Data Structures - Basic Concepts

Overview

- Programming Paradigms
- Values, Sets, and Arrays
- Indexer, Iterators, and Pattern Structures

References

- Bruno R. Preiss: Data Structures and Algorithms with Object-Oriented Design Patterns in C++. John Wiley & Sons, Inc. (1999)
- Richard F. Gilberg and Behrouz A. Forouzan: Data Structures - A Pseudocode Approach with C. 2nd Edition. Thomson (2005)
- Russ Miller and Laurence Boxer: Algorithms Sequential & Parallel. 2nd Edition. Charles River Media Inc. (2005)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

Programming Paradigms

- Imperative style:

  program = algorithms + data

- Functional style:

  program = function • function

- Logic programming style:

  program = facts + rules

- Object-oriented style:

  program = objects + messages

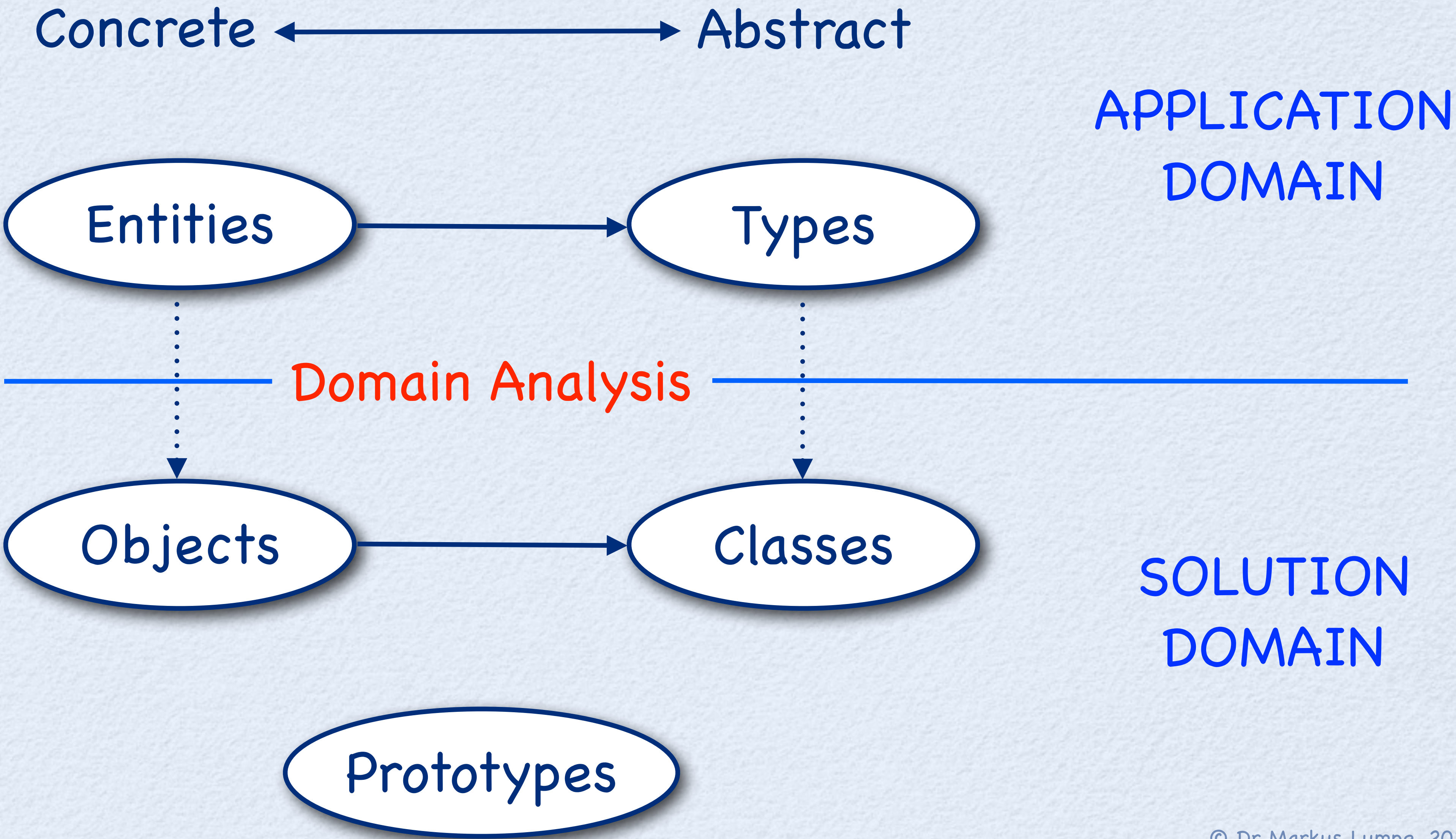
- Other styles and paradigms:

  blackboard, events, pipes and filters, constraints, lists, …

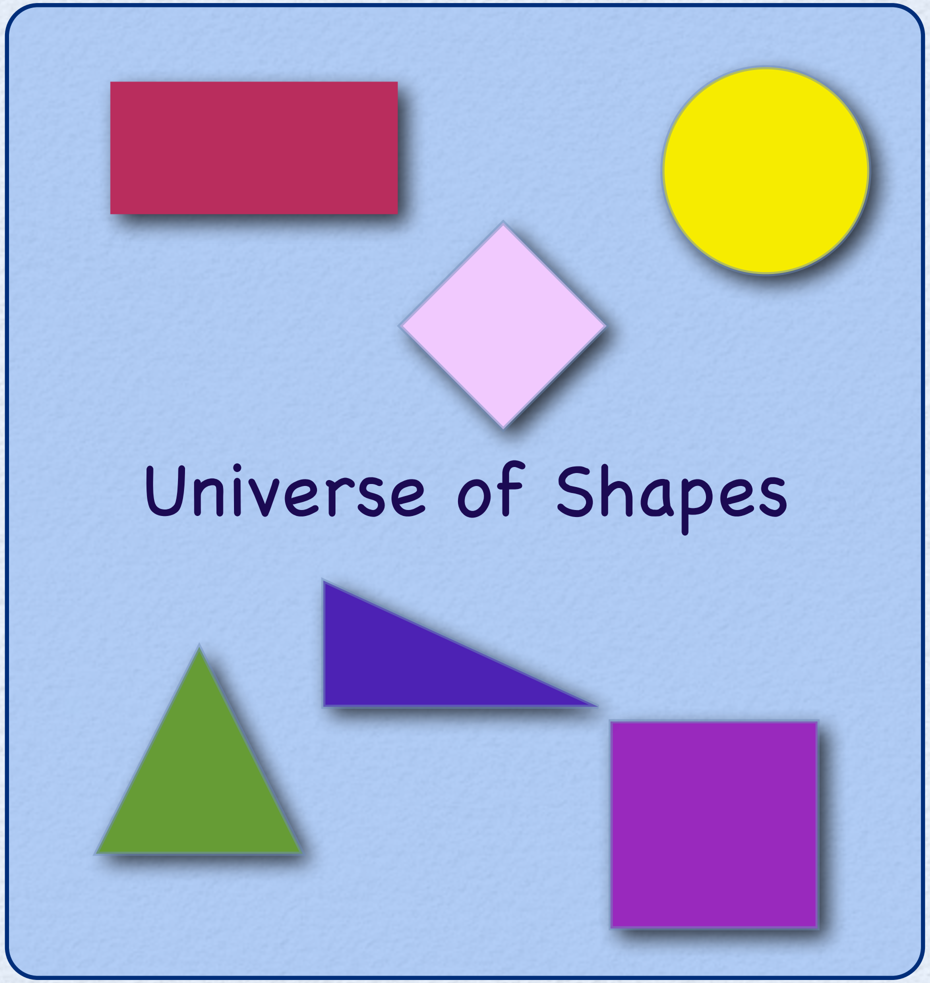
Object-Oriented Software Development

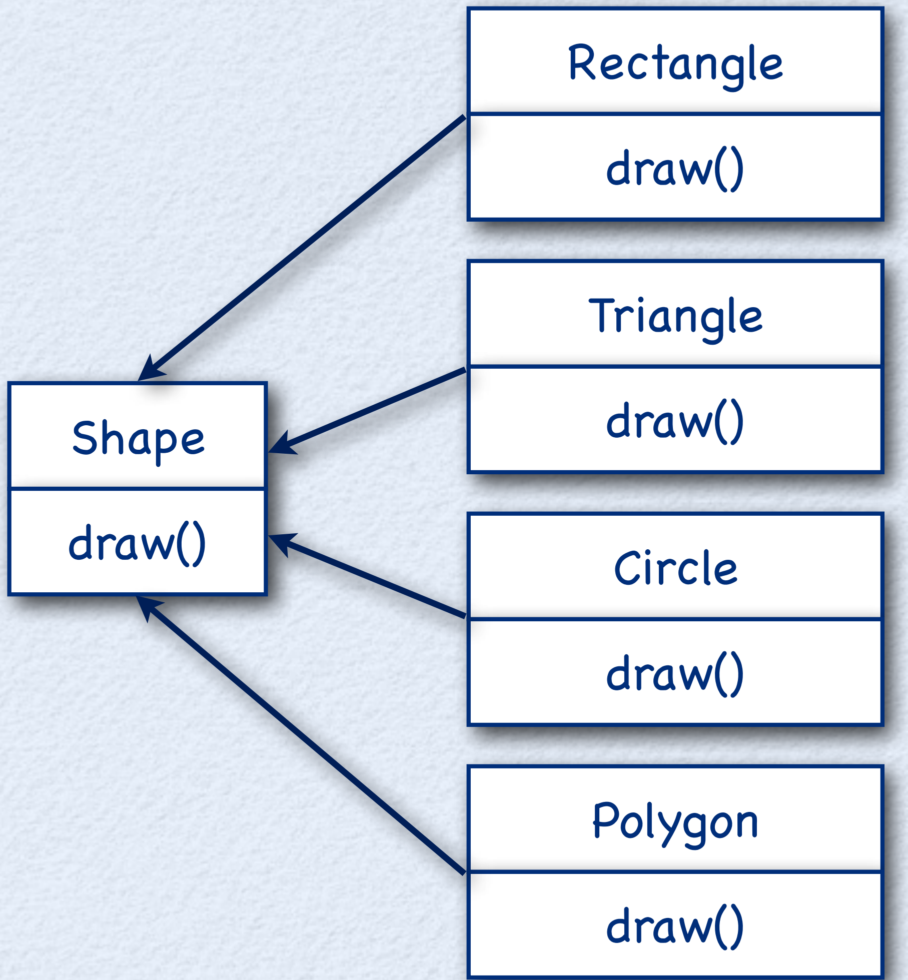
- Object-oriented programming is about
  - Object-oriented software development
  - Using an object-oriented programming language
- Object-oriented software development is
  - An evolutionary step refining earlier techniques
  - A revolutionary idea perfecting earlier methods

Object-Oriented Design



Concrete vs. Abstract





Why is object-oriented software development popular?

- The object-oriented development approach
  - Naturally captures real life
  - Scales well from trivial to complex tasks
  - Focuses on responsibilities, reuse, and composition

Values

- In computer science we classify as a value everything that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.
- In computer science, as in mathematics, an "expression" is used (solely) to denote a value.
- Which kinds of values are supported by a specific programming environment depends heavily on the underlying paradigm and its application domain.
- Most programming environments provide support for some basic sets of values like truth values, integers, real number, records, lists, etc.

Constants

- Constants are named abstractions of values.
- Constants are used to assign an user-defined meaning to a value.
- Examples:
  - EOF = -1
  - TRUE = 1
  - FALSE = 0
  - PI = 3.1415927
  - MESSAGE = "Welcome to DSP"
- Constants do not have an address, that is, they do not have a location.
- At compile time, applications of constants are substituted by their corresponding definition.

