# Lecture 4

## Classes

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

# Classes I



**Class**: C++ mechanism for defining own abstract data types.

- Classes may have data, function or type members.
- Classes are defined using either the class or struct keyword.
- A class defines a new type and a new scope.

**member function**: Class member that is a function.

- Ordinary member functions are bound to an object of the class type through the implicit this pointer.
- Static member functions are not bound to an object and have no this pointer.

# Classes II

**access specifier**: defines if the following members are accessible to users of the class or only to friends and members

- Each specifier sets the access protection for the members declared up to the next label.
- Specifiers may appear multiple times within the class.
- public, private, or protected

# Data abstraction and Encapsulation I

**abstract data type**: data structure (like a class) using encapsulation to hide its implementation.

**data abstraction**: Programming technique that focuses on the interface to a type.

- Allows programmers to ignore the details of how a type is represented and to think instead about the operations that the type can perform.

# Data abstraction and Encapsulation II

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

**Encapsulation**: Separation of implementation from interface

encapsulation hides implementation details of a type

- In C++, encapsulation is enforced by preventing general user access to the private parts of a class.
- Access specifiers enforce abstraction and encapsulation

# Definitions



**Interface**: The operations supported by a type.

**Implementation**: The (usually private) members of a class that define the data and any operations that are not intended for use by code that uses the type.

# Interface and Implementation Rules

- Well-designed classes separate their interface and implementation, defining the interface in the public part of the class and the implementation in the private parts.
- Data members ordinarily are part of the implementation.
- Function members are part of the interface when they are operations that users of the type are expected to use and part of the implementation when they perform operations needed by the class but not defined for general use.

# Data abstraction and Encapsulation III



- Advantages of abstraction and encapsulation
  - Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object
  - The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code

# Class Definitions

- **incomplete type**: A type that has been declared (forward declaration) but not yet defined.
  - It is not possible use an incomplete type to define a variable or class member.
  - It is legal to define references or pointers to incomplete types.
- Hints:
  - Use typedefs/using to streamline classes
  - Use only few (overloaded) member functions
  - Prefer inlined member functions (even it is only a hint to the compiler)
- Why a class definitions ends with a semicolon:
  - class Sales_item { /* … */ } accum, trans;

# Const and Mutable data members



**const member function**: member function that may not change an object's ordinary (i.e., neither static nor mutable) data members.

- The *this* pointer in a const member is a pointer to const.
- A member function may be overloaded based on whether the function is const.

**mutable data member**: Data member that is never const, even when it is a member of a const object.

- A mutable member can be changed inside a const function.

# Class Scope

- **class scope**: Each class defines a scope.
  - Class scopes are more complicated than other scopes—member functions defined within the class body may use names that appear after the definition.
  - Two different classes have two different class scopes (even if they have the same member list)
- Member definitions
  - double MyClass::accumulate() const { }
- Parameter lists and function bodies are in class scope
- Function return types are not always in class scope
  - MyClass::index MyClass::accumulate(index number ) const { }

# Name Lookup



- **Name lookup**: The process by which the use of a name is matched to its corresponding declaration
- Class definitions are processed in two phases
  - First, the member declarations are compiled
  - Only after all the class members have been seen are the definitions themselves compiled

# Constructors

**Constructor**: Special member function that is used to initialize newly created objects.

The job of a constructor is to ensure that the data members of an object have safe, sensible initial values.

# Constructors

- Constructors are special member functions that are executed whenever we create new objects of a class type
- Constructors have the same name as the class and may not specify a return type
- Constructors may be overloaded
- Constructors may not be declared as const

# Constructor Initializer List I



- **constructor initializer list**: Specifies initial values of the data members of a class.
  - The members are initialized to the values specified in the initializer list before the body of the constructor executes.
  - Class members that are not initialized in the initializer list are implicitly initialized by using their default constructor.
  - `Image( size_t cols = 0, size_t rows = 0 ) : cols_(cols), rows_(rows) {}`
- Members that must be initialized in the constructor list
  - Members of class type without default constructor
  - const or reference type members

# Constructor Initializer List II



- Members are initialized in order of their definitions
- Initializers may be any expression
- Initializers for data members of class type may call any of its constructors
- Hint: prefer to use default arguments in constructors because this reduces code duplication

# Default Constructor



- **default constructor**: The constructor that is used when no initializer is specified.
- **synthesized default constructor**: The default constructor created (synthesized) by the compiler for classes that do not define any constructors.
  - This constructor initializes members of class type by running that class's default constructor
  - members of built-in type are uninitialized.
- Common mistake when trying to use the default constructor:
  - MyClass myobj(); // defines a function, not an object!

