C Object-oriented Programming

C.1 Overview

- Motivation for the OO paradigm
- Software-design methods
- Basic terms of OO programming
- Basic concepts of the OO paradigm

C.2 References

ABC83. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshot and R. Morrison, “An Approach to Persistent Programming”, *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, 1983.

Boo94. Grady Booch, *Object-Oriented Analysis and Design (with Applications)*, Benjamin/ Cummings, Redwood (CA), 1994.

CoY91a. P. Coad, E. Yourdon. Object-Oriented Analysis. Prentice Hall, 1991.

Coa91b. P. Coad, E. Yourdon. Object-Oriented Design. Prentice Hall, 1991.

Cox86. Brad J. Cox. *Object Oriented Programming.* Addison Wesley, 1986.

CW85. Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol. 17, No. 4, Dec. 1985.

Jac92. I. Jacobson. Object-Oriented Software Enineering — A Use Case Driven Approach. Addison-Wesley, 1992.

MaM88. Ole Lehrmann Madsen, Birger Møller-Pedersen, "What object-oriented programming may be — an what it does not have to be", *ECOOP ’88 – European Conference on OO Programming*, pp. 1 - 20, S. Gjessing, K. Nygaard [Eds.]; Springer Verlag, Oslo, Norway, Aug. 1988.

Mey86. Bertrand Meyer, "Genericity versus Inheritance", *Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA ’87*, pp. 391 - 405, Portland (Oreg., USA), published as *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986.

Mey88. Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall Inc., Hemel Hempstead, Hertfordshire, 1988.

C.2 References (2)

Oes97. B. Oestereich. Objektorientierte Softwareentwicklung: Analyse und Design. Oldenbourg, 1997.

Rum91. J. Rumbaugh. Object-Oriented Modelling and Design. Prentice Hall, 1991.

Str91. Bjarne Stroustrup. *The C++ programming language*, 2. ed., Addison-Wesley, 1991.

Str93. Bjarne Stroustrup, "A History of C++", ACM SIGPLAN Notices, Vol. 28, No. 3, pp.271 - 297, Mar. 1993.

Weg87. Peter Wegner, “Dimensions of Object-Based Language Design”, *OOPSLA ’87 – Conference Proceedings*, pp. 1-6, San Diego (CA, USA), published as *SIGPLAN Notices*, Vol. 22, No. 12, Dec. 1987.

Weg90. Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming”, *ACM OOPS Messenger*, No. 1, pp. 8-84, Jul. 1990.

C.3 Motivation for the OO Paradigm

1 Goals

- Increasing complexity of large software
  - "industrial-strength" software [Boo94]
    - ➤ impossible for one developer to comprehend all details of its design
    - ➤ very long life span
    - ➤ many users depend on their proper functioning
    - ➤ many people responsible for maintenance and enhancement

➔ Software crisis
  - Hardwareincreasingly capable
  - Software becomes larger and larger
  - Costs for maintenance and enhancement rise dramatically
  - Not enough good software developers to create the software users need

1 Goals (2)

- Increase the productivity of programmers
  - Design patterns for repeatedly occurring problems
  - Reusage of existing software
  - Better extensibility of software by modularization and clear interfaces
  - Incremental development from small & simple to huge & complex systems
  - Better control over complexity and costs of software maintenance
- Shift from the needs of the machine to abstractions of the problem domain
  - Better understanding of the problem
  - Terminology of the problem domain is reflected in the software solution
    - ➤ better understanding of the solution

C.4 Software-Design Methods

1 Classification [Boo94]

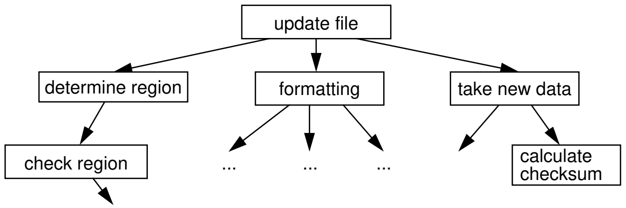
- Top-down structured design (composite design)
- Data-driven design
- Object-oriented design

2 Top-Down Structured Design (Composite Design)

- Units of decomposition: Subroutine
- **Algorithmic decomposition**
- Not suitable for structuring today's large and complex software systems
- Top-down structured design cannot describe:
  - – data abstraction & information hiding
  - – concurrency
- Problems arise when applications are very complex or when object-oriented languages have to be used
- Widely used technique