Primitive Values

- Primitive values are values whose representation cannot be further decomposed. We find that some of these values are implementation and platform dependent.
- Examples:
    - Truth values,
    - Integers,
    - Characters,
    - Strings,
    - Enumerands,
    - Real numbers.

-1

"Hello World!"

3.14159

false

Red

Composite Values

- Composite values are built up using primitive values and composite values. The layout of composite values is in general implementation dependent.
- Examples:
    - Records
    - Arrays
    - Enumerations
    - Sets
    - Lists
    - Tuples
    - Files

CompositeValues.cpp

```cpp
struct Point
{
    int x;
    int y;
} aPoint = { 1, 2 };

enum Color
{
    Red,
    Blue,
    Green
};
```

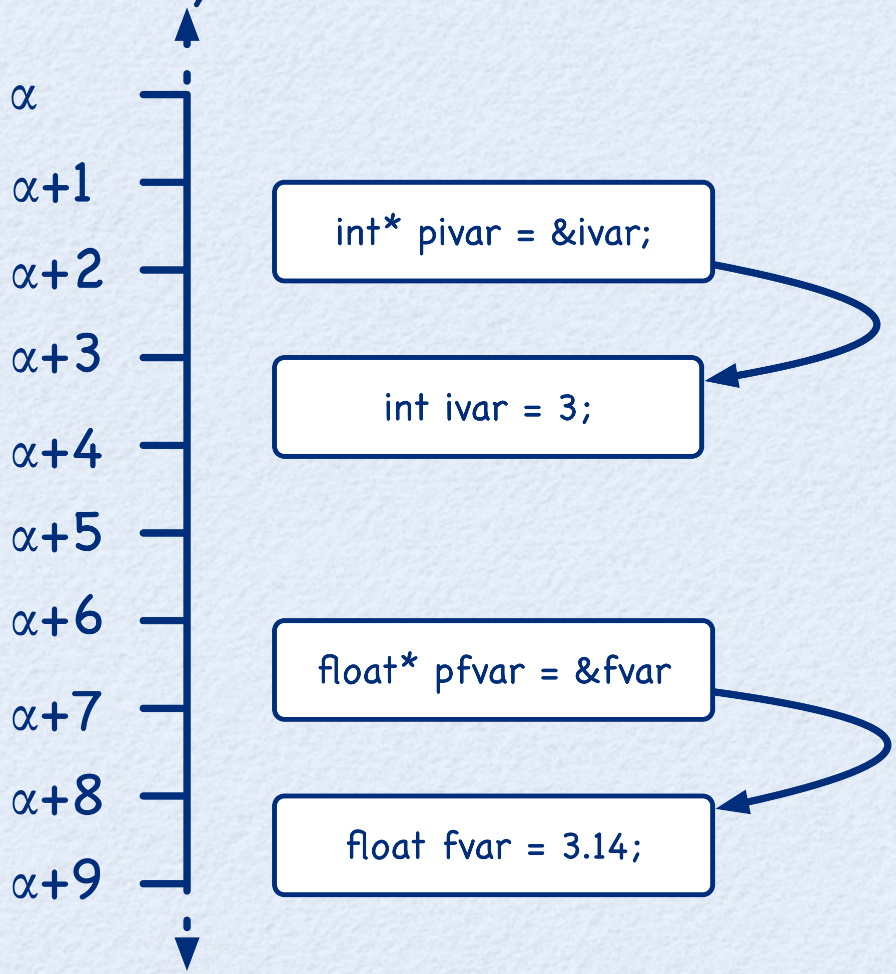
Line: 11 Column: 1 C++ Tab Size:

Pointers

- Pointers are references to values, i.e., they denote locations of a values.
- Pointers are used to store the address of a value (variable or function) – pointer to a value, and pointers are also used to store the address of another pointer – pointer to pointer.
- In general, it not necessary to define pointers with a greater reference level than pointer to pointer.
- In modern programming environments, we find pointers to variables, pointers to pointer, function pointers, and object pointers, but not all programming languages provide means to use pointers directly (e.g., Java).

