OOP345S1A

Object Oriented Software Development Using C++

Sukhbir Singh Tatla

sukhbir.tatla@Senecacollege.ca

Agenda

- **Review**
- **Procedural VS Object Oriented Programming**
- **Introduction to Object Oriented Programming Concepts**
- Data Types and Operators in C++
- Use of Arrays, Pointers and Structures in C++
- Summary

High Level Programming Language

- High-level languages represent a giant leap towards easier programming.

- The syntax of High Level languages is similar to English.

- Historically, we divide High Level languages into two groups:
  - Procedural languages
  - Object-Oriented languages (OOP)

Procedural Programming

- Traditional programming languages were procedural.
- –C, Pascal, BASIC, Ada and COBOL
- Programming in procedural languages involves choosing data structures (appropriate ways to store data), designing algorithms, and translating algorithm into code.
- Procedural programming separates the data of the program from the operations that manipulate the data.
- This methodology requires sending data to procedure/functions.

Object Oriented Programming

- Object-oriented programming is centered on creating objects rather than procedures/ functions.
- •Objects are a melding of data and procedures that manipulate that data.
- •Data in an object are known as attributes.
- •Procedures/functions in an object are known as methods.
- Object-oriented programming combines data and behavior (or method). This is called encapsulation.
- Only an object’s methods should be able to directly manipulate its attributes.
- This indirect access is known as a programming interface.

Procedural Languages V/S Object – Oriented Languages

Procedural Languages	Object Oriented Languages
Early high-level languages are typically called procedural languages.	Most object-oriented languages are high-level languages
Procedural languages are characterized by sequential sets of linear commands.	Programmers code using “blueprints” of data models called classes.
The focus of such languages is on structure.	The focus of OOP languages is not on structure, but on modeling data.
Examples include C, COBOL, Fortran, LISP, Perl, HTML, VBScript	Examples of OOP languages include Smalltalk, C++,.NET and Java.

Object Oriented Paradigm

(Paradigm: a way of seeing and doing things)

- Object Oriented (OO) Programming (P):
  - Organizing software as a collection of objects with a certain state and behavior.
- Object Oriented Design:
  - Based on the identification & organization of objects.
- Object Oriented Methodology
  - Construction of models
  - The development of Software is a modeling process
- Object Oriented Modeling and Design
  - Modeling objects based on the real world
  - Using models to design independently of a programming language .

