C Object-oriented Programming

C.1 Overview

- Motivation for the OO paradigm
- Software-design methods
- Basic terms of OO programming
- The Evolution of the object model
- Fundamental 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.

GHJ+97. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, 10th print, Addison-Wesley, 1997

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.

C.2 References (2)

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.

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
  - ◆ Hardware increasingly 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)
- Object-oriented design

2 Classes of Programming Languages

… at least the most important ones

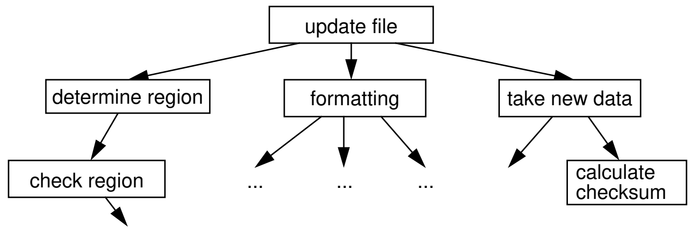
- Procedural / imperative
- Functional
- Object-oriented

3 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
- Procedural languages ideally suited for implementations

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

- Example:

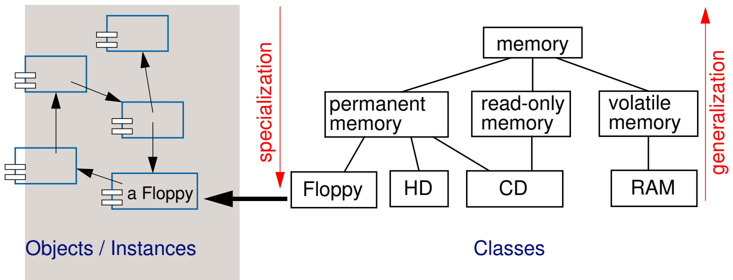


4 Object-oriented Design

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 Object-oriented Programming

1 Definition (Grady Booch)

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

cooperative collections of objects,

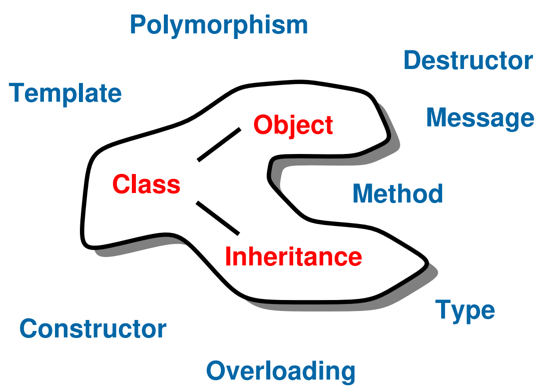
each of which represents an

instance of some class,

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

inheritance relationships.

2 Basic Terms



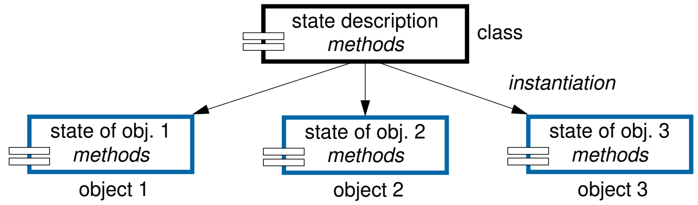
3 Objects & Methods

- 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);
```

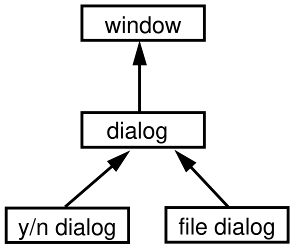
8 Objects and Classes in Java

★ **Essential Differences to C++**

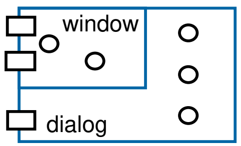
- No static instantiation
- Dynamic instantiation → only references (pointers) to objects
  - ◆ access to object components through object reference and operator **.**
- No need to delete objects explicitly
  - ◆ automatic garbage collection
- Methods are implemented always in the class declaration
  - ◆ 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

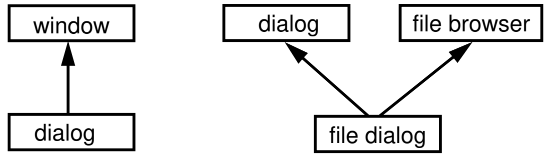


instance of an
inheriting class

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)

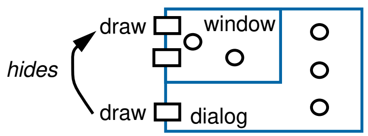
★ **Program-technical point of view**

- 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**

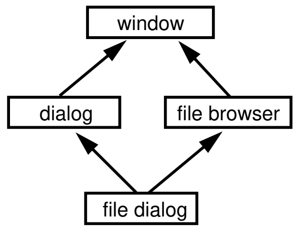
- Reimplementation of a method:
  - ➤ hides the method of the superclass



  - ➤ 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)**

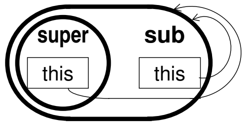
```
// Class window
class Window
{
  protected:
    int x, y, width, height;
  public:
    virtual void init(int x, int y, int w, int h){ initialize }
    virtual void move(int x, int y) { move window }
    virtual void display() { display window }
    virtual void delete() { remove window }
};

// 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();
```

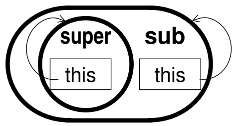
- This is also true if an object invokes a method at itself!
  - ◆ Example:
    - – `move()` finally calls `display()` to redraw the window
    - – `BorderedWindow` inherits `move()` from `Window`
    - – invoking `move()` at an instance of `BorderedWindow` finally calls `display()` of `BorderedWindow`



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; }
```