler!

Review: Deep Copy

- Consider the following class with a pointer member

```
class Stack {
  int d_capacity, d_size;
  string* d_stack;
public:
  Stack( int _capacity = 10 ) :
       d_capacity{_capacity}, d_size{0},
       d_stack{new string[_capacity]}
  {}
  ~Stack(){ delete [] d_stack;}
  Stack& push( const string& s);
  string pop();
  string top() const;
  void print() const;
};
```

Review Deep Copy

Stack example without defining a deep copy is in error

- Define a deep copy

```
Stack::Stack( const Stack& oS ) :
      d_capacity{oS.d_capacity}, d_size{oS.d_size} {
    d_stack = new string[d_capacity];
    for ( int i=0; i<d_size; ++i ) {
      d_stack[i] = oS.d_stack[i];
    }
  }
```

Rule of 3/5

If a class needs a non-default copy constructor, it also needs a non-default destructor and assignment operator

- **Assignment operator prototype**
  - operator and not a constructor as we are assigning to an existing object
  - return type is a reference to the assigned to object as we want to chain assignment

```
Stack& Stack::operator=( const Stack& oS )
```

- **Rule of 3 has become rule of 5 in some cases with C++11 for move ctor and move assignment (to be discussed later)**

Deep Assignment

- **Must check for self assignment!**

```
Stack& Stack::operator=( const Stack& oS ) {
    if ( this != &oS ) {
      delete [] d_stack;
      d_size = oS.d_size;
      d_capacity = oS.d_capacity;
      d_stack = new string[d_capacity];
      for ( int i=0; i<d_size; ++i ) {
        d_stack[i] = oS.d_stack[i];
      }
    }
    return *this;
  }
```

Review: Copy Constructor and Class Hierarchies

- **Default Copy Constructor**
  - Calls copy constructor of base class first
- **Defined copy constructor**
  - Must explicitly call copy constructor of base class

```
class House : protected Building {
  …
public:
  House( const House& _oHouse )
    : Building{_oHouse}, d_noOccu{_oHouse.d_noOccu} {}
};
```

Assignment Operator and Class Hierarchies

- **Default assignment operator**
  - Calls assignment operator of base class first
- **Defined assignment operator**
  - Must explicitly call assignment operator of base class

```
class House : protected Building {
public:
  const House& operator=( const House& _oHouse ) {
    // Should always check against self-assignment
    if ( this != &_oHouse ) {
      Building::operator=( _oHouse );
      d_noOccu = _oHouse.d_noOccu;
    } return *this; }
};
```

Exceptions

- Key concept in object-oriented programming
- Supports Robustness

- **Advantages**
  - Code where the error occurs and code to deal with the error can be separated
  - Exceptions can be used with constructors and other functions/operators which can not return an error code
  - Properly implemented exceptions lead to better code

Basic Exception Concepts

- **try**
  - Try executing some block of code
  - See if an error occurs
- **throw**
  - An error condition occurred
  - Throw an exception
- **catch**
  - Handle an exception thrown in a try block

C++ Exception Syntax

– Syntax is again very similar to Java
– Except for empty throw (rethrows the currently handled exception) and catch(…) (catch all)

```
try-block:
    try compound-statement handler-list
handler-list:
    handler handler-list_opt
handler:
    catch ( exception-declaration ) compound-statement
exception-declaration:
    type-specifier-list declarator
    type-specifier-list abstract-declarator
    type-specifier-list
    ...
throw-expression :
    throw assignment-expression_opt
```

An Example

```
size_t szA; int* iA;
try {  // try block
  cin >> szA;
  if ( cin.fail() ) {
    string line; getline(cin,line); throw line;
  }
  iA = new int[szA];
  cout << "Array of size " << szA
       << " successfully allocated." << endl;
  delete[] iA;
} catch ( string inLine ) {
  cerr << "Error: Not an integer:" << inLine <<endl;
  throw; // re-throw exception
} catch (...) { // Catch anything else
}
```

– Note: In C++, the argument for throw can be of any type. No requirement for it to be a subclass of an exception.