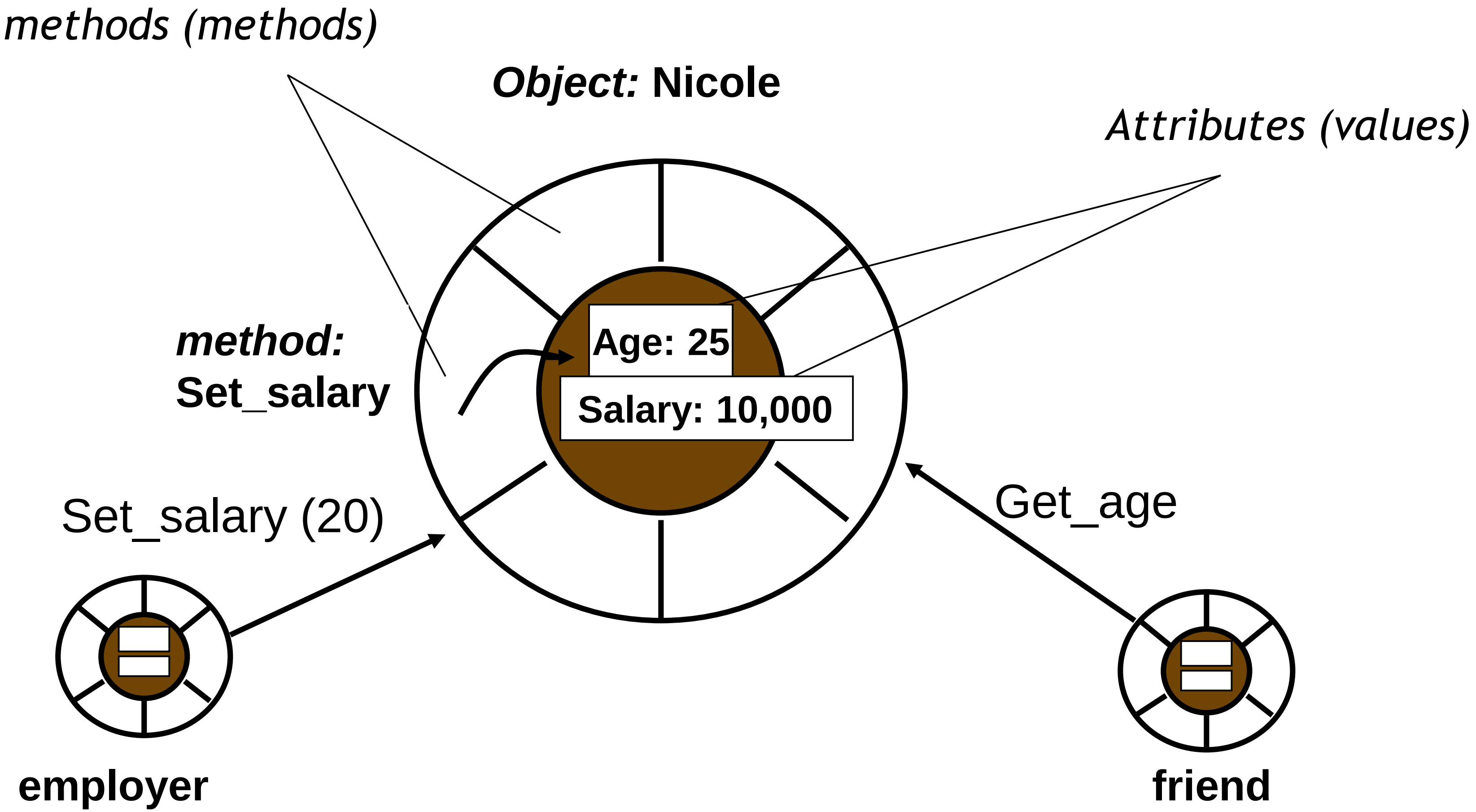
- Object (instance)
  - State (fields)
  - Behavior (methods)
  - Identity
- Class
  - code describing implementation of an object

- Data Abstraction
- Modularity
- Encapsulation
- Inheritance
- Polymorphism

Objects

- Object: An object contains both data and methods that manipulate that data.
  - An object is active, not passive; it does things
  - An object is responsible for its own data
  - But: it can expose that data to other objects
- An object is a complex data type that has an **identity**, contains other data types called **attributes** and modules of code called **operations** or **methods**
- Attributes and associated values are **hidden** inside the object.
- Any object that wants to obtain or change a value associated with other object, must do so by sending a **message** to one of the objects (invoking a method)
- Example: A “CheckingAccount” might have
  - A balance (the internal state of the account)
  - An owner (some object representing a person)

Objects


methods (methods)
Object: **Nicole**
Attributes (values)
method:
Set_salary
Age: 25
Salary: 10,000
Set_salary (20)
Get_age
employer
friend

Classes

- Classes are templates that have methods and attribute names and type information, but no actual values!
- A class is a prototype for creating objects.
- When we write a program in an object-oriented language like C++, Java, we define classes, which in turn are used to create objects.
- A class has a constructor for creating objects
- Objects are generated by these classes and they actually contain values.
- We design an application at the class level.
- When the system is running objects are created by classes as they are needed to contain state information.
- When objects are no longer needed by the application, they are eliminated.