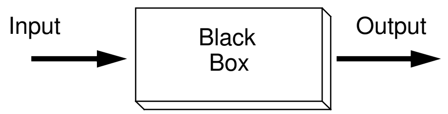
2 Top-Down Structured Design (2) (Composite Design)

- Example:



3 Data-driven Design

- Fundamental concepts from Jackson et al.
- Structure of software is based on mapping inputs to outputs
  ➔ Application mainly in information management
- Problems with time-critical events

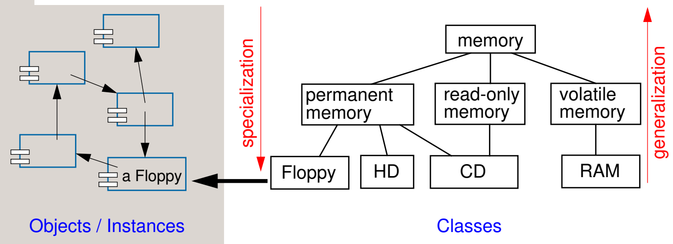


4 Object-oriented Design

Bertrand Meyer:[Mey88]
Computing systems perform certain actions on certain objects;
to obtain flexible and reusable systems,
it is better to base the structure of software
on the objects than on the actions.

4 ... Object-oriented Design (2)

- Software system is modeled as a collection of cooperating objects
- Each object is
  an instance of a class
  in a hierarchy of classes
- Example of a class hierarchy:



4 ... Object-oriented Design (3)

- Concepts reflected in the structure of modern programming languages
  - ➤ Smalltalk
  - ➤ C++
  - ➤ Eiffel
  - ➤ Java
  - ➤ Ada
- General basis: object-oriented decomposition
- Advantages:
  - + Reusage of common mechanisms
    ➔ software becomes smaller
  - + Modifications and improvements of the software become easier
  - + Results are less complex
  - + Better understanding of the principal's ideas

C.5 The Evolution of the Object Model

1 Generations of Programming Languages

- First generation (1954 - 1958)
  - ◆ Mathematical Expressions (FORTRAN I, ALGOL 58)
- Second generation (1959 - 1961)
  - ◆ Subroutines, separate compilation (FORTRAN II)
  - ◆ Block structure, data types (ALGOL 60)
  - ◆ Data description, file handling (COBOL)
  - ◆ List processing, pointers, garbage collection (Lisp)
- Third generation (1962 - 1970)
  - ◆ various successors to ALGOL 60 (ALOGOL 68, Pascal)
  - ◆ Classes, data abstraction (Simula)

2 Generations of Programming Languages (2)

- *The Generation Gap* (1970 - 1980)

  many different languages were invented,
  but few endured …
  - ◆ C
  - ◆ Modula
  - ◆ Ada
- Object-oriented programming languages (1980 - today)
  - ◆ Smalltalk (successor to Simula)
  - ◆ C++ (evolved from C and Simula)
  - ◆ Eiffel (evolved from Simula and Ada)
  - ◆ Java

C.6 Object-oriented Programming

1 Definition (Grady Booch)

OOP is a method of implementation in which programs are organized as

cooperative collections of objects,

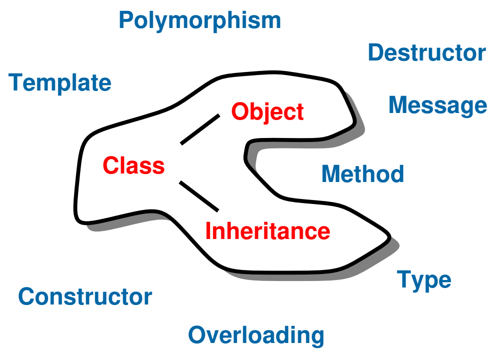
each of which represents an

instance of some class,

and whose classes are all members of a hierarchy of classes united via

inheritance relationships.

