Lecture 6

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Copy Control – C++ Primer Chapter 13

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Copy control

copy control: Special members that control what happens when objects of class type are copied, assigned, and destroyed.

Rule of Three/Five: Rule of thumb: if a class needs a nontrivial destructor then it almost surely also needs to define its own copy constructor, an assignment operator and in the C++11 standard move constructor + move-assignment operator.

Copy constructor I

- **copy constructor**: Constructor that initializes a new object as a copy of another object of the same type.
  - Copy constructor is applied implicitly to pass objects to/from a function by value.
  - If we do not define the copy constructor, the compiler synthesizes one for us.
- The copy constructor is used to
  - Explicitly or implicitly initialize one object from another of the same type
  - Copy an object to pass it as an argument to a function
  - Copy an object to return it from a function
  - Initialize the elements in a sequential container
  - Initialize elements in an array from a list of element initializers

Copy constructor II

- **synthesized copy constructor**: The copy constructor created (synthesized) by the compiler for classes that do not explicitly define the copy constructor.
  - The synthesized copy constructor memberwise initializes the new object from the existing one.
- **memberwise initialization**: Term used to described how the synthesized copy constructor works.
  - The constructor copies, member by member, from the old object to the new.
  - Members of built-in or compound type are copied directly.
  - Those that are of class type are copied by using the member's copy constructor.
- Hint: to prevent copies, a class must explicitly declare its copy constructor as private (or = delete it)

Assignment Operator I

- **assignment operator**: The assignment operator can be overloaded to define what it means to assign one object of a class type to another of the same type.
  - The assignment operator must be a member of its class and should return a reference to its object.
  - The compiler synthesizes the assignment operator if the class does not explicitly define one.

Assignment Operator II

- **synthesized assignment operator**: A version of the assignment operator created (synthesized) by the compiler for classes that do not explicitly define one.
  - The synthesized assignment operator memberwise assigns the right-hand operand to the left.
- **memberwise assignment**: Term used to describe how the synthesized assignment operator works.
  - The assignment operator assigns, member by member, from the old object to the new.
  - Members of built-in or compound type are assigned directly.
  - Those that are of class type are assigned by using the member's assignment operator.

Destructor I

- **Destructor**: Special member function that cleans up an object when the object goes out of scope or is deleted.
  - The compiler automatically destroys each member.
  - Members of class type are destroyed by invoking their destructor
  - no explicit work is done to destroy members of built-in or compound type.
  - In particular, the object pointed to by a pointer member is not deleted by the automatic work done by the destructor.
- Example
  - Folder
  - ~MyClass() {}

Operator Overloading – C++ Primer Chapter 14

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Defining an Overloaded operator

- **overloaded operator**: Function that redefines one of the C++ operators to operate on object(s) of class type.
- Overloaded operators must have an operand of class type
  - This rule enforces the requirement that an overloaded operator may not redefine the meaning of the operators when applied to objects of built-in types
- Precedence and associativity are fixed
- Short-circuit evaluation is not preserved
- Overloaded functions that are members of a class may appear to have one less parameter than the number of operands
  - Operators that are members have an implicit this pointer that is bound to the first operand
- http://www.cppreference.com/wiki/operator_precedence

Overloaded operator design

- Do not overload operators with built-in meanings
  - It is usually not a good idea to overload the comma, address-of, logical AND, or logical OR operators
  - If you nevertheless do so, the operators should behave analogously to the synthesized operators
- Classes that will be used as key type of an associative container should define the < and == operator
  - In most cases then it is also a good idea to define the >, <=, >=, != operators
- When the meaning of an overloaded operator is not obvious, it is better to give the operation a name

Member vs. Nonmember Implementation

- The operators =, [], (), -> must be defined as members
- Like assignment, compound assignment operators ordinarily ought to be members of the class
- Other operators that change the state of their object or that are closely tied to their given type – such as increment, decrement, and dereference –usually should be members
- Symmetric operators, such as the arithmetic, equality, relational, and bitwise operators are best defined as ordinary nonmember functions

Input and output operators I

- Try to be consistent with the standard IO library
- Output operators should print the contents of the object with minimal formatting, they should not print a newline
- IO Operators must be nonmember functions
  - If they would be members, the left-hand operand would have to be an object of our class type
  - However, it is an istream or ostream in order to support normal usage
- Example
  - friend std::istream& operator>>(std::istream&, Sales_item&);
  - friend std::ostream& operator<<(std::ostream&, const Sales_item&);

Input and output operators II

- Input operators must deal with the possibility of errors and end-of-file
- Handling input error
  - The object being read into should be left in a usable and consistent state
  - Set the condition states of the istream parameter if necessary

Arithmetic and Relational operators

- Notes
  - Implement also the compound assignment operators
  - Operator == and < are used by many generic algorithms
- Example
  - MyClass operator+(const MyClass& lhs,const MyClass& rhs);
  - bool operator==(const MyClass& lhs,const MyClass& rhs);

Assignment and subscript operators

- Assignment operators can be overloaded
- Assignment and subscript operators must be class member functions
- Assignment should return a reference to *this
- In order to support the expected behavior for nonconst and const objects, two versions of a subscript operator should be defined:
  - one that is a nonconst member and returns a reference and one that is a const member and returns a const reference

Member access operators

- Operator arrow must be defined as a class member function
- The overloaded arrow operator must return either a pointer to a class type or an object of a class type that defines its own operator arrow
- The dereference operator is not required to be a member, but usually it is a good design to make it one

Increment and Decrement operators

- For consistency with the built-in operators, the prefix operations should return a reference to the incremented or decremented object
  - MyClass& operator++();
- For consistency with the built-in operators, the postfix operations should return the old (unincremented or undecremented) value
  - That value is returned as a value, not a reference
  - MyClass operator++(int);

Conversion operators

- **conversion operators**: Conversion operators are member functions that define conversions from the class type to another type.
  - Conversion operators must be a member of their class.
  - They do not specify a return type and take no parameters.
  - They return a value of the type of the conversion operator.
  - That is, operator int returns an int, operator MyClass returns a MyClass, and so on.
- Example
  - Operator int() const { return ival; }

class types conversions I

- **class-type conversion**: Conversions to or from class types.
  - Non-explicit constructors that take a single parameter define a conversion from the parameter type to the class type.
  - Conversion operators define conversions from the class type to the type specified by the operator.
- Why conversions are useful
  - Supporting mixed-type expressions
  - Conversions reduce the number of needed operators

class types conversions II

- The compiler automatically calls the conversion operator
  - In expressions: obj >= dval
  - In conditions: if (obj)
  - When passing an argument to or returning values from a function: int i = calc(obj);
  - As operands in overloaded operators: cout << obj << endl;
  - In an explicit cast: ival = static_cast<int>(obj) + 3;

class types conversions III

- An implicit class-type conversion can be followed by a standard conversion type
  - obj >= dval // obj converted to int and then converted to double
- An implicit class-type conversion may not be followed by another implicit class-type conversion
- Standard conversions can precede a class-type conversion

Conversions and operators

- Never define mutually converting classes
- Avoid conversions to the built-in arithmetic types
- If you want to define a conversion to such a type
  - Do not define overloaded versions of the operators that take arithmetic types. If users need to use these operators, the conversion operation will convert objects of your type, and then the built-in operators are used
  - Do not define a conversion to more than one arithmetic type. Let the standard conversions provide conversions to the other arithmetic types