Class & Objects

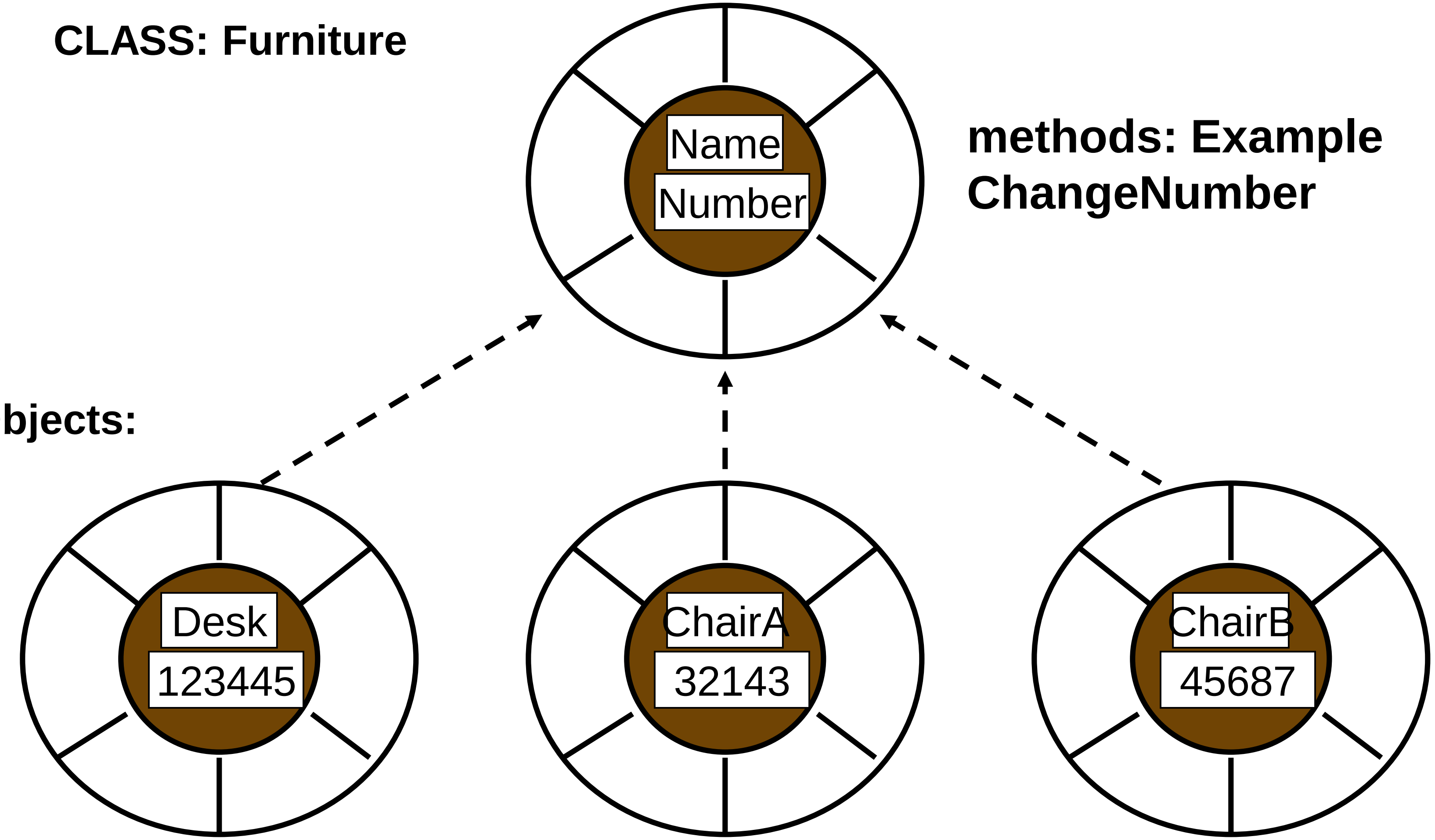
- Every object belongs to (is an instance of) a class
- An object may have fields, or variables
  - The class describes those fields
- An object may have methods
  - The class describes those methods
- A class is like a template, or cookie cutter
  - You use the class’s constructor to make objects
- An Abstract Data Type (ADT) bundles together:
  - some data, representing an object or "thing"
  - the operations on that data
- The operations defined by the ADT are the only operations permitted on its data.

Class & Objects

CLASS: Furniture

methods: Example
ChangeNumber

Objects:



Approximate Terminology

- instance = object
- field = instance variable
- method = function
- sending a message to an object = calling a function
- These are all *approximately* true

Data Abstraction

- Focus on the meaning of the operations (behavior), to avoid over-specification.
- The representation details are confined to only a small set of procedures that create and manipulate data, and all other access is indirectly via only these procedures.
- Facilitates code evolution.