2 Basic Terms



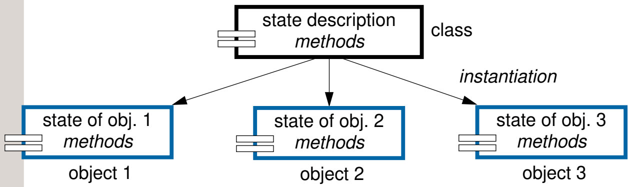
3 Objects & Methods

- Software developer's view:
  - ◆ an object is a "thing" from the problem domain
    - ➤ has a state
    - ➤ has behavior
    - ➤ has a unique identity
- Program-technical point of view:
    - ➤ an encapsulated unit of data and functions that operate on this data
    - ➤ an object has a clear interface (operations = **methods**)

➔ **object-based programming languages** [Weg87]

4 Classes

- Software developer's view:
  - ◆ a class is a set of objects with common structure and common behavior
- Program-technical point of view:
  - ◆ a class is a template for objects
    - ➤ each object is an instance of a class
    - ➤ object creation = *instantiation*



➔ **class-based programming languages**
= objects & classes

5 Objects and Classes in C++

- Class declaration similar to a structure declaration in C
- Access to members of an object (instance variables and methods) with the operators `.` or `->`, like the access to structure components
- Example:

```
// Class counter
class Counter
{
  private:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};
```


6 Methods in C++

- Definition within a class declaration:
  - ➤ method is handled as ***inline*** function
- Definition separate from the class declaration
  - ➤ assignment to class with the ***scope*** operator `::`
  - ➤ method invocations are handled like normal function calls
- Example:

```
class Counter {
  private:
    int value;
  public:
    void incr(); void decr(); int get_value();
};

void Counter::incr()          { value++; }
void Counter::decr()          { value--; }
int Counter::get_value()      { return value; }
```

7 Instantiation in C++

- Instantiation of Objects either
  - ➤ statically at compile time, or
  - ➤ dynamically during run time

★ **Static Instantiation**

- By object definition
- Example:

```
void main()
{
  Counter c1;          // object c1 of class Counter
  Counter *pc1;        // pointer to an object of class Counter
  ...
}
```


7 Instantiation in C++ (2)

★ **Dynamic Instantiation**

- C++ operators `new` and `delete`
- Example:

```
class Counter
{ ... };

void main()
{
  Counter c1;             // create object c1 statically
  Counter *pc1;           // pointer to an object of class Counter
  ...
  pc1 = new Counter;
  pc1->incr();
  c1.incr();
  ...
  delete pc1;
  ...
}
```


7 Instantiation in C++ (3)

★ **Constructor**

➔ Method for the initialization of objects

- method name = class name
  - ➤ method is automatically invoked during instantiation
- Example:

```
class Counter {
  private:
    int value;
  public:
    Counter(int c){ value = c; }        // constructor
    void incr()   { value++; }
    ...
};
...
Counter c1(20);        // create c1, initialize value to 20
cp = new Counter(30);
```

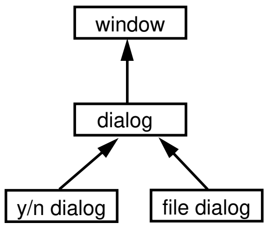

8 Objects and Classes in Java

★ **Essential Differences to C++**

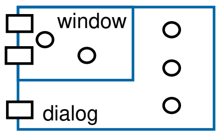
- No static instantiation
- Dynamic instantiation ➔ only references (pointers) to objects
  - ◆ access to object components through object reference and operator `.`
- No need to delete objects explicitly
  - ◆ automatic garbage collection
- Methods are implemented always in the class declaration
  - ◆ but no in-line mechanism
- No pointer arithmetic

9 Inheritance

- Relationship among classes where one class shares the structure and/or behavior defined in another class / other classes



inheritence hierarchy

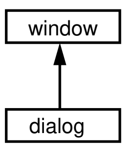


instance of an inheriting class

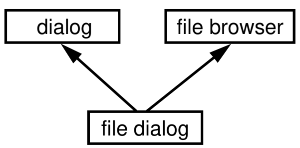
9 Inheritance (2)

★ **Terms**

- **Superclass / base class:** class from which another class inherits
- **Subclass:** class which inherits from other class(es)
- **Single inheritance:** subclass has exactly one superclass
- **Multiple inheritance:** subclass has several superclasses



single inheritence



multiple inheritence

9 Inheritance (3)

★ **Software developer's view**

- Specialization / generalization of classes
- Common aspects of classes are collected in a superclass
- Hierarchy of abstractions:
  - ◆ from more general classes to specialized classes and vice versa
- Documentation of the relationship between classes