Memory, Values, and Pointers

Memory:



Values:	Deference/Address :
pivar == α+3	*pivar == 3
pivar == 3	&pivar == α+3
pfvar == α+8	*pfvar == 3.14
fvar == 3.14	&fvar == α+8

Sets

- A set is a collection of elements (or values), possibly empty.
- All elements satisfy a possibly complex characterizing property. Formally, we write:

  $$\{ x \mid P(x) = \text{True} \}$$

  to define a set, where all elements satisfy the property P.
- The basic axiom of set theory is that there exists an empty set, $\emptyset$, with no elements. Formally,

  $$\forall x, \quad x \notin \emptyset$$

  In words, "for every x, x is not an element of $\emptyset$."

Sets are collections of values.

Inductive Reasoning

- To define a set and to capture what qualifies values to be members of the set, we can use inductive reasoning and formally verify properties about members of the set.

- Algebraically, we can define a set using induction on the structure of expressions and induction on the length or structure of expressions as a means to verify (prove) properties of the set and the elements thereof.

- **Note:** We can construct infinitely many values from a given finite recipe - inductive specification.

Inductive Specification

- Sometimes it is difficult to define a set explicitly, in particular if the elements of the set have a complex structure.
- However, it may be easy to define the set in terms of itself. This process is called inductive specification or recursion.
- Example:

  Let the set S be the smallest set of natural numbers satisfying the following two properties:

  - $0 \in S$, and
  - Whenever $x \in S$, then $x + 3 \in S$.

  The first property is called base clause and the second property is called inductive/recursive clause. An inductive specification may have multiple base and inductive clauses.

The "Smallest Set"

- If we use inductive specification, we always define the smallest set that satisfies all given properties. That is, inductive specification is free of redundancy.
- It is easy to see that there can be only one such set:

  If S1 and S2 both satisfy all given properties, and both are the smallest, then we have S1 ⊆ S2 (since S1 is the smallest), and S2 ⊆ S1 (since S2 is the smallest), hence S1 = S2.

The Set of Strings

$S = \epsilon \mid aS$, where

- ϵ is the empty string and
- $a \in \Sigma$, with Σ being the alphabet over S.

- Examples:
  - ϵ, ϵa, $\epsilon aaaaaaaaa$ where a is some character in the alphabet Σ ($a \in \Sigma$)

Regular Sets of Strings

- Operations for building sets of strings:
  - Alternation

    $$S_1 \mid S_2 = \{ s \mid s \in S_1 \vee s \in S_2 \}$$

  - Concatenation

    $$S_1 \cdot S_2 = \{ s_1 s_2 \mid s_1 \in S_1, s_2 \in S_2 \}$$

  - Iteration

    $$S^* = \{\epsilon\} \mid S \mid S \cdot S \mid S \cdot S \cdot S \mid \dots$$

    $$= S_0 \mid S_1 \mid S_3 \mid S_3 \mid \dots$$

- A set of strings over Σ is said to be regular if it can be built from the empty set $\varnothing$ and the singleton set $\{a\}$ (for each $a \in \Sigma$), using just the operations of alternation, concatenation, and iteration.

Indexed Sets

- Sets are unordered collections of data elements.
- In order to obtain an ordering relation over the elements of a given set, we can assign each element in that set a unique element of another ordered set I:

$$S_I = \{\ a_i \mid a \in S,\ i \in I\ \}$$

S_I is called the "indexed set" of S.

Some Indexed Sets

- Let A = { a, b, c, d } and I = $\mathcal{N}$, then

$$A_I = \{ a_1, b_2, c_3, d_4 \}$$

- Let A = { a, b, c, d } and I = (S×S, <), then

$$A_I = \{ a_{\text{“}1\text{”}}, b_{\text{“}2\text{”}}, c_{\text{“}3\text{”}}, d_{\text{“}4\text{”}} \}$$