Encapsulation

- Each objects methods manage it’s own attributes.
- This is also known as ***hiding***.
- An object A can learn about the values of attributes of another object B, only by invoking the corresponding method (message) associated to the object B.
- In other words, Incorporation into a class of data & operations in one package
- Data can only be accessed through that package
- Example:
  - Class: Lady
  - Attributes: Age, salary
  - Methods: get_age, set_salary

Message Passing & Associations

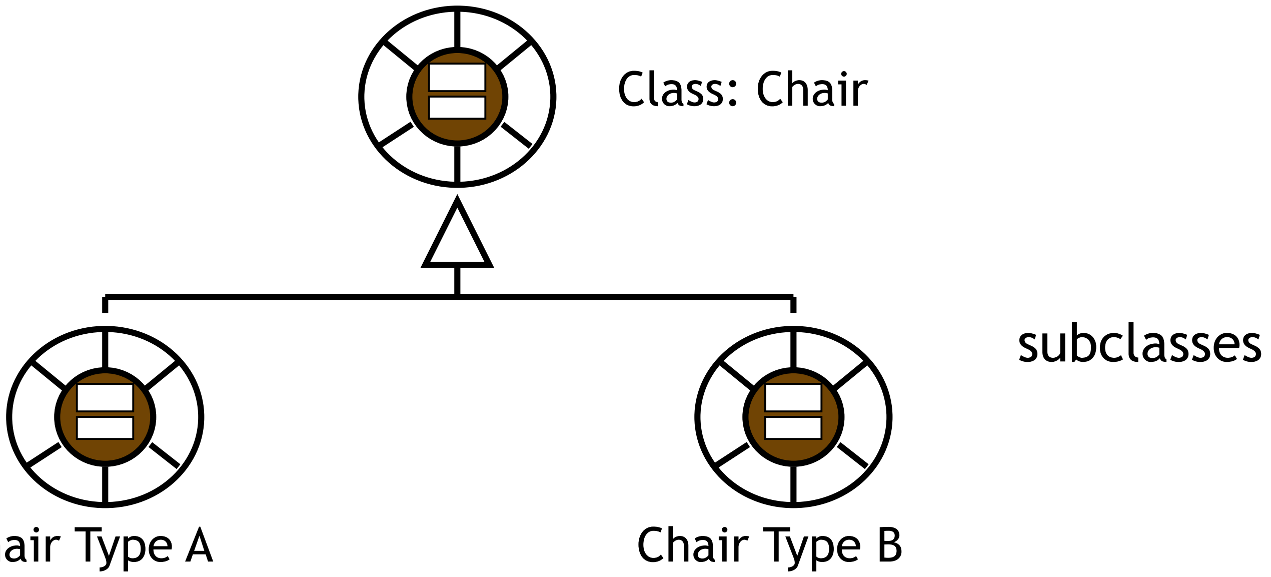
- Methods are associated with classes but classes don’t send messages to each other.
- Objects send messages.
- A static diagram (class diagram) shows classes and the logical associations between classes, it doesn´t show the movement of messages.
- An association between two classes means that the objects of the two classes can send messages to each other.
- Aggregation: when an object contains other objects (a part-whole relationship)