9 Inheritance (4)

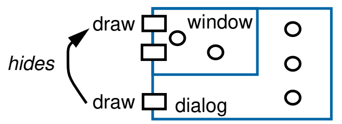
★ **Program-technical point of view**

- Extension of an existing class implementations
  - ➤ additional methods
  - ➤ additional data
- Code reusage:
  no reimplementation of inherited data and methods necessary
- Reimplementation of a method is *possible*, if the method of the superclass is not appropriate for the subclass
- Methods of the superclass can be invoked at an object of the subclass
- Modifications of a superclass effect all subclasses (central maintenance)

9 Inheritance (5)

★ Reimplementation

- Reimplementation of a method:
  - ➤ hides the method of the superclass

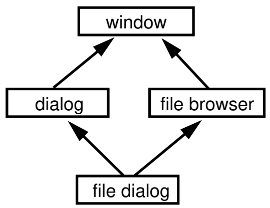


  - ➤ default behavior: invocation of the subclasses' method
  - ➤ invocation of the reimplemented method of the superclass?

9 Inheritance (6)

★ Multiple Inheritance

- Problems:
  - ➤ naming conflicts of variables or methods of the different superclasses
  - ➤ inheritance of the same superclass through different paths



- Application:
  - ➤ less important for code reusage
  - ➤ very important to describe type conformance (see section about typing)

10 Inheritance in C++

- Subclass inherits variables and methods of the superclass
- Subclass may modify superclass
  - ➤ additional methods and variables
  - ➤ modified methods
- Methods of the subclass may access *public* and *protected* components of the superclass
  - ➤ public superclass
    - ➔ the *interface* of the superclass is inherited
  - ➤ private superclass
    - ➔ the *interface* of the superclass is *not* inherited
    - ➔ objects of the subclass are not type-conform
- *private* data and methods of the superclass are not visible for methods of the subclass

10 Inheritance in C++ (2)

★ Example (1)

```
// Class counter
class Counter
{
  protected:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};

// Subclass resettable counter
class RCounter : public Counter
{
  private:
    int initial;
  public:
    RCounter(int v)      { initial = v; value = v; }
    void reset()         { value = initial; }
};
```

10 Inheritance in C++ (3)

★ **Example (2)**

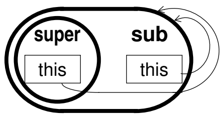
```
// Class window
class Window
{
  protected:
    int x, y, width, height;
  public:
    virtual void init(int x, int y, int w, int h){ initialize }
    virtual void move(int x, int y) { move window }
    virtual void display() { display window }
    virtual void delete() { remove window }
};

// Subclass bordered window
class BorderedWindow: public Window
{
  public:
    virtual void display() { display bordered window }
    virtual void change_width(int x) { change width }
    virtual void change_hight(int y) { change hight }
};
```


11 Dynamic Binding

- Decision which method to execute at run time (dynamic)

```
Window w = new BorderedWindow();
w->display();
```

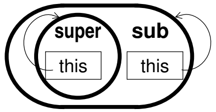
- This is also true if an object invokes a method at itself!
  - Example:
    - **move()** finally calls **display()** to redraw the window
    - **BorderedWindow** inherits **move()** from **Window**
    - invoking **move()** at an instance of **BorderedWindow** finally calls **display()** of **BorderedWindow**



the pointer *this* always references the "whole object" and not just the part of the superclass

11 Dynamic Binding (2)

- Without dynamic binding "true inheritance" is not possible
  → self reference (pointer *this*) is not adjusted correctly



★ **Static Binding**
Decision which implementation of a method is taken at compile time (depending on the type of the pointer)

- In C++ only "virtual" methods are bound dynamic
  - other methods are generally bound static
- In Java all methods are bound dynamic
  - static binding can be enforced by the keyword **final** in the method declaration
  - such methods cannot be reimplemented in subclasses

```
public final void incr() { value += step; }
```


C.7 Basic Concepts of the OO Paradigm

- Abstraction
  - Encapsulation
  - Abstract data type
- Modularization
- Hierarchy
- Typing
  - Hierarchy of types
  - Polymorphism
  - Genericity
- Concurrency
- Persistence

1 Abstraction

Fundamental concept for solving complex problems

- Emphasize details which are relevant for the entire solution
- Suppress details which are (for the moment) immaterial or diversionary

→ Object orientation

- important:
  - ➤ Signature of an object } Outside view
  - ➤ Semantics of an object } Outside view
    - ➥ contract model: Outside view = contract with other objects
- unimportant:
  - ➤ Implementation of an object

- Describe the abstraction first and think about the implementation later

2 Encapsulation

= Information Hiding
Concealing of the implementation of an abstraction from the users of the abstraction

- Complement to abstraction
  - ➤ Abstraction exposes the external properties of an object
  - ➤ Encapsulation hides the internals
- Fundamental for abstraction

  B. Liskov *For abstraction to work, implementations must be encapsulated*

- Encapsulation & object orientation
  - ◆ Representation of the object state
  - ◆ Implementation of the methods

→ Abstract data type

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abstract Data Type

? How can we provide a complete, precise & unambiguous description of an abstraction?

- ADT: Model to describe
  - ➤ Properties of data structures (Semantics!)
  - ➤ Operations of the data + semantic effects
  - ➤ NOT: Implementation of the data structures
- Specification → Description of all essential properties
- Overspecification: Details about the representation/implementation

→ ADT & object orientation

- ◆ Class = Implementation of an ADT
- ◆ Data abstraction: Object state is accessible only through methods

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abstract Data Type (2)

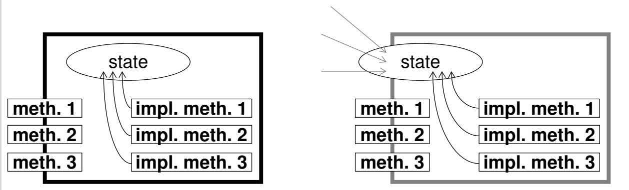
- Example: Specification of an ADT “Stack”

```
TYPES
        STACK[X]

FUNCTIONS
        empty: STACK[X] → BOOLEAN
        new: → STACK[X]
        push: X × STACK[X] → STACK[X]
        pop: STACK[X] ⇸ STACK[X]
        top: STACK[X] ⇸ X

PRECONDITIONS
        pre pop(s: STACK[X]) = (not empty(s))
        pre top(s: STACK[X]) = (not empty(s))

AXIOMS
        ∀x:X, s:STACK[X]:
        empty(new())
        not empty(push(x,s))
        top(push(x,s)) = x
        pop(push(x,s)) = s
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3 Abstract Data Type (3)

- Data abstraction in OOP
  - ➔ the object state is accessible only through the methods of the object



object as implementation of an ADT — object without data abstraction

- Data abstraction in C++ and Java
  - ◆ Scope rules (private / protected)

4 Modularity

Partitioning a program into individual components can reduce its complexity

- ◆ Problem partitions are easier to comprehend
- ◆ Different problem partitions may be assigned to different developer teams
- ◆ Module = separate unit in development

- **most important: partitioning creates boundaries = interfaces**
  - ➔ well-defined
  - ➔ documented

  } invaluable for the comprehension of the whole problem

- Many programming languages make a difference between interface and implementation of a module
- Structured Design: Grouping of subprograms
- OOD: Grouping of classes and objects (from the design's logical structure)

5 Hierarchy

- ◆ Abstraction & encapsulation help to hide details of components
- ◆ Modularity helps to cluster related abstractions

- Comprehension of large problems still difficult
  - ➔ too many abstractions
  - ➔ additional means to organize abstractions
- Abstractions often form hierarchies
  - ➤ common properties → more general abstractions
  - ➤ differences → specialization
  - ➔ Hierarchy: Ordering of abstractions
- Hierarchy & object orientation
  - ➤ Class structure: Inheritance → "is a" hierarchy
  - ➤ Object structure: Aggregation → "part of" hierarchy

6 Typing

- Concept derives primarily from the theory of ADTs

  Booch: *Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at most, they may be interchanged only in very restricted ways*

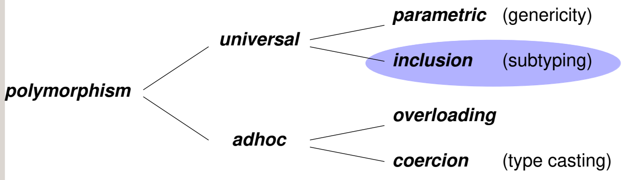
- Typing enables a programming language to check and enforce design decisions
  - ➔ essential for programming-in-the-large [Weg87]
- Strong typing:
  Conformance of all types in an expression is guaranteed
  - ➔ Static typing:
    conformance checked completely at compilation time
    - ➤ less flexibility
      - cannot detect compatible types
      - cannot support dynamic binding
  - ➔ More flexibility with polymorphism and genericity

7 Type Hierarchy

- Often 1:1 relationship between classes and types — but not necessary
  - ➤ different classes may implement the same type
  - ➤ one class may implement different types
- Hierarchy of classes: superclass ← subclass
  - ◆ Objective: code inheritance
  - ◆ Subclass not necessarily confirming to the type of the superclass
- Hierarchy of types: supertype ← subtype
  - ◆ Objectives: – behavior inheritance
    – declaration of conformable types (➔ polymorphism)
- Type inheritance (subtyping) as means for deriving types
  - ➔ Relationship between types becomes comprehendible
    - ➤ allows the identification of conformable types

8 Polymorphism

- Polymorphism = *the ability to take several forms*
  - ➤ various types for values or variables
  - ➤ various types for the parameters of functions
  - ➤ various types for the operands of an operator
- Example:
  operator "+" works with operands of type **`int`** and **`real`**
- Classification



9 Polymorphism in C++

- Overloading polymorphism
  - ➤ Function-name overloading
  - ➤ Operator overloading
- Inclusion polymorphism
  - ➤ public inheritance
- Coercion polymorphism
  - ➤ Cast operator

9 Polymorphism in C++ (2)

★ **Function-name Overloading**

- several functions with the same name but different signature within one scope
- compiler selects the proper function
  - ➔ static binding (*early binding*)
- Example:

```
int max (int a, int b) { return((a>b):a?b); }
float max (float a, float b) { return((a>b):a?b); }
char *max (char *a, char *b) {
        if ((strcmp(a,b)> 0) return a; else return b; }
```

- Special cases:
  - ➤ several constructors for a class
  - ➤ operator overloading
  - ➤ overloading of a method in the context of inheritance

Polymorphism in C++ (3)

★ Cast Operator / Coercion Polymorphism

- In most programming languages automatic type conversion for basic types (e.g. short → int → double ← float)
- Cast operator (cast method) allows the definition of an explicit conversion of the class type into another type

```
class::operator type(void) { ... }
```

- Cast operator is like a normal unary operator

Polymorphism in C++ (4)

★ Cast operator — Example

```
class time
{
  private:
    int hour, min, sec;
  public:
    ...                                  // Konstruktor

    operator long () {                   // cast-Operator
      return(hour*3600 + min*60 + sec);
    }
};
main() {
  time now(1,10,2);
  long t;

  t = now + 10;                     // ⇒ t == 4212
...
```


Polymorphism in C++ (5)

★ Inclusion Polymorphism — *THE Polymorphism in OOP*

Inheritance + Virtual Methods + Object References

- Object reference (pointer) has a type (= class)
  - ◆ instances of this class and all of its subclasses may be assigned to the ref.
  - ◆ on method invocation, the actual implementation of the method is designated not by the class of the pointer but by the class of the current object

➔ You may always assign any type-conformable object to an object reference and everything will work

- ◆ you may pass it as parameter to a method
- ◆ the programmer of the method did not need to know anything about your new subtype — as long as it conforms to the supertype his method expects

Polymorphism in C++ (6)

★ Virtual Methods & Inclusion Polymorphism — Example:

```
class geo_obj {                     // general superclass
  public:
    virtual void draw();
};

class circle : public geo_obj {     // subclass
  public:
    void draw();
}

class square : public geo_obj {     // subclass
  public:
    void draw();
}

main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```

10 Types & C++ Abstract Classes

- Superclass declares methods and its signatures but does not define them
  - – **pure virtual functions**
  - ➔ superclass defines only a type
- Subclasses define various implementations of the methods
  - ➔ each subclass is one implementation of the type
- Not possible to instantiate objects from the superclass
- Example:

```
class geo_obj {                    // abstract calss
  public:
    virtual void draw() = 0;       // pure virtual function
};
class circle : public geo_obj {   // subclass
  public:
    void draw() { ... }
}
```