# Sales_item.h: class definition revisited

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

```
#pragma once

#include <iostream>
#include <string>

class Sales_item {
friend bool operator==(const Sales_item&, const Sales_item&);
public:
    Sales_item(const std::string &book):
              isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);

    Sales_item& operator+=(const Sales_item&);

    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;

};
```

# Implicit class-type conversions

- **conversion constructor**: A nonexplicit constructor that can be called with a single argument.
  - A conversion constructor is used implicitly to convert from the argument's type to the class type
  - MyClass(string &s) {}
- **explicit constructor**: Constructor that can be called with a single argument but that may not be used to perform an implicit conversion.
  - A constructor is made explicit by prepending the keyword explicit to its declaration.
  - explicit MyClass( string &s ) {}

# 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

# 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
  - `~MyClass() {}`

# 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

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

- 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);`

# Sales_item.h: class definition (revisited)

```
#ifndef SALESITEM_H
#define SALESITEM_H

#include <iostream>
#include <string>

class Sales_item {
friend bool operator==(const Sales_item&, const Sales_item&);
public:
    Sales_item(const std::string &book):
              isbn(book), units_sold(0), revenue(0.0) { }
    Sales_item(std::istream &is) { is >> *this; }
    friend std::istream& operator>>(std::istream&, Sales_item&);
    friend std::ostream& operator<<(std::ostream&, const Sales_item&);

    Sales_item& operator+=(const Sales_item&);

    double avg_price() const;
    bool same_isbn(const Sales_item &rhs) const
        { return isbn == rhs.isbn; }
    Sales_item(): units_sold(0), revenue(0.0) { }
private:
    std::string isbn;
    unsigned units_sold;
    double revenue;

};

#endif
```

# Sales_item.h: class implementation I



```
Sales_item operator+(const Sales_item&, const Sales_item&);

inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue &&
          lhs.same_isbn(rhs);
}

inline bool
operator!=(const Sales_item &lhs, const Sales_item &rhs)
{
    return !(lhs == rhs); // != defined in terms of operator==
}

using std::istream; using std::ostream;

// assumes that both objects refer to the same isbn
inline
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

// assumes that both objects refer to the same isbn
inline
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);  // copy lhs into a local object that we'll return
    ret += rhs;           // add in the contents of rhs
    return ret;           // return ret by value
}
```

# Sales_item.h: class implementation II

```
inline
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    if (in)
        s.revenue = s.units_sold * price;
    else
        s = Sales_item();  // input failed: reset object to default state
    return in;
}

inline
ostream&
operator<<(ostream& out, const Sales_item& s)
{
    out << s.isbn << "\t" << s.units_sold << "\t"
        << s.revenue << "\t" <<  s.avg_price();
    return out;
}

inline
double Sales_item::avg_price() const
{
    if (units_sold)
        return revenue/units_sold;
    else
        return 0;
}
```

# 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

# Friends



- **Friend**: Mechanism by which a class grants access to its nonpublic members.
  - Both classes and functions may be named as friends.
  - friends have the same access rights as members.
  - A friend declaration introduces the named class or nonmember function into the surrounding scope
  - A friend function may be defined inside the class, then the scope of the function is exported to the scope enclosing the class definition

# Static class members

- **static member**: Data or function member that is not a part of any object but is shared by all objects of a given class.
- Advantages of static members compared to globals
  - The name of a static member is in the scope of the class, thereby avoiding name collisions with members of other classes or global objects
  - Encapsulation can be forced since a static member can be private, a global object cannot
  - It is easy to see by reading the program that a static member is associated with a particular class. This visibility clarifies the programmer’s intention
- Static member functions have no this pointer