Inheritance Concept: Classes form a Hierarchy

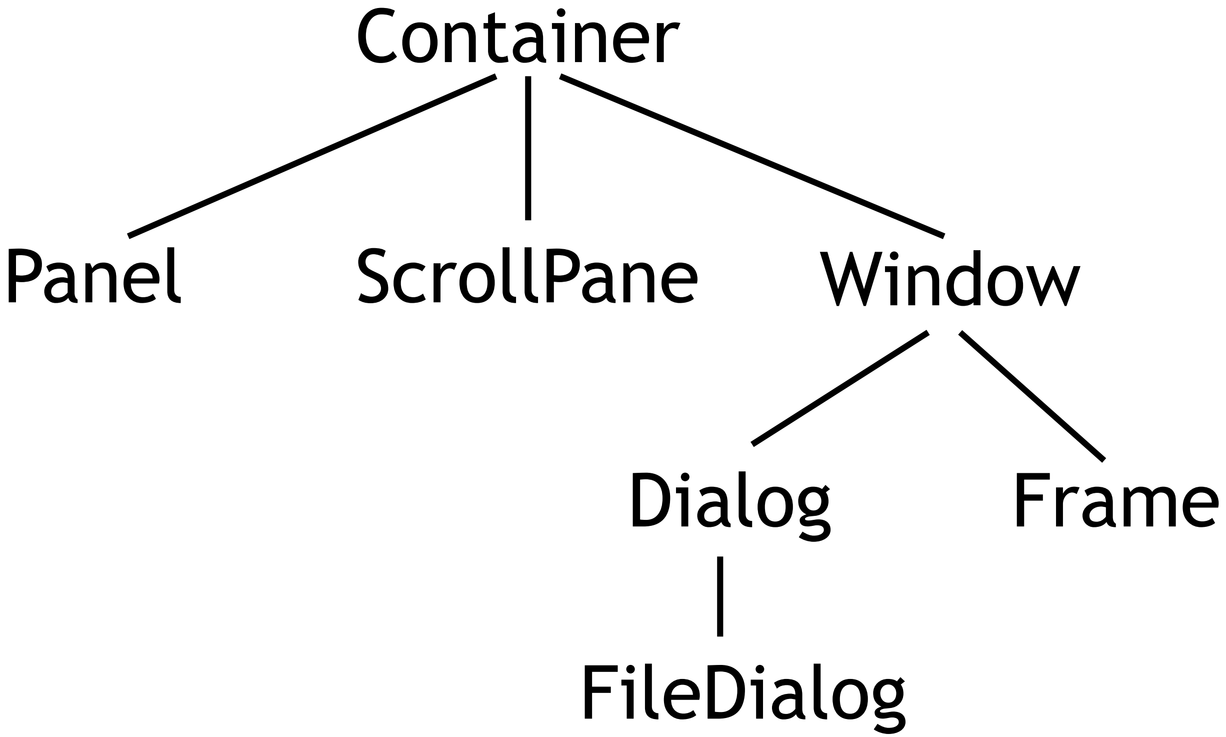
- Classes are arranged in a treelike structure called a hierarchy
- The class at the root is named Object or root class
- Every class, except Object/Root Class, has a superclass
- A class may have several ancestors, up to Root Class/Object
- When you define a class, you specify its superclass
  - If you don’t specify a superclass, Root Class/Object is assumed
- Every class may have one or more subclasses

Class Hierarchies & Inheritance

- Classes can be arranged in hierarchies so that more classes inherit attributes and methods from more abstract classes
- Class hierarchy diagrams

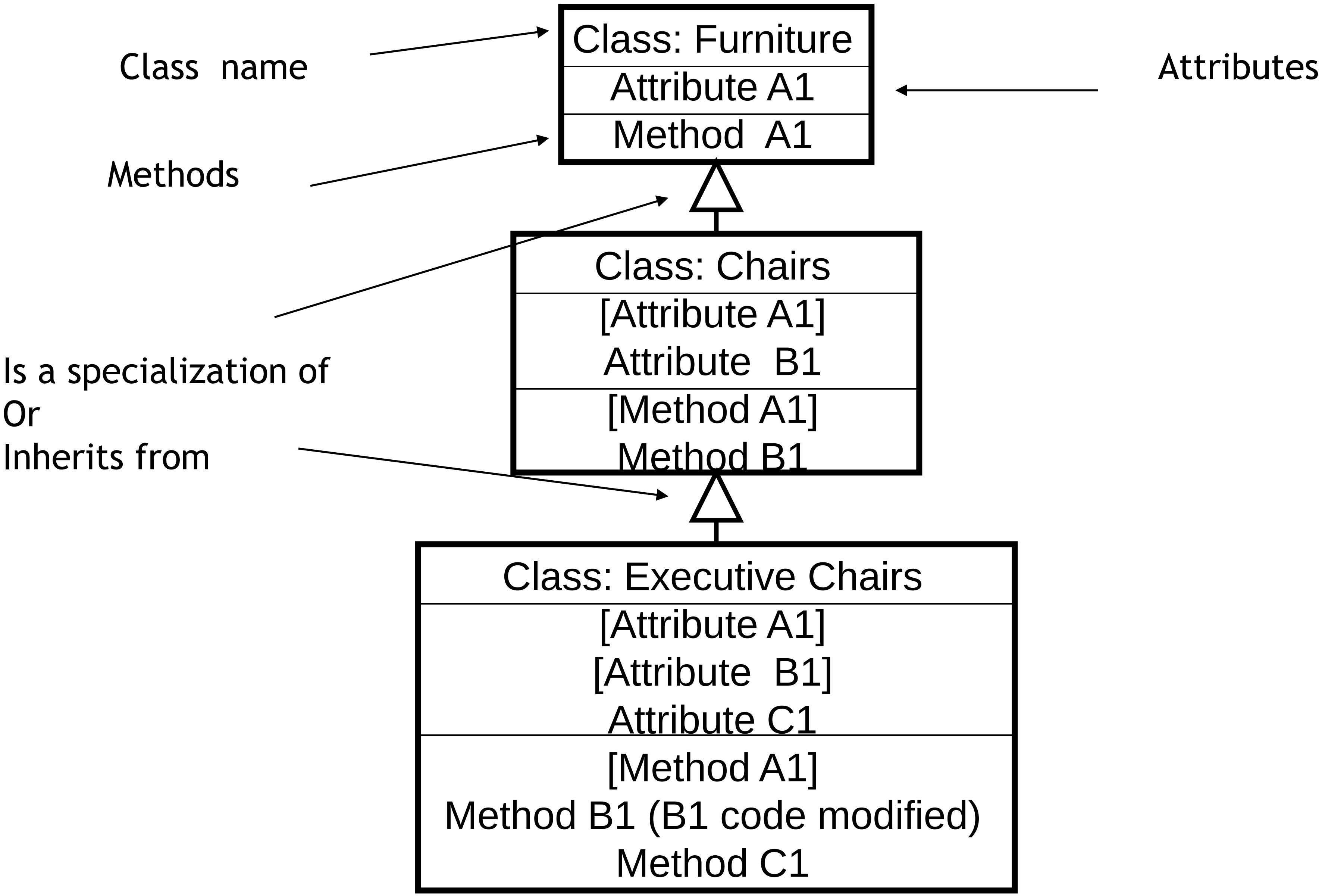


Example of (part of) a hierarchy



A FileDialog is a Dialog is a Window is a Container

Class Inheritance & Specialization



Class name

Attributes

Methods

Is a specialization of
Or
Inherits from

Class: Furniture
Attribute A1
Method A1

Class: Chairs
[Attribute A1]
Attribute B1
[Method A1]
Method B1

Class: Executive Chairs
[Attribute A1]
[Attribute B1]
Attribute C1
[Method A1]
Method B1 (B1 code modified)
Method C1

Public, Private & Protected

- Attributes can be public or private:
  - **Private:** it can only be accessed by its own methods
  - **Public:** it can be modified by methods associated with any class (violates encapsulation)
- Methods can be public, private or protected:
  - **Public:** it’s name is exposed to other objects.
  - **Private:** it can’t be accessed by other objects, only internally
  - **Protected:** (special case) only subclasses that descend directly from a class that contains it, know and can use this method.

Method signature

- Things which an object can do; the “verbs” of objects. In code, usually can be identified by an “action” word -- Hide, Show.
- It is the method’s name and the parameters that must be passed with the message in order for the method to function.
- The parameters are important because they assure that the method will function properly.
- Additionally they allow a compiler or interpreter to discriminate between two different methods with the same name.

Polymorphism

- Means that the same method will behave differently when it is applied to the objects of different classes
- It also means that different methods associated with different classes can interpret the same message in different ways.
- Example: an object can send a message PRINT to several objects, and each one will use it’s own PRINT method to execute the message.

Binding

- Associating a method call with the method code to run
- Resolving ambiguity in the context of overloaded methods
- Choices for binding time
- Static: Compile-time
- Dynamic : Run-time

Agenda

- Review
- Procedural VS Object Oriented Programming
- Introduction to Object Oriented Programming Concepts
- **Data Types and Operators in C++**
- **Use of Arrays, Pointers and Structures in C++**
- **Summary**

C++: Data Types

- C++ is a programmer’s language - need to know the basics…
- There are 6 atomic data types:
- char - character (1 byte)
- int - integer (usually 4 bytes)
- float - floating point (usually 4 bytes)
- double - double precision floating point (usually 8 bytes)
- bool - true or false (usually 4 bytes)
- void - explicitly says function does not return a value
  and can represent a pointer to any data type
- Size of the data types depends on machine architecture
  e.g. 16 bit, 32 bit or 64 bit words
- Other data types are derived from atomic types e.g. long int
- Can use ‘typedef’ to alias your own data type names;
  defining C++ classes creates new types