11 Types & Java Abstract Classes

- As in C++
  - • instantiation not possible
  - • missing parts have to be completed in the subclass
- Example:

```
abstract class geo_obj {              // abstract class
  public abstract void draw();        // pure virtual function
};
class circle extends geo_obj {        // subclass
  public void draw() { ... }
}
```


12 Types & Java Interfaces

- 2 possibilities to declare a type
  - ◆ by a class definition
    - ➔ class inheritance automatically leads to type inheritance
  - ◆ by a interface declaration
    - ➔ separate type declaration
- Example:

```
public interface Printable {
    public void Print();
}

public class MyPoint extends Object implements Printable {
    ....
    public void Print() {
        System.out.println("x="+x+" y="+y);
    }
}
```


12 Types & Java Interfaces (2)

- Inheritance + multiple inheritance for interfaces
- One class may implement several types
- Type conformance is transitive
- Exceptions are an element of the type interface
- Examples:

```
interface Streamable extends FileIO, Printable {
    // additional Methods
    public void test() throws TestException;
}

class Test implements Streamable, Testinterface {
    ...
}
```

Genericity

Concept that allows the definition of the type of some unit of the programming language at invocation or instantiation by a parameter

- OOP: generic classes ➔
  - generic class ➔ instantiation (+ parameterization) ➔ actual class
  - actual class ➔ instantiation of objects
- Example:
  - ◆ general stack class
    - ➤ int stack
    - ➤ real stack
    - ➤ string stack
- Most benefit of genericity can also be achieved by inheritance
- Implemented in Ada, Eiffel and C++ (*Templates*)

Genericity & C++: Templates

- Objective: definition of a class without finalization of types
  - ➤ dynamic type checking at run time or
  - ➤ static type checking at compile time + parameterizable classes
- Template = parameterizable class
- Example:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
              { index = 0; array = new T[n]; }
    void push(T elem)
              { array[index++] = elem; }
    T pop(void)
              { return(array[index--]); }
};
```


Genericity & C++: Templates

- Parameterization of class on object instantiation
- Examples:
  - ◆ Instantiation and usage of a stack with elements of type float:

```
stack <float> s(10);
float a;
s.push(a);
a = s.pop( );
```

  - ◆ Instantiation and usage of a stack with elements of type integer:

```
stack <int> s(10);
...
```


Concurrency

Several threads of control are processed in parallel on different processors or quasi-parallel on one single processor

- Concurrency is orthogonal to object orientation (in general)

but: concurrent solutions are much more complex than sequential ones

- Granularity: concurrency / objects (capsules)
  - ➤ Finer grained concurrency
    - ➔ even object-internal concurrency
  - ➤ Coarser grained concurrency
    - ➔ just object-external concurrency
- Integration of concurrency control into OO programming languages
  - ➤ orthogonal languages
  - ➤ non-orthogonal languages
    - – uniform / non-uniform languages

17 Concurrency & Java

- Thread concept and synchronization mechanisms are integrated
  - ➥ non-orthogonal, non-uniform language
- Creation of threads via thread class
- Example:

```
class MyClass implements Runnable {
        public void run()  {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();    // create object
Thread t1 = new Thread(o1);    // create thread to run in o1