Static Members

- **Static class attributes**
  - Sharing a variable between all instances of a class
  - Same concept than a static variable in a function
- **Static class methods**
  - Global functions; static member functions exist without object
    - no object to access, no this, no non-static attributes, no non-static methods (similar to Java)
  - Access modifiers can be applied
- **Note:**
  - Static variables are not initialized in a constructor but default initialized the same way as global variables

Initialization of Static Class Variables

- **Static class variables must be defined and initialized outside the class**
  - Might be used without an object of the class!
- **Useful convention**
- **Declare in header file (as usual):**

```
class MountainBike {
  static const float WHEELSIZE; …
}
```

- **Define in cpp file OUTSIDE any method!**

```
static const float MountainBike::WHEELSIZE = 26.0f;
```

In-class Initialization of Static Class Variables

- **const Types initialized from constant expression can be initialized in the class**
  - Before C++11 only const integral and enumeration types could be initialized in class with a constant expression
  - use constexpr to clarify
  - can only use literal types (e.g., no strings)

```
class MountainBike {
  static constexpr float WHEELSIZE = 26.0f;
…
}
```

Inline Functions

- **Inline functions (methods) avoid overhead for function call at run-time**
  - Inline functions (methods) are “copied” and “pasted” into code
  - Access methods should (typically) be inlined to avoid overhead of function calls
- **Example**

```
class Matrix3D {
  double d_elements[9];
public:
  inline double& element( int _row, int _col );
}
```

Restrictions on Inlining

- **Inline method must be available when used**
  - Define in header file together with declaration
  - 2 possible variations, use the second (separation of class functionality and method implementation.)

```
// header file
class Matrix3D {
  double d_elements[9];
  inline void element( int _row, int _col, double _val ) {
    d_elements[ _row * 3 + _col ] = _val;
  }
  inline double element( int _row, int _col );
}
double Matrix3D::element( int _row, int _col ) {
  return d_elements[ _row * 3 + _col ]; }
```

More Restrictions on Inlining

- **Inline is a compiler directive**
  - Inlining can save substantial overhead, function calls are expensive
  - Compiler may choose to ignore inline
  - Compiler switches are important, e.g., in Visual C++ debug mode methods are usually not inlined
    - Often makes debug mode useless for matrix and image classes which use a lot of inlined accessor methods

Friends

- **Friend keyword changes access rights**
  - Friend can be applied to classes, global or member functions and global or member operators
- **Application**
  - A set of classes which deal with a common issue
    - Similar to java package accessibility
- **Example**

```
class Matrix3D;
class Vector3D {
  friend class Matrix3D;
…
}
```

Example: Friendly Matrix Vector Multiply

```
class Vector3D {
  friend class Matrix3D;
  double d_components[3]; … }

class Matrix3D {
  double d_elements[9];  … }

Vector3D Matrix3D::Multiply( Vector3D& _vec ) {
  Vector3D res;
  for ( int row=0; row<3; row++ ) {
    res.d_components[row] = 0.0;
    for ( int col=0; col<3; col++ ) {
      res.d_components[row] += d_elements[row*3+col] *
         _vec.d_components[col];
  }}
  return res;
}
```

Less Friendly

- **Friend keyword can be applied to a specific function or operator**
  - limits access to protected and private members to specific operator or function
- **Example**

```
class Matrix3D;
class Vector3D {
  friend Vector3D Matrix3D::Multiply( Vector3D& );
}
```

- **Note: Previous implementation example works with the above declaration as well**

Limitation of Friendship

- **Friend is not inherited, e.g.:**
  - B has friend access to A
  - childA is derived class from A
  - childB is derived class from B
  - childB cannot access A
  - childA cannot be accessed by B
- **Friend is not transitive, e.g.:**
  - C has friend access to B
  - B has friend access to A
  - C does not have access to A

```
class B;
class A {
  friend class B;
}
class childA : A;
class childB : B;
```

```
class C;
class B {
  friend class C;
}
class A {
  friend class B;
}
```

Next

OO

- **Object-oriented design**
  - Polymorphism
    - Virtual Functions, Ch. 15.3, 15.7
    - Abstract classes, Ch. 15.4
    - Dynamic cast, Ch. 19.2.1