C++: Variables and Scope

```
int a, b, c;
a = 1;
b = c = 0x3F;
```

```
float iAmAFloat = 1.234;
double iAmADouble = 1.2e34;
```

```
{
  int i, a;

  for (i=0; i<10; i++) {
    a = i;
    int b = i;
  }
  b = 2;
}
```

‘a’ declared outside loop braces

‘b’ declared inside braces; ‘b’ is in scope inside braces

‘a’ used inside braces, OK

‘b’ is unknown outside braces: ‘b’ is out of scope, ERROR

C++: Operators

- Obvious: +, -, *, /
  Shorthand: +=, *=, -=, /=
- Modulus: %
- Decrement: --
- Increment: ++
- Relational: ==, !=, <, >, <=, >=
- Logical: !, &&, ||, &, |, ^, ~

- 5%3 evaluates to 2 (the remainder of division)

```
int a, b;

a++;
b--;
```

means the same as

```
a = a + 1;
b = b - 1;
```

C++: Statements

- A statement is a part of the program that can be executed
- Statement categories:
- Selection
- Iteration
- Jump
- Expression
- Try (exception handling; look it up)
- Statements specify actions within a program. Generally they are responsible for control-flow and decision making: e.g. if (some condition) {do this} else {do that}

C++: Arrays and Strings

- Arrays: indexed collections of identical-type objects.
- Array index always start on 0
- Arrays can be used in two different ways: primitive arrays and vectors.
- Arrays can be Single Dimensional or Multi-Dimensional.
- Strings:

```
#include<string>
   string s=“Hello World!”;
   int size=s.length();
   cout<< s[4] <<endl;          // result:“o”
   cout<< s <<endl;
```

C++: Pointers

- How to Declare a pointer:

```
int *ptr;
```

- & : unary operator that returns the address of the object, it is placed before.

```
int x=5;            int *ptr;
ptr=&x;             cout << ptr << endl;   // output: 0013FF7C
```

- * : unary de-referencing operator which can access data/object being pointed.

```
*ptr = 10;
```

C++: Pointers Cont’d

Legal Pointer Syntax

int x=10

Declare a pointer:

int *ptr=&x

After declare:

*ptr=15

OR

int *ptr

ptr=&x

Illegal Pointer Syntax

- int *ptr //run time error
  *ptr=&x

OR

- ptr=x or int *ptr=x

C++: Pointers Cont'd

- *ptr = x // Symantically incorrect

- What happens bellow ?

```
int x=5;
int *ptr = &x;
*ptr +=1;
*ptr++;
```

C++: Structures

- A Structure is a container, it can hold a bunch of things.
- These things can be of any type.
- Structures are used to organize related data (variables) into a nice neat package.
- Example: Student Record:

Name	a string
HW Grades	an array of 3 doubles
Test Grades	an array of 2 doubles
Final Average	a double

Structure: Members

- Each *thing* in a structure is called *member*.
- Each *member* has a name, a type and a value.
- Names follow the rules for variable names.
- Types can be any defined type.

Structure: Definition Example

```
struct StudentRecord {
        char *name;                     // student name
        double hw[3];                   // homework grades
        double test[2];                 // test grades
        double ave;                     // final average
};
```

Structure: Accessing Members

- You can treat the members of a struct just like variables.
- You need to use the *member access operator* '.' (pronounced "dot"):

```
void main ()
{
        StudentRecord stu;
        cout << stu.name << endl;
        stu.hw[2] = 82.3;
        stu.ave = total/100;
}
```

Pointer to Structure

- Pointers to structures are used often.
- There is another member access operator used with pointers: ->

```
        StudentRecord *sptr;
        sptr = &stu;
        cout << "Name is" << sptr->name;
OR      cout << "Name is" << (*sptr).name;
        cout << "Ave is " << sptr->ave;
OR      cout << "Ave is " << (*sptr).ave;
```

Other Stuff to do with a Struct

- You can also associate special functions with a structure (called member functions).
- A C++ class is very similar to a structure, we will focus on classes later.
- Classes can have (data) members
- Classes can have member functions.
- Classes can also hide some of the members (functions and data).

THANK YOU!

Any Questions Please?