t1.start();                    // start thread

Thread t2 = new Thread(o1);    // create second thread to run in o1

t2.start();                    // start second thread
```


17 Concurrency & Java (2)

★ **Synchronization**

- Monitors: exclusive execution of methods of an object
  - ◆ Example:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

- Conditions: releasing a monitor while waiting on an event

18 Persistence

★ **Motivation for Persistence** [ABC83]

- “active” data → programming language facilities / run-time environment
- “passive” data → DBMS or file system

➔ 2 different views of data

➔ Disadvantages for the programmer
  - ➤ Conversion between active and passive data necessary
  - ➤ Data type protection of programming language is lost

18 Persistence (2)

★ **General Definition**

- Persistence is the property of data through which
  its existence transcends time
  (i. e. it continues to exist after its creator ceases to exist)
  and/or space
  (i. e. its location moves from the address space in which it was created).

★ **Spectrum of Persistence**

1. Transient results in expression evaluation
2. Local variables in procedure activations
3. Global variables and heap items whose extent is different from their scope
4. Data that exists between executions of a program
5. Data that exists between various versions of a program
6. Data that outlives the program

18 Persistence (3)

★ **Persistence in Object-oriented Systems**

- Objects survive the termination of the environment (thread, application execution) in which they were instantiated or used
- Powerful mechanism in OO operating systems for
  - ➤ Data storage
  - ➤ Data transport
- Examples
  - ➤ File systems
  - ➤ Database systems
  - ➤ Persistent communication objects
- Properties of the object-oriented programming model are automatically inherited by all mechanisms which are constructed on its basis

C.8 OOA & OOD

1 Overview

- **OOA: *What* shall my system do?**
  - ◆ Basis: Analysis of the “real world”
    - ➤ Components, terms, tasks
    - ➤ Requirements and constraints
  - ◆ Abstraction from unimportant aspects & implementation details
- **OOD: *How* does my system do it?**
  - ◆ Transformation from the analysis model to an implementable model
  - ◆ Add aspects of the implementation environment
  - ◆ Structural and strategic decisions
    - ➤ Threads, distribution, IPC, error handling, garbage collection, …
- **Design patterns:** Guidelines for the OOD

2 OOAD Methods

- Notation
- Process
- Well-known examples for OOAD methods: Booch, OMT and UML
- CASE tools support OOA and OOD
  - ➤ Diagrams
  - ➤ Documentation
  - ➤ Support for multiple users
  - ➤ Consistency checks
  - ➤ Generation of code skeletons

3 OOA — Process

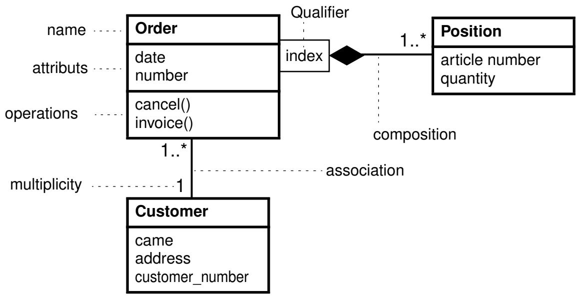
- Requirement Analysis
  - ➤ Determine problem domain
  - ➤ Specify goals from the user’s point of view
  - ➤ Performance and architectural requirements
- Use cases
  - ➤ Describe interaction between “user” and application
  - ➤ Participants: actor & use case
  - ➤ Substantiates the requirements analysis
- Find objects
  - ➤ Look for terms of the problem domain (nouns in the description)
- Organize objects (requirements model -> analysis model)
  - ➤ different object types (interface objects, entity objects, control objects)
- Refine structure

4 OOD

- Transition from OOA to OOD often very smooth
  - When implementation aspects cannot be kept back any longer
- Transformation from the analysis model to an implementable model
- Class design
  - Find software classes for the classes of the analysis model
  - Keep up class boundaries (Analysis -> Design -> Implementation)
    - Traceability
- System design
  - problem-independent aspects
    (distribution, concurrency, resource management, …)
- Program design
  - Programming language
  - Error messages, exceptions, performance optimizations

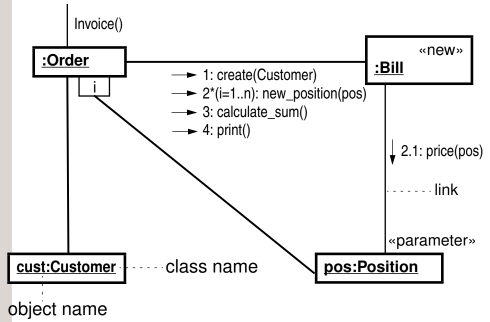
5 UML Notation(1): Class Diagrams

- Class with its attributes (variables) and operations (methods)
- Relations between classes
- Example:



6 UML-Notation(2): Collaboration Diagrams

- Model collaboration of objects to fulfil a certain task
- Example:



7 Design Patterns

- A design pattern describes a solution for a class of problems
- Components of a pattern: Name, problem class, solution, consequences
- Example:
  Observer Design Pattern: Other objects depend on the state of an object and have to be informed about